HPCLATAM 2012
Proceedings
ISSN 2422-5207

HPCLATAM 2012

V Latin American Symposium on High Performance Computing

Buenos Aires, Argentina

July 23-24, 2012

Conference Proceedings

E. Mocskos, S. Nesmachnow (editors)

July 2012

HPCLATAM 2012
V Latin American Symposium on High Performance Computing
Buenos Aires, Argentina
July 23-24, 2012

Conference Proceedings
Esteban Mocskos, Sergio Nesmachnow (editors)
ISSN 2422-5207

FOREWORD/PREFACE

This compilation publishes the papers presented in HPCLATAM 2012, the V Latin American Symposium on High Performance Computing.

The use and development of High Performance Computing in Latin America is steadily growing. The new challenges coming from the use of the computing capabilities of clusters, grids, and distributed systems for HPC, help to promote the research and innovation in this area.

Building on the great success of the previous four editions, in 2012 the Latin American Symposium on High Performance Computing grew to include three major events: the V HPCLATAM2012 International Symposium (Buenos Aires, July 23-24), the High Performance Computing School (ECAR 2012, Buenos Aires, July 25 to August 3), and the HPC Day (La Plata, August 30) within the 41st Argentine Conference of Informatics (41 JAIIO).

The HPCLATAM2012 International Symposium provided a regional forum fostering the growth of the HPC community in Latin America through the exchange and dissemination of new ideas, techniques, and research in High Performance Computing. The symposium featured invited talks from academy and industry, short- and full-paper sessions presenting both mature work and new ideas in research and industrial applications. The submitted articles presented contributions in the areas of Parallel Algorithms and Architectures, High Performance Applications, Tools and Environments for High Performance System Engineering, Graphics Processing Units in High Performance Computing, Distributed and Grid Computing, and Parallelism and Data Sharing on Multi-core Architectures.

In our opinion, the articles published in this book are valuable contributions to the development of high performance computing in Latin America.

Esteban Mocskos, Sergio Nesmachnow (editors)

July 2012

Esteban Mocskos, Universidad de Buenos Aires, Argentina, emocskos@dc.uba.ar

Sergio Nesmachnow, Universidad de la República, Uruguay, sergion@fing.edu.uy

HPCLATAM 2012

V Latin American Symposium on High Performance Computing

ORGANIZATION/COMMITEES

CHAIRS

- Esteban Mocskos, Universidad de Buenos Aires, Argentina
- Sergio Nesmachnow, Universidad de la República, Uruguay

TECHNICAL PROGRAM COMMITTEE

- Adrián Cristal, Barcelona Supercomputing Center, Spain
- Andrés More, Intel, Argentina
- Alvaro Coutinho, Universidade Federal do Rio de Janeiro, Brasil
- Carlos Garcia Garino, Universidad Nacional de Cuyo, Argentina
- Cristian Perfumo, University of Newcastle, Australia
- Diego Crupnicoff, Mellanox Technologies, USA
- Eduardo Bringa, Universidad Nacional de Cuyo, Argentina
- Gerson Geraldo Cavalheiro, Universidade Federal de Pelotas, Brasil
- Gonzalo Hernández Oliva, Universidad de Valparaíso, Chile
- Francisco Brasileiro, Universidade Federal de Campina Grande, Brasil
- Marcela Printista, Universidad Nacional de San Luis, Argentina
- Mariano C. González Lebrero, Universidad de Buenos Aires, Argentina
- Mariano Vázquez, Barcelona Supercomputing Center, Spain
- Mario Storti, Universidad Nacional del Litoral, Argentina
- Miguel Angel Cavaliere, Tenaris Siderca and Universidad de Buenos Aires, Argentina
- Nicolás Wolovick, Universidad Nacional de Córdoba, Argentina
- Pablo Mininni, Universidad de Buenos Aires, Argentina
- Alejandro Soba, CNEA-CONICET, Argentina
- Patricia Tissera, IAFE, Universidad de Buenos Aires, Argentina
- Ricardo Medel, Intel, Argentina
- Roberto Bevilacqua, Comisión Nacional de Energía Atómica, Universidad de Buenos Aires, Universidad Nacional de San Martin, Argentina

HPCLATAM 2012

V Latin American Symposium on High Performance Computing Proceedings

INDEX

HPCLatAm 2012, pp. 1-16 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

Parallel Adaptive Simulation of Coupled Incompressible Viscous Flow and Advective-Diffusive Transport Using Stabilized FEM Formulation

Andre Rossa[1] and Alvaro Coutinho[2]

[1] Engineering Simulation and Scientific Software,
Avenida Presidente Vargas, 3131, 20210-031,
Rio de Janeiro, Brazil
andre.rossa@esss.com.br
http://www.esss.com.br
[2] High-Performance Computing Center, Department of Civil Engineering,
Federal University of Rio de Janeiro,
Cidade Universitária, Centro de Tecnologia, Bloco I, Sala I-248, 21941-972
Rio de Janeiro, Brazil
alvaro@nacad.ufrj.br
http://www.nacad.ufrj.br

Abstract. In this work we study coupled incompressible viscous flow and advective-diffusive transport of a scalar. Both the Navier-Stokes and transport equations are solved using an Eulerian approach. The SUPG/PSPG stabilized finite element formulation is applied for the 8-node isoparametric hexahedron. The implementation is held using the `libMEsh` finite element library which provides the support for adaptive mesh refinement and coarsening and parallel computation. The Rayleigh-Bénard natural convection and the planar lock-exchange density current problems are solved to assess the adaptive parallel performance of the numerical solution.

Keywords: Stabilized FEM formulation, incompressible flows, adaptive meshes, parallel computing.

1 Introduction

The numerical simulation of current engineering problems would not be feasible without the advent of parallel computing. Even with the development of techniques for mesh adaptation, the fastest available processors are not able to solve, within a practical period of time, problems that have large amounts of degrees of freedom. High-performance computing (HPC) have enabled the solution of problems with a large number of unknowns and high complexity (often involving multiple scales and multiple physics) by clusters of computers installed in universities as well in industry research centers.

HPCLatAm 2012, pp. 2-16 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

To make HPC be efficiently used, a set of algorithms and computational methods have been developed over the last decade that made possible high fidelity solution of complex problems. Processors with multiple cores with shared memory, clusters of personal computers in which each processor has its own memory (distributed memory) and more recently, the hybrid memory architectures are present on the daily work of engineers and researchers.

Although improving the processing capacity parallel computation have added complexity to the computer codes programing. According to [1] scaling performance is particularly problematic because the vision of seamless scalability cannot be achieved without having the applications scale automatically as the number of processors increases. However, for this to happen, the applications have to be programmed to exploit parallelism efficiently. Therefore, parallel computing resources should be used rationally in order to obtain compatible performances.

Good simulation practice suggests that the applications and algorithms employed in HPC should be optimized for this purpose. Currently there are available (mostly freely distributed) several programs to perform different tasks inherent to parallel computing. The domain partitioning and load balancing, information exchange between processors, algebraic operations and linear preconditioned systems solving are some of the necessary operations and have specific computational libraries that can be incorporated into the implementation of a numerical simulator.

In order to keep the focus on the issues related to the numerical problem, we use the `libMesh` framework, which is a C++ library for parallel adaptive mesh refinement/coarsening numerical multiphysics simulations based on the finite element method [2]. The library has been developed since 2002 by a group of researchers from CFDLab, Department of Aerospace Engineering and Engineering Mechanics, University of Texas at Austin, and is available as open source software (`http://libmesh.sourceforge.net/`).

In this work we implement stabilized finite element formulations for the Navier-Stokes and advective-diffusive transport in `libMesh`. Parallel adaptive simulations of coupled problems confirm the mesh size reduction potential and the ability to capture the solution lower scales as well the development of the interface between the fluids in evolution problems.

The remaining of this work is organized as follows. In the next section the dimensionless governing equations for the coupled viscous flow and transport is presented together with correspondent stabilized SUPG/PSPG finite element method (FEM) formulation. Details of the adaptive mesh refinement/coarsening (AMR/C) in the context of the `libMesh` library as well some aspects of the parallel solution of precondiotioned linear systems are presented in Section 3. Section 4 presents the results of the parallel adaptive simulation of the Rayleigh-Bnard natural convection and a density current in a planar lock-exchange configuration. The paper ends with the main conclusions.

HPCLatAm 2012, pp. 3-16 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

2 Mathematical Formulation

2.1 Governing Equations

Assuming an unsteady incompressible viscous flow and the Bousinessq approximation, the dimensionless Navier-Stokes, continuity and scalar transport equations[3] can be written in a non-conservative way following a Eulerian description as

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\nabla\mathbf{u} - \frac{1}{Re}\nabla^2\mathbf{u} + \nabla p = \frac{Gr}{Re^2}\phi\mathbf{e} \quad \text{in} \quad \Omega \times [0,t] \ , \tag{1}$$

$$\nabla \cdot \mathbf{u} = 0 \quad \text{in} \quad \Omega \times [0,t] \ , \tag{2}$$

$$\frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla\phi - \frac{1}{D}\nabla^2\phi = 0 \quad \text{in} \quad \Omega \times [0,t] \ . \tag{3}$$

defined in the simulation domain Ω which is surrounded by the smooth boundary Γ. The time is t, $\mathbf{u} = (u, v, w)^T$ is the velocity field, p is the pressure and ϕ the scalar being transported and

$$\mathbf{e} = \frac{\mathbf{g}}{\|\mathbf{g}\|} \tag{4}$$

is an unit vector aligned with the gravity $\mathbf{g}$.

In (1) Re and Gr are the Reynolds and Grashof numbers. The parameter D in equation (3) represents a dimensionless diffusive constant depending on the nature of the scalar being transported (e.g., the Peclet number for the temperature transport).

The essential and natural boundaries conditions are:

$$\begin{aligned} \mathbf{u} &= \mathbf{g} && \text{on } \Gamma_{\mathbf{g}} \ , \\ \mathbf{n} \cdot \left[\frac{1}{Re} \left(\nabla\mathbf{u} + (\nabla\mathbf{u})^T \right) - p\mathbf{I} \right] &= \mathbf{h} && \text{on } \Gamma_{\boldsymbol{\sigma}} \ , \\ \phi &= \overline{\phi} && \text{on } \Gamma_{\phi}, \\ -\mathbf{n} \cdot \nabla\phi &= q && \text{on } \Gamma_q \end{aligned} \tag{5}$$

where $\mathbf{n}$ is the unit outward normal vector on the boundary and $\mathbf{I}$ is the 3×3 identity matrix.

The initial conditions are:

$$\begin{aligned} \mathbf{u}(\mathbf{x}, 0) &= \mathbf{u_0} \ , \\ \phi(\mathbf{x}, 0) &= \phi_0 \end{aligned} \tag{6}$$

where the initial velocity field $\mathbf{u_0}$ is divergent free.

[3] Details on how the physical quantities may be normalized in order to arrive at dimensionless equations can be found at [3].

HPCLatAm 2012, pp. 4-16 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

In gravity current problems, the concept of buoyancy velocity u_b is largely used (see [5]). It may be defined as

$$u_b := \sqrt{g^{'} h_v} \tag{7}$$

where h_v is a scale length related to the vertical dimension of the simulation domain (usually taken as the domain height) and $g^{'}$ is called the reduced gravity given by

$$g^{'} := g\frac{\rho_1 - \rho_2}{\rho_\infty} \tag{8}$$

where g is the absolute value of the gravitational acceleration, ρ_∞ is the reference density, ρ_1 is the density of the "heavy" fluid and ρ_2 is the density of the "light" fluid.

When one uses the buoyancy velocity as a reference velocity and h_v as the scale length, the Reynolds number may be computed directly from the Grashof as follow

$$Re = \sqrt{Gr}\ . \tag{9}$$

For particle-driven problems (a class of gravity current phenomenon), where the transported scalar is the density ρ, the diffusivity constant is given by the product of the Schmidt Sc and Grashof numbers. Taking it into account, the Navier-Stokes and advective-diffusive equations may be rewritten as

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\nabla\mathbf{u} - \frac{1}{\sqrt{Gr}}\nabla^2\mathbf{u} + \nabla p = \rho\mathbf{e}\ , \tag{10}$$

$$\frac{\partial \rho}{\partial t} + \mathbf{u}\cdot\nabla\rho - \frac{1}{ScGr}\nabla^2\rho = 0\ . \tag{11}$$

2.2 Stabilized Finite Element Formulation

Given a suitably defined finite-dimensional trial solution and weight functions spaces for velocity and pressure

$$\begin{aligned} S^h_{\mathbf{u}} &= \left\{ \mathbf{u}^h \mid \mathbf{u}^h \in \left[H^{1h}(\Omega)\right]^3,\ \ \mathbf{u}^h \doteq \mathbf{g}^h \ \ \text{em} \ \ \Gamma_{\mathbf{g}} \right\}\ , \\ V^h_{\mathbf{w}} &= \left\{ \mathbf{w}^h \mid \mathbf{w}^h \in \left[H^{1h}(\Omega)\right]^3,\ \ \mathbf{w}^h \doteq 0 \ \ \text{em} \ \ \Gamma_{\mathbf{g}} \right\}\ , \\ S^h_p &= V^h_p = \left\{ q^h \mid q^h \in H^{1h}(\Omega) \right\} \end{aligned} \tag{12}$$

where $H^{1h}(\Omega)$ is the finite-dimensional space function square integrable into the element domain, the stabilized SUPG/PSPG FEM formulation for the non-dimensional Navier-Stokes and continuity equations (1) and (2) can be written

HPCLatAm 2012, pp. 5-16 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

as: Find $\mathbf{u}^h \in S^h_{\mathbf{u}}$ and $p^h \in S^h_p$ such as, $\forall \mathbf{w}^h \in V^h_{\mathbf{w}}$ and $\forall q^h \in V^h_p$,

$$\begin{aligned}
&\int_\Omega \mathbf{w}^h \cdot \left[\left(\frac{\partial \mathbf{u}^h}{\partial t} + \mathbf{u}^h \nabla \mathbf{u}^h\right) - \mathbf{l}^h\right] d\Omega + \frac{1}{Re}\int_\Omega \left(\nabla \mathbf{w}^h\right)^T \cdot \nabla \mathbf{u}^h \mathbf{I} d\Omega - \\
&\int_\Omega \nabla \mathbf{w}^h p^h \mathbf{I} d\Omega - \int_\Gamma \mathbf{w}^h \cdot \mathbf{h}^h d\Gamma + \int_\Omega q^h \nabla \cdot \mathbf{u}^h d\Omega + \\
&\sum_{e=1}^{n_el} \int_{\Omega^e} \left(\tau_{SUPG} \mathbf{u}^h \nabla \mathbf{w}^h\right) \cdot \left[\left(\frac{\partial \mathbf{u}^h}{\partial t} + \mathbf{u}^h \nabla \mathbf{u}^h\right) + \nabla p^h - \mathbf{l}^h\right] d\Omega^e + \\
&\sum_{e=1}^{n_el} \int_{\Omega^e} \left(\tau_{PSPG} \nabla q^h\right) \cdot \left[\left(\frac{\partial \mathbf{u}^h}{\partial t} + \mathbf{u}^h \nabla \mathbf{u}^h\right) + \nabla p^h - \mathbf{l}^h\right] d\Omega^e = 0 \,.
\end{aligned} \tag{13}$$

The first four integrals in (13) arise from the classical Galerkin weak formulation for the Navier-Stokes equations. The fifth integral represents the classical Galerkin formulation for the continuity equation. The summations over the elements are the SUPG and the PSPG stabilizations for the Navier-Stokes equation. The parameters adopted for both stabilizations were obtained from [4] and are defined as follows

$$\tau_{SUPG} = \tau_{PSPG} = \left[\left(2\frac{\left\|\mathbf{u}^h\right\|}{\mathfrak{h}}\right)^2 + 9\left(\frac{4}{Re\mathfrak{h}^2}\right)^2\right]^{-\frac{1}{2}} . \tag{14}$$

The dimensionless stabilizations parameters are local (element level) so, the velocity modulus $\left\|\mathbf{u}^h\right\|$ is calculated for each element e and $\mathfrak{h}$ is an element length measure based in its volume $\mathfrak{V}$ as shown bellow

$$\mathfrak{h} = \sqrt[3]{\frac{6\mathfrak{V}}{\pi}} . \tag{15}$$

The discretized dimensionless body force are represented by $\mathbf{l}^h$.

For the dimensionless advective-diffusive transport we adopt the same assumptions, so given the following finite-dimensional trial solution and weight functions spaces for the scalar

$$\begin{aligned}
S^h_\phi &= \left\{\phi^h \mid \phi^h \in H^{1h}(\Omega), \quad \phi^h \doteq \overline{\phi}^h \ \text{ in } \ \Gamma_\phi\right\} , \\
V^h_w &= \left\{w^h \mid w^h \in H^{1h}(\Omega), \quad w^h \doteq 0 \ \text{ em } \ \Gamma_\phi\right\}
\end{aligned} \tag{16}$$

the stabilized FEM formulation can be written as: Find $\phi^h \in S^h_\phi$ such as, $\forall w^h \in V^h_w$,

$$\begin{aligned}
&\int_\Omega w^h \cdot \left(\frac{\partial \phi^h}{\partial t} + \mathbf{u}^h \cdot \nabla \phi^h\right) d\Omega + \frac{1}{D}\int_\Omega \left(\nabla w^h\right)^T \cdot \nabla \phi^h d\Omega - \int_\Gamma w^h q d\Gamma + \\
&\sum_{e=1}^{n_el} \int_{\Omega^e} \left(\tau_{SUPG} \mathbf{u}^h \cdot \nabla w^h\right) \cdot \left[\left(\frac{\partial \phi^h}{\partial t} + \mathbf{u}^h \cdot \nabla \phi^h\right)\right] d\Omega^e = 0 \,.
\end{aligned} \tag{17}$$

HPCLatAm 2012, pp. 6-16 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

The three first integrals in (17) come from the Galerkin weak formulation. The integral into the summation over the elements is the SUPG stabilization. The non-dimensional stabilization parameter is computed similarly to (14), that is:

$$\tau_{SUPG} = \left[\left(2\frac{\left\| \mathbf{u}^h \right\|}{\mathfrak{h}} \right)^2 + 9\left(\frac{4}{D\mathfrak{h}^2} \right)^2 \right]^{-\frac{1}{2}} . \tag{18}$$

In the stabilized formulations (17), an additional stabilization is added to handle instabilities in the numerical solution of flows with presence of strong gradients of the scalar being transported. [6] present a discontinuity capturing term which is calculated as follows:

$$\sum_{e=1}^{n_el} \int_{\Omega^e} \delta\left(\phi^h\right) \nabla w^h \cdot \nabla \phi^h d\Omega^e . \tag{19}$$

Because the δ parameter is a function of the scalar, (17) can be understood as a nonlinear diffusion operator. In this work, the δ parameter was adapted from [6] as follows in dimensionless form

$$\delta\left(\phi^h\right) = \left| \frac{1}{\phi^*} R\left(\phi^h\right) \right| \left(\sum_{i=1}^{3} \left| \frac{1}{\phi^*} \frac{\partial \phi^h}{\partial x_i} \right|^2 \right)^{\beta/2-1} \frac{\mathfrak{h}}{2}^{\beta} \tag{20}$$

where ϕ^* is a dimensionless value of the scalar (usually taken as 1) and $R\left(\phi^h\right)$ is an approximation for the actual residual defined as:

$$R\left(\phi^h\right) = \frac{\partial \phi^h}{\partial t} + \mathbf{u}^h \cdot \nabla \phi^h . \tag{21}$$

The β parameter can be set as 1 or 2.

2.3 Discretized Systems

Adopting the implicit backward Euler scheme for the time discretization together with a fixed point linearization, the final discrete system of (13) and (17) results in

$$\begin{aligned} &\left(\mathbf{M} + \mathbf{M}_\tau\right) \mathbf{u}^{n+1,k+1} + \Delta t \left(\mathbf{N}\left(\mathbf{u}^{n+1,k}\right) + \mathbf{N}_\tau\left(\mathbf{u}^{n+1,k}\right) + \mathbf{K} \right) \mathbf{u}^{n+1,k+1} - \\ &\Delta t \left(\mathbf{G} - \mathbf{G}_\tau \right) \mathbf{p}^{n+1,k+1} = \Delta t \left(\mathbf{f}\left(\phi^n\right) + \mathbf{f}_\tau\left(\phi^n\right) \right) + \left(\mathbf{M} + \mathbf{M}_\tau\right) \mathbf{u}^n , \end{aligned} \tag{22}$$

$$\begin{aligned} &\Delta t \mathbf{G}^T \mathbf{u}^{n+1,k+1} + \mathbf{M}_\xi \mathbf{u}^{n+1,k+1} + \Delta t \left(\mathbf{N}_\xi\left(\mathbf{u}^{n+1,k}\right) \mathbf{u}^{n+1,k+1} + \mathbf{G}_\xi \mathbf{p}^{n+1,k+1} \right) = \\ &\Delta t \mathbf{f}_\xi\left(\phi^n\right) + \mathbf{M}_\xi \mathbf{u}^n , \end{aligned} \tag{23}$$

HPCLatAm 2012, pp. 7-16 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

$$
\begin{aligned}
&\left(\mathbf{M}+\mathbf{M}_{\tau}\right)\boldsymbol{\phi}^{n+1,k+1}+\\
&\Delta t\left(\mathbf{N}\left(\mathbf{u}^{n+1}\right)+\mathbf{N}_{\tau}\left(\mathbf{u}^{n+1}\right)+\mathbf{K}+\mathbf{K}_{\delta}\left(\boldsymbol{\phi}^{n+1,k}\right)\right)\boldsymbol{\phi}^{n+1,k+1}= \qquad (24)\\
&\left(\mathbf{M}+\mathbf{M}_{\tau}\right)\boldsymbol{\phi}^{n}\ .
\end{aligned}
$$

In the matrix systems (22), (23) and (24) $\mathbf{u}$, $\mathbf{p}$ and $\boldsymbol{\phi}$ are the nodal vectors of the correspondent unknowns $\mathbf{u}^h$, $\mathbf{p}^h$ and $\boldsymbol{\phi}^h$, and Δt stands for the time-step size. The super indexes $n+1$ and n mean the current and previous time-steps while $k+1$ and k are respectively the current and previous nonlinear iterations counter.

For the matrices where the advective operator appears, i.e., Galerkin advection, SUPG mass and advection, and PSPG advection, the velocity components are evaluated at each integration point. $\mathbf{M}$ is the mass matrix, $\mathbf{K}$ is the viscous/diffusive matrix, $\mathbf{N}(\mathbf{u})$ is the nonlinear advection matrix, $\mathbf{G}$ and $\mathbf{G}^T$ are the gradient and its transpose (divergent) matrices. $\mathbf{f}$ is the body force vector. $\mathbf{K}_{\delta}(\boldsymbol{\phi})$ is the nonlinear discontinuity capturing matrix. The matrices and vectors with the subscripts τ and ξ mean the SUPG and PSPG terms.

3 Computational Aspects

3.1 Mesh Adaptivity

The mesh adaptivity together with high-performance computing (parallel processing) play a key role to enable numerical simulations of actual engineering/industrial problems within an acceptable time without exhausting the processing capacity of current computers.

Particularly for the density current problem, AMR/C is a important tool to capture and track the flow structure at the front, where the Kelvin-Helmholtz billows occur. From an initial coarse mesh, the adaptivity refinement process begins near the interface between the two fluids and it follows its development.

In `libMesh` the mesh refinement can be accomplished by element subdivision (h-refinement), increasing the local polynomial degree (p-refinement) as well a combination of both methods (hp-refinement). Although there is an extensive literature devoted to obtaining reliable a posteriori estimators that are more closely linked to the operators and governing equations [7, 8], in `libMesh` the error indicator is focused on local indicators that are essentially independent of the physics [2].

`libMesh` uses a statistical refinement/coarsening scheme based on the ideas presented in [9] where the mean μ and the standard deviation σ of the error indicator "population" are computed. Using refinement and coarsening fractions (r_f and r_c), the elements are flagged for refinement and coarsening as showed in Fig. (1).

This scheme is suitable for evolution problems where, in the beginning, a small error is evenly distributed. Throughout the simulation the error distribution spreads and the AMR/C process starts. Whether the solution approaches

HPCLatAm 2012, pp. 8-16 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

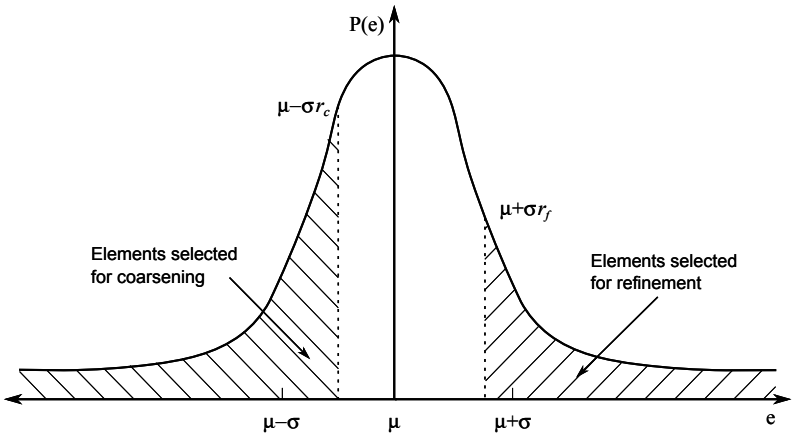


Fig. 1. Statistical refinement: elements in hatched areas are flagged to AMR/C process.

its steady-state, the distribution of error also reaches the steady-state, stopping the AMR/C process.

The elements are refined through a "natural refinement" scheme: elements of dimension d, with the exception of the pyramids, produce 2^d elements of the same type after refinement. The degrees of freedom are constrained at the hanging nodes on element interfaces. This approach yields a tree data structure formed by the "parents" and their "children" elements. The elements present in the initial mesh (level-0 elements) have no parents as well the active elements (those that are part of the current simulation mesh) have no children, so the latter are the current high-level elements. The element level is determined recursively from its parents and the user should determine the maximum refinement level.

The frequency of refinement/coarsening is an user's responsibility. When the mesh adapts it is optimized for the state at the current time [10]. One should find a frequency of adaptivity which will balance the computational effort and quality of results as there is a computational cost associated with the adaptivity process. When the mesh is adapted, the field solution should be projected (interpolated) and a simulation should be performed on the new mesh.

In this work we use a class of errors estimators based on derivative jump (or flux jump) of the transported scalar calculated at the elements interface called Kelly's Error Estimator [11] to perform the h-refinement. The refinement and coarsening fractions for the statistical strategy as well the adaptivity frequency are set independently for each simulation problem.

3.2 Domain Decomposition

In this paper, we consider the standard partitioning domain without overlap, as shown in Fig. (2), where the elements related to each of the sub-domains are assigned to different processors. That is, the simulation domain Ω^h is divided into a discrete set of sub-domains Ω_p^h such as $\bigcup \Omega_p^h = \Omega^h$ and $\bigcap \Omega_p^h = \emptyset$.

HPCLatAm 2012, pp. 9-16 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

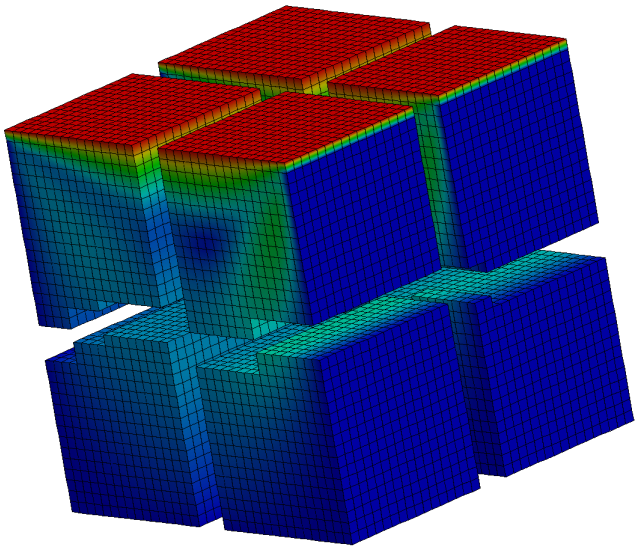


Fig. 2. Simulation domain decomposition in 8 sub-domains.

In AMR/C computations at new stage adaptation, regions of the domain will have an increase in mesh element density while in others, the number of elements will decrease. These dynamic mesh adjustments result for some processors in significant increasing (or decreasing) work therefore causing unbalanced load [1].

Libraries such as METIS and ParMETIS were developed aiming implementing efficient partitioning mesh schemes. The first is a serial library partitioning, while the second is based on parallel MPI. Both can also reorder the unknowns in unstructured grids to minimize the fill-in during LU factorization. The ParMETIS extends the functionality provided by METIS and includes routines for parallel computations with adaptive meshes refinement and large-scale numerical simulations.

3.3 Parallel Solution of Preconditioned Linear Systems

The support for the numerical solution of the differential equations in the parallel architecture environment is provided by PETSc. It provides structures for efficient storage (vectors, arrays, for example), as well ways to handle it. The `libMesh` uses compressed sparse row (CSR) structure. The PETSc has a number of methods for solving linear sparse system as GMRES and BiConjugate Gradient method (BiCG) and several types of preconditioners as ILU(k) and Block-Jacobi. Options for reordering the linear system as the Reverse Cuthill-McKee method (RCM, [12]) are also available.

In this work we use Block-Jacobi sub-domain preconditioning. It is one of the most widely used schemes due to its easiness of implementation. There are

HPCLatAm 2012, pp. 10-16 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

no overlap between the blocks. The incomplete factorization may be applied to each of them without extra cost in communication. However, in adaptive simulations, the new local factorization should be performed every time the mesh is modified since the adaptivity changes the group of elements residing in each sub-domain [13]. It is usual to refer to the Block-Jabobi preconditioning strategy in conjunction with the ILU factorization with a certain level k of fill-in by BILU(k) [14].

The communication between the processors, such as required for the algebraic operations or during assembly of arrays of elements are supported by a set of PETSc library routines which is designed for parallel computing using the MPI API.

4 Numerical Results

The simulations were performed in a SGI Altix ICE 8400 cluster with 640 cores (Intel Nehalem). This machine has 1.28 TB of distributed memory. The processing nodes are connected by InfiniBand. The cluster is located at the High-Performance Computing Center (NACAD) of the Federal University of Rio de Janeiro, Brazil.

4.1 Parallel Adaptive Simulation of the Rayleigh-Bénard Problem

In this example we consider the Rayleigh-Bénard natural convection in a container with geometric domain $\Omega = [0, 4] \times [0, 1] \times [0, 1]$. This problem consists to solve a natural convection phenomenon of a fluid which initially at rest ($t = 0$) produces a sequence of adjacent convection cells along the longitudinal direction (x axis) due to the temperature difference between its upper (cold) and lower (hot) walls.

No-slip boundary conditions are imposed in all the walls and the pressure is prescribed as $p(2.0, 0.5, 0.0) = 0.0$. The dimensionless cold temperature is $T_c = -0.5$ and the hot $T_h = 0.5$. The physical problem is defined setting the Reynolds Number as $Re = 4,365$, Grashof number as $Gr = 41,666.66$, the Peclet number as $Pe = 3,142.8$ and Froude number[4] as $Fr = 0.6432$.

Only one refinement/coarsening level was allowed at every 25 time-steps. For the statistical adaptivity scheme the refinement fraction is $r_f = 0.6$ and coarsening fraction is $r_c = 0.01$. The linear tolerance for GMRES(30) together with the BILU(1) and reordering by RCM method is 1.0×10^{-6}. The nonlinear tolerance is 1.0×10^{-5} and the constant time step size is $\Delta t = 5.0$.

The steady-state velocity vectors are shown in Fig.(3) and the temperature over the final adapted mesh is plotted in Fig.(4).

[4] Besides not introduced in the Navier-Stokes equations presented in section 2.1, the Froude number is used here to take into account the fluid's weight in the calculation. More details about how to incorporate the Froude number in the dimensionless Navier-Stokes equations may be found in [3].

HPCLatAm 2012, pp. 11-16 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

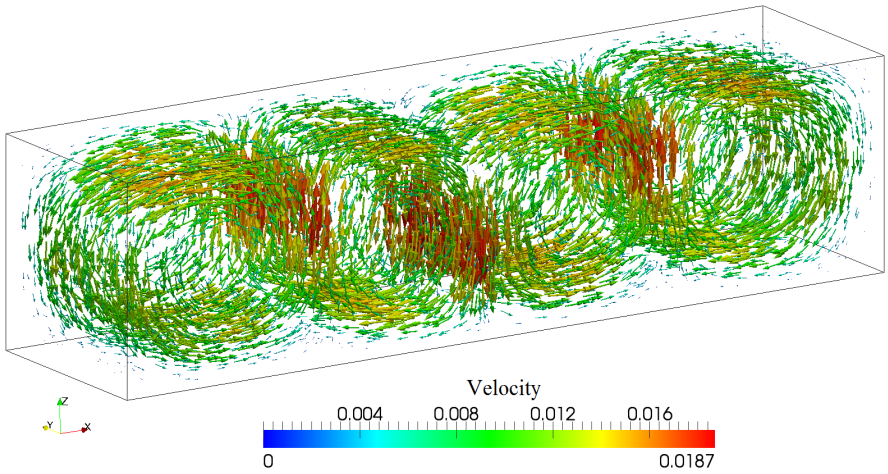


Fig. 3. Steady-state velocity vectors.

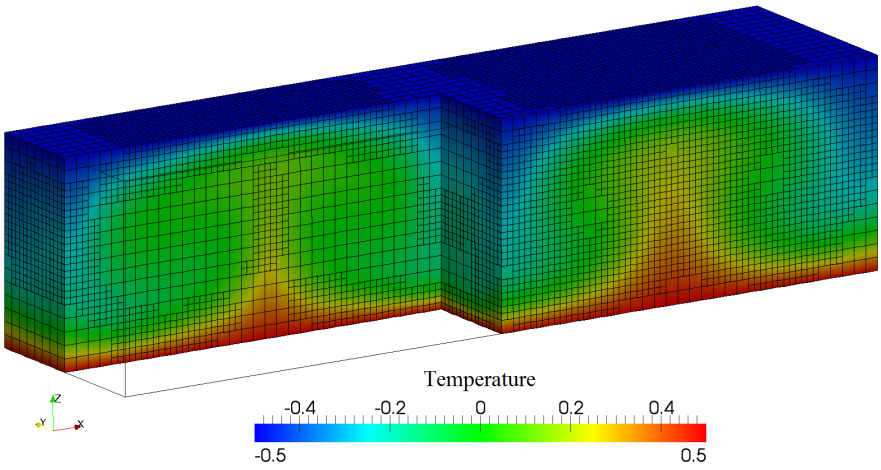


Fig. 4. Temperature at steady-state and final adapted mesh.

The Fig. (5) presents the speedup for the total simulation time, the total time for solving the Navier-Stokes and transport problems and the AMR/C procedure considering in the numerator the time spent with 16 CPU's, i.e.,

$$S_p = \frac{\tau_{16}}{\tau_p} \, . \tag{25}$$

HPCLatAm 2012, pp. 12-16 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

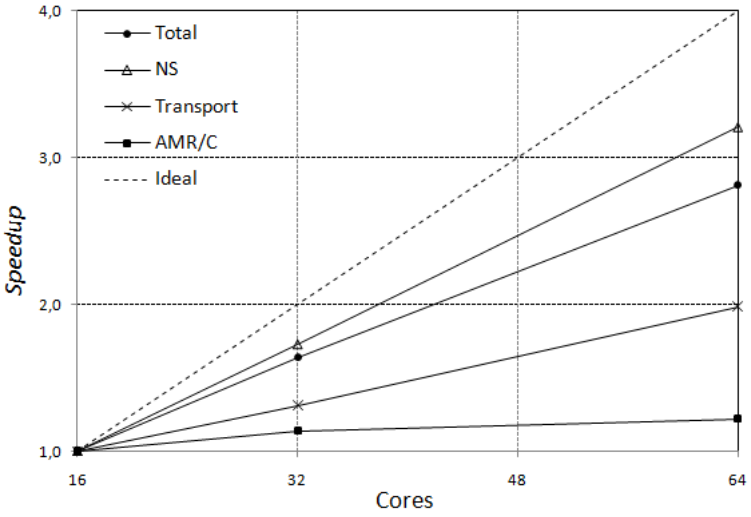


Fig. 5. Speedup for the Rayleigh-Bénard problem.

The AMR/C time does not reach 10% of the total simulation time. We may observe from the results of Fig.(5) that the present simulation achieves a good parallel performance, that is, speed up around 3 for the total simulation with 64-cores run with respect to 16-cores run (over 3 for the Navier-Stokes simulation).

Despite the good overall performance, it is observed that the adaptive procedure does not scale as well as the linear solvers ($S_{64} = 1.22$). The poor performance of the AMR/C procedure in the current `libMesh` release is due to the fact that all mesh data are replicated on all cores, which increases memory requirements and communication per core.

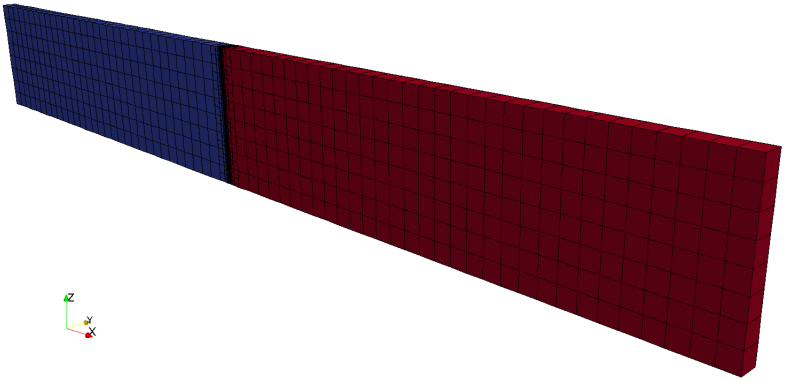
4.2 Temperature-driven gravity current with AMR/C

For the simulation of a temperature-driven gravity current with mesh adaptivity, we consider a slice domain[5] $\Omega = [0, L] \times [0, H/8] \times [0, H]$ where the nondimensional length and height are $L = 0.8$ and $H = 0.1$. The left half of the channel is initially filled with the cold fluid and the right half is filled with hot fluid. The Fig. (6) shows the initial configuration and a detail of the refinement at the center of the domain.

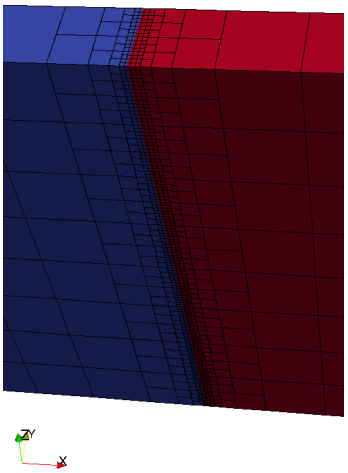
The dimensionless cold temperature is set to $T_c = -0.5$ and the hot $T_h = 0.5$. We consider no-slip boundary conditions on the bottom, left and right walls.

[5] To emulate a 2D simulation domain from a mesh composed of 3D elements, the slice domain is positioned parallel to the xz plane and the perpendicular direction $(0, y, 0)$ is discretized with only one element except at regions where the mesh adapts. For all nodes on the mesh $v_y = 0.0$ is imposed.

HPCLatAm 2012, pp. 13-16 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina



(a)



(b)

Fig. 6. Initial lock-exchange configuration: (a) View of the slice domain and (b) Mesh detail.

Free-slip boundary conditions are imposed on the top one. Thus free and no slip fronts may be considered in the same simulation.

We do not consider the reduced gravity for the definitions for the dimensionless parameters and set the Reynolds number as $Re = 1.0 \times 10^6$ and the Grashof is set to $Gr = 1.0 \times 10^{10}$. For this simulation, we disregard the diffusion term of the transport equation (3). So, the Peclet number does not need to be defined. We set the exponent of the nonlinear diffusion operator (20) as $\beta = 1$ and the time step is set to $\Delta t = 0.025$.

We compare the results from the present adaptive simulation with those obtained using fixed structured mesh with characteristic length $l = 0.00078125$ (given by the hexahedron edge). The dimensionless distance of the front head

HPCLatAm 2012, pp. 14-16 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

between the two fluids $X = |x - 0.4|$ is tracked over time t. The distance from the initial position ($x = 0.4$) of both fronts using the fixed mesh is plotted in Fig. (7)

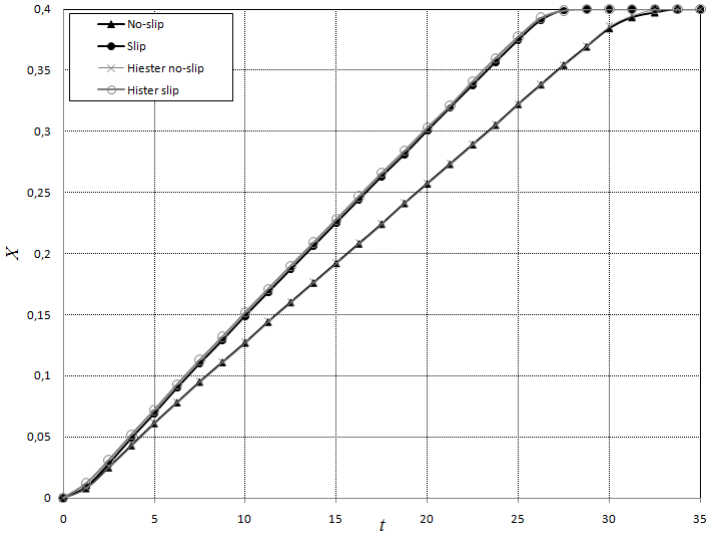


Fig. 7. Plot of the dimensionless distance of top and bottom fronts.

As expected, free-slip front (top) reaches the vertical wall before the no-slip (bottom) does. Figure (7) shows a good agreement between our results and those obtained by the reference [10]. The results from [10] were obtained using a fixed 2D mesh formed by triangles which characteristic length is $l = 0.00025$.

The Fig (8) shows the temperature distributions and the meshes at two different times with AMR/C at every 10 time-steps. The refinement fraction is set as $r_f = 0.95$ and the coarsening fraction is $r_c = 0.01$. In order to prevent the size of the elements become too small, we allowed only 4 refinement-levels. The Kelvin-Helmholtz billows are captured by the mesh as the front evolves after the release.

Through the mesh adaptivity simulation, the largest number of elements reached is approximately 30,000. If a fixed structured mesh had been used, to achieve the same refinement level, it would take approximately 131,000 hexaedrons. Therefore, with mesh adaptation we can, in this problem, compute a solution with one order of magnitude less elements without compromising the solution accuracy.

HPCLatAm 2012, pp. 15-16 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

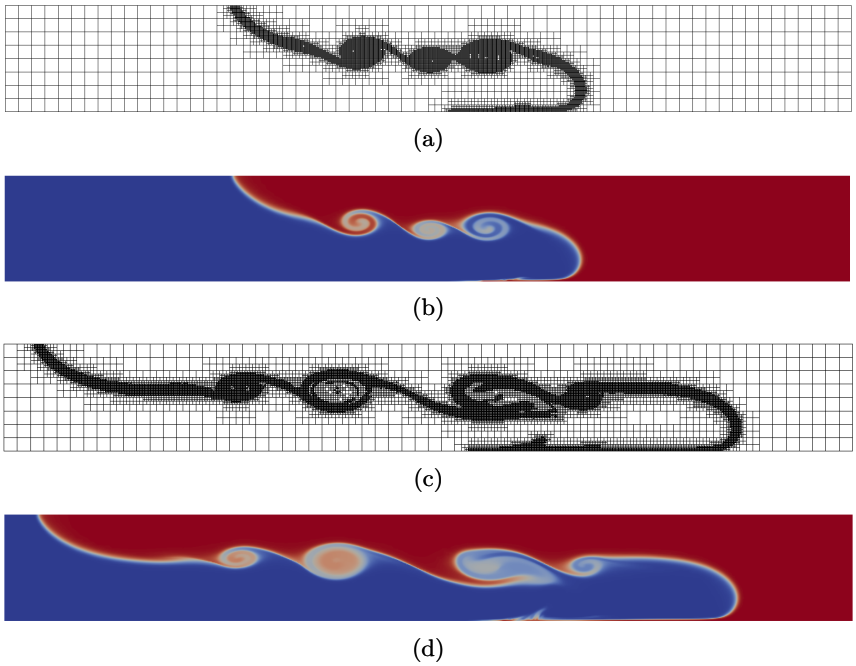


Fig. 8. Adaptive meshes and temperature distribution: (a) Adaptive mesh at $t = 12.5$, (b) Temperature distribution at $t = 12.5$, (c) Adaptive mesh at $t = 25.0$ and (d) Temperature distribution at $t = 25.0$.

5 Conclusions

The `libMesh` framework was used to implement the stabilized SUPG/PSPG finite element formulation for the parallel adaptive solution of incompressible viscous flow and advective-diffusive transport using a three-linear hexahedral element. Good numerical results were obtained for parallel executions with adaptive meshes. AMR/C allows the representation of multiple flow scales and improves resolution where needed. It was possible to track the Kelvin-Helmholtz billows present in the temperature-driven gravity current problem.

References

1. Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., White, A.: Sourcebook of Parallel Computing. Morgan Kaufmann Publishers (2003)
2. Kirk, B.S., Peterson J.W., Stone R., Carey G.F.: `Libmesh`: a c++ library for parallel adaptive mesh refinement/coarsing simulations. Journal Engineering with Computers, 22, 237-254 (2006)
3. Griebel, M., Dornseifer, T., Neuhoeffer, T.: Numerical Simulation in Fluid Dynamics: A Practical Introduction. SIAM (1997)

HPCLatAm 2012, pp. 16-16 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

4. Tezduyar, T.: Stabilized finite element formulations for incompressible flows computation. Advances in Applied Mechanics, 28, 1-44 (1992)
5. Härtel, C., Meiburg, E., Necker, F.: Analysis and direct numerical simulation of the flow at a gravity-current head. Part 1. Flow topology and front speed for slip and no-slip boundaries. Journal of Fluid Mechanics, 418, 189-212 (2000)
6. Bazilevs, Y., Calo, V.M., Tezduyar, T., Hughes, T.J.R.: YZβ Discontinuity Capturing for Advection-Dominated Processes with Application to Arterial Drug Delivery. International Journal for Numerical Methods in Fluids, 54, 593-608 (2007)
7. Bank, R., Welfert B.: A posteriori error estimates for the stokes problem. Journal of Numerical Analysis, 28, 591-623 (1991)
8. Ainsworth, M., Oden, J.: A Posteriori Error Estimation in Finite Element Analysis. Wiley Interscience (2000)
9. Carey, G.: Computational grids: generation, adaptation, and solution strategies. Taylor & Francis (1997)
10. Hiester, H., Piggot, M., Allison, P.: The impact of mesh adaptivity on the gravity current front speed in a two-dimensional lock-exchange. Ocean Modeling, 38, 1-21 (2011)
11. Kelly, D., Gago, J., Zienkiewicz, O., Babuska, I.: A posteriori error analysis and adaptive processes in the finite element method: Part I - Error analysis. International Journal for Numerical Methods in Engineering, 19, 1593-1619 (1983)
12. Liu, W., Sherman, A.: Comparative Analysis of the Cuthill-McKee and the Reverse Cuthill-McKee Ordering Algorithms for Sparse Matrices. Journal on Numerical Analysis, 13, 198-213 (1976)
13. Camata, J.J., Rossa, A.L., Valli, A.M.P., Catabriga, L., Carey, G.F., Coutinho, A.L.G.A.: Reordering and incomplete preconditioning in serial and parallel adaptive mesh refinement and coarsening flow solutions. International Journal for Numerical Methods in Fluids, 69, 802-823 (2012)
14. Benzi, M.: Preconditioning Techniques for Large Linear Systems: A Survey. Journal of Computational Physics, 182, 418-477 (2002)

HPCLatAm 2012, pp. 17-33 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

A Numerical Algorithm for the Solution of Viscous Incompressible Flow on GPU's

Santiago Costarelli[1], Mario Storti[1], Rodrigo Paz[1], Lisandro Dalcín[1], and Sergio Idelsohn[1,2,3]

[1] CIMEC-INTEC-CONICET-UNL, Guemes 3450, 3000 Santa Fe, Argentina
santi.costarelli@gmail.com, http://www.cimec.org.ar
[2] International Center for Numerical Methods in Engineering (CIMNE), Technical University of Catalonia (UPC), Gran Capitán s/n, 08034 Barcelona, Spain,
[3] Institució Catalana de Recerca i Estudis Avançats (ICREA), Barcelona, Spain

Abstract. Graphic Processing Units have received much attention in last years. Compute-intensive algorithms operating on multidimensional arrays that have nearest neighbor dependency and/or exploit data locality can achieve massive speedups. This work discuss a solver for the pressure problem in applications using immersed boundary techniques in order to account for moving solid bodies. The solver is based on standard Conjugate Gradients iterations and depends on the availability of a fast Poisson solver on the whole domain to define a preconditioner.

Keywords: Graphics Processing Units; Incompressible Navier-Stokes; Poisson equation

1 Introduction

Graphics Processing Units (GPU) are computer co-processors used in desktop computers and workstations to off-load the renderization of complex graphics from the main processor (CPU). They have evolved to complex systems containing many processing units, a large amount of on-board memory and a computing power in the order of teraflops. They are instances of massively parallel architectures and Single Instruction Multiple Data (SIMD) paradigms.

Recently, GPU's are becoming increasingly popular among scientists and engineers for High Performance Computing (HPC) applications [1–3, 8, 9, 11–15, 19, 20]. This tendency motivated GPU manufacturers to develop General Purpose Graphics Processing Units (GPGPU) targeting the HPC market.

In the pursuit of more realistic visualization algorithms for video games and special effects, solving Partial Differential Equations (PDE) has become a necessary ingredient [6, 7, 10, 18, 22]. Numerical schemes employed in these applications usually sacrifice accuracy for speed, resulting in very fast implementations when comparing to engineering codes.

The resolution of Computational Fluid Dynamics (CFD) problems on GPU's requires of specialized algorithms due to the particular hardware architecture of these devices. Algorithms that fall in the category of Cellular Automata (CA) are the best fitted

HPCLatAm 2012, pp. 17-33 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

for GPU's. For instance, explicit Finite Volume or Finite Element methods, jointly with *immersed boundary* techniques [21] to represent solid bodies, can be used on structured cartesian meshes. In the case of incompressible flows, it is not possible to develop a purely explicit algorithm, due to the essentially non-local nature of the incompressibility condition.

Segregated algorithms solve an implicit Poisson equation for the pressure field, being this stage the most time-consuming in the solution procedure. Using fast Poisson solvers like Multigrid (MG) or Fast Fourier Transform (FFT) is tempting but treating moving solid bodies becomes cumbersome in the case of MG or unsuitable for FFT. To surpass these difficulties, Molemaker *et.al* proposed in [13] the Iterated Orthogonal Projection (IOP) method which requires a series of projections on the complete grid (fluid and solid) to enforce the incompressibility and boundary conditions.

In this work an alternative to IOP, the Accelerated Global Preconditioning (AGP), is proposed. The solver is based on using a Preconditioned Conjugate Gradients (PCG) algorithm, so that, it is an accelerated iterative method in contrast to the stationary scheme used in IOP. In addition, AGP method iterates only on pressure, whereas IOP iterates on both pressure and velocity.

2 The Accelerated Global Preconditioning

A preconditioning for embedded problems that is based on solving the problem in the complete mesh is presented. Suppose a situation like in figure 1, with a solid body described by the boundary Γ_{bdy}. This is embedded in a structured FEM grid of constant mesh size h. The Poisson problem *outside* the body has to be solved, so that this is done by assembling the matrices of those finite elements that are in the fluid region. In order to do that, the center of the elements are checked wether they fall inside or outside the body. In this way the body is approximated by a *staircase geometry* as is shown in gray in the figure. In a FEM context the imposition of the homogeneous Neumann condition is done by simply assembling only those elements that are in the fluid part (filled in gray in the figure). The other elements that are not in gray are *ghost elements* and are not assembled for the solution of the Poisson problem. Only the pressure in the nodes connected to some element that is assembled are relevant, i.e. those that are marked in blue and red. Those that are marked in green are *ghost* and then they are not computed. Those that are computed are classified as *interior* and *boundary*. interior to the fluid, (subindex F) are those that are surrounded by computed elements, or conversely that are not connected to ghost elements. The rest are marked as *boundary* (subindex B, filled in red in the figure). So the Poisson problem is

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{1}$$

and the splitting of nodes induces a matricial splitting like this

$$\begin{bmatrix} \mathbf{A}_{FF} & \mathbf{A}_{FB} \\ \mathbf{A}_{BF} & \mathbf{A}_{BB} \end{bmatrix} \begin{bmatrix} \mathbf{x}_F \\ \mathbf{x}_B \end{bmatrix} = \begin{bmatrix} \mathbf{b}_F \\ \mathbf{b}_B \end{bmatrix} \tag{2}$$

For the definition of the preconditioning P, the whole matrix $\tilde{P}$ for all elements (fluid and ghost) is assembled, and the right hand side vector is extended as 0 on the

HPCLatAm 2012, pp. 17-33 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

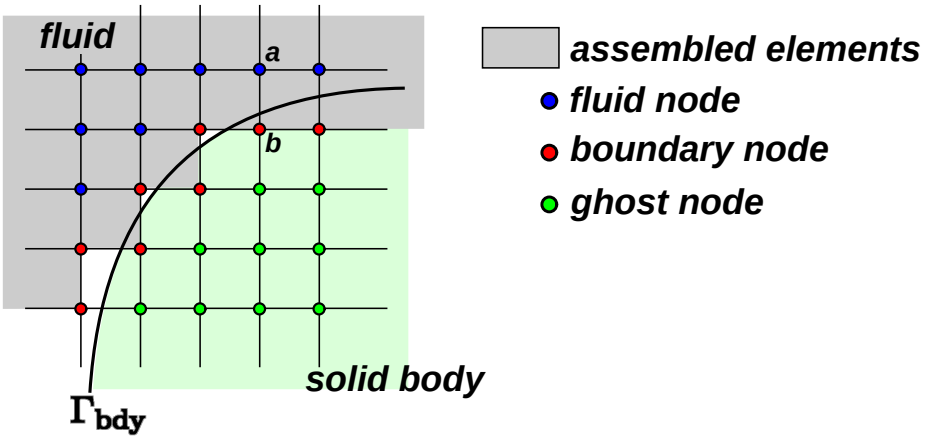


Fig. 1. Description of nodes and elements used in the AGP preconditioning.

ghost elements. The system is solved and then the ghost values are discardeed, i.e. the preconditioning is defined formally as $\mathbf{y}_{FB} = \mathbf{P}\mathbf{x}_{FB}$, where $\mathbf{y}_{FB}$ is the solution of

$$\begin{bmatrix} \tilde{\mathbf{P}}_{FF} & \tilde{\mathbf{P}}_{FB} & \tilde{\mathbf{P}}_{FG} \\ \tilde{\mathbf{P}}_{BF} & \tilde{\mathbf{P}}_{BB} & \tilde{\mathbf{P}}_{BG} \\ \tilde{\mathbf{P}}_{GF} & \tilde{\mathbf{P}}_{GB} & \tilde{\mathbf{P}}_{GG} \end{bmatrix} \begin{bmatrix} \mathbf{x}_{FB} \\ \mathbf{x}_G \end{bmatrix} = \begin{bmatrix} \mathbf{y}_{FB} \\ \mathbf{0}_G \end{bmatrix}. \tag{3}$$

Note that a differente symbol $\tilde{\mathbf{P}}$ is used for the discrete Laplace operator in this equation, since it is assembled on differente elements. However it can be seen that

- $\tilde{\mathbf{P}}_{FF} = \mathbf{A}_{FF}$ since the F nodes are those for which all elements are assembled in the Poisson problem.
- $\tilde{\mathbf{P}}_{FB} = \mathbf{A}_{FB}$, and $\tilde{\mathbf{P}}_{BF} = \mathbf{A}_{BF}$ since for instance, such a coefficient would link nodes as a and b in the figure. This coefficient comes from the assembly of all the elements that are connected to a *and* b, but since a is an F node, it means that all elements connected to a are assembled.
- $\tilde{\mathbf{P}}_{FG} = \tilde{\mathbf{P}}_{GF} = 0$ since F nodes are only connected to fluid elements and G are only connected to ghost elements, so that they can not share an element.

So

$$\begin{bmatrix} \mathbf{A}_{FF} & \mathbf{A}_{FB} & \mathbf{0} \\ \mathbf{A}_{BF} & \tilde{\mathbf{P}}_{BB} & \tilde{\mathbf{P}}_{BG} \\ \mathbf{0} & \tilde{\mathbf{P}}_{GB} & \tilde{\mathbf{P}}_{GG} \end{bmatrix} \begin{bmatrix} \mathbf{x}_{FB} \\ \mathbf{x}_G \end{bmatrix} = \begin{bmatrix} \mathbf{y}_{FB} \\ \mathbf{0}_G \end{bmatrix}. \tag{4}$$

$\mathbf{x}_G$ can be eliminated from the bottom line, and then

$$\begin{bmatrix} \mathbf{A}_{FF} & \mathbf{A}_{FB} \\ \mathbf{A}_{BF} & \tilde{\mathbf{P}}_{BB} - \tilde{\mathbf{P}}_{BG}\tilde{\mathbf{P}}_{GG}^{-1}\tilde{\mathbf{P}}_{GB} \end{bmatrix} \mathbf{x}_{FB} = \mathbf{y}_{FB}, \tag{5}$$

so that an explicit expression for the preconditioning matrix is obtained

$$\mathbf{P} = \begin{bmatrix} \mathbf{A}_{FF} & \mathbf{A}_{FB} \\ \mathbf{A}_{BF} & \tilde{\mathbf{P}}_{BB} - \tilde{\mathbf{P}}_{BG}\tilde{\mathbf{P}}_{GG}^{-1}\tilde{\mathbf{P}}_{GB} \end{bmatrix}. \tag{6}$$

HPCLatAm 2012, pp. 17-33 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

A first consequence of this expression is that a lot of eigenvalues of the preconditioned matrix will be 1. Consider the space of all vectors $\mathbf{x}$ such that the B component is null, then

$$\begin{aligned} \mathbf{A}\mathbf{x} &= \mathbf{P}\mathbf{x}, \\ \mathbf{P}^{-1}\mathbf{A}\mathbf{x} &= \mathbf{x}, \end{aligned} \tag{7}$$

so that $\mathbf{x}$ is an eigenvector with eigenvalue 1.

2.1 Numerical experiment computing condition numbers

The condition number of matrices for the Poisson problem have been computed with and without preconditioning.

- N_x ranges from 8 to 64.
- The Poisson problem is computed selecting the quadrangles whose center fall outside the body.
- In all cases the domain is the unit square with periodic boundary conditions.
- The bodies considerer are: cylinder of radius 0.2, a vertical strip of width 0.5, and a square of side 0.5.
- Ths condition numbers are computed with Octave.

Note that in all cases the non preconditioned matrix condition number grows as $O(N_x^2)$, whereas with the preconditioning it remains constant.

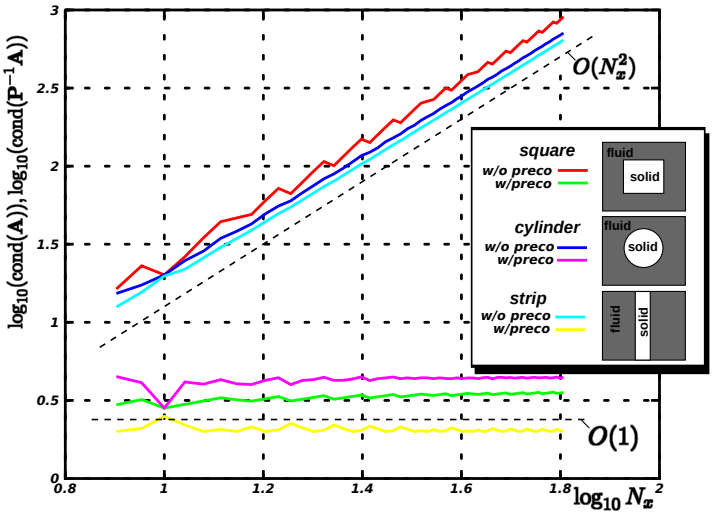


Fig. 2. Condition number of Poisson problem with and without FFT preconditioning

HPCLatAm 2012, pp. 17-33 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

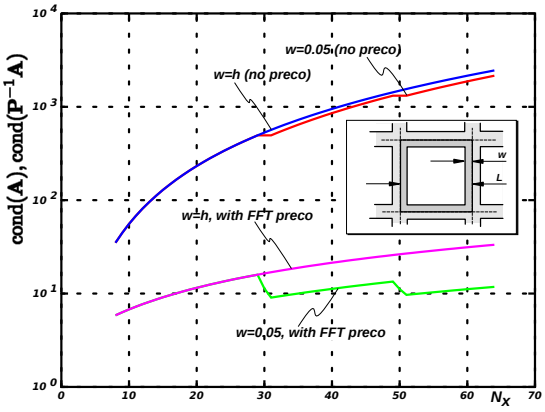


Fig. 3. Condition number for Poisson problem on a square, with and without FFT preconditioning.

2.2 The thin wall case

Consider now the case where the fluid occupies the *interior* of a square $\max(|x - L/2|, |y - L/2|) < L/2 - w$ where w is the width of the wall (see figure 3). In the figure the condition number for the Poisson problem with and without preconditioning are shown. The case where the width of the wall varies so as to have always one element (in fact two elements due to the periodic b.c.'s) separating the squares is considered. On the other hand if a fixed value $w = 0.05$ then the condition number of the preconditioned case is kept bounded.

2.3 Computation of the condition number in terms of the eigenvalues of the Steklov operators

The eigenvalues of the Steklov operators can be computed in closed form for the case of a cylinder in an infinite flow. Recall that the convergence of the AGP scheme is controlled by the condition number of the preconditioned operator

$$\mathrm{cond}(P^{-1}A) = \frac{\max|\,\mathrm{eig}(P^{-1}A)|}{\min|\,\mathrm{eig}(P^{-1}A)|} \tag{8}$$

As all are positive and definite operators, the eigenvalues λ are real and positive, and $0 \leq \lambda \leq 1$. Also there are a lot of eigenvalues that are unity $\lambda = 1$, they correspond to the space of functions that satisfy the boundary condition $(\partial\phi/\partial n) = 0$ at Γ. However, the Krylov methods iterate only on the space perpendicular to it, and it can be shown that the condition number of the preconditioned Steklov operator must be computed

$$\tilde{\mathcal{S}} = (\mathcal{S}_F + \mathcal{S}_S)^{-1}\mathcal{S}_F. \tag{9}$$

$$\kappa(\tilde{\mathcal{S}}) = \frac{\max[\mathrm{eig}(\tilde{\mathcal{S}})]}{\min[\mathrm{eig}(\tilde{\mathcal{S}})]} \tag{10}$$

For simple geometries like a semi-infinite plane, a strip, and a cylinder the eigenvalues of $\mathcal{S}_F$, $\mathcal{S}_S$ can be explicitely computed. In addition, it turns out that the eigenfunctions are the same, so that the spectral decomposition of the sum $\mathcal{S}_F + \mathcal{S}_S$ and the preconditioned operator are available. Recall that the Steklov $\mathcal{S}_F : V_\Gamma \to V_\Gamma$, operator is defined as $w = \mathcal{S}_F(v)$

$$\begin{aligned} &\Delta\phi = 0, \text{ in } \Omega_F \\ &\phi_\Gamma = v, \text{ and } w = (\partial\phi/\partial n)|_\Gamma \end{aligned} \tag{11}$$

where $V_\Gamma = \{\text{real valued functions on } \Gamma\}$, $\hat{\mathbf{n}}$ is the normal to Γ exterior to Ω_F. The same definition, *mutatis mutandis*, applies to $\mathcal{S}_S$.

The semiplane. The geometry consists on a semiplane

$$\Omega_F = \{\mathbf{x}/x < 0\}, \quad \Omega_S = \{\mathbf{x}/x > 0\}, \quad \Gamma = \{\mathbf{x}/x = 0\} \tag{12}$$

By symmetry of translation in the y direction the eigenfunctions must be plane waves of the form $v = \mathrm{e}^{iky}$ with k real. The solution to the Poisson problem on the fluid and solid are

$$\begin{aligned} &\phi = \mathrm{e}^{iky}\mathrm{e}^{-|k|x}, \ \mathbf{x} \in \Omega_S, \\ &\phi = \mathrm{e}^{iky}\mathrm{e}^{|k|x}, \ \mathbf{x} \in \Omega_F, \end{aligned} \tag{13}$$

and then

$$\mathrm{eig}\{\mathcal{S}_F(v)\} = \mathrm{eig}\{\mathcal{S}_S(v)\} = |k|, \tag{14}$$

so that all the eigenvalues of $\tilde{\mathcal{S}}$ are 0.5, and $\kappa(\tilde{\mathcal{S}}) = 1$.

The cylinder. The domain is $\Omega_S = \{||\mathbf{x}|| = R\}$. By rotational symmetry the eigenfunctions must be

$$\begin{aligned} v_{n,+} &= \cos(n\theta) \\ v_{n,-} &= \sin(n\theta) \end{aligned} \tag{15}$$

where r, θ are polar coordinates at the center of the cylinder. The solution at both domains are

$$\phi_{n,\pm,F} = r^{-n}\begin{Bmatrix}\cos(n\theta)\\ \sin(n\theta)\end{Bmatrix}, \quad \phi_{n,\pm,S} = r^{n}\begin{Bmatrix}\cos(n\theta)\\ \sin(n\theta)\end{Bmatrix}, \tag{16}$$

so that the eigenvalues and eigenfunctions of both operators are the same

$$\lambda(n, \pm, F/S) = \frac{n}{R} \tag{17}$$

and again $\lambda_{n,\tilde{\mathcal{S}}} = 1/2$, $\kappa(\tilde{\mathcal{S}}) = 1$.

The infinite strip The domain is $\Omega_S = \{|x| \leq w/2\}$ where w is the width of the strip. The space V_Γ are pairs of functions on both sides of the strips. By translation invariance in the y direction

$$v = \begin{cases} a\,\mathrm{e}^{iky}, & \text{for } x = -w/2 \\ b\,\mathrm{e}^{iky}, & \text{for } x = +w/2 \end{cases} \tag{18}$$

it can be shown by symmetry $x \to -x$ that the eigenfunctions are the symmetric ($a = b$) and antisymmetric ($a = -b$) combinations, so that

$$v_{k,\pm} = \begin{cases} \pm\mathrm{e}^{iky}, & \text{for } x = -w/2, \\ \mathrm{e}^{iky}, & \text{for } x = +w/2. \end{cases} \tag{19}$$

The corresponding solution for the symmetric modes at $\Omega_{F,S}$ are

$$\phi_{k,+} = \begin{cases} \mathrm{e}^{iky+|k|(w/2-x)}, & \text{for } |x| \geq w/2, \\ \dfrac{\cosh(kx)}{\cosh(kw/2)}\mathrm{e}^{iky}, & \text{for } |x| \leq w/2, \end{cases} \tag{20}$$

and the corresponding eigenvalues

$$\begin{aligned} \lambda(k,+,F) &= |k|, \\ \lambda(k,+,S) &= k\tanh(kw/2). \end{aligned} \tag{21}$$

And for the antisymmetric eigenfunctions,

$$\phi_{k,-} = \begin{cases} \mathrm{sign}(x)\mathrm{e}^{iky+|k|(w/2-x)}, & \text{for } |x| \geq w/2, \\ \dfrac{\sinh(kx)}{\sinh(kw/2)}\mathrm{e}^{iky}, & \text{for } |x| \leq w/2, \end{cases} \tag{22}$$

and the corresponding eigenvalues

$$\begin{aligned} \lambda(k,-,F) &= |k|, \\ \lambda(k,-,S) &= k\coth(kw/2). \end{aligned} \tag{23}$$

Both the symmetric and antisymmetric eigenvalues van be seen in figure 4. Note that the eigenvalues of the fluid operator $\lambda(k,\pm)$ are the same for the symmetric and antisymmetric case and independent of the strip width w, since the fluid domain is completely decoupled by the strip, and then the eigenvalues of the Steklov operator $\mathcal{S}_F$ are the same as those of those of each semiplane $x > w/2$ and $x < w/2$.

On the other hand, with respect to the eigenvalues of the Steklov operator of the strip (solid domain) $\mathcal{S}_S$, note that both symmetric and antisymmetric eigenvalues behave like $\sim |k|$ for $kw \to \infty$. This is because for kw large means that the wavelength of the eigenfunction is much smaller than the width of the strip and then again, the behavior of the eigenvalues is the same as for a infinite semiplane.

However, when kw is small the behavior of the symmetric and antisymmetric branches is very different. Remember that, as per definition of the Steklov operator, given an eigenfunction u it must be imposed as a Dirichlet boundary condition on the interface

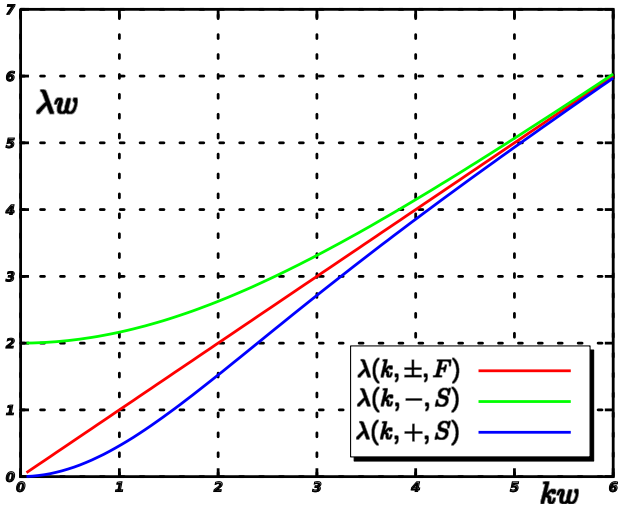


Fig. 4. Eigenvalues of Steklov operators for the solid strip case.

Γ, solve the Laplace equation for the field ϕ in the corresponding domain and compute the flux $v = (\partial\phi/\partial n)$. As u is an eigenfunction, v should be proportional to u and the proportionality constant is the eigenvalue λ. First consider the symmetric branch. If a sinusoidal value is imposed on the left boundary $x = -w/2$ (see figure 6) then for the symmetric mode the same Dirichlet boundary condition must be imposed on the other boundary $x = +w/2$. As a result, facing points inside the strip like A, A' or B, B' have equal values of temperature ϕ imposed and then the heat flow is very low, which means a small eigenvalue. On the other hand, for the antisymmetric mode, opposing points have the same absolute temperature but of opposed sign, and the heat flow is very high (the red arrows in the figure). This explains why for low wavenumber k the eigenvalues of the symmetric mode are smaller, with a behavior $\lambda \propto k^2$ for $k \to 0$. For the antisymmetric mode for low wavenumber the eigenvalue is larger with a behavior $\lambda w \to 2$ for $kw \to 0$.

This last limit is simple to understand. Effectively, for very small wavenumber conduction in the y direction can be neglected, and so for an eigenfunction $u = \pm\cos ky$ at $x = \pm w/2$ the solution is

$$\phi = \frac{2x}{w}\cos ky, \tag{24}$$

so that

$$v = (\partial\phi/\partial n)|_{x=w/2} = \frac{2}{w}\cos ky = \frac{2}{w}u. \tag{25}$$

So that, $\lambda w = 2$.

HPCLatAm 2012, pp. 17-33 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

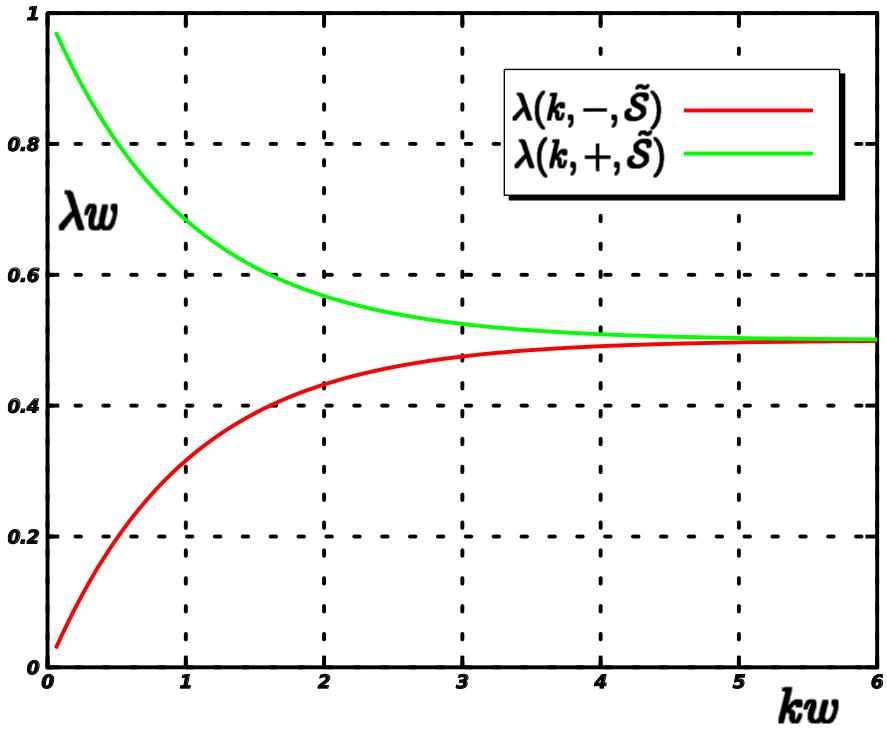


Fig. 5. Eigenvalues of preconditioned Steklov operator for the symmetric and antisymmetric branches.

The eigenvalues of the preconditioned Steklov operators are plot in figure 5. They are given by the expressions

$$\lambda(k,+,\tilde{S}) = \frac{|k|}{|k| + k\tanh(kw/2)},$$
$$\lambda(k,-,\tilde{S}) = \frac{|k|}{|k| + k\coth(kw/2)}. \tag{26}$$

Note that for both symmetric and antisymmetric mode the eigenvalues g to $1/2$ for $kw \to \infty$, as for an infinite semiplane. On the other hand, for small kw the eigenvalues of the symmetric mode are larger than $1/2$, and those for the antisymmetric modes are smaller. So, if the eigenvalues for the symmetric modes are considered the condition of the preconditioned Steklov operator is 2, whereas if we consider the antisymmetric modes the condition number tends to ∞ since the smallest eigenvalue tends to 0 for $kw \to 0$.

This is expected, since the FFT preconditioning is based on solving the Poisson equation on the whole domain, fluid and solid, instead in solving only on the fluid. Thus, this preconditioning is good whenever the fluxes in the solid domain are small. But this is exactly the case for the symmetric modes, and the opposite happens for the antisymmetric modes.

HPCLatAm 2012, pp. 17-33 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

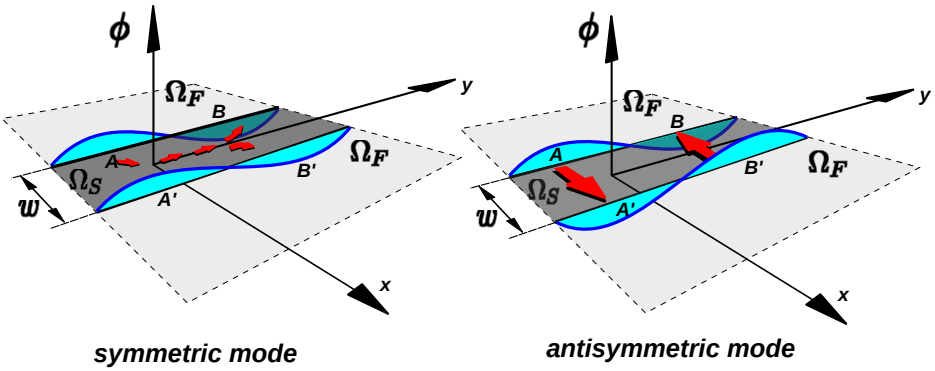


Fig. 6. Explanation of the behavior of the Steklov eigenvalues for large wavelengths.

Estimation of the condition number for thin walls The analysis of the infinite strip shows that a situation where the FFT preconditioning is deteriorated is when there are thin walls in the solid geometry, since in that case the modes that are antisymmetric about the axis of the solid produce large heat fluxes in the solid, and this is an indication of bad performance of the preconditioning. So for elongated solid geometries with, say, a typical length of L and a typical width of $w \ll L$ an estimate of the condition number of the preconditioned operator can be obtained by taking L as the maximal wavelength and the estimate gives a condition number of

$$\kappa(\tilde{\mathcal{S}}) \sim \frac{|k_{\min}| + k_{\min} \coth(k_{\min} w/2)}{|k_{\min}|} \tag{27}$$

where $k_{\min} = 2\pi/L$. By algebraic manipulation this can be simplified to

$$\kappa(\tilde{\mathcal{S}}) \sim 1 + \coth(\frac{\pi w}{L}) \sim \frac{L}{\pi w}, \tag{28}$$

i.e. the condition number is proportional to the aspect ratio of the solid domain.

3 CUDA implementation

A basic description of the CUDA implementation will be given here. Complete details can be found elsewhere [4, 5].

Code (1) shows the pseudocode for the complete set of steps required under a fractional-step method to solve checkerboard problems on pressure-velocity decoupling under discrete schemas as finite differences or volume methods.

HPCLatAm 2012, pp. 17-33 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

Algorithm 1 - Pseudocode used to explain briefly the steps requiered by our problem.

```
for i=1:TimeSteps {
  1- Update solid position.
  2- Impose solid velocities on current velocity field.
  3- Solve momentum equations.
  4- Time integration (Adams-Bashfort)
  5- Impose solid velocities on current velocity field.
  6- Compute velocity divergence.
  7- CG+FFT solver.
    7.1- Fluid+solid FFT solver.
    7.2- Solving Poisson equation for pressure (↩
        contributions accounted only by solid-free nodes).
  8- Compute pressure gradients (contributions accounted ↩
     only by solid-free nodes).
  9- Update solid-free nodes using pressure gradients.
}
```

Some comments about the algorithm follows:

- Step 1 applies only in the case of moving bodies. If the bodies are at rest this step is irrelevant.
- Step 2 consists in imposing the velocities of the solid to those velocity cells that fall inside the given body.
- In step 7.2 only the contributions of cells not connected to the solid are assembled.
- In steps 8-9 pressure gradients are obtained and used to update velocity nodes whose pressure neighbours are not in the solid.

The results obtained by the algorithm are shown on Table (1). The GPGPU used was a NVIDIA Tesla C2050 under a CPU intel i7 950. CG iterations were limited on 3, where an absolute error tolerance of $10^{-2} \sim 10^{-3}$ was reached.

	Simple [segs/MCels]	Double [segs/MCels]
32x32x32	0.17	0.20
64x64x64	0.043	0.062
128x128x128	0.026	0.044

Table 1. Performance obtained per time step using NVIDIA Tesla C2050. CG iterations: 3.

The most consuming steps are those on solving momentum equations and the Poisson step.

HPCLatAm 2012, pp. 17-33 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

In this work the NVIDIA cuFFT library [16, 17] has been used to perform FFT's, Thrust and CUSP API's to manage vector and linear algebra operations, and VTK for visualization the results obtained.

4 Numerical experiments

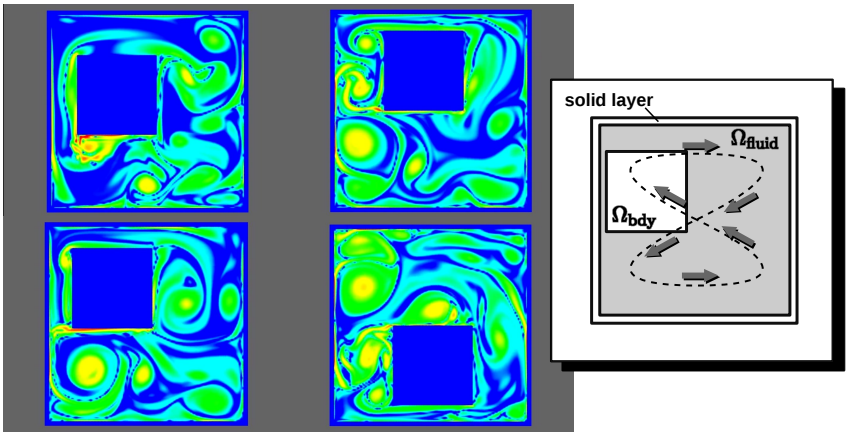


Fig. 7. Colormap of $\log_{10}(|\omega|)$ for a square of side $L_s = 0.4$[m] moving in a square domain of side $L = 1$[m]. The square moves forming a Lissajous *8*-shaped curve.

Numerical simulations of several flows involving moving bodies are shown in figures 7-11. In all cases (except for the case of the example in section §4) the flows represent a body moving inside a square or cubic cavity of length side 1[m]. In order to circumvent the restriction of periodic boundaries intrinsic to the FFT solver, a thin layer (2.5% of the square or cubic domain side length) is defined as a fixed body. In all cases the color corresponds to $\log_{10}(|\omega|)$, i.e. the absolute magnitude of the vorticity vector $\omega = \nabla \times \mathbf{u}$ in logarithmic scale. This quantity helps in the visualization of boundary layers, since the magnitude of vorticity has variations of several orders of magnitude in flows with boundary layers at high Reynolds numbers. In 2D cases the mesh was 128×128 and in 3D cases $128 \times 128 \times 128$. In all cases the side of the domain (square in 2D, cube in 3D) was $L = 1$[m] and the kinematic viscosity was $\nu = 6.33 \times 10^{-5}[\mathrm{m}^2/\mathrm{s}]$.

Square moving in curved trajectory. The body is a square of side $L_s = 0.4$[m], and the center of the body (x_c, y_c) describes an *8*-shaped Lissajous curve, described by

$$x_c = \frac{L}{2} + A\cos(2\omega t), y_c = \frac{L}{2} + A\cos(\frac{\pi}{2} + \omega t),$$
$$\omega = 1[\mathrm{s}^{-1}],\ A = 0.2[\mathrm{m}] \tag{29}$$

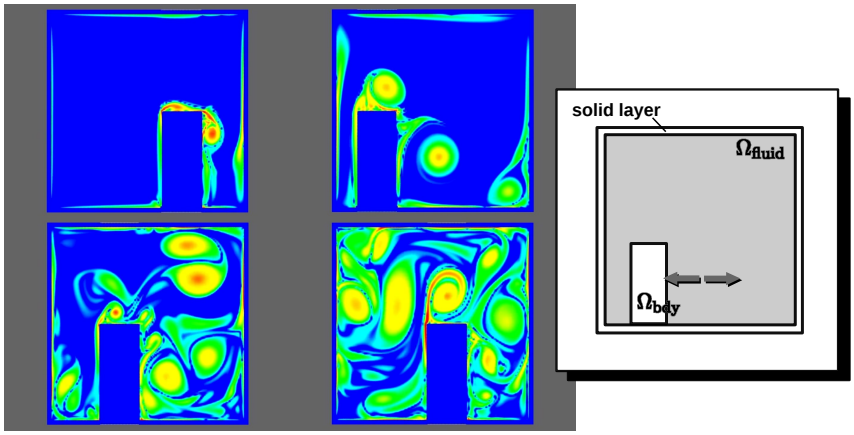


Fig. 8. Colormap of $\log_{10}(|\omega|)$ for a rectangle sliding on the bottom of the domain.

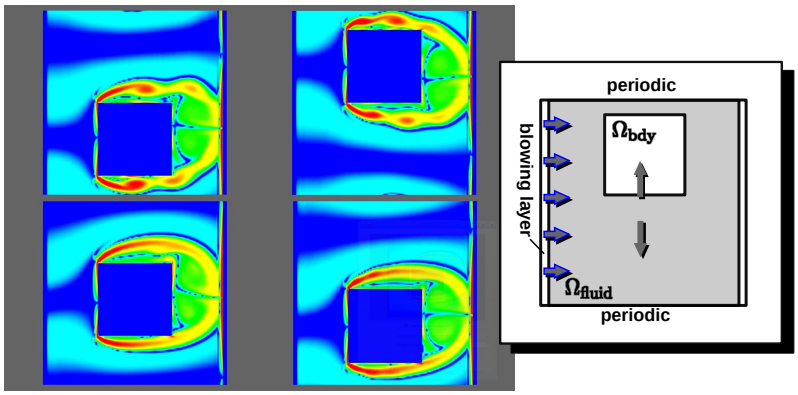


Fig. 9. Colormap of $\log_{10}(|\omega|)$ for a square body performing harmonic motion in the vertical direction with a cross flow in the horizontal direction.

As the body displaces fluid high levels of vorticity can be observed at the vertices. As the simulation progresses large vortices remain rotating in the fluid with long filamentary vorticity layers that are a characteristic 2D feature (they are unstable in 3D).

Moving rectangular obstacle. The body is a rectangle of height $H = 0.5$[m] and width $W = 0.2$[m]. An harmonic horizontal displacement as follows

$$\begin{aligned} x_c &= (L/2) + A\cos(\omega t), \\ \omega &= 1[\mathrm{s}^{-1}],\ A = 0.3[\mathrm{m}], \end{aligned} \tag{30}$$

is imposed. As the body displaces fluid a large concentration of vorticity is observed in the upper corner of the body, with characteristic trailing filamentary vortex layers that detach from the corners.

HPCLatAm 2012, pp. 17-33 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

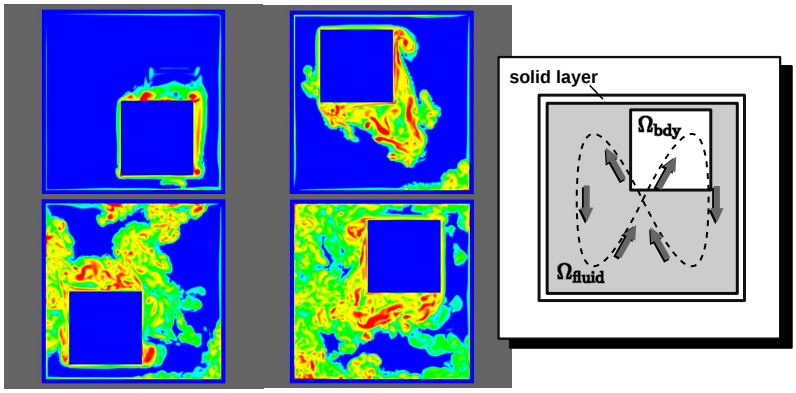


Fig. 10. Colormap of $\log_{10}(|\omega|)$ for a cube moving in a Lissajous 8-shaped curve.

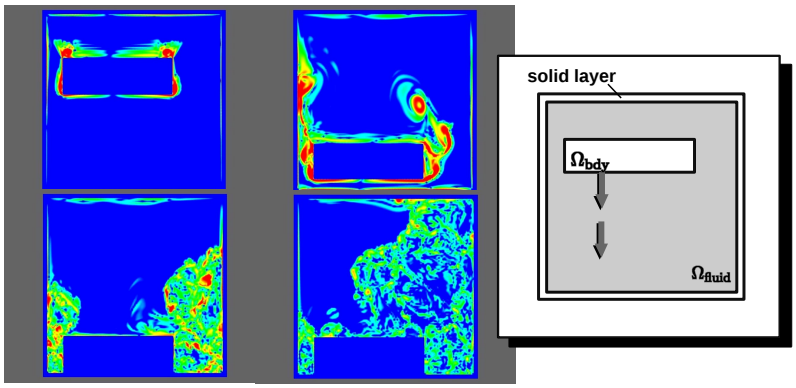


Fig. 11. Colormap of $\log_{10}(|\omega|)$ for a falling block.

Square moving vertically with mean horizontal flow. In this example the exterior boundary of the computational domain is not at rest, but rather it is intended to generate a mean flow that impinges on the body. This freestream flow is obtained with a layer of width 0.025[m] at the left and right sides were a positive x velocity of $u = 1[\mathrm{m/s}]$ is imposed. Periodic boundary conditions are imposed in the vertical y direction. The body is a square of side $L_s = 0.4[\mathrm{m}]$, the center of the body (x_c, y_c) is centered in the x direction and experiences an harmonic vertical movement

$$\begin{aligned} y_c &= (L/2) + A\cos(\omega t), \\ \omega &= 0.5[\mathrm{s}^{-1}],\ A = 0.2[\mathrm{m}]. \end{aligned} \tag{31}$$

An accelerating boundary layer is formed at the left side facing the fluid stream. The boundary layer accelerate towards the corners and detach there. If the vertical movement were at a constant velocity then the flow would be equivalent to a fixed body with an impinging stream at an angle of attack. A notable feature of the flow is that when

the body reaches the extreme positions in the y direction the vortex layers become unstable and start shedding vortices, whereas when the body is moving the vortex layer stabilizes.

Moving cube This is a 3D case. The center (x_c, y_c, z_c) of a cube of side $L_s = 0.4[\mathrm{m}]$ is describing a Lissajous 8-shaped figure in the $z = 0.66[\mathrm{m}]$ plane, as follows

$$\begin{aligned} x_c &= L/2 + A\cos(\omega t), \\ y_c &= L/2 + A\cos(\frac{\pi}{2} + 2\omega t), \\ z_c &= 0.66[\mathrm{m}], \\ \omega &= 2[\mathrm{s}^{-1}], A = 0.4[\mathrm{m}] \end{aligned} \tag{32}$$

This is similar to the case §4 but 3D. The large filamentary vortex layers are no more present, but instead there is a large amount of small eddies characteristic of a 3D flow.

Falling block The body is a parallelepiped block of dimensions $L_x = L_z = 0.6[\mathrm{m}]$, $L_y = 0.2[\mathrm{m}]$. The center of the body is initially at $(x_c, y_c, z_c) = (0.4125, 0.7, 0.5)[\mathrm{m}]$ and starts falling vertically with a velocity of $1[\mathrm{m/s}]$. As the body falls it displaces a large quantity of fluid that forms a turbulent region expanding from both sides of the block.

5 Conclusions

A new method called Accelerated Global Preconditioning for solving the incompressible Navier-Stokes equations with moving bodies was presented. The algorithm is based on a pressure segregated, staggered grid, Finite Volume formulation and uses an FFT solver for preconditioning the CG solution of the Poisson problem. Theoretical estimates of the condition number of the preconditioned Poisson problem are given, and several numerical examples are presented validating these estimates. The algorithm is specially suited for implementation on GPU hardware. The condition number of the preconditioned Poisson equation does not degrade with refinement. The algorithm allows computing 3D problems in real time on moderately large meshes for many problems of practical interest in the area of Computational Fluid Dynamics.

6 Acknowledgment

This work has received financial support from

- Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET, Argentina, PIP 5271/05),
- Universidad Nacional del Litoral (UNL, Argentina, grant CAI+D 2009-65/334),
- Agencia Nacional de Promoción Científica y Tecnológica (ANPCyT, Argentina, grants PICT-1506/2006, PICT-1141/2007, PICT-0270/2008), and

– European Research Council (ERC) Advanced Grant, Real Time Computational Mechanics Techniques for Multi-Fluid Problems (REALTIME, Reference: ERC-2009-AdG).

The authors made extensive use of *Free Software* as GNU/Linux OS, GCC/G++ compilers, Octave, and *Open Source* software as VTK among many others. In addition, many ideas from these packages have been inspiring to them.

Bibliography

[1] Adams, S., Payne, J., Boppana, R.: Finite difference time domain (FDTD) simulations using graphics processors. HPCMP Users Group Conference 0, 334–338 (2007)

[2] Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. pp. 1–11. ACM, New York, NY, USA (2009)

[3] Corrigan, A., Camelli, F.F., Löhner, R., Wallin, J.: Running unstructured grid-based CFD solvers on modern graphics hardware. International Journal for Numerical Methods in Fluids (2010), (in press)

[4] Costarelli, S.: Resolución de las ecuaciones de navier-stokes utilizando cuda. Tech. rep., Universidad Nacional del Litoral (2011), http://www.cimec.org.ar/ojs/index.php/cimec-repo/article/view/3735

[5] Costarelli, S., Paz, R., Dalcin, L., Storti, M.: Resolución de las ecuaciones de navier-stokes utilizando cuda. In: Muller, O., Signorelli, J., Storti, M. (eds.) Mecánica Computacional. vol. XXX, pp. 2979–3008. AMCA (2011), http://www.cimec.org.ar/ojs/index.php/mc/article/view/3965

[6] Crane, K., Llamas, I., Tariq, S.: Chapter 30 - Real-Time Simulation and Rendering of 3D Fluids (2008)

[7] Elcott, S., Tong, Y., Kanso, E., Schröder, P., Desbrun, M.: Stable, circulation-preserving, simplicial fluids. In: SIGGRAPH Asia '08: ACM SIGGRAPH ASIA 2008 courses. pp. 1–11. ACM, New York, NY, USA (2008)

[8] Elsen, E., LeGresley, P., Darve, E.: Large calculation of the flow over a hypersonic vehicle using a GPU. J. Comput. Phys. 227(24), 10148–10161 (2008)

[9] Goddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Wobker, H., Becker, C., Turek, S.: Using GPU's to improve multigrid solver performance on a cluster. Int. J. Comput. Sci. Eng. 4(1), 36–55 (2008)

[10] Irving, G., Guendelman, E., Losasso, F., Fedkiw, R.: Efficient simulation of large bodies of water by coupling two and three dimensional techniques. ACM Trans. Graph. 25(3), 805–811 (2006)

[11] Klöckner, A., Warburton, T., Bridge, J., Hesthaven, J.: Nodal discontinuous galerkin methods on graphics processors. Journal of Computational Physics 228(21), 7863–7882 (2009)

[12] Lastra, M., Mantas, J.M., Ure na, C., Castro, M.J., García-Rodríguez, J.A.: Simulation of shallow-water systems using graphics processing units. Math. Comput. Simul. 80(3), 598–618 (2009)

[13] Molemaker, J., Cohen, J.M., Patel, S., Noh, J.: Low viscosity flow simulations for animation. In: SCA '08: Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation. pp. 9–18. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2008)

[14] Mossaiby, F., Rossi, R., Dadvand, P., Idelsohn, S.: Opencl-based implementation of an unstructured edge-based finite element convection-diffusion solver on graphics hardware. International Journal for Numerical Methods in Engineering 89, 1635 1651 (2012)

[15] Mullen, P., Crane, K., Pavlov, D., Tong, Y., Desbrun, M.: Energy-preserving integrators for fluid animation. In: SIGGRAPH '09: ACM SIGGRAPH 2009 papers. pp. 1–8. ACM, New York, NY, USA (2009)

[16] Nvidia, C.: Compute unified device architecture (CUDA) (2010), http://developer.nvidia.com/category/zone/cuda-zone

[17] Nvidia, C.: CUFFT library (2010), http://developer.nvidia.com/cufft

[18] P.Rinaldi, Bauza, C.G., Vénere, M., Clausse, A.: Paralelización de autómatas celulares de aguas superficiales sobre placas gráficas. In: Cardona, A., Storti, M., Zuppa, C. (eds.) Mecánica Computacional Vol. XXVII. vol. XXVII, pp. 2943–2957 (2008)

[19] Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.m.W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. pp. 73–82. ACM, New York, NY, USA (2008)

[20] Thibault, J.C., Senocak, I.: CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows. In: AIAA (ed.) 47th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition (Disc 1) (2009)

[21] Wang, X., Wang, C., Zhang, L.: Semi-implicit formulation of the immersed finite element method. Computational Mechanics 49, 421–430 (2012)

[22] Wu, E., Liu, Y., Liu, X.: An improved study of real-time fluid simulation on GPU: Research articles. Comput. Animat. Virtual Worlds 15(3-4), 139–146 (2004)

HPCLatAm 2012, pp. 34-48 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

Parallel Computing Applied to Satellite Images Processing for Solar Resource Estimates

Rodrigo Alonso[1] and Sergio Nesmachnow[2]

[1] Instituto de Física, Facultad de Ingeniería, Universidad de la República
[2] Centro de Cálculo, Facultad de Ingeniería, Universidad de la República

Abstract. This article presents the application of parallel computing techniques to process satellite imagery information for solar resource estimates. A distributed memory parallel algorithm is introduced, which is capable to generate the required inputs from visible channel images to feed a statistical solar irradiation model. The parallelization strategy consists in distributing the images within the available processors, and so, every image is accessed only by one process. The experimental analysis demonstrate that a maximum speedup value of 2.32 is achieved when using four computing resources, but beyond that point the performance rather decrease due to hard-disk input/output velocity.

Keywords: parallel computing, satellite images, solar resource assessment

1 Introduction

Interest in renewable energies—such as solar and wind energy and its related applications—has strongly increased in the recent years. In Uruguay, a country with no conventional energy resources such as coal, oil, natural gas or potentially fissile materials, renewable energies are seen as a way to reduce the dependence of international oil and energy prices and availability. The Uruguayan government has set some ambitious national objectives concerning renewable energies, which are projected to contributed in more than a 50% of the country primary energy supply by the year 2015. A primordial step for the successful introduction of renewable energy is resource assessment. Even though the important part that is expected for renewable energies to contribute into the primary energy mix, the assessment of the available solar resource at the national territory has been initiated only recently.

The first solar map of Uruguay was built in 2009 based on a well-established correlation between irradiation and insolation ground measurements [1]. The spatial and temporal resolution obtained from methodologies based on correlating ground-data is very limited, and the method usually requires applying interpolation techniques. Such interpolation techniques provide limited accuracy, even over small distances. Perez et al. [9] showed that simple satellite-based irradiation models are able to achieve better accuracy for hourly irradiance than interpolation techniques over distances as short as 30 km. In fact, from an end-user perspective, it is preferable to rely on satellite hourly estimates than using ground data from stations located more than 30 km away of the target point.

HPCLatAm 2012, pp. 34-48 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

Historically, models to assess irradiance estimates using satellite information were classified into two categories: statistical [7] and physical [8]. Statistical models use regression techniques between satellite data and ground measurements. As a result, they require reliable ground measurements to tune some coefficients to a target region. On the other hand, physical models intend to describe the physical processes that occur at the atmosphere. Satellite-based solar resource estimation is also quite recent in Uruguay. The first local implementation of a irradiation model was done in 2011 [2]. A statistical model was adjusted for the Uruguayan territory and it was able to perform hourly irradiation estimations with a spatial resolution of 2km and an uncertainty of 19.8%.

For research purposes, is usually required to process several times a big amount of satellite images. Typically, processing all the 91950 images database demands more than a day of computing time. The main contribution of this article is to present a study devoted to show how parallel computing techniques help to compute efficiently the satellite inputs required for a satellite-based solar resource model. A distributed-memory parallel algorithm was developed to assess mean satellite observed brightness in site neighborhoods that are distributed through the target territory. The utility of the proposed parallel algorithm relies in the possibility of performing in significantly reduced execution times the processing of the complete image database, which could be useful, for example, to process several times the data-bank varying the neighborhood size.

The rest of the article is organized as follows, in section 2 a brief description of the model and the satellite data-bank is presented. Section 3 explains the main design considerations for the parallel algorithms, while the implementation details of the proposed algorithm are described in section 4. The experimental analysis is reported in section 5. Finally, section 6 presents the conclusions of the research and the main lines for future work.

2 Problem description

This section briefly describes the model implemented in this article to assess solar irradiation from satellite imagery, and the required satellite information. Also, a description of the satellite data-bank and its information is offered.

2.1 Satellite-based model for solar resource estimates

The first model that use satellite information to estimate the available solar resource at ground level adjusted for the Uruguayan territory was the one by Justus et al. [6]. This model is a parametrization to estimate solar radiation from satellite data proposed in 1979 by Tarpley et al. [11], modified in 1986 by Tarpley and colaborators to his actual version, due to some bias problems noticed in the previous model. We will refer to that second version as *JPT model* hereafter. The JPT model is, in fact, an statistical model that utilizes visible channel satellite information to provide and estimation of the total amount of solar energy at a given point—specified by his latitude and longitude—at an hourly scale.

HPCLatAm 2012, pp. 34-48 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

Taking into account the statistical conception of the model, some parameters must be adjusted for a target region using measurements from both satellite radiometer and ground pyranometers. The JPT model proposes the multiple regression presented in Equation 1 where the parameters a, b, c and d are the regression coefficients.

$$I = I_{sc}\left(\frac{r_0}{r}\right)^2\left(a\cos\theta_z + b\cos^2\theta_z + c\cos^3\theta_z\right) + d\left(B_m^2 - B_0^2\right) \quad (1)$$

In Equation 1, I_{sc} is the hourly value of the solar constant ($I_{sc} = 4920\,\mathrm{kJ/m^2}$), $\cos\theta_z$ is the cosine of the zenital angle and $(r_o/r)^2$ is a factor that accounts to the Sun-Earth distance. All these variables could be calculated knowing the spatial position of a site $\{\phi, \psi\}$ (latitude and longitude, respectively) and a given time $\{n, h\}$ (day of year and hour) [5].

The information required from satellite images for both operational purposes and model adjustment, are the values of B_m and B_0. B_m is the actual brightness of a site at a certain time, and B_0 is the brightness for the same site and time but in clear-sky condition. So that, we will refer to B_m simply as a site brightness and to B_0 as a clear-sky brightness. Once the values of $\cos\theta_z$, brightness, and clear-sky brightness are calculated for every hour for each site, it is possible to make an estimation of the hourly irradiation, or, if the hourly integral for measurement is available for a specific site, it is possible to perform an adjustment of the coefficients a, b, c, and d by using a standard least square technique.

Due to the non-homogeneous shape of clouds and their quick movement within an hour, the B_m values are averaged in a small cell of a site. If more than one image is available for the hour, the B_m hourly value is assessed by the mean value of the values obtained at each image. In order to compute the clear-sky brightness, a parametrization is trained based on the B_m values. Thus, the computation of the B_m values by averaging the counts in a small cell of a site is the base step of the process.

In this article, we focus on the implementation of a parallel algorithm to perform the computation of the brightness values for equally spaced sites in latitude and longitude through the Uruguayan territory.

2.2 Satellite image data-bank

Image database consist of observations of the Geostationary Operational Environmental Satellite (GOES) located at geostationary orbit at 75 degrees West. The series of satellites that operated in that position is called GOES-East. An image for the visible channel and five spectral bands are available. The images were downloaded from the Comprehensive Large Array-data Stewardship System (CLASS) website that is administrated by the National Oceanic and Atmospheric Administration (NOAA), and available at `http://www.class.noaa.gov`. Images from the time period 2000 to date were acquired. The spatial resolution of the images is about 2km between pixels for the target region and, in average, there are two images per hour. GOES-East physical device has changed over time so that the database is compose with GOES8, GOES12 and GOES13 images.

HPCLatAm 2012, pp. 34-48 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

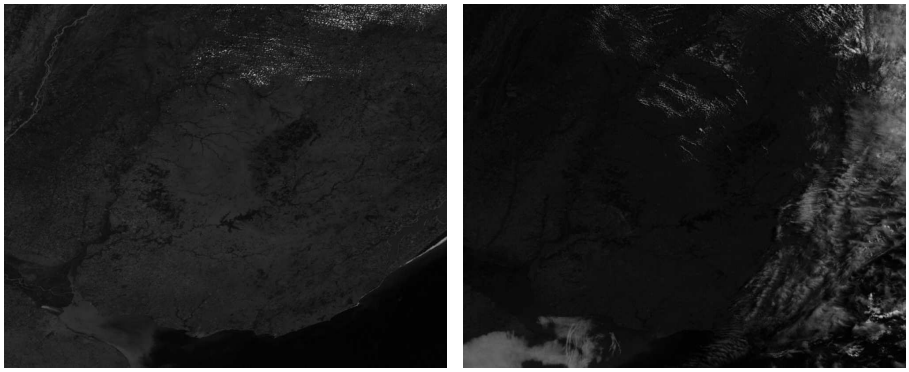
In total numbers, more than 90.000 images such as the ones presented in Fig. 1 must be processed in order to compute solar irradiation estimations for all the data-bank period. Without using parallel computing, processing this amount of images could take about 15 hours to perform in a single PC/server machine. Table 1 shows the composition of the satellite data-bank up to April 2012.

Table 1. Satellite database composition

satellite	start date	end date	images
GOES 8	01/01/2000	31/03/2003	24750
GOES 12	01/04/2003	14/04/2010	51900
GOES 13	14/04/2010	30/04/2012	15300
total	01/01/2000	30/04/2012	91950

The images acquired are in NetCDF format, a standard machine-independent data format that support the creation, access, and sharing of array-oriented scientific data [10]. Visible channel information is recorded in each file as a data matrix. Also, every file have his own navigation information due to the fact that GOES satellite might present orientation movements. Two additional matrices that correspond with the latitude (lat) and longitude (lon) are available for each image file. A position (i, j) in the matrices lat and lon correspond to the latitude and longitude information of the brightness count at the same matrix position.

The spatial window of the target region varies between 30 and 35 degrees South, and 53 and 59 degrees West, including the Uruguayan territory. A total of $(5 \times 30) \times (6 \times 30) \times 90.000 = 2.43 \times 10^9$ averages are needed to compute B_m at cells that are spaced by 5 minutes in latitude-longitude intervals.



(a) Clear-sky image. (b) Partially cloudy image.

Fig. 1. Two examples of visible channel satellite images.

HPCLatAm 2012, pp. 34-48 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

3 Parallel computing for satellite image processing

This section describes the main details about the parallel model and the design of the proposed parallel algorithm.

3.1 Context and parallel model

The parallel implementation described in this article is the first step for migrating the system to a C platform using parallel computing strategies, in order to implement an efficient operational model for solar image processing. The operational model will allow researchers to perform efficiently experiments related to the characterization and prediction of solar energy availability, climate research (fog, precipitation, cloud classification, etc.), agricultural products like the estimation of evapotranspiration, and other satellite assessed products. As a result, not only the main parallel algorithm was developed, but also a set of common libraries that are used by other similar algorithms or may be used by future ones. The implemented libraries are independent from any parallelization scheme.

Processing a single image takes a reduced execution time (about half a second), so each image can be efficiently processed by an individual processor. Applying parallel processing within each image is not useful from a performance-oriented point of view, since it severely reduces the granularity of each task, and causes that several processes simultaneously try to access to the same image or to write to the same output file.

The real complexity of the tackled problem relies on the large number of images to process. Thus, a data-parallel scheme was adopted to efficiently solve the problem. Several processes are used, each one of them conceived to execute in a different node in a distributed-memory cluster infrastructure. A master-slave parallel model was adopted as shown in Fig. 2.

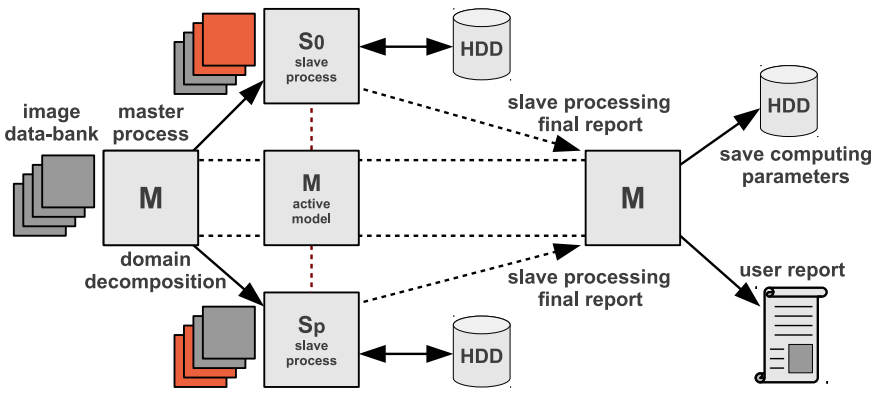


Fig. 2. Master-slave parallel model for the satellite image processing algorithm.

HPCLatAm 2012, pp. 34-48 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

The master process is in charge of performing the domain-decomposition and the distribution of work to slave processes. The domain decomposition approach divides the total amount of images into p subsets of time consecutive images, where p is the total amount of processes. At the beginning of its execution, every process receives a reference of the first and last image it is supposed to compute.

An *active* master-slave model is used: the master process also works in the image processing, performing the same operation than the slave processes.

Cell size may not be the same as the spacing between cells. Spacing between cells correspond with the spatial resolution of the resulting irradiation map for every image. On the other hand, cell size is concerned with the accuracy which with a B_m value, calculated for one particular image time, could represent all the time interval between two images. In this article, we work with 5×5 cell size and 5×5 latitude-longitude spacing between cells. These parameters are introduced via plain text file by the user, jointly with other configuration options. An example of a plain text file with input parameters to feed the algorithm is presented in Fig.3.

```
/home2/rodrigoa/satellite/TRAW/                    % Origin folder
/home2/rodrigoa/satellite/T000/                    % Destiny folder
1                                                  % Spectral band to process
2                                                  % Amount of years to process
2005                                               % First year to process
2009                                               % Second year to process
35 0 0 59 0 0                                      % Init of the target region
30 0 0 53 0 0                                      % Finish of the target region
0 5 0 0 5 0                                        % Cell's latitude and longitud spacing
0 5 0 0 5 0                                        % Cell's latitude and longitud size
```

Fig. 3. Example of plain text file with user parameters for the proposed algorithm.

4 Parallel implementation of the image processing algorithm

This section describes the details of the implemented parallel algorithm. It presents a description of the set of common libraries implemented and their characteristics, the parallelization scheme, and other features of the parallel implementation.

4.1 Brief description of implemented libraries

Three libraries were built: (a) a *processing* library, which implements all the specific processing duties, (b) an *assigment* library to assess the distribution of duties in the parallelization, and (c) a *calibration* library to perform satellite calibration and count-to-radiance conversion.

HPCLatAm 2012, pp. 34-48 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

The processing library implements all the specific functionalities required for image processing. This library includes the specials features to interact with NetCDF files, to write down the information to disk, and the logic to process every image. The reference to the start and finish image is received as a parameter. The images are scanned and the pixel values are accumulated in the corresponding cell based on the latitude and longitude information for the pixel. The average is computed for each every cell by performing the quotient between the accumulated value and the total amount of pixels counted for that cell. Finally, the average matrix of equally spaced cell is saved to disk.

In the current version of the processing software, the user is able to specify which years of images want to process. The assignment library is able to count how many images were requested to process by scanning hard-disk drive directories. Also, it has the internal logic to assess the domain-decomposition and the load balancing that is explained in the next subsection.

Finally, a library for satellite calibration was implemented. A set of coefficients is applied to the brightness count values to assess satellite observed radiance or to compensate the radiance due to satellite sensor degradation. These coefficients differ depending on the physical device and time. A different calibration procedure can be applied if a better one is available. Thus, a modular approach was followed to design the calibration library in order to easily allow changing this module.

4.2 Load balancing

Taking into account the uniform processing model for images, a static load balancing scheme was used. All the images in the data-bank have the same size and the processing of each separately image is exactly the same.

The parallel algorithm was conceived to execute in a dedicated parallel computing infrastructure. Thus, there is no a priori reason to think that a given process will result overloaded. In the proposed parallel algorithm, load balancing is performed simply by dividing the domain into subsets containing the same amount of images. The experimental evidence showed that in practice, some processes usually complete its assigned processing before some other ones, when executing the algorithm in operational mode. However, the deviation from the ideal equally-time processing is never larger than 20 images when processing a entire year (approximately 7500 images), corresponding to a negligible value of less than 10 seconds of execution time.

In the static load balancing scheme used, the distribution of images is done by the assignment library, which is able to count all the actual images at the directories. Then, the image assignments are performed based on the available quantity of processes p and the total amount of images required to process. The responsibilities for each process is assigned by generating six arrays, which keep the information about the starting and finishing image for each process. Thus, two arrays indicate the starting and finishing year, two more indicate the starting and finishig month and, finally, the last two arrays got the information about the starting and finishing image's index in the corresponding starting and

HPCLatAm 2012, pp. 34-48 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

finishing folder. The size of these arrays is equal to p, and are such that, when the k-process, $k :: k \in \{0 \ldots (p-1)\}$, evaluates them in the k position, they specify the first and last image of the assignation to that process.

4.3 Algorithm description

The parallel algorithm was implemented in C. The parallelism was implemented following the MPI standard for parallel and distributed programming [4], by using the 1.2.7p1 version of the well-known MPICH implementation.

The algorithm is composed by four main stages: (a) initialization and data parallel distribution (b) upload of images from hard-disk, (c) computation of the B_m values for each assigned image, and (d) save processed data to hard-disk. A final step is done by the master process to save some final parameters and information regarding the size of the cell, the amount of cells, and the grid used in the processing. A graphical explanation of this scheme is presented in Fig. 4.

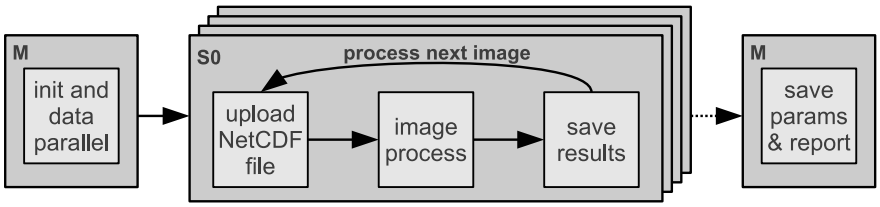


Fig. 4. Flow scheme of the stages in the image processing algorithm.

The master process is in charge of performing the initialization phase of the algorithm. The master reads the user data, initializes some parameters of the system (e.g. paths, filenames, cell size and spacing), counts the images in the data-bank, and generates the data for domain decomposition. After that, the master process send the references about the images, as well as user and system variables, to each one of the p slave process. Once the initialization phase is done, every slave process knows all the data needed to work independently. In particular, each slave have references to the first and the last image to process and where it has to write down to hard-disk the information about the processing of each assigned image.

The output of the processing includes six plain text files. Four of them contain the grid values that correspond to the brightness for each cell, and three subproducts from the image calibration. The last two files are the grid mask that indicate if a given value in the grid is corrupted or not, and the amount of pixels counted for each cell in the grid. When a process finishes its assigned processing, it sends a notification to the master, jointly with a resume of the processing performed. Since all processes inform the master when they stop working, the master knows when all the requested processing is done.

Fig. 5 shows an example of the (optional) user output that the master process writes after the initialization stage just before creating and launching the slave processes. In order to illustrate the domain decomposition performed, the sample case in Fig. 5 shows that the algorithm is executed with four processes executing on four processing units ($p = 4$) and the parameters already presented in Fig. 3. In the presented example, each process is in charge of processing 3892 images ($q = 3892$), and 3 out of 4 processes has to compute one extra image ($r = 3$). A total number of 15.571 NetCDF files are processed. This kind of output is useful to inform the user about the system parameters, as a way to know if they are adequate before processing.

```
-------------------------------------------------------------------------------------------
----- rank = [0] :: Processing data -------------------------------------------------------
-------------------------------------------------------------------------------------------
Path: /home2/rodrigoa/satelite/TRAW/
Channel to process:      [01.nc]           Years to process: [ 2005, 2009]
Amount of images found:  [15571]           Assigment: q = [3892] :: r = [3]
                                           Amount of nodes = [4]
Amount of cells to process:                Ci = [30] :: Cj = [36] :: Ct = [1080]
                                           Latitude region =  [-35.000000 ... -30.166667]
                                           Longitude region = [-59.000000 ... -53.166667]
                                           incLAT = [0.083333] :: incLON = [0.083333]
Channel images per month:    [ 826, 718, 744, 594, 539, 403, 469, 560, 411, 710, 840, 862]
                               842, 707, 683, 596, 568, 486, 542, 510, 679, 770, 716, 796]
-------------------------------------------------------------------------------------------
```

Fig. 5. Example of the (optional) master process user output after the initialization stage, just before launching the slave processes. All system parameters are reported.

5 Performance evaluation

This section describes the computational platform and the image test-set used in the experimental analysis. After that, the methodology followed in the experimental analysis is described. Finally, the results of the performance analysis are presented and discussed, with special emphasis on the speedup analysis of the parallel algorithm.

5.1 Execution platform

The experimental analysis of the proposed parallel algorithm was performed in a server with two Intel quad-core Xeon processors at 2.6 GHz, with 8 GB RAM, CentOS Linux, and Gigabit Ethernet. The infrastructure is part of the Cluster FING, Facultad de Ingeniería, Universidad de la República, Uruguay; cluster website: `http://www.fing.edu.uy/cluster`).

5.2 Test set images

A test-set of images was used to carry out the performance analysis for the satellite data-bank processing algorithm. NetCDF files that correspond to satellite images of the year 2011 were used. A total number of 7670 images comprises the data-bank information for that year, which represent a total disk capacity of about 60GB. Hard-disk drive used to store the test set was local to the node used to run the evaluation, to avoid the performance degradation due to transferring large image files through NFS.

5.3 Methodology

The experimental evaluation studies the execution time of the parallel algorithm when varying the number of working processes between 2 and 8. This subsection introduces the performance metrics used to evaluate the parallel algorithm and the methodology used in the analysis.

Performance metrics. The most common metrics used by the research community to evaluate the performance of parallel algorithms are the *speedup* and the *efficiency*.

The speedup evaluates how much faster a parallel algorithm is than its corresponding sequential version. It is computed as the ratio of the execution times of the sequential algorithm (T_1) and the parallel version executed on m computing elements (T_m) (Equation 2). When applied to non-deterministic algorithms, the speedup should compare the *mean* values of the sequential and parallel execution times (Equation 3). The ideal case for a parallel algorithm is to achieve linear speedup ($S_m = m$), but the most common situation is to achieve sublinear speedup ($S_m < m$), mainly due to the times required to communicate and synchronize the parallel processes.

The efficiency is the normalized value of the speedup, regarding the number of computing elements used to execute a parallel algorithm (Equation 4). This metric allows the comparison of algorithms eventually executed in non-identical computing platforms. The linear speedup corresponds to $e_m = 1$, and in the most usual situations $e_m < 1$.

$$S_m = \frac{T_1}{T_m} \quad (2) \qquad S_m = \frac{E[T_1]}{E[T_m]} \quad (3) \qquad e_m = \frac{S_m}{m} \quad (4)$$

Statistical analysis of execution times. In order to reduce the effect of non-determinism in the execution, a well-know side effect of parallel programming [3], fifty independent execution of the parallel program were performed for each value of p. For every execution the total time spent for processing the image data-bank was recorded, and the average execution time of the fifty executions was computed for every value of p, in order to compute an accurate estimation for the operation time of the test.

In addition, an estimation of the time required by the sequential algorithm was computed by averaging fifty executions. The proposed parallel algorithm does not modify the algorithmic structure of the sequential version, since only the image processing is performed in parallel and no additional components are included. From the algorithmic point of view, the required communication between master and slave processes to start the execution and return the results is identical to a standard function calling in a sequential algorithm.

The time required for both the initialization of the MPI environment and the data parallel distribution are negligible in comparison with the time needed to processed the images. The final stage of recoding the system parameters and the grid data, which is performed by the master process, cannot be parallelized. It is included in the *serial fraction* of the parallel algorithm, since it demands the same execution time that when using a sequential algorithm. Taking into account the previous comments, the time that a sequential algorithm will require to perform the processing is almost equal to the time that the proposed parallel algorithm needs when using a single processing element($p = 1$).

With the aforementioned execution time values, estimations for the speedup and efficiency metrics are computed to evaluate the performance of the proposed parallel algorithm for solar image processing.

Results and discussion. Table 2 reports the best, average, and standard deviation (σ) values of the execution times computed in the fifty independent executions performed for both the sequential and the parallel algorithm for different values of the number of processes p. The corresponding speedup values for different values of p are also reported.

Table 2. Performance metrics for different values of p.

processes	execution time (s)			metric	
	best	avg	σ	speedup	efficiency
1 (sequential)	4555	4786	380	–	–
2	2587	2720	316	1.76	0.88
3	2047	2123	214	2.25	0.75
4	1989	2062	130	2.32	0.58
5	2124	2200	136	2.18	0.44
6	2127	2360	293	2.03	0.34
7	2324	2370	36	2.02	0.29
8	2438	2484	32	1.93	0.24

According to the straightforward domain decomposition approach applied in the proposed parallel algorithm, the performance was expected to increase when increasing the number of processes. However, the analysis of the execution time results reported in Table 2 indicates that this behavior only holds for $p < 5$.

HPCLatAm 2012, pp. 34-48 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

Fig. 6 shows examples of the estimated average execution time values obtained in the fifty executions of the sequential and the parallel algorithm with $p = 2$, $p = 4$ and $p = 8$. The figure shows that the majority of the execution time values are aligned, and the standard deviation tends to reduce when more processes are used.

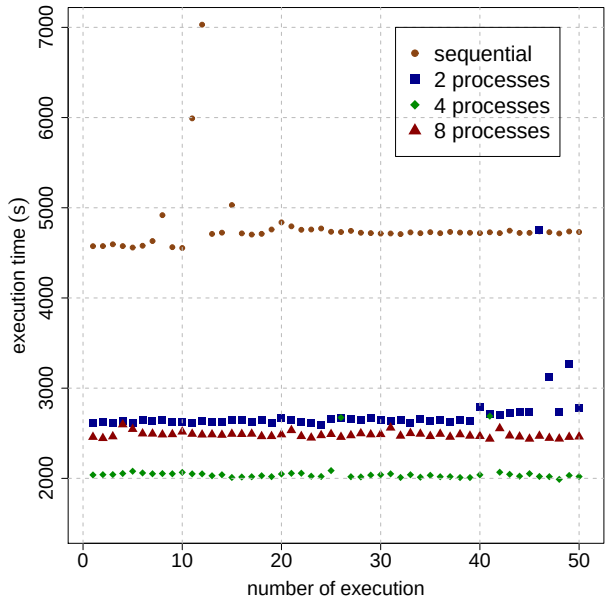


Fig. 6. Example of the times recorded for fifty executions of the proposed parallel algorithm for $p = 1$, $p = 2$, $p = 4$, and $p = 8$.

Regarding the speedup and efficiency metrics, a speedup value of 1.76 was achieved when working with $p = 2$ and then it is slightly improved using more processes, obtaining a maximum speedup value of 2.32 at $p = 4$. Beyond this point, using more processes causes the execution times to deteriorate. The most probable explanation for this phenomena is that the input/output capacity of the hard-disk drive severely affects the computational efficiency of the proposed parallel algorithm. Due to the large data transferred, the maximum bandwidth capacity of the data bus is almost reached using $p = 3$ and fully achieved using $p = 4$. In addition, when a large number of processes (e.g. $p > 6$) is used, the idle times due to unbalanced processing also contributes to reduce the computational efficiency of the proposed parallel algorithm.

The variation of the best and average execution times for different values of p is presented in Fig. 7. The variation of the values for the speedup and efficiency metric for different values of p is presented in Fig. 8 (a) and (b), respectively.

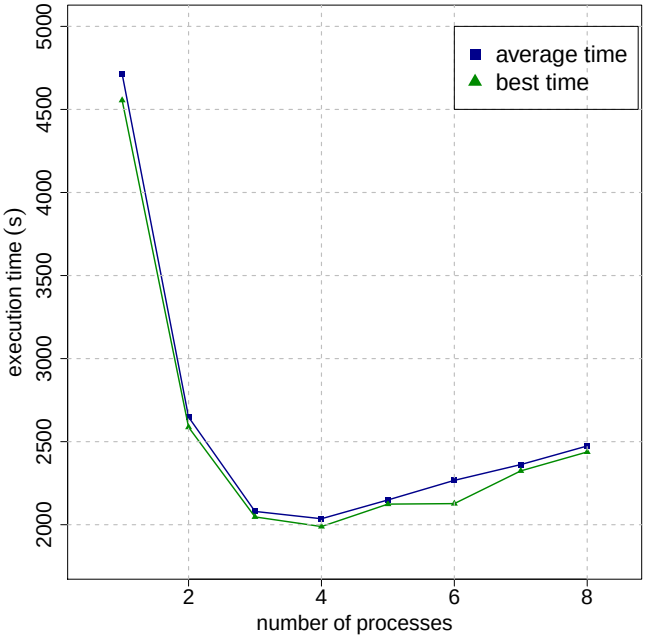
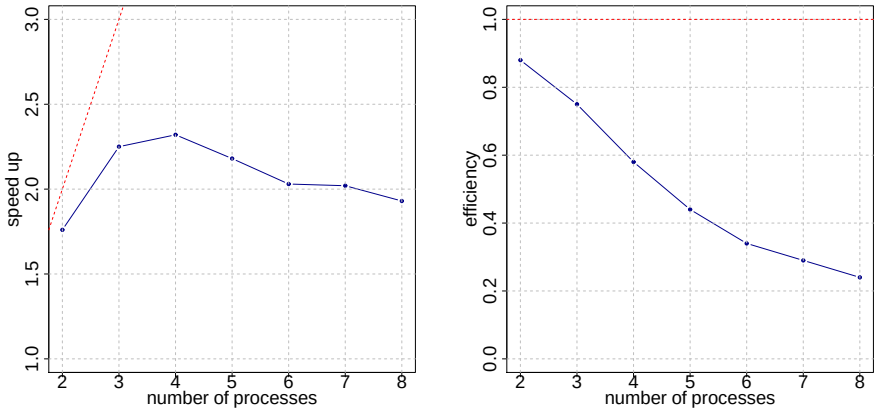


Fig. 7. Best and average execution times vs. processes used.



(a) speedup vs. processes. (b) efficiency vs. processes.

Fig. 8. Estimated speedup and efficiency vs. processes used. A red line was drawn to show the ideal linear speedup/efficiency situation

HPCLatAm 2012, pp. 34-48 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

6 Conclusions and future work

This article has presented a parallel implementation for an algorithm to process satellite image information for solar resource estimates.

Nowadays, satellite applications in Uruguay are at a research stage. As a consequence, satellite imagery is being processed several times. Processing each image individually does not take many computational effort and half a second is often spent. The complexity of the problem lies in the big amount of images that should be processed. The computation of all the database involves to work with more than 90000 images and a suitable automatic solution to process such a volume of information is needed. In this line of work, the parallel computing strategy described in this article was applied to reduce the computation time of a GOES-East satellite data-bank.

A parallel master-slave algorithm was developed in order to solve the particular problem of computing the inputs for a satellite-based solar irradiation model. Also, a set of common libraries were implemented to address specific processing tasks. The proposed scheme is based on image domain-decomposition in order to fully exploit the master-slave parallel model when executing in a cluster infrastructure.

The experimental analysis show that significant improvements in the execution times are obtained with the parallel algorithm when compared with the sequential version. A maximum speedup value of 2.32 was reached by the proposed parallel algorithm when using four processes, but when using more than four processes the computational efficiency reduces and the execution times increase. The phenomena is explained by the limited input/output hard-disk drive bandwidth of the infrastructure used. In its current implementation, the algorithm shall be used splitting the work on four processes, to take the best advantage of the parallel platform. Although the proposed parallel algorithm does not scale appropriately for more than four processes, it is a promising first step in the quest of designing an efficient automatic tool for solar image processing in our research context.

The main lines for future work are related to improving the computational efficiency of the proposed method, by addressing the main issues that conspire against a good scalability behavior. We plan to implement both a dynamic load balancing to execute in non dedicated infrastructures, and the parallelization of hard-disk drive access as a strategy to reduce the negative effects of hard-disk bounded bandwidth. The distribution of images in different physical hard-disk drives will aloud to launch processes in nodes with local access to them. This is expected to increase the efficiency of the parallelization when using more than four processes, and thus, to decrease execution times in such situation. Another promising line for future work is the possibility of executing the image processing in grid infrastructures, in order to take advantage of the large resource availability of distributed computing platforms.

Acknowledgments

The work of R. Alonso has been partially supported by ANII and CSIC, Uruguay. The work of S. Nesmachnow has been partially supported by ANII and PEDECIBA, Uruguay.

References

1. G. Abal, M. D'Angelo, J. Cataldo and A. Gutierrez, *Mapa Solar del Uruguay.* IV Conf. Latinoamericana de Energía Solar (IV ISES-CLA), 2010 (text in Spanish).
2. R. Alonso, G. Abal, R. Siri, P. Musé and P. Toscano, *Solar irradiation assessment in Uruguay using Tarpley's model and GOES satellite images.* Proceedings of the 2011 ISES Solar World Congress. 2011.
3. I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison Wesley, 1995.
4. W. Gropp, E. Lusk and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface.* MIT Press Cambridge, MA, USA, 1999.
5. M. Iqbal, *An introduction to Solar Radiation*, Academic Press, 1983.
6. C. Justus, M. Paris and J. Tarpley, *Satellite-measured insolation in the United States, Mexico, and South America.* Remote Sensing of Environment, vol. 20, pag. 57-83, 1986.
7. M. Noia, C. Ratto and R. Festa, *Solar irradiance estimation from geostationary satellite data: 1. Statistical Models*, Solar Energy 51, 449–456, 1993.
8. M. Noia, C. Ratto and R. Festa, *Solar irradiance estimation from geostationary satellite data: 2. Physical Models*, Solar Energy 51, 457–465, 1993.
9. R. Perez, R. Seals and A. Zelenka, *Comparing satellite remote sensing and ground network measurements for the production of site/time specific irradiance data*, Solar Energy 60, 89–96, 1997.
10. R. Rew and G. Davis, *NetCDF: An Interface for Scientific Data Access*, IEEE Computer Graphics and Applications **10**(4), 76–82, 1990.
11. J. Tarpley, *Estimating Incident Solar Radiation at the Surface from Geostationary Satellite Data.* Journal of Applied Meteorology, vol. 18, pag. 1172-1181, 1979.

HPCLatAm 2012, pp. 49-64 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

Parallel conversion of satellite image information for a wind energy generation forecasting model

Germán Gadea, Andrés Flevaris, Juan Souteras, Sergio Nesmachnow,
Alejandro Gutiérrez, and Gabriel Cazes

Universidad de la República, Uruguay
{ggadea,aflevaris,jsouteras,sergion,aguti,agcm}@fing.edu.uy

Abstract. This paper presents an efficient parallel algorithm for the problem of converting satellite imagery in binary files. The algorithm was designed to update at global scale the land cover information used by the WRF climate model. We present the characteristics of the implemented algorithm, as well as the results of performance analysis and comparisons between two approaches to implement the algorithm. The performance analysis shows that the implemented parallel algorithm improves substantially against the sequential algorithm that solves the problem, obtaining a linear speedup.

1 Introduction

Wind prediction is crucial for many applications in environmental, energy, and economic contexts. The information about wind is important for weather forecasting, energy generation, aircrafts and ship traffic, dispersal of spilled fuel prediction, coastal erosion, and many other issues. In the last thirty years, researchers have made important advances in methods and models for wind prediction [1,4].

The Weather Research and Forecasting (WRF) model [8] is a flexible and efficient mesoscale numerical weather prediction system developed by a collaborative partnership including several centers, administrations, research laboratories and universities in the USA. With a rapidly growing community of users, WRF is currently in operational use at several centers, labs and universities through the globe, including our research group at Instituto de Mecánica de los Fluidos e Ingeniería Ambiental (IMFIA) de la Facultad de Ingeniería, Universidad de la República, Uruguay.

In our context, WRF is applied to analyze the availability of wind energy, which depends on the wind speed, in order to perform accurate forecasting in the range of 24-48 hours, required for the integration of wind energy into the power grid. These predictions are a valuable help for the power grid operators to make critical decisions, such as when to power down traditional coal-powered and gas-powered plants.

Soil information is very relevant for wind forecasting using WRF, since the terrain type directly affects the wind received by the generators (usually placed at 100 m from the ground). The soil information used by the WRF is outdated, and in the case of Uruguay, the last actualization was performed in the early 1990's. Thus, there is a specific interest on updated soli information in order to improve the accuracy of wind forecasting.

In the WRF model, the information about soil is stored in files using a proprietary binary format. so, in order to perform the soil information update, it is needed to convert the information obtained from satellite images to the binary format used in WRF. The conversion process also includes performing a change of projection due to the input information from satellites has a different projection than the output information.

In order to perform the soil information update not only for our country, but at planetary scale, a large number of satellite images need to be processed. In this context, applying high performance computing (HPC) techniques is a valuable strategy to reduce the execution time required to process the large volume of information in the images. More than 300 images have to be processed in the world scenario, with a total size of 27 GB, and the sequential algorithm demands 1710 minutes of execution time.

The main contributions of the research reported in this article are: i) to introduce two parallel versions—using shared memory and distributed memory approaches—of an algorithm that process the satellite images to update the soil information used for wind prediction in the WRF model, and ii) to report an exhaustive experimental analysis that compares the computational efficiency of the shared and distributed memory parallel versions. The experimental results demonstrate that both parallel implementations achieve good computational efficiency, and the distributed memory is the most efficient method for parallelization.

The rest of the manuscript is organized as follows. Next section present a review of related work on parallel algorithms to process satellite images for wind prediction. Section 3 describes the design and details of the proposed parallel algorithm. The description of the two variants implemented are presented in Section 4. The experimental analysis that compares the two parallel versions and the sequential one is presented in Section 5. Finally, Section 6 summarizes the conclusions of the research and formulates the main lines for future work.

2 Related work

Several works have recently addressed the satellite image processing problem using high-performance computing techniques. These works are mainly focused on processing data received directly from the satellites, making a classification of pixels in order to obtain land cover information [3,5]. Maulik and Sarkar [3] proposed a strategy for satellite image classification by grouping the pixels in the spectral domain. This method allows performing the detection of different land cover regions. A parallel implementation of the proposed algorithm following the master-slave paradigm was presented in order to perform the classification efficiently. The experimental analysis on different remote sensing data performed on INRIA PlaFRIM cluster varying the number of processors from 1 to 100, demonstrated that the proposed parallel algorithm is able to achieve linear speedup when compared against a sequential version of the algorithm.

Nakamura et al. [5] described how the researchers at Tokyo University of Information Sciences receive MODIS data to be used in one of the major fields of research: the analysis of environmental changes. Several applications to analyze environmental changes are developed to execute on the satellite image data analysis system, which is implemented in a parallel distributed system and a database server.

Sadykhov et al. [9] described a parallel algorithm based on fuzzy clustering for processing multispectral satellite images to enforce discrimination of different land covers and to improve area separation. A message passing approach was used as basis of parallel calculation because it allows simple organization of interaction between of calculating processes and synchronization. Experimental testing of developed algorithms and techniques has been carried out using images received from Landsat 7 ETM+ Satellite. However, the article does not report experimental analysis focused on evaluating the performance improvements when using parallel computing techniques.

Plaza et al. [6,7] described a realistic framework to study the parallel performance of high-dimensional image processing algorithms in the context of heterogeneous networks of workstations (NOWs). Both papers provided a detailed discussion on the effects that platform heterogeneity has on degrading parallel performance in the context of applications dealing with large volumes of image data. Two representative parallel image processing algorithms were thoroughly analyzed. The first one minimizes interprocessor communication via task replication. The second one develops a polynomial-time heuristic for finding the best distribution of available processors along a fully heterogeneous ring. A detailed analysis of parallel algorithms is reported, by using an evaluation strategy based on comparing the efficiency achieved by an heterogeneous algorithm on a fully heterogeneous NOW. For comparative purposes, performance data for the tested algorithms on Thunderhead (a large-scale Beowulf cluster at NASA Goddard Space Flight Center) are also provided. The experimental results reveal that heterogeneous parallel algorithms offer a surprisingly simple, platform-independent, and scalable solution in the context of realistic image processing applications.

Unlike the previously commented articles, our research corresponds to a later stage, which took the land cover information generated by some source and convert it to another format to feed the WRF model. We have designed and implemented two parallel implementations of the conversion algorithm, using shared memory and distributed memory approaches. Both parallel implementations of the conversion algorithm are currently operative in our cluster infrastructure (Cluster FING), allowing to perform an efficient conversion of satellite images downloaded from NASA satellites, in order to update the information used by the WRF climate model.

3 Conversion of satellite images to binary files

This section describe the main decisions taken to design the parallel conversion algorithm, and the main features of the parallel model used.

3.1 Design considerations

The proposed algorithm implements the conversion of satellite images from the Terra and Aqua (NASA) satellite to binary files supported by the WRF model.

In the design phase of the algorithm, it was necessary to analyze the input and output files of the conversion process and the way that the data is contained. After that, the design of a strategy for converting the information from the input format to the output format was devised.

In order to reduce the large execution times required by a sequential implementation of the algorithm when processing a large number of images, a parallel implementation was conceived in order to assure a more efficient processing. The parallel implementation is capable to convert the full domain requiring significantly lower execution times, allowing the researchers to scale up and processing world-size scenarios.

3.2 Data-parallel: domain decomposition

The work domain of the WRF model is a grid that covers all the world. Each cell in this grid represents an area of 600×600 kilometers of land cover. Thus, a straightforward domain decomposition is suggested by using the WRF grid. Since the WRF grid is the output of the conversion process, an *output domain descomposition* is used.

The domain decomposition is achieved by generating small cells that are represented by matrices with dimensions 1200×1200 (rows by columns). This data partition is important because it divides the amount of data that each process in the parallel algorithm has to work with.

3.3 Parallel model

Taking into account the characteristics of the algorithm to be parallelized, the selected domain partition, and mainly because there is no need to use border information in order to generate one cell of the output domain, a master-slave model was adopted to implement the parallel algorithms. The use of this model to implement the communication between the processes seems to be appropriate for the conversion algorithm, because the slave processes participating in the conversion process do not have to share information between each other. Figure 1 presents a graphic representation of the proposed parallel algorithm, showing the interaction between the different processes.

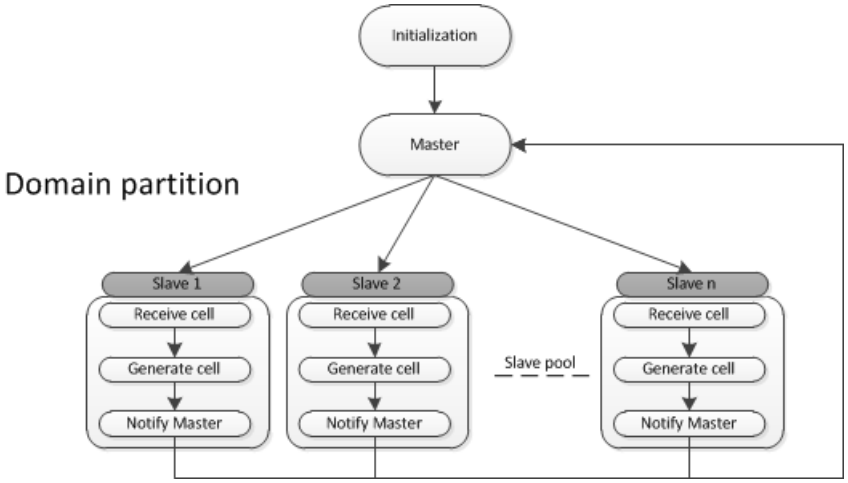


Fig. 1. Scheme of the proposed parallel algorithm.

In the proposed parallel model, the master process initializes all data structures, variables and control structures, and then continues fully dedicated to control the execution of the conversion process and implementing a dynamic load balancing schema for assigning tasks to the slave processes. On the other hand, the slaves processes receive the assigned work or cell, and then they execute the processing tasks in order to convert the land cover data, from the satellite format to binary format.

Two variants of the proposed algorithm were implemented with focus on two different paradigms of parallel computing. One variant was implemented using shared memory, and the other using distributed memory. Both algorithm variants follows the same general approach, but they are designed to execute on different parallel computing infrastructures. The shared memory algorithm is conceived to specifically execute on a multicore computer, while the distributed memory algorithm is able to execute on a distributed infrastructure, such as a cluster of computers.

Both algorithms have been tested in a hybrid cluster infrastructure formed by many multicore computers. Testing the algorithms in the same environment makes possible an analysis comparing the two implementations.

3.4 Load balancing

The two versions of the implemented algorithm gain efficiency by applying a correct load balance strategy. Taking advantage of the domain partition selected, the proposed strategy for load balancing is conceived to be performed in two steps. First, at the initialization step, the master process statically assigns to each slave process a cell to generate. This initial assignment assures that all the slave processes have a work to perform at the beginning of their execution. After that, in each execution step, while the master process has cells to generate and one of the slave processes finish its work, the master dynamically assigns to that slave another cell. The dynamic cell assignment performed by the master process keeps the conversion process running and generating cells, while minimizing the idle time. Each time that one of the slaves processes finishes the generation of a cell, the master process immediately assigns a new one to be generated, and the slave keeps working.

4 Parallel implementation of the conversion algorithm

This section presents the implemented parallel algorithms for the conversion process. In the shared memory algorithm, the master and slaves processes are threads, which are controlled and synchronized using mutexes In the distributed memory algorithm, the master and slaves are processes, which use message passing to perform communication and synchronization.

4.1 Data structures

The common data structures used by the both implemented algorithms are the ones listed in the Data structure frames 1.1 and 1.2.

HPCLatAm 2012, pp. 49-64 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

```
struct descriptorHDF{
  int h;
  int v;
  char fileName[1024];
  char gridName[64];

  // coord SINUSOIDALES
  long double lowerLeftLat_syn;
  long double lowerLeftLon_syn;
  long double lowerRightLat_syn;
  long double lowerRightLon_syn;
  long double upperLeftLat_syn;
  long double upperLeftLon_syn;
  long double upperRightLat_syn;
  long double upperRightLon_syn;

  // coord GEOGRAFICAS
  long double lowerLeftLat_geo;
  long double lowerLeftLon_geo;
  long double lowerRightLat_geo;
  long double lowerRightLon_geo;
  long double upperLeftLat_geo;
  long double upperLeftLon_geo;
  long double upperRightLat_geo;
  long double upperRightLon_geo;
};
```

Data structure 1.1. descriptorHDF

The data structure descriptorHDF contains the fields to save the necessary information about the satellite imagery files. This data structure saves the coordinates in geographic projection and in sinusoidal projection. Both groups of coordinates indicate the area covered by the file. The data structure also contains the fields 'h' and 'v' that indicates the horizontal and vertical position of the HDF file at the satellite imagery grid, respectively. Other fields are used for the file name and for the object that contains the information to be converted.

```
struct celdaSalida{
  char nombreArchivoSalida[256];
  long double lowerLeftLat;
  long double lowerLeftLon;
  long double lowerRightLat;
  long double lowerRightLon;
  long double upperLeftLat;
  long double upperLeftLon;
  long double upperRightLat;
  long double upperRightLon;
};
```

Data structure 1.2. celdaSalida

The data structure celdaSalida is used in the algorithms to indicate a given slave process which cell of the output grid it must generate. The data structure contains fields for the cell coordinates, and the output binary file name. This structure is communicated between the master and slaves processes when the master assigns a cell to be generated by the slave process.

4.2 Shared memory algorithm

The shared memory version of the conversion algorithm uses a pool of threads, implemented using the standard POSIX thread library (pthread). The algorithm is divided in three procedures, the procedure that runs the master thread, other for each slave thread, and the main procedure. The conversion algorithm has two phases. In the first phase, the main procedure initializes all the threads, mutexes and data structures. The data structures used by the algorithm are an array of 'descriptorHDF' and a matrix of 'celdaSalida'. The shared memory algorithm uses a set of global variables:

- turn: used by each slave thread to indicate the master when it has finished working.
- finish: used to indicates the slave treads to exit.
- iSlave, jSlave: this are two arrays of integer used to indicate which cell generates a slave. The indexes i and j indicate the position on the matrix 'celdaSalida'

In the second phase of the algorithm, the master thread first assigns one cell to each slave, and then it waits for the slave answers, to assign a new cell to the requesting slave. When all cells have been assigned, the master thread waits until all slaves finish their work, and then it sets the 'finish' variable to true, causing all slaves to exit. On the other hand, the slaves threads are implemented as a loop that generate cells and performs the conversion until the 'finish' variable is set to true.

On each loop step, the slave processes execute three main tasks. Initially, the assigned cell is received; after that, the assigned cell is generated, and finally the master process is notified that the cell has been generated and the slave process is available to get a new cell assigned. A pseudocode of the algorithm is presented in Algorithm 1 (main program), Algorithm 2 (master process), and Algorithm 3 (slave processes).

Algorithm 1 Main program

```
1: initialize mutexes
2: create and initialize master thread
3: create and initialize slave threads
4: //parallel execution
5: wait(for the exit of all threads)
6: destroy(all structures used)
```

4.3 Distributed memory algorithm

The parallel algorithm has been implemented to be executed in a distributed infrastructure such as a cluster of computers, using the C language and the Message Passing Interface library (MPI) [2]. MPI allows simple organization, communication, and synchronization of the master and the slaves processes.

For the distributed execution, the set of satellite images to convert (with a total size of 27 GB), was stored in the file system of the cluster, that is accessible by all running processes.

Algorithm 2 Master thread

```
1: for all slave thread do
2:     assing(cell to generate)
3:     unlock(slave)
4: end for
5: while has cell to be generated do
6:     wait(slave answer)
7:     receive(answer)
8:     assign(new cell to the slave who answered)
9:     unlock(slave who answered)
10: end while
11: while are slave working do
12:     wait(slave answer)
13: end while
14: finish ← true
```

Algorithm 3 Slave thread

```
1: isThereWork ← true
2: while isThereWork do
3:     lock(until work is assigned)
4:     if not finish then
5:         i ← iSlave[my_id]
6:         j ← jSlave[my_id]
7:         search(HDF files to use)
8:         stitch(HDF files)
9:         proyectAndSubset(cell to generate)
10:        generateBinaryFile()
11:        wait(turn to alert the master)
12:        turn ← my_id
13:        alert(master)
14:    else
15:        isThereWork ← false
16:    end if
17: end while
```

The creation of distributed processes is configured and built using the available TORQUE manager in the cluster FING. The number of processes to create, the cluster nodes to be used, the distribution of processes between nodes, and the priority of the scheduled job are indicated in a configuration file.

The structure of the algorithm closely matches the shared memory algorithm already described. In a first step, the master process initializes the data structures, and in the second stage, the master process begins to perform dynamic load balancing by assigning the work to the slave processes. Then, it waits the responses from the slaves processes, and then assigns new work to idle slaves. On the other hand, the slave processes execute a loop with the following steps: expect a cell to generate, running the conversion process of the received cell, and finally notifies the end of processing to the master process.

Slaves remain in this loop until the master process indicates to finish the execution. A pseudocode of the algorithm is presented in the algorithm 4.

One of the most important tasks in the communication between the master process and slave processes is sending the array with the information of the input files (descriptorHDF). Since this is a large amount of information organized in an array of structs, the use of the common functions of MPI for sending data (MPI_Send) is ineffective, since they cause corruption in the information. For properly implement the communication, it was necessary to use specific MPI functions for sending arrays by performing a serialization of structures (functions MPI_Buffer_attach, MPI_Buffer_dettach and MPI_Bsend). The master process has to wait for messages sent by the slave processes to report that they have finished processing the assigned work, so the master process is blocked by applying the MPI_Recv function using the MPI_ANY_SOURCE flag, which allows receiving messages from any process. To find out which process is the source of communication, the master reads the status parameter returned by the operation and so is obtained the rank that identifies the slave process.

Algorithm 4 Distributed memory algorithm

```
1:  initialize MPI structures
2:  master process do
3:      initialize matrix of celdaSalida
4:      initialize array of descriptorHDF
5:  master process send array of descriptorHDF to all slaves
6:  if master process then
7:      for i = 1 → #process do
8:          send(cell[i], process[i])
9:      end for
10:     while not receive all slaves answers do
11:         waitForAnswer(answer)
12:         slave_id ← answer.slave_id
13:         cell ← answer.cell
14:         markCellAsGenerated(cell)
15:         if not allCellsGenerated() then
16:             cell ← nextCell()
17:             send(cell, process[slave_id])
18:         end if
19:     end while
20:     send finish message to all slaves processes
21: else if slave process then
22:     while not receive finish message do
23:         cell_id ← receive()
24:         generate(cell)
25:         sendMaster(cell_id, my_id)
26:     end while
27: end if
```

HPCLatAm 2012, pp. 49-64 (full paper)
ISSN 2422-5207
Session: High performance scientific computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

5 Experimental evaluation

This section presents the results of the experimental evaluation for the two parallel versions of the proposed algorithm.

5.1 Development and execution platform

Both implementations of the parallel conversion algorithms were developed in C. The distributed algorithm uses MPICH version 1.2.7, and the shared memory algorithm was implemented using the pthread library. The experimental evaluation was performed in a server with two Intel quad-core Xeon processors at 2.6 GHz, with 8 GB RAM, CentOS Linux, and Gigabit Ethernet (Cluster FING, Facultad de Ingeniería, Universidad de la República, Uruguay; cluster website: `http://www.fing.edu.uy/cluster`).

5.2 Data used in the experimental evaluation

The input of the conversion process is a set with more than 300 image files in HDF format, with a total size of 28 GB of information. As a baseline reference, converting all these images with a sequential process takes 1710 minutes.

5.3 Performance metrics

The most common metrics used by the research community to evaluate the performance of parallel algorithms are the *speedup* and the *efficiency*.

The speedup evaluates how much faster a parallel algorithm is than its corresponding sequential version. It is computed as the ratio of the execution times of the sequential algorithm (T_S) and the parallel version executed on m computing elements (T_m) (Equation 1). The ideal case for a parallel algorithm is to achieve linear speedup ($S_m = m$), but the most common situation is to achieve sublinear speedup ($S_m < m$), mainly due to the times required to communicate and synchronize the parallel processes.

The efficiency is the normalized value of the speedup, regarding the number of computing elements used to execute a parallel algorithm (Equation 2). The linear speedup corresponds to $e_m = 1$, and in the most usual situations $e_m < 1$.

We have also studied the *scalability* of the proposed parallel algorithm, defined as the ratio of the execution times of the parallel algorithm when using one and m computing elements (Equation 3).

$$S_m = \frac{T_S}{T_m} \quad (1) \qquad e_m = \frac{S_m}{m} \quad (2) \qquad Sc_m = \frac{T_1}{T_m} \quad (3)$$

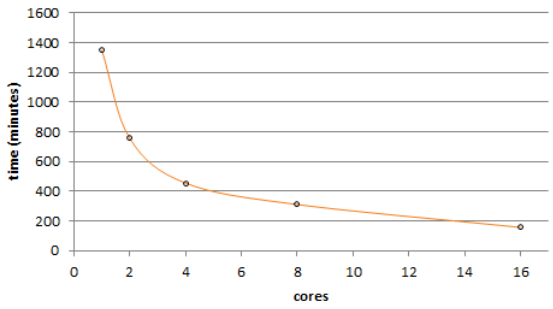
5.4 Performance evaluation and discussion

This subsection presents and analyzes the performance results of the implemented shared memory and distributed memory parallel algorithms. All the execution time results reported correspond to the average values computed in five independent execution performed for each algorithm and experimental scenario.

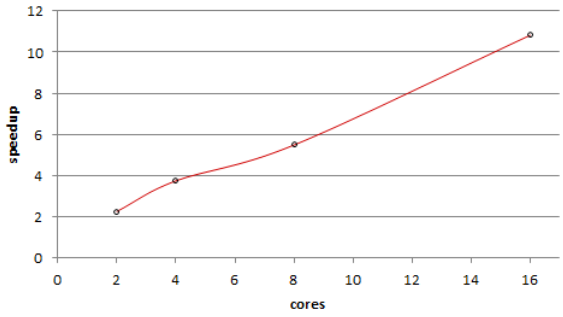
Shared memory implementation. The experimental analysis of the shared memory parallel implementation was performed on a single host, varying the number of cores used. Table 1 reports the execution time results and the performance metrics for the for the shared memory implementation, and Figures 2 and 3 graphically summarize the results.

# cores (m)	**time** (minutes)	**speedup** (S_m)	**efficiency** (e_m)	**scalability** (Sc_m)
1	1349.00	1.27	1.27	1.00
2	683.00	2.26	1.13	1.78
4	456.00	3.75	0.94	2.96
8	311.67	5.49	0.69	4.33
16	158.00	10.82	0.68	8.54

Table 1. Performance metrics for the shared memory implementation.

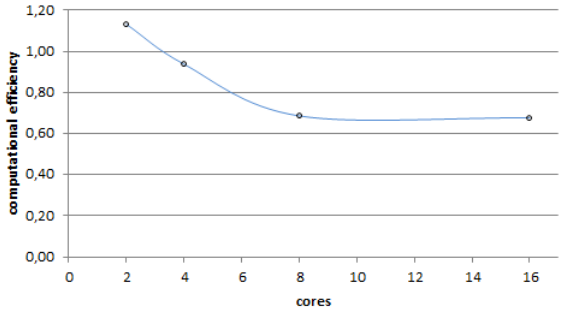


(a) Execution time

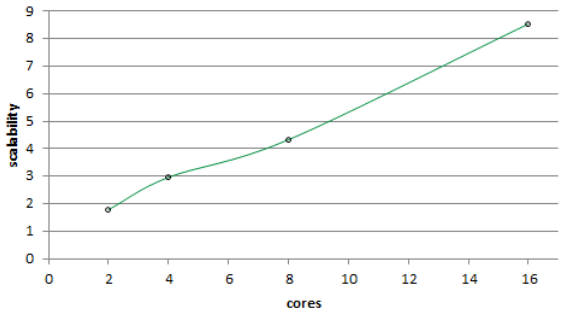


(b) Speedup

Fig. 2. Execution time and speedup for the shared memory implementation.



(a) Efficiency



(b) Scalability

Fig. 3. Efficiency and scalability for the shared memory implementation.

The results in Table 1 indicate that acceptable improvements in the execution times are obtained when using several cores in the shared memory parallel implementation of the conversion algorithm. Fig. 2(a) graphically shows the reduction in the execution times. While the sequential algorithm takes 1710 minutes to perform and the parallel version running on a single core takes 1349 minutes, the execution time is reduced down to 158 minutes when using the maximum number of cores available in the execution platform (16). Fig. 2(b) indicates that when using up to four cores, the parallel algorithm comes to obtain a linear speedup, and even superlinear speedup when two cores are used, mainly due to the time improvements in the image loading process. Using more cores yields to a sublinear speedup behavior, and both the computational efficiency and the scalability of the algorithm reduces, as shown in Fig. 3. Several factors can be mentioned as possible explanations for this behavior, including the overhead for the thread management, the simultaneous bus access, and also that the infrastructure used has four cores per processor, so the use of the memory bus for accessing the images have a larger impact when using more than four cores.

Overall, efficient results are obtained for the shared memory implementation of the parallel conversion process. Speedup values up to 10.82 are obtained when using 16 cores, and the efficiency values are almost linear.

Distributed memory implementation The experimental analysis for the distributed memory implementation was performed on a cluster infrastructure, varying the number of hosts and also the number of cores used in each host. Table 2 reports the execution time results and the performance metrics for the for the distributed memory implementation, and Fig. 4 and 5 graphically summarize the results.

# host	**# cores** (m)	**time** (minutes)	**speedup** (S_m)	**efficiency** (e_m)	**scalability** (Sc_m)
1	1	1313,00	1,30	1,30	1,00
	2	783,00	2,18	1,09	1,68
	4	421,33	4,06	1,01	3,12
	8	260,67	6,56	0,82	5,04
	16	158,67	10,78	0,67	8,28
2	2	805.00	2.12	1.06	1.63
	4	408.66	4.18	1.05	3.21
	8	188.33	9.08	1.13	6.97
	16	138.00	2.39	0.77	9.51
4	4	356.67	4.79	1.20	3.68
	8	192.33	8.89	1.11	6.83
	16	110.00	15.55	0.97	11.94

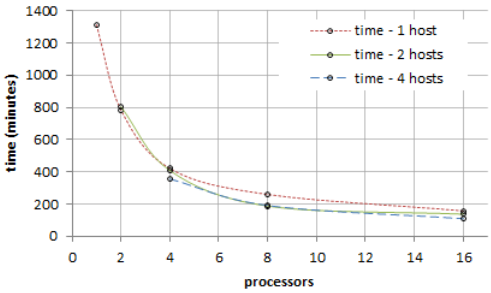
Table 2. Performance metrics for the distributed memory implementation.

The results in Table 2 indicate that notable improvements in the execution times are obtained by the distributed memory parallel implementation of the conversion algorithm, specially when distributing the processing in several hosts.

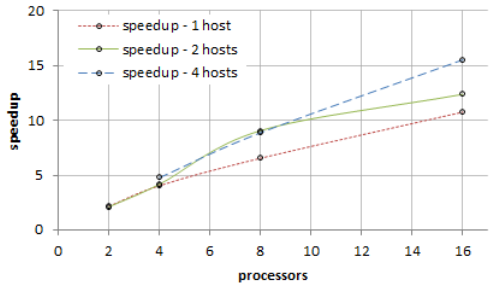
The sequential algorithm takes 1710 minutes to execute in the cluster infrastructure. When using one host, the best performance was reached when using 16 cores, with an execution time of 159 minutes. Using up to four cores, the algorithm comes to obtain a linear speedup as shown in Fig. 4(b). Just like for the sared memory implementation, using more than four cores yields to a sublinear speedup behavior, and both the computational efficiency and scalability of the algorithm reduces. Better results are obtained when distributing the processing in two hosts. The execution time is reduced to 138 minutes when using 16 cores. In this case, the linear speedup/computational efficiency behavior holds up to the use of eight cores, as shown in Figures 4(b), 5(a).

Finally, when using four hosts, the best results of the analysis are obtained. The execution time using 16 cores distributed in four hosts is 110 minutes, almost 16 times faster than the sequential one. The linear speedup/efficiency behavior holds with up 16 cores The scalability of the algorithm also increases up to a value of 11.94.

The previously presented results show that when using the distributed memory algorithm, the better approach is to distribute the execution across several hosts, rather than using additional cores into a single host. The possible reason for this behavior is that when more hosts are involved, more resources are available for the distributed image processing (i.e. memory, disk access, number of open files per process). This is consistent with the large number of files and memory used by the algorithm.



(a) Execution time



(b) Speedup

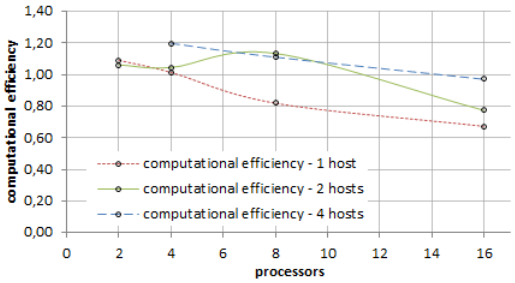
Fig. 4. Execution time and speedup for the distributed memory implementation.

Comparison: shared memory vs. distributed memory. By comparing the two implemented algorithms, the main conclusion is that the distributed memory version reaches better performance in the four hosts scenario. As it was already commented, this situation occurs due to the use of distributed resources, which improves the efficiency since there is a minimal communication between the master and the slaves, and no data exchange is performed between slaves. At the same time, by distributing the resources utilization, the waiting time for input/output and the time added by the operating system tasks do not impact significantly in the efficiency. The experimental results suggest that even better performance could be obtained when using an increasing number of hosts.

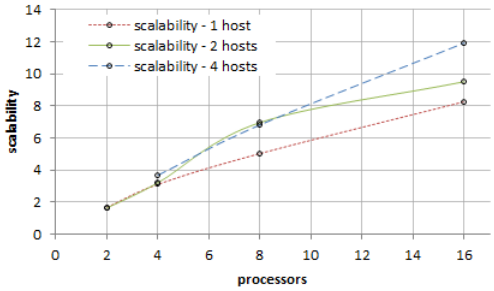
6 Conclusions and future work

This article presented an efficient algorithm for converting the full domain of land cover satellite images to the binary files within the WRF model. The proposed method is an important contribution, as it helps to efficiently generate better wind forecasts.

The implemented parallel algorithm has already been used to generate updated binary WRF files, an the results are now used for wind forecasting at wind farm Emanuelle Cambilargiu, Uruguay. In addition, the binaries for full world are published, and the outcome of this research is currently under examination by experts from NCAR (National Center for Atmospheric Research, USA) to be included in future releases of WRF.



(a) Efficiency



(b) Scalability

Fig. 5. Efficiency and scalability for the distributed memory implementation.

The parallel implementation of the conversion process provided an accurate and efficient method to perform the image processing. Two implementations, using shared and distributed memory, were implemented and analyzed. Regarding to the performance results, both algorithms allowed to obtain significant reductions in the execution times when compared with a traditional sequential implementation. The shared memory implementation did not scale well when more than four cores are used within the same host. On the other hand, the distributed memory algorithm have the best efficiency results: while the sequential algorithm took about 28 hours to perform the conversion, the distributed memory algorithm executing on 4 hosts takes 110 minutes to complete the process. A linear speedup behavior was detected for the distributed memory algorithm, and speedup values of up to **15.55** were achieved when using 16 cores distributed on four hosts. The computational efficiency is almost one, meaning that we are in presence of an almost-ideal case of performance improvement. The values of the scalability metric indicate that the distributed memory implementation of the proposed parallel algorithm scales well when more hosts are used, mainly due to the minimal need of communications between the master and slave processes. The presented results suggest that adding more hosts would improve even more the efficiency, and it also allow to tackle more complex image processing problems.

The main lines for future work are related with further improving the computational efficiency of the proposed implementations and studying the capability of tackling more complex versions of the satellite image processing problem. Regarding the first line, specific details about input/output performance and data bus access should be studied, in order to overcome the efficiency loss of the shared memory implementation. In addition, the scalability of the distributed memory implementation should be further analyzed, specially to determine the best approach for tackling more complex image processing problems (e.g. with better spatial and time resolution), eventually by using the computer power available in large distributed computing infrastructures, such as grid computing and volunteer-computing platforms.

7 Acknowledgments

The work of S. Nesmachnow, A. Gutierrez, and G. Cazes was partially funded by ANII and PEDECIBA, Uruguay.

References

1. J. Cornett and D. Randerson. *Mesoscale wind prediction with inertial equation of motion*. Number 48 in NOAA technical memorandum ERL ARL ;. Air Resources Laboratory, Environmental Research Laboratories, Las Vegas, 1975.
2. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with Message-Passing Interface*. MIT Press, 1999.
3. U. Maulik and A. Sarkar. Efficient parallel algorithm for pixel classification in remote sensing imagery. *Geoinformatica*, 16(2):391–407, April 2012.
4. C. Monteiro, R. Bessa, V. Miranda, A. Botterud, J. Wang, and G. Conzelmann. Wind power forecasting : state-of-the-art 2009. *Information Sciences*, page 216, 2009.
5. Akihiro Nakamura, Jong Geol Park, Kotaro Matsushita, Kenneth J. Mackin, and Eiji Nunohiro. Development and evaluation of satellite image data analysis infrastructure. *Artif. Life Robot.*, 16(4):511–513, February 2012.
6. Antonio Plaza, Javier Plaza, and David Valencia. Impact of platform heterogeneity on the design of parallel algorithms for morphological processing of high-dimensional image data. *J. Supercomput.*, 40(1):81–107, April 2007.
7. Antonio J. Plaza. Parallel techniques for information extraction from hyperspectral imagery using heterogeneous networks of workstations. *J. Parallel Distrib. Comput.*, 68(1):93–111, January 2008.
8. R. Rozumalski. *WRF Environmental Modeling System - User Guide*. National Weather Service SOO Science and Training Resource Center, release 2.1.2.2 edition, May 2006.
9. R. Sadykhov, A. Dorogush, Y. Pushkin, L. Podenok, and V. Ganchenko. Multispectral satellite images processing for forests and wetland regions monitoring using parallel mpi implementation. *Envisat Symposium*, pages 1–6, 2010.

HPCLatAm 2012, pp. 65-80 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

Facial Recognition Using Neural Networks over GPGPU

Juan Pablo Balarini, Martín Rodríguez, and Sergio Nesmachnow

Centro de Cálculo, Facultad de Ingeniería
Universidad de la República, Uruguay
{jbala87, martinr87}@gmail.com, sergion@fing.edu.uy

Abstract. This article introduces a parallel neural network approach implemented over Graphic Processing Units (GPU) to solve a facial recognition problem, which consists in deciding where the face of a person in a certain image is pointing. The proposed method uses the parallel capabilities of GPU in order to train and evaluate a neural network used to solve the abovementioned problem. The experimental evaluation demonstrates that a significant reduction on computing times can be obtained allowing solving large instances in reasonable time. Speedup greater than 8 is achieved when contrasted with a sequential implementation and classification rate superior to 85 % is also obtained.

Keywords: Face recognition, Neural Networks, Parallel Computing, GPGPU.

1 Introduction

Face recognition can be described as the ability to recognize people given some set of facial characteristics. Nowadays, it has become a popular area of research in computer vision and image analysis, mainly because we can find such recognition systems in objects of everyday life such as cellphones, security systems, laptops, PCs, etc. [21,22]. Another key element is that the high computing power now available makes these image recognition systems possible.

Using an image of a human face, an algorithm is proposed to evaluate and decide where that face is pointing. Each image is classified into one of four classes according to the direction where is facing (those classes are: left, right, up and straight).

For certain types of problems, artificial neural networks (ANN) have proven to be one of the most effective learning methods [2], built of complex webs of interconnected neurons, where each unit takes a number of real-valued inputs and produces a single real-valued output. Also the Backpropagation algorithm is the most commonly used ANN learning technique, which is appropriate for problems where the target function to be learned is defined over instances that can be described by a vector of predefined features (such as pixel values), also the target function output may be discrete-valued, real-valued, or a vector of several real or discrete-valued attributes. Additionally the training examples may contain errors, and fast evaluation of the learned function may be required. All this makes ANN a good option in image recognition problems. A survey of practical applications of ANNs can be found on [14].

One of the main inconvenient of ANNs is the time needed to perform the training phase, which generally is quite high for complex problems. As the number of hidden layers and neurons grows, the required time for the learning process of the ANN and for the evaluation of a new instance grows exponentially. On the other hand, the rate of successful classification of new instances increases as well. Note that generally, the more training examples the network is provided, the more effective it will be (and the longer it will take too). Therefore it is of special interest to perform training with a large number of neurons in the hidden layer and with a significant number of training examples, but with a relatively low training time.

It is interesting to note that every neuron in each layer can make their calculations independently of others in the same layer. This means that for any given layer, parallel computations can take place, and some parallel architecture could be used to take advantage of this.

Promising [20] work is being made in the area of general purpose GPU computing, principally in problems with parallel nature. GPU implementations allow obtaining significant reduction in the execution times of complex problems when compared with traditional sequential implementations on CPU [9]. Despite the fact GPUs were originally designed for the sole purpose of rendering computer graphics they have evolved into a general purpose computing platform with enough power and flexibility to make many computationally-intensive application perform better than on a CPU [12,13]. This can be explained by the significant disparity between CPUs and GPUs which rises every year. In this work, we propose an algorithm that takes advantage of this parallel architecture to obtain an algorithm that outperforms another that uses a sequential implementation.

This work focuses in the field of machine learning, high performance parallel computing, and using graphics processing units for general purpose computing, as it develops an algorithm that significantly improves the ANN training and classification time, as contrasted with a sequential algorithm.

The main contributions of this article are that a parallel face recognition algorithm is obtained that develops good classification rates in reasonable execution times, also this algorithm can be easily modified to recognize other features of a human face without changing those execution times. Furthermore, it demonstrates how the GPGPU platform is a very good option when it is desired to improve the execution time of a certain problem.

The rest of the paper is organized as follows. Section 2 introduces GPU computing and the CUDA programming model, section 3 presents a conceptual framework then, section 4 presents related work. Section 5 introduces the proposed solution and provides implementation details. In section 6 an experimental analysis is made and finally, section 7 presents the work conclusions and future work.

HPCLatAm 2012, pp. 65-80 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

2 GPU Computing

GPUs were originally designed to exclusively perform the graphic processing in computers, allowing the Central Process Unit (CPU) to concentrate in the remaining computations. Nowadays, GPUs have a considerably large computing power, provided by hundreds of processing units with reasonable fast clock frequencies. In the last ten years, GPUs have been used as a powerful parallel hardware architecture to achieve efficiency in the execution of applications.

2.1 GPU Programming and CUDA

Ten years ago, when GPUs were first used to perform general-purpose computation, they were programmed using low-level mechanism such as the interruption services of the BIOS, or by using graphic APIs such as OpenGL and DirectX. Later, the programs for GPU were developed in assembly language for each card model, and they had very limited portability. So, high-level languages were developed to fully exploit the capabilities of the GPUs. In 2007, NVIDIA introduced CUDA [11], a software architecture for managing the GPU as a parallel computing device without requiring mapping the data and the computation into a graphic API.

CUDA is based in an extension of the C language, and it is available for graphic cards GeForce 8 Series and superior. Three software layers are used in CUDA to communicate with the GPU (see Fig. 1): a low-level hardware driver that performs the CPU-GPU data communications, a high-level API, and a set of libraries such as CUBLAS for linear algebra and CUFFT for Fourier transforms.

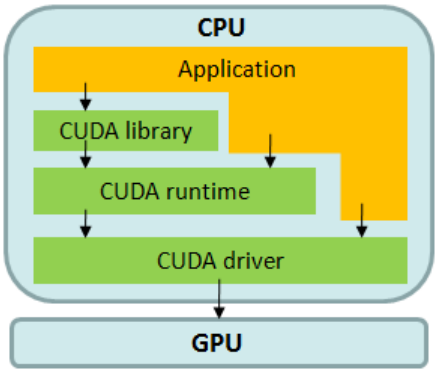


Fig. 1. CUDA *Architecture*

For the CUDA programmer, the GPU is a computing device able to execute a large number of threads in parallel. A procedure to be executed many times over different data can be isolated in a GPU-function using many execution threads. The function is compiled using a specific set of instructions and the resulting program (*kernel*) is loaded in the GPU. The GPU has its own DRAM, and the data are copied from it to the RAM of the host (and vice versa) using optimized calls to the CUDA API.

The CUDA architecture is built around a scalable array of multiprocessors, each one with eight scalar processors, one multithreading unit, and a shared memory chip. The multiprocessors are able to create, manage, and execute parallel threads, with reduced overhead. The threads are grouped in *blocks* (with up to 512 threads), which are executed in a single multiprocessor, and the blocks are grouped in *grids*. When a CUDA program calls a grid to be executed in the GPU, each one of the blocks in the grid is numbered and distributed to an available multiprocessor. The multiprocessor receives a block and splits the threads in *warps*, a set of 32 consecutive threads. Each warp executes a single instruction at a time, so the best efficiency is achieved when the 32 threads in the warp executes the same instruction. Each time that a block finishes its execution, a new block is assigned to the available multiprocessor.

The threads access the data using three memory spaces: a *shared memory* used by the threads in the block; the *local memory* of the thread; and the *global memory* of the GPU. Minimizing the access to the slower memory spaces (the local memory of the thread and the global memory of the GPU) is a very important feature to achieve efficiency. On the other side, the shared memory is placed within the GPU chip, thus it provides a faster way to store the data.

3 Face Recognition Using Artificial Neural Networks in GPU

3.1 Face Pointing Direction

The face pointing direction problem consists in recognizing where a human face is pointing (up, left, right and center) in a certain image. This problem has many practical applications such as detecting where a driver is looking while driving (raising an alarm if the driver fell asleep), a computer mouse for impaired people that moves according to head movements (i.e. face direction), a digital camera software that only takes a picture if all individuals are looking at the camera, etc. Traditional methods to solve this problem include ANNs [17, 2], evolutionary algorithms [15, 16], problem specific heuristics, etc., but in general, sequential implementations are used.

3.2 Artificial Neural Networks

ANNs provide a general practical method for learning real-valued, discrete-valued, and vector-valued functions from examples. Several algorithms (such as *backpropagation*) can be used to tune network parameters to best fit a training set of input-output pairs. ANNs are robust to errors in the training data and has been successfully applied to problems such as image recognition, speech recognition, and learning robot control strategies [2].

Figure 2 presents the general schema of an ANN. There is a set of *neurons* connected with each other. Each neuron receives several input data, perform a linear combination (result a) and then produces the result of the neuron, which is the evaluation of some function $f(x)$ for the value $x = a$.

HPCLatAm 2012, pp. 65-80 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

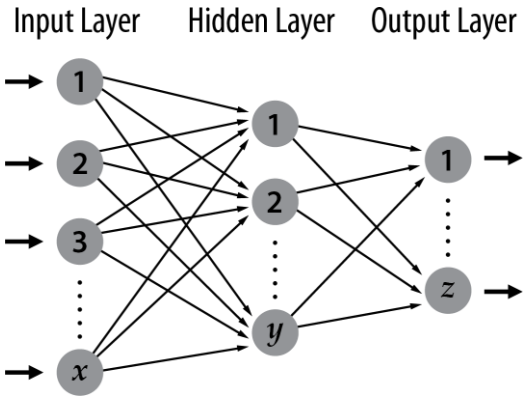


Fig. 2. Schema of an ANN.

The neurons are grouped in several layers:

- Input layer: receives the problem input
- Hidden layer/s: receives data from other neurons (typically from input layer or from another hidden layer), and forwards the processed data to the next layer. In an ANN, there may be multiple hidden layers with multiple neurons each.
- Output layer: this layer may contain multiple neurons and it determines the output of the processing for a certain problem instance.

Fig. 3 shows a schema for a neuron. First, a linear combination of the neuron input data (x_i, weights w_i, $i \in [1, n]$, and an independent coefficient w_0) is made. Then the output is evaluated at some well-known *activation function*, to produce the neuron output.

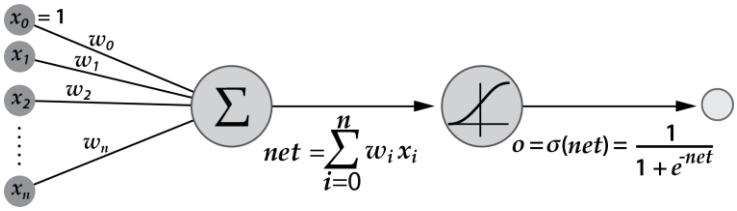


Fig. 3. Schema of a single neuron in an ANN.

3.3 Face Recognition Using GPGPU

In this article, an ANN is used to solve the face recognition problem, trained with the backpropagation algorithm. Backpropagation learns the weights for a multilayer network with a fixed set of units and interconnections, by applying the gradient descent method to minimize the squared error between the network output values and the target values for these outputs. The learning problem faced by backpropagation implies searching in a large space defined by all possible weight values for all neurons. The backpropagation method applied in this work (stochastic gradient descent version for feedforward networks) is described in Algorithm 1.

HPCLatAm 2012, pp. 65-80 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

Backpropagation (training_examples, n, n_{in}, n_{out}, n_{hidden})

Each training example is a pair of the form$\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network of input values, and $\vec{t}$ is the vector of target network output values.
n is the learning rate, n_{in} is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units.
The input unit from unit i into unit j is denoted x_{ji} , and the weights from unit i to unit j is denoted w_{ji}.

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- initialize all network weights to small random numbers
- while the termination condition is not met, do

for each $\langle \vec{x}, \vec{t} \rangle$ in training_examples,do
{propagate the input forward through the network}
input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network
{propagate the errors backward through the network}
for each network output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

for each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh} \delta_k$$

update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}, \text{ where } \Delta w_{ji} = n \delta_j x_{ji}$$

Algorithm 1. Stochastic gradient descent version of the backpropagation algorithm for feed-forward networks.

Algorithm 1 begins by constructing a network with the desired number of hidden and output units and initializing all network weights to small random numbers. Given this fixed network structure, the main loop of the algorithm iterates over the training examples. For each training example, it applies the network to the example, computes the gradient with respect to the error on this example, and then updates all weights in the network. This gradient step is iterated (using the same training examples multiple times) until the network performs acceptably well [2]. For evaluating a single instance (not to train the network), only the propagation of the input data through the network is made. The presented ANN uses neurons of sigmoid type with activation function:

$$F(x) = \frac{1}{1+e^{-x}} \quad (1)$$

To solve the face recognition problem in GPU, a specific version of the backpropagation algorithm was implemented. The generic schema in Algorithm 1 was adapted to execute on a GPU, mainly taking into account the communication restrictions between the GPU processing units. Before calling a function that runs on GPU a function-dependent domain decomposition is applied. In general, certain GPU threads are assigned to execute over certain neurons on the ANN. The domain decompositions are always performed to maximize the parallel execution, (i.e. each GPU thread can work independent from each other), and to avoid serializations in the memory access. A detailed description of domain decomposition is presented in section 5.2.

Figure 4 presents a schema of the ANN training, showing that the train() function is a concatenation of functions that execute in parallel.

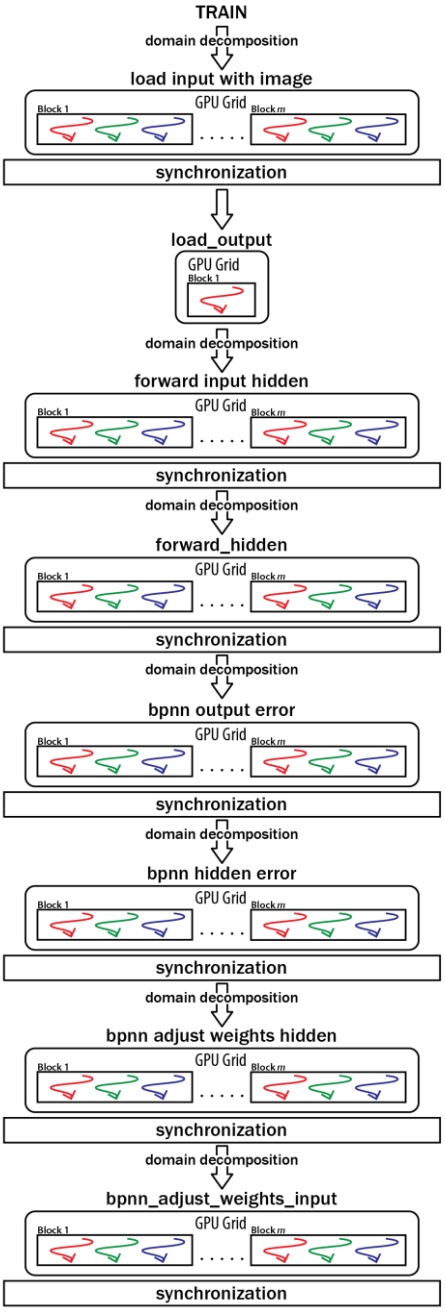


Fig. 4. ANN training: parallel approach.

4 Related Work

Lopes and Ribeiro [9] presented an analysis of an ANN implementation executing on GPU, showing how the training and classification times can be reduced significantly (ranging from 5× to 150×, depending on the complexity of the problem). They also conclude that the GPU scales better than the CPU when handling large datasets and complex problems. In this work the authors recognizes two sources of parallelism: the outputs of the neurons can be computed in parallel, and all the samples (patterns) on a dataset can be processed independently. The parallel ANN takes advantage of parallelism in the three training phases (forward, robust learning and backpropagation). Several problems were tackled in the experimental analysis, including solving $f(x) = \sin(x)/x$, and several classification and detection problems such as: the two-spirals problem, the sonar problem, the covertype problem, the poker hand problem, the ventricular arrhythmias problem and face recognition of the Yale face database [18], containing 165 gray scale faces images of 64×64 pixels of 15 individuals.

Jang et al. [6] introduced a parallel ANN implementation using CUDA applied to text recognition on images. Processing times up to 5 times faster than a CPU implementation were obtained. In this work, parallelism is achieved through computing in parallel all the linear combinations made when some neuron calculates their output, and also when the sigmoid function is computed on each neuron. In this case, text detection was performed over three image sizes (320×240, 571×785 and 1152×15466), always using 30 neurons on the hidden layer.

Solving a similar problem, Izotov et al. [7], used an ANN-based algorithm to recognize handwritten digits, using a CUDA-based implementation. The training time improvements were about 6 times less than an algorithm executed on CPU, and on instance classification there were reductions of about 9 times in execution time. This work represented some ANN features as matrixes and took advantage of the CUBLAS library (a linear algebra library over CUDA driver level) for calculating matrix multiplications (thus achieving parallelism) using 8-bit gray scale 28×28 pixel image instances of handwritten digits from the public domain MNIST database [19].

Nasse et al. [8] solved the problem of locating the direction of a human face in space, using ANN on a GPU. Parallelization of the solutions was achieved through dividing the image into several rectangles, and computing each one in parallel. This implementation obtains classification values up to 11 times faster than an implementation which runs on CPU, and was trained using 6000 non-faces and 6000 faces with three different sizes (378×278, 640×480 and 800×600 pixels).

Shufelt and Mitchell [4] solved a similar problem to the one proposed in this work (deciding whether an image is of a certain person or not) by using a sequential method. Their proposal was the starting point for the parallel solution presented here.

The analysis of the related work allows concluding that ANN implementations that execute over a GPU can obtain very good results when contrasted with a CPU-only implementation. Taking this fact into account, our purpose here is to develop an ANN to recognize certain features of a human face in a short period of time. If training time can be reduced considerably by using parallel GPU infrastructures, the solution will overcome one of the main disadvantages of traditional ANN implementations.

5 Implementation Details

The proposed parallel implementation applies the ideas in the sequential algorithm by Shufelt and Mitchell [4] for recognizing if a given picture is of a certain person. Thus, the method has to be slightly modified to obtain the expected solution for the face recognition problem. Moreover, in the parallel implementation, the possibility of working with a second layer of hidden neurons was included.

5.1 Software modules

The proposed solution uses five modules, which implement all functions needed by the algorithm. First, facetrain.cu contains the main method and is the module that performs the calls to the other functions. Then, backprop.cu implements the ANN and all the auxiliary functions needed to work with it. The proper interaction between the ANN and the images is solved in imagenet.cu. The pgmimage.cu library is used to work with images in pgm format. Finally, constants.h contains the entire configuration for the correct execution of the algorithm.

Training is a key element in the algorithm. The diagram in Fig. 5 explains the required steps needed to train the ANN for the entire trainset.

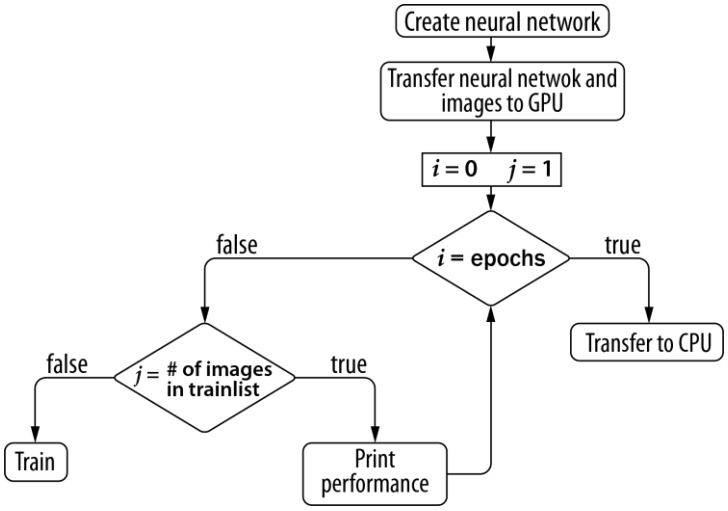


Fig. 5. ANN training: functional schema.

The GPU architecture is best exploited when performing training and classification. For example (see Fig. 6), when forwarding from input layer to hidden layer, the parallel algorithm creates as many blocks as neurons are in the hidden layer (each block will work with one neuron on the hidden layer), and each thread in a block will compute the linear combination of the weight that goes to that hidden neuron by the data that comes from the input layer (the algorithm works with the same number of threads per block as input neurons are). Similar levels of parallelism are achieved in the rest of the functions, obviously changing the role of each block and each thread in a block.

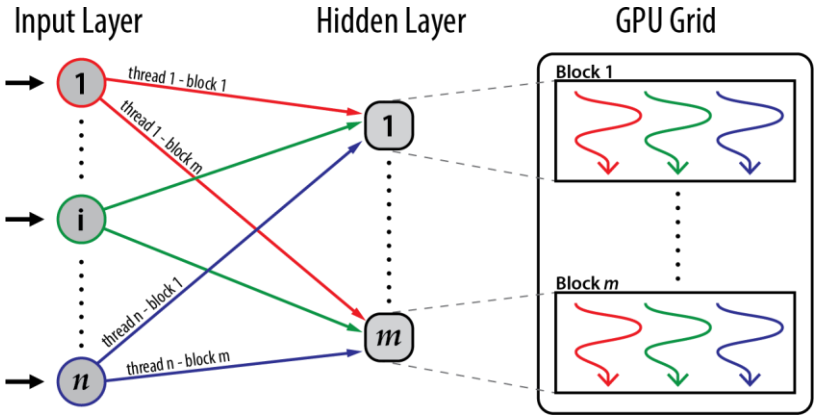


Fig. 6. Parallelism example: forwarding from input layer to hidden layer.

5.2 Tasks Executed on GPU

Key parameters such as the grid and block size used for the function invocations performed on GPU are of special interest for the overall performance of the algorithm. The following functions are called in both the training and evaluation tasks. The function *load_input_with_image()* is called with as many block as rows the image has, and as many threads per block as columns the image has. This function loads each image into the neurons of the input layer for later use (each block loads a row of the image). After that, *forward_input_hidden()* computes the outputs of the neurons in the hidden layer, using the data from the input layer. It is called with as many blocks as the number of hidden neurons in the network and as many threads per block as the GPU allows (1024 in our case). Each block computes the output of one neuron on the hidden layer, which is the linear combination performed by several threads (see Section 2). The function *forward_hidden()* works like the previous function, obtaining the output of the hidden layer. Next, *load_output()* loads the expected output of the ANN for a certain image. It is invoked with one block with one thread because of its simplicity.

The function *evaluate_performance()* computes the error of an image and checks if the output of the ANN matches the expected one. It is called with one block with one thread, due to the small processing performed. The functions *bpnn_output_error()* and *bpnn_hidden_error()* are used to compute the error in the output and hidden layer, respectively. The first one is called with one block and with as many threads per block as output neurons the ANN has. The second one with as many blocks as hidden neurons are and with as many threads per block as output neurons are. The function *bpnn_adjust_weights_hidden()* adjust the weights from the hidden layer to the output layer. It is called with as many blocks as output neurons are, and with as many threads per block as number of hidden neurons are plus one. For each block, it adjusts the weights that go from hidden neurons (as many as threads) to output neurons. Finally, *bpnn_adjust_weights_input()* adjusts weights that go from input layer to output layer. It works like the previous function, with the difference that the number of threads must be larger, due to the number of neurons the input layer has.

5.3 Other GPU considerations

Throughout the provided implementation, all constants have their type defined. This implementation decision was made because a precision loss was detected when performing conversions (e.g. double to float), affecting the numerical efficacy of the proposed algorithm. Since all GPU memory must be contiguous, static structures are used, because when transferring data from CPU to GPU the *cudaMemcpy* function copies only contiguous memory directions. Moreover, the CPU stack size had to be enlarged to 512 MB in order to allow storing 128×120 images or larger.

In addition, certain implementation decisions were taken to improve the final performance of the proposed algorithm. First, it was decided to use shared memory in certain GPU kernels, to hide the latency of global memory access. Another element to take into account is that most threads running on GPU performed several calculations, in order to avoid the limit for the number of threads per block in the execution platform (1024 in CUDA compute capabilities 2.0).

A weakness of the implementation is that heavily relies on the used hardware (especially with the compute capabilities of the graphics card). This will impact on the number of threads per block that can be created and in the use of certain CUDA functions (i.e. use of the *atomicAdd* function with data of float type).

6 Experimental Analysis

This section reports the results obtained when applying the parallel GPU implementations of the face pointing direction problem for a set of problem instances. A comparative analysis with a sequential implementation is performed, and the obtained speedups (the quotient between execution time in the sequential implementation and execution time in the parallel implementation) are also reported.

6.1 Development and Execution Platform

The GPU algorithm was developed on an AMD Athlon II X3 445 processor at 3.10 GHz, with 6 GB DDR3 RAM memory at 1333 MHz, a MSI 880GM-E41 motherboard, and a GeForce GTS 450 GPU with 1 GB of RAM.

The experimental analysis of the proposed algorithm executions was carried out on two platforms. Validation experiments using small-sized images were performed on the development platform, using a 500 GB SATA-2 disk with RAID 0 running Ubuntu Linux 11.10 64 bits Edition.

The limited computing power of the development platform did not allow taking full advantage of the parallel features proposed by the algorithm. Thus, a more comprehensive set of experiment including large images were performed in a more powerful platform, the *execution platform*, consisting in a Core i7-2600 processor at 3.40 GHz processor with 16 GB DDR3 RAM memory, with Fedora 15 64 bits Edition operating system, and a GeForce GTX 480 GPU with 1536 MB of RAM.

6.2 Problem Instances

Both sets of problem instances used for training and classification were obtained from the work by Shufelt and Mitchell [4]. These are images of different people in different poses. For each person and pose, the image comes in three sizes: 32×30 pixels, 64×60 and 128×120 pixels. There are about 620 images, which are divided in three sets, one for the network training and the other two to measure its effectiveness (trainlist and testlist, respectively). Also, a scaling of the images was performed in order to carry out executions with larger images to contrast these executions with the previous ones. The scaling was performed in two sizes: 256×240 and 512×480.

This variety of instance types (different image sizes) allows to take advantage of the GPU platform to the maximum because for instances of small size (i.e. 32×30 pixels, 64×60 pixels) both platforms perform similarly, while for images of larger sizes a clear difference between the two platforms begins to notice. This is mainly because the GPU platform has much more processing units than the CPU (yet at a lower speed), so if many calculations at once are required, there will be more available computing resources in GPU than in CPU.

6.3 Results and Discussion

To validate the algorithm it was considered that the rate of correctly classified images for new instances should be greater than 80% and the speedup gain over the sequential algorithm should be at least 2.

All values shown below correspond to algorithm executions with a training set consisting of 277 images (training set) and two test sets of 139 and 208 images respectively (train1 set and train2 set). In first instance, training over the ANN is made with every image in the training list, then performance is evaluated using images from test set 1 and 2 (this concludes a cycle). This is performed 100 times (100 epochs) to complete an execution cycle. The presented values correspond to the average of 50 execution cycles with an ANN with 100 neurons in the hidden layer.

Solution Quality

Since the proposed parallel implementation does not modify the algorithmic behavior of the sequential implementation, the obtained results with the GPU implementation are nearly the same than those obtained with the sequential version for all the studied instances. Table 2 and table 4 show correctly classified instances (in percentage) for both sequential and parallel algorithm and the learning rate and momentum constants that were used. Classification rates close to 80% are achieved for images of 512×480 pixels in both development and execution platform. For the development platform best results are obtained with images of 32×30 pixels where classification rate close to 93% is obtained as for the execution platform classification rate close to 92% is achieved for both 32×30 and 64×60 and 128×120 pixel images.

Execution Times

This subsection reports and discussed the execution time and performance results for the GPU implementation of the proposed algorithm. All execution times reported are the averages and its correspondent standard deviation values, computed in 50 independent execution of the parallel algorithm for each scenario.

Validation experiments on the development platform. Table 1 reports the execution times (in seconds) for the sequential and parallel implementations of the algorithm and the values of the speedup metric, in the development platform. Table 2 shows the classification rates obtained for both the sequential and parallel implementation, and the corresponding constants used for learning rate and momentum.

Table 1. Execution times (in seconds) in the development platform

image size	sequential	parallel	speedup
32×30	51.94 ± 0.57	56.22 ± 0.02	0.92
64×60	474.38 ± 18.90	147.54 ± 0.61	3.21
128×120	3665.99 ± 21.60	667.41 ± 2.26	5.49
256×240	16103.48 ± 80.41	3174.47 ± 1.84	5.07
512×480	67114.06 ± 132.66	14760.04 ± 1.55	4.54

Table 2. Correctly classified instances in the development platform

image size	sequential	parallel	learning rate	momentum
32×30	93.16%	92.91%	0.30	0.30
64×60	88.60%	88.83%	0.30	0.30
128×120	87.63%	86.79%	0.30	0.30
256×240	86.44%	86.56%	0.10	0.20
512×480	79.11%	79.80%	0.03	0.20

Experimental analysis on the execution platform. Table 3 reports the execution times for both sequential and parallel implementation, as well as the speedup obtained in experiments performed in the execution platform. Table 4 shows the classification rates obtained for each implementation and the corresponding constants used for learning rate and momentum.

Table 3. Execution times (in seconds) in the execution platform

image size	sequential	parallel	speedup
32×30	19.56 ± 0.24	34.29 ± 0.14	0.57
64×60	111.06 ± 2.94	61.69 ± 0.85	1.8
128×120	502.54 ± 7.95	224.09 ± 0.08	2.24
256×240	8633.87 ± 17.09	1019.13 ± 0.15	8.47
512×480	34540.95 ± 24.65	4777.90 ± 0.53	7.23

Table 4. Correctly classified instances in the execution platform

image size	sequential	sequential	learning rate	momentum
32×30	92.07%	91.96%	0.30	0.30
64×60	92.20%	91.84%	0.30	0.30
128×120	87.16%	91.35%	0.30	0.30
256×240	86.55%	86.55%	0.10	0.20
512×480	80.31%	78.63%	0.03	0.20

Speedup comparison. Fig. 7 summarizes the acceleration when using a GPU implementation, contrasted with using a sequential implementation in CPU, for the different image sizes and platforms used in the experimental analysis. The *speedup* evaluates the quotient between the execution time of the sequential implementation and the execution time in the parallel implementation in GPU.

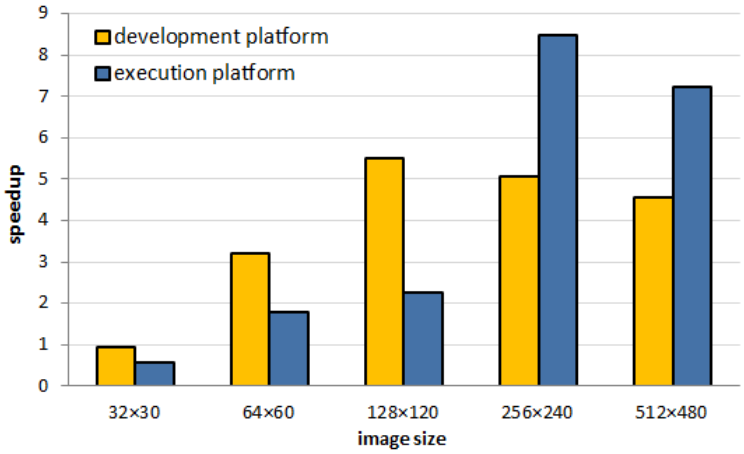


Fig. 7. Speedup comparison.

The speedup values in Fig. 7 indicate that the best acceleration is obtained for images with dimension 256×240 pixels, where the algorithm reaches the compute capabilities of the graphic card. The results in Tables 1 and 3 indicate that significant improvements on the execution times are obtained when using the parallel version of the algorithm with images of size 64×60 or larger. When solving images of size 32×30, the GPU implementation was unable to outperform the execution times of the CPU-only implementation, mainly due to the overhead introduced by thread creation and management and the use of the GPU memory. However, when solving larger problem instances, significant improvements in execution times are achieved, especially for images of size 256×240 pixels, where a speedup of 8.47 is obtained.

The previous results indicate that the parallel implementation of the face recognition algorithm executing on GPU provides significant reductions on the execution times over a traditional sequential implementation in CPU, especially when large images are processed.

HPCLatAm 2012, pp. 65-80 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

7 Conclusions and Future Work

ANNs have proven to be suitable for solving many real world problems. However, the large execution times required in the training phase sometimes exclude ANNs from being an option when using large datasets or when solving complex problems.

Nowadays, parallel computing on GPUs allows achieving important performance improvements over CPU implementations. In this article, a parallel GPU algorithm was proposed for solving the face recognition problem using ANNs.

The parallel GPU algorithm was designed and implemented to take advantage of the specific features of GPU infrastructures, in order to provide an accurate and efficient solution to both the training process using the well-known backpropagation algorithm, and the face recognition problem itself.

The overall parallel strategy used is based on many threads running on GPU, each one working with several neurons, and trying to maintain the threads as independent as possible. Every kernel function was designed to take advantage of the execution platform, optimized to obtain the best performance (e.g. some kernels are assigned to perform more than one neuron calculations to avoid the overhead of thread creation). Also, shared memory was exploited in order to avoid global memory access latency.

The experimental analysis demonstrates that the parallel algorithm in GPU allowed obtaining significant improvements in the execution times when compared with a traditional sequential implementation. Speedup values up to **8.47** were obtained when solving problem instances with images of 256×240 pixels, and 7.23 for images of 512×480 pixels. These results confirm that to take advantage of the GPU computing power, the algorithm should be used to process images of considerable sizes.

The main contributions of this article include a parallel face recognition algorithm in GPU that is able to obtain accurate classification rates in reasonable execution times. The algorithm can be easily modified to recognize other features of a human face, without significant changes in the expected execution times.

The research reported in this article demonstrates that the GPGPU platform is a very good option to speed up the resolution of complex problems. Furthermore, the results indicate how the growing technological evolution of graphic cards helps to tackle more complex classification problems using ANNs, which can be solved accurately and in reduced execution times.

The main lines for future work include further improving the computational efficiency of the presented algorithm and tackling other classification/image processing problems using ANNs implemented on GPU. Regarding the first line of work, improved execution time results can be obtaining by adjusting the parameters of each kernel invocation, to avoid problems such as thread divergence or better use of the GPU resources (i.e. shared memory) for larger images. Also, some algorithm constants (such as *momentum* and *learning rate*) could be auto-tuned by the algorithm to obtain the best classification rates as possible. Regarding the second line, it will be of special interest to implement GPU algorithms to recognize generic features of people images, such as skin color, if it has sunglasses or not, etc., configurable at run time.

References

1. Kyong, K. and Jung, K.: GPU Implementation of Neural Network. Pattern Recognition, vol. 37, no. 6, pp. 1311-1314. Pergamon (2004)
2. Mitchell, Tom: Machine Learning. McGraw Hill (1997)
3. Bishop, Christopher M.: Pattern Recognition and Machine Learning. Springer (2006)
4. Mitchell, T. and Shufelt, J.: Neural Networks for Face Recognition, http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ ftp/faces.html. Accessed June 2012.
5. Neural Network on GPU, http://www.codeproject.com/Articles/24361/A-Neural-Network-on-GPU Accessed June 2012
6. Jang, H., Park, A. and Jung, K.: Neural Network Implementation Using CUDA and OpenMP, Proc. of Computing: Techniques and Applications, pp.155-161. IEEE (2008)
7. Izotov, P., Kazanskiy, N., Golovashkin, D. and Sukhanov, S.: CUDA-enabled implementation of a neural network algorithm for handwritten digit recognition, Optical Memory & Neural Networks, vol. 20, no. 2, pp.98-106. Allerton Press, Inc (2011)
8. Nasse, F., Thurau, C. and Fink, G.: Face Detection Using GPU-Based Convolutional Neural Networks, Proc. of Computer Analysis of Images and Patterns, pp. 83-90. Springer Berlin (2009)
9. Lopes, N. and Ribeiro, B.: An Evaluation of Multiple Feed-Forward Networks on GPUs, International Journal of Neural Systems, vol. 21, no. 1, pp. 31-47. World Scientific Publishing Company (2011)
10. LeCun, Y., Bottou, L., Orr, G. and Muller, K.: Efficient Backprop in Neural Networks-Tricks of the Trade, Springer Lecture Notes in Computer Sciences, vol. 1524, pp. 5-50. Springer (1998)
11. NVIDIA. CUDA C Programming Guide Version 4.1, http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf. Accesed June 2012
12. Steinkrau, D., Simard, P. and Buck, I.: Using GPUs for machine learning algorithms, Proc. of 8th Int. Conf. on Document Analysis and Recognition, pp. 1115–1119 (2005)
13. Catanzaro, B., Sundaram, N. and Keutzer, K.: Fast support vector machine training and classification on graphics processors, Proc. of 25th International Conference on Machine Learning, 2008, pp. 104–111. ACM (2008)
14. Rumelhart, D., Widrow, B. and Lehr, M.: The basic ideas in neural networks, Communications of the ACM, 37(3) pp. 87-92. ACM (1994)
15. Huang, S., Fu, L., Hsiao, P.: A framework for human pose estimation by integrating data-driven Markov chain Monte Carlo with multi-objective evolutionary algorithm, Proc. of Int. Conf. on Robotics and Automation, pp. 3748–3753 (2006)
16. Murphy-Chutorian, E., Trivedi, M.: Head Pose Estimation in Computer Vision: A Survey, IEEE Trans. on Patt. Analysis and Machine Intelligence, 2009, pp. 607–626. IEEE (2009)
17. Bishop, Christoper M.: Neural Networks for Pattern Recognition. Clarendon Press, Oxford (1995)
18. Yale Face Database, cvc.yale.edu/projects/yalefaces/yalefaces.html. Accessed June 2012
19. LeCun, Y. and Cortes, C.: The MNIST Database of Handwritten Digits, MNIST Handwritten Digit Database, http://yann.lecun.com/exdb/mnist. Accessed June 2012
20. CUDA Spotlights, http://developer.nvidia.com/cuda-spotlights. Accessed June 2012
21. Ng, C., Savvides, M. and Khosla, P.: Real-time face verification system on a cell-phone using advanced correlation filters, Proc. of 4th IEEE Workshop on Automatic Identification Advanced Technologies, pp. 57–62. IEEE (2005)
22. Venkataramani, K., Qidwai, S. and Vijayakumar, B.: Face authentication from cell phone camera images with illumination and temporal variations, IEEE Trans. on Systems, Man, and Cybernetics, Part C, vol. 35, pp. 411 – 418. IEEE (2005)

HPCLatAm 2012, pp. 81-95 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

Parallel implementations of the MinMin heterogeneous computing scheduler in GPU

Mauro Canabé and Sergio Nesmachnow

Centro de Cálculo, Facultad de Ingeniería
Universidad de la República, Uruguay
{mcanabe,sergion}@fing.edu.uy

Abstract. This work presents parallel implementations of the MinMin scheduling heuristic for heterogeneous computing using Graphic Processing Units, in order to improve its computational efficiency. The experimental evaluation of the four proposed MinMin variants demonstrates that a significant reduction on the computing times can be attained, allowing to tackle large scheduling scenarios in reasonable execution times.

Keywords: GPU computing, heterogeneous computing, scheduling.

1 Introduction

In the last fifteen years, distributed computing environments have been increasingly used to solve complex problems. Nowadays, a common platform for distributed computing usually comprises a heterogeneous collection of computers. This class of infrastructures includes *grid computing* and *cloud computing* environments, where a large set of heterogeneous computers with diverse characteristics are combined to provide pervasive on demand and cost-effective processing power, software, and access to data, for solving many kinds of problems [7,18].

A key problem when using such heterogeneous computing (HC) environments consists in finding a scheduling strategy for a set of tasks to be executed. The goal is to assign the computing resources by satisfying some efficiency criteria, usually related to the total execution time or resource utilization [4,13]. The *heterogeneous computing scheduling problem* (HCSP) became specially important due to the popularization of heterogeneous distributed computing systems [5,8].

Traditional scheduling problems are NP-hard [9], thus classic exact methods are only useful for solving problem instances of very reduced size. Heuristics methods are able to get efficient schedules in reasonable times, but they still require long execution times when solving large instances of the scheduling problem. These execution times (i.e., in the order of an hour) can be extremely high for performing on-line scheduling in realistic HC infrastructures.

High performance computing techniques can be applied to reduce the execution times required to perform the scheduling. The massively parallel hardware in Graphic Processor Units (GPU) has been successfully applied to speed up the computations required to solve problems in many application areas [11], showing an excellent trade-off between cost and computing power [16].

HPCLatAm 2012, pp. 81-95 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

The main contribution of this work is the development of four parallel implementations on GPU for a the classic and effective scheduling heuristic MinMin [12]. The experimental evaluation of the proposed parallel methods demonstrates that a significant reduction on the computing times can be attained when using the parallel GPU hardware. This performance improvement allows solving large scheduling scenarios in reasonable execution times.

The manuscript is structured as follows. Next section introduces the HCSP mathematical formulation, and the heuristics studied in this work. A brief introduction to GPU computing is presented in Section 3. Section 4 describes the four proposed implementations of the MinMin heuristic on GPU. The experimental evaluation of the proposed methods is reported in Section 5, where the efficiency results are also analyzed. Finally, Section 6 summarizes the conclusions of the research and formulates the main lines for future work.

2 Heterogeneous computing scheduling

This section presents the HCSP and its mathematical formulation. It also provides a description of the class of list scheduling heuristics, and describes the MinMin method parallelized in this work.

2.1 HCSP formulation

An HC system is composed of many computers, also called *processors* or *machines*, and a set of tasks to be executed on the system. A task is the atomic unit of workload, so it cannot be divided into smaller chunks, nor interrupted after it is assigned to a machine. The execution times of any individual task vary from one machine to another, so there will be competition among tasks for using those machines able to execute them in the shortest time.

Scheduling problems mainly concern about time, trying to minimize the time spent to execute all tasks. The most usual metric to minimize in this model is the *makespan*, defined as the time spent from the moment when the first task begins execution to the moment when the last task is completed [13].

The following formulation presents the mathematical model for the HCSP aimed at minimizing the makespan:

- given an HC system composed of a set of machines $P = \{m_1, \ldots, m_M\}$ (dimension M), and a collection of tasks $T = \{t_1, \ldots, t_N\}$ (dimension N) to be executed on the system,
- let there be an *execution time function* $ET : T \times P \rightarrow \mathbf{R}^+$, where $ET(t_i, m_j)$ is the time required to execute the task t_i in the machine m_j,
- the goal of the HCSP is to find an assignment of tasks to machines (a function $f : T^N \rightarrow P^M$) which minimizes the *makespan*, defined in Equation 1.

$$\max_{m_j \in P} \sum_{\substack{t_i \in T: \\ f(t_i)=m_j}} ET(t_i, m_j) \tag{1}$$

In the previous HCSP formulation all tasks can be independently executed, disregarding the execution order. This kind of applications frequently appears in many lines of scientific research, specially in Single-Program Multiple-Data applications used for multimedia processing, data mining, parallel domain decomposition of numerical models for physical phenomena, etc. The independent tasks model also arises when different users submit their (obviously independent) tasks to execute in grid computing and volunteer-based computing infrastructures -such as TeraGrid, WLCG, Berkeley's BOINC, Xgrid, etc. [2]-, where non-dependent applications using domain decomposition are very often submitted for execution. Thus, the relevance of the HCSP version faced in this work is justified due to its significance in realistic distributed HC and grid environments.

2.2 List scheduling heuristics

The class of *list scheduling* heuristics comprises many deterministic scheduling methods that work by assigning priorities to tasks based on a particular criterion. After that, the list of tasks is sorted in decreasing priority and each task is assigned to a processor, regarding the task priority and the processor availability. Algorithm 1 presents the generic schema of a list scheduling method.

Algorithm 1 Schema of a list scheduling algorithm.

```
1: while tasks left to assign do
2:     determine the most suitable task according to the chosen criterion
3:     for each task to assign, each machine do
4:         evaluate criterion (task, machine)
5:     end for
6:     assign the selected task to the selected machine
7: end while
```

Since the pioneering work by Ibarra and Kim [10], where the first algorithms following the generic schema in Algorithm 1 were introduced, many list scheduling techniques have been proposed to provide easy methods for tasks-to-machines scheduling. This class of methods has also often been employed in hybrid algorithms, with the objective of improving the search of metaheuristic approaches for the HCSP and related scheduling problems.

The simplest list scheduling heuristics use a single criterion to perform the tasks-to-machines assignment. Among others, this category includes: *Minimum Execution Time* (MET), which considers the tasks sorted in an arbitrary order, and assigns them to the machine with lower ET for that task, regardless of the machine availability; *Opportunistic Load Balancing* (OLB), which considers the tasks sorted in an arbitrary order, and assigns them to the next machine that is expected to be available, regardless of the ET for each task on that machine; and *Minimum Completion Time* (MCT), which tries to combine the benefits of OLB and MET by considering the set of tasks sorted in an arbitrary order and assigning each task to the machine with the minimum ET for that task.

Trying to overcome the inefficacy of these simple heuristics, other methods with higher complexity have been proposed, by taking into account more complex and holistic criteria to perform the task mapping, and then reduce the makespan values. This work focuses on one of the most effective heuristics in this class:

– **MinMin**, which greedily picks the task that can be completed the soonest. The method starts with a set U of all *unmapped* tasks, calculates the MCT for each task in U for each machine, and assigns the task with the minimum overall MCT to the best machine. The mapped task is removed from U, and the process is repeated until all tasks are mapped. MinMin improves upon the MCT heuristic, since it does not consider a single task at a time but all the unmapped tasks sorted by MCT and by updating the machine availability for every assignment. This procedure leads to balanced schedules and also allows finding smaller makespan values than other heuristics, since more tasks are expected to be assigned to the machines that can complete them the earliest.

The computational complexity of MinMin heuristic is $O(N^3)$, where N is the number of tasks to schedule. When solving large instances of the HCSP, large execution times are required to perform the task-to-machine assignment (i.e. several minutes for a problem instance with 10.000 tasks). In this context, parallel computing techniques can be applied to reduce the execution times required to find the schedules.

GPU computing has been used to parallelize many algorithms in diverse research areas. However, to the best of our knowledge, there have been no previous proposals of applying GPU parallelism to list scheduling heuristics.

3 GPU computing

GPUs were originally designed to exclusively perform the graphic processing in computers, allowing the Central Process Unit (CPU) to concentrate in the remaining computations. Nowadays, GPUs have a considerably large computing power, provided by hundreds of processing units with reasonable fast clock frequencies. In the last ten years, GPUs have been used as a powerful parallel hardware architecture to achieve efficiency in the execution of applications.

GPU programming and CUDA. Ten years ago, when GPUs were first used to perform general-purpose computation, they were programmed using low-level mechanism such as the interruption services of the BIOS, or by using graphic APIs such as OpenGL and DirectX [6]. Later, the programs for GPU were developed in assembly language for each card model, and they had very limited portability. So, high-level languages were developed to fully exploit the capabilities of the GPUs. In 2007, NVIDIA introduced CUDA [15], a software architecture for managing the GPU as a parallel computing device without requiring to map the data and the computation into a graphic API.

CUDA is based in an extension of the C language, and it is available for graphic cards GeForce 8 Series and superior. Three software layers are used in CUDA to communicate with the GPU (see Fig. 1): a low-level hardware driver that performs the CPU-GPU data communications, a high-level API, and a set of libraries such as CUBLAS for linear algebra and CUFFT for Fourier transforms.

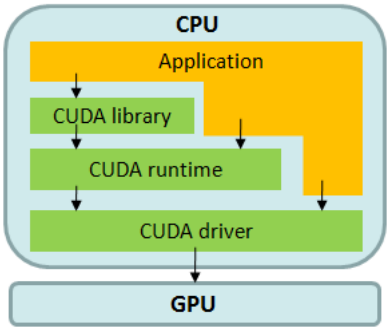


Fig. 1. CUDA architecture.

For the CUDA programmer, the GPU is a computing device which is able to execute a large number of threads in parallel. A specific procedure to be executed many times over different data can be isolated in a GPU-function using many execution threads. The function is compiled using a specific set of instructions and the resulting program (named *kernel*) is loaded in the GPU. The GPU has its own DRAM, and the data are copied from the DRAM of the GPU to the RAM of the host (and viceversa) using optimized calls to the CUDA API.

The CUDA architecture is built around a scalable array of multiprocessors, each one of them having eight scalar processors, one multithreading unit, and a shared memory chip. The multiprocessors are able to create, manage, and execute parallel threads, with reduced overhead. The threads are grouped in *blocks* (with up to 512 threads), which are executed in a single multiprocessor, and the blocks are grouped in *grids*. When a CUDA program calls a grid to be executed in the GPU, each one of the blocks in the grid is numbered and distributed to an available multiprocessor. When a multiprocessor receives a block to execute, it splits the threads in *warps*, a set of 32 consecutive threads. Each warp executes a single instruction at a time, so the best efficiency is achieved when the 32 threads in the warp executes the same instruction. Each time that a block finishes its execution, a new block is assigned to the available multiprocessor.

The threads access the data using three memory spaces: a *shared memory* used by the threads in the block; the *local memory* of the thread; and the *global memory* of the GPU. Minimizing the access to the slower memory spaces (the local memory of the thread and the global memory of the GPU) is a very important feature to achieve efficiency. On the other hand, the shared memory is placed within the GPU chip, thus it provides a faster way to store the data.

HPCLatAm 2012, pp. 81-95 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

4 MinMin implementations on GPU

The GPU architecture is better suited to the Single Instruction Multiple Data execution model for parallel programs. Thus, GPUs provide an ideal platform for executing parallel programs based on algorithms that use the domain decomposition strategy, especially when the algorithms execute the same set of instructions for each element of the domain.

The generic schema for a list scheduling heuristic presented in Algorithm 1 applies the following strategy: for each unassigneed task the criterial are evaluated on all machines and the task that best meets the criteria is selected and assigned to the machine which generates the minimum MCT. Clearly, this schema is an ideal case for applying a task-based or machine-based domain decomposition to generate parallel versions of the heuristics.

The four MinMin implementations on GPU designed in this work are based on the same generic parallel strategy. For each unassigned task, the evaluation of the criteria for all machines is performed in parallel on the GPU, building a vector that stores the identifier of the task, the best value obtained for the criteria, and the correspondent machine to get that value. The indicators in the vector are then processed in the reduction phase to obtain the best value that meets the criteria, and then the best pair (task, machine) is assigned. It is worth noting that the processing of the indicators to obtain the optimum value in each step is also performed using the GPU. A graphical summary of the generic parallel strategy applied in the parallel MinMin algorithms proposed in this article is presented in Fig. 2.

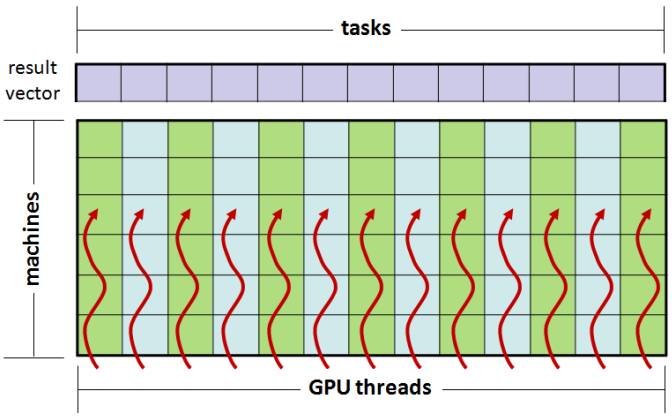


Fig. 2. Generic parallel strategy for MinMin on GPU.

Four variants of the proposed MinMin implementation in GPU were designed:

1. *Parallel MinMin using one GPU* (MinMin-1GPU), which executes on a single GPU, applying the aforementioned generic procedure;
2. *Parallel MinMin in four GPUs with domain decomposition using pthreads* (MinMin-4GPU-PT), which applies a master-slave multithreading programming approach implemented with posix threads (pthreads) that executes the same algorithm on four GPUs independently. The employed domain partition strategy splits the domain (i.e. the set of tasks) into N equally sized parts (being N the number of GPUs used, four in our case), so that each task belongs to only one subset. Thus, each GPU performs the MinMin algorithm on a subset of the tasks input data on all machines, and a master process consolidate the results after each GPU finishes its task;
3. *Parallel MinMin in four GPUs with domain decomposition using OpenMP* (MinMin-4GPU-OMP), which applies the same master-slave strategy than the previous variant, but the multithreading programming is implemented using OpenMP. The only difference between this implementation and the previous variant lies in how the threads are handled, in this case they are automatically managed and synchronized using OpenMP directives included in the implementation. The code for loading input data, dumping the resulting data, performing the domain partition, and implementing the GPU kernel are identical to the one used in MinMin-4GPU-PT;
4. *Parallel synchronous MinMin in four GPUs and CPU* (MinMin-4GPU-sync), which also applies a domain decomposition but it follows an hybrid approach. In each iteration, each GPU performs a single step of the MinMin algorithm, then a master process running in CPU assesses the result computed by each GPU and select the one that best meets the proposed criteria (i.e. MCT minimization), and finally the information of the selected assignment is updated in each GPU. This variant applies a multitheading approach implemented using pthreads to manage and synchronize the threads.

Figure 3 describes the parallel strategy used in the proposed implementations MinMin-4GPU-PT and MinMin-4GPU-OMP, where the CPU threads are defined and handled by using pthreads and OpenMP, respectively. Figure 4 describes the parallel strategy used in the synchronous implementation MinMin-4GPU-sync.

A specific data representation was used to accelerate the execution of the sequential implementation of the MinMin heuristic, in order to perform a fair comparison with the execution times of the GPU implementations. The sequential implementation use a data matrix where each row represents a task and each column represents a machine. Thus, when performing the processing for tasks (rows), the entries are loaded to the cache of the processing core, allowing a faster way to access the data.

HPCLatAm 2012, pp. 81-95 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

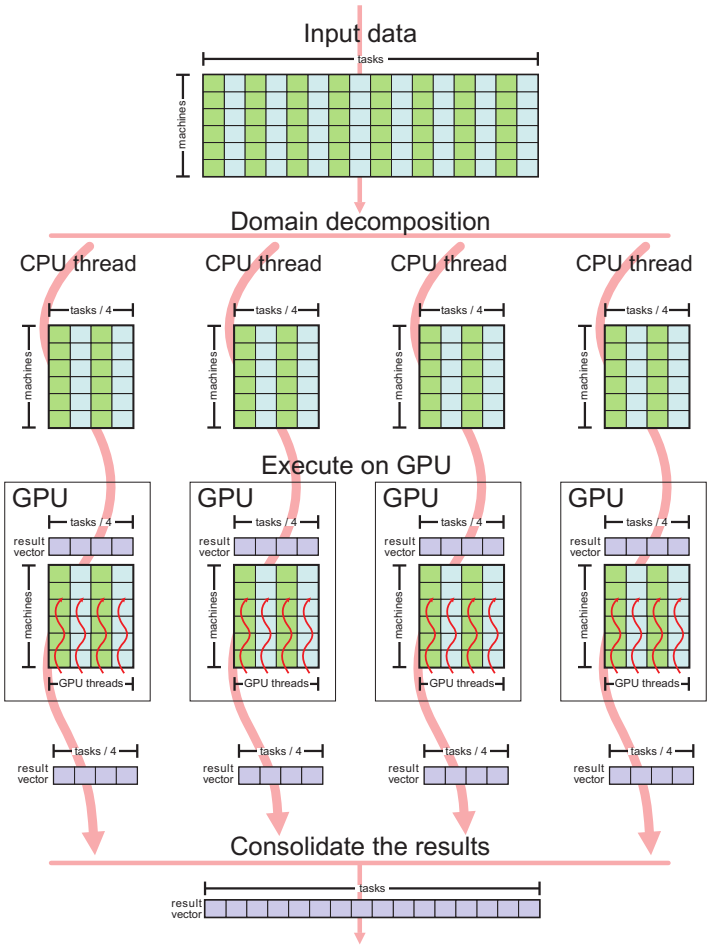


Fig. 3. Parallel strategy used in MinMin-4GPU-PT and MinMin-4GPU-OMP.

For parallel algorithms executing on GPU, loading the data matrix in the same way reduces the computational efficiency. Adjacent threads would access to the data stored in contiguous rows, but these are not stored contiguously, thus they cannot be stored in shared memory. When the data matrix is loaded so that each column represent a task and each row represent a machine, two adjacent threads in GPU access to the data stored in contiguous columns. These data are stored in contiguous memory locations, so they can be loaded in the shared memory, allowing to perform a faster data access for each thread, and therefore improving the execution of the parallel algorithm on GPU.

Preliminary experiments were also performed using a domain decomposition strategy that divides the data *by machines* rather than by tasks, but this option was finally discarded due to scalability issues as the problem size increases.

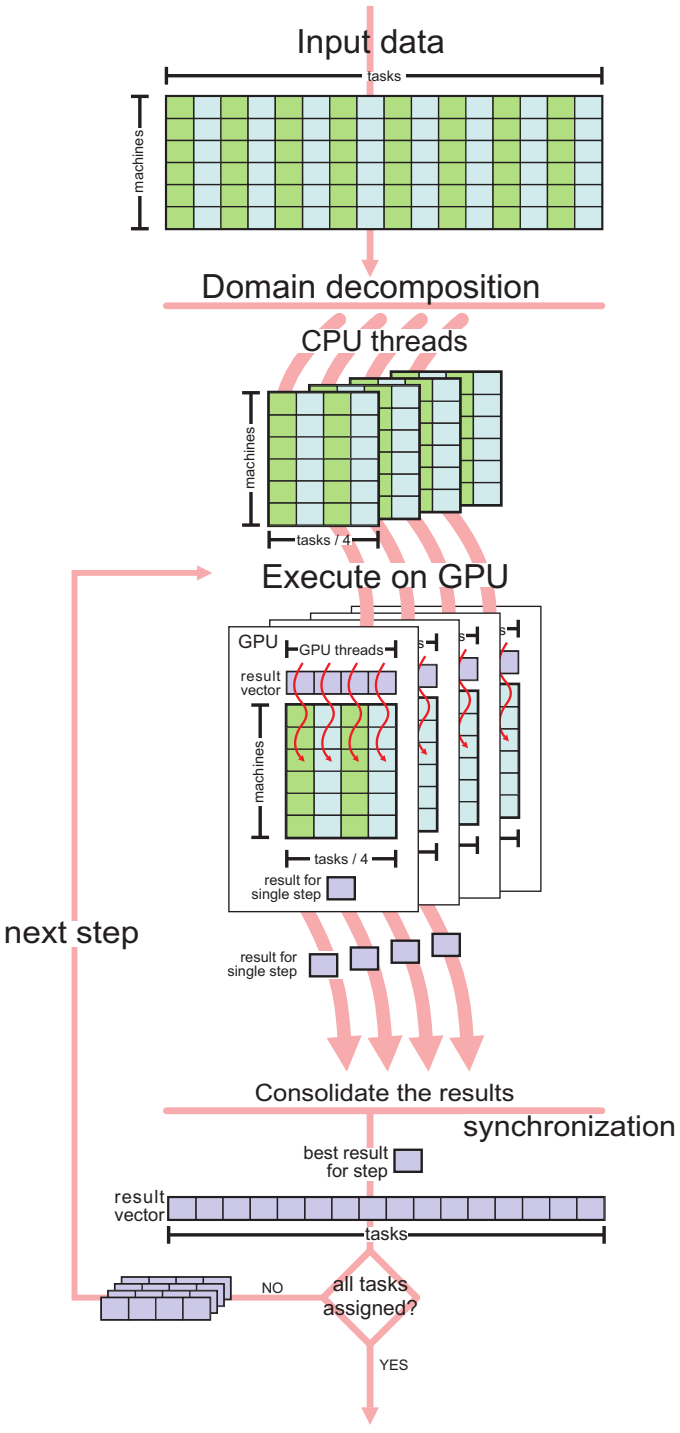


Fig. 4. Parallel strategy used in MinMin-4GPU-sync.

5 Experimental analysis

This section presents the experimental evaluation of the proposed MinMin implementations on GPU.

5.1 HCSP scenarios

No standardized benchmarks or test suites for the HCSP have been proposed in the related literature [17]. Researchers have often used the suite of twelve instances proposed by Braun et al. [3], following the expected time to compute (ETC) performance estimation model by Ali et al. [1].

ETC takes into account three key properties: machine heterogeneity, task heterogeneity, and consistency. *Machine heterogeneity* evaluates the variation of execution times for a given task across the HC resources, while *task heterogeneity* represents the variation of the tasks execution times for a given machine. Regarding the consistency property, in a *consistent* scenario, whenever a given machine m_j executes any task t_i faster than other machine m_k, then machine m_j executes all tasks faster than machine m_k. In an *inconsistent* scenario a given machine m_j may be faster than machine m_k when executing some tasks and slower for others. Finally, a *semi-consistent* scenario models those inconsistent systems that include a consistent subsystem.

For the purpose of studying the efficiency of the GPU implementations as the problem instances grow, the experimental analysis consider a test suite of large-dimension HCSP instances, randomly generated to test the scalability of the proposed methods. This test suite was designed following the methodology by Ali et al. [1]. The set includes the 96 medium-sized HCSP instances with dimension (tasks×machines) 1024×32, 2048×64, 4096×128 and 8192×256 previously solved using an evolutionary algorithm [14], and new large dimension HCSP instances with dimensions 16384×512, 32768×1024, 65536×2048, and 131072×4096, specifically created to evaluate the GPU implementations presented in this work.

These dimensions are significanlty larger than those of the popular benchmark by Braun et al. [3] and they better model present distributed HC and grid systems. The problem instances and the generator program are publicly available to download at `http://www.fing.edu.uy/inco/grupos/cecal/hpc/HCSP`.

5.2 Development and execution platform

The parallel MinMin heuristics were implemented in C, using the standard `stdlib` library. The experimental analysis was performed on a Dell PowerEdge (QuadCore Xeon E5530 at 2.4 GHz, 48 GB RAM, 8 MB cache), with CentOS Linux 5.4 and a NVidia Tesla C1060 GPU (240 cores at 1.33 GHz, 4GB RAM) from the Cluster FING infrastructure, Facultad de Ingeniería, Universidad de la República, Uruguay (cluster website `http://www.fing.edu.uy/cluster`).

5.3 Experimental results

This section reports the results obtained when applying the parallel GPU implementations of the MinMin list scheduling heuristic for the HSCP instances tackled in this article.

In the experimental evaluation, we study two specific aspects of the proposed parallel MinMin implementations on GPU:

- *Solution quality*: The proposed parallel implementations modify the algorithmic behavior of the MinMin heuristic, so the makespan results obtained with the GPU implementations are not the same than those obtained with the sequential versions for the studied HCSP instances. We evaluate the relative gap with respect to the traditional (sequential) MinMin for each method, as defined by Eq. 2, where $makespan_{PAR}$ and $makespan_{SEQ}$ are the makespan values computed using the parallel and the sequential MinMin implementation, respectively.

$$GAP = \frac{makespan_{PAR} - makespan_{SEQ}}{makespan_{SEQ}} \quad (2)$$

- *Execution times and speedup*: We analyze the wall-clock execution times and the speedup for each parallel MinMin implementation with respect to the sequential one. The speedup metric evaluates how much faster a parallel algorithm is than its corresponding sequential version. It is computed as the ratio of the execution times of the sequential algorithm (T_S) and the parallel version executed on m computing elements (T_m) (Equation 3). The ideal case for a parallel algorithm is to achieve linear speedup ($S_m = m$), but the most common situation is to achieve sublinear speedup ($S_m < m$), mainly due to the times required to communicate and synchronize the parallel processes. However, when using GPU infrastructures very large speedup values have been often reported.

$$S_m = \frac{T_S}{T_m} \quad (3)$$

Table 1 reports the average execution times (in seconds), the average GAP values and the average speedup for each of the four parallel MinMin implementations on GPU studied, and a comparison with the sequential implementation in CPU. The results in Table 1 correspond to the average values for all the HCSP instances solved for each problem dimension studied, and the comparison is performed considering the optimized sequential algorithms using the specialized data representation described in Section 4.

dimension	MinMin	MinMin-1GPU			MinMin-4GPU-PT		
	$t(s)$	$t(s)$	GAP	$speedup$	$t(s)$	GAP	$speedup$
1024×32	0.07	0.23	0.10%	0.31	0.88	20.00%	0.08
2048×64	0.39	0.37	-0.16%	1.06	0.95	28.33%	0.41
4096×128	2.25	1.02	0.13%	2.20	1.19	20.66%	1.89
8192×256	15.67	4.77	0.04%	3.28	2.03	19.90%	7.73
16384×512	119.74	24.83	-0.46%	4.82	6.08	20.45%	19.69
32768×1023	848.62	176.85	-0.11%	4.80	22.77	16.33%	37.28
65536×2048	6352.94	1049.34	0.11%	6.05	110.44	20.14%	57.52
131072×4096	49764.76	7253.88	0.04%	6.86	690.67	17.64%	72.03
dimension	**MinMin**	**MinMin-4GPU-OMP**			**MinMin-4GPU-sync**		
	$t(s)$	$t(s)$	GAP	$speedup$	$t(s)$	GAP	$speedup$
1024×32	0.07	0.83	20.00%	0.09	1.03	-0.07%	0.07
2048×64	0.39	0.89	28.33%	0.44	1.26	0.07%	0.31
4096×128	2.25	1.01	20.66%	2.21	1.95	-0.17%	1.15
8192×256	15.67	1.82	19.90%	8.62	4.16	0.05%	3.77
16384×512	119.74	5.84	20.45%	20.51	14.83	-0.28%	8.07
32768×1023	848.62	22.79	16.33%	37.23	60.16	-0.16%	14.11
65536×2048	6352.94	108.93	20.14%	58.32	292.75	-0.16%	21.70
131072×4096	49764.76	690.85	17.64%	72.05	2236.72	0.40%	22.25

Table 1. Experimental results for the GPU implementations.

The results in Table 1 show that significant improvements on the execution times of MinMin are obtained when using the GPU implementations for problem instances with more than 8.000 tasks. When solving the low-dimension problem instances, the GPU implementations were unable to outperform the execution times of the sequential MinMin, mainly due to the overhead introduced by the threads creation and management, and the use of the GPU memory. However, when solving larger problem instances that model realistic large grid scenarios, significant improvements in the execution times are achieved, specially for the problem instances with dimension 65536×2048 and 131072×4096.

Regarding the computational efficiency, Fig. 5 summarizes the speedup values for the GPU implementations for each problem dimension faced.

The evolution of the speedup values in Fig. 5 indicates that the four GPU implementations obtained small accelerations for the HCSP instances with dimension less than 8192×256. However, as the dimension of the problem instances grow (16384×512, 32768×1024, 65536×2048, and 131072×4096), reasonable speedup values are obtained for the parallel implementations. The best speedup values were computed for the two largest problem dimensions, with a maximum of **72.05** for the parallel asynchronous MinMin implementation on four GPUs using OpenMP threads.

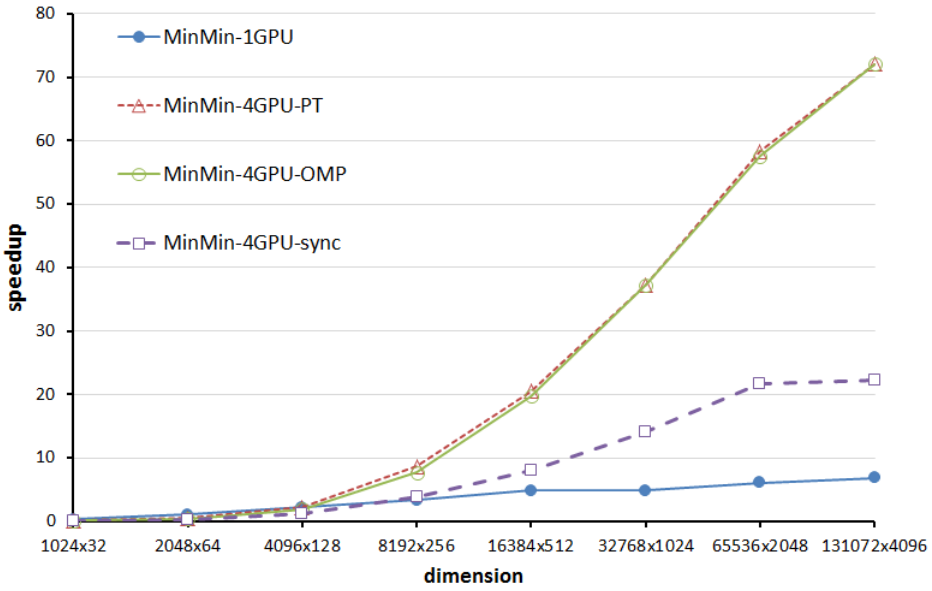


Fig. 5. Speedup for the MinMin GPU implementations.

The four studied MinMin variants in GPU provide different trade-off values between the quality of solutions and execution time required. The asynchronous implementations applying domain decomposition using four GPUs (MinMin-4GPU-PT and MinMin-4GPU-OMP) have the largest speedup values, but the results quality are from 16% to 20% worst than the sequential MinMin implementation. Despite the aforementioned reductions in the solution quality, these methods are able to compute the solutions in reduced execution times (i.e. about 10 minutes in the larges scenario studied, when scheduling 131072 tasks on 4096 machines), thus they can be useful to rapidly solve large scheduling scenarios. On the other hand, the parallel synchronous version of MinMin using four GPUs computed exactly the same solution than the sequential MinMin, but it improves the execution time in a factor of up to 22.25× for the largest instances tackled in this work.

The previously commented results indicate that the proposed parallel implementation of the MinMin list scheduling heuristic in GPU are accurate and efficient methods for scheduling in large HC and grid infrastructures. All parallel variants provides promising reductions in the execution times when solving large instances of the scheduling problem.

6 Conclusions and future work

This article studied the development of parallel implementations in GPU for a weel-known effective list scheduling heuristic algorithm, namely MinMin, for scheduling in heterogeneous computing environments.

The four proposed algorithms were developed using CUDA, following a simple domain decomposition approach that allows scaling up to solve very large dimension problem instances. The experimental evaluation solved HCSP instances with up to 131072 tasks and 4096 machines, a dimension far more larger than the previously tackled in the related literature.

The experimental results demonstrated that the parallel implementations of MinMin on GPU provide significant accelerations over the time required by the sequential implementations when solving large instances of the HCSP. On the one hand, the speedup values raised up to a maximum of **72.05** for the parallel asynchronous MinMin implementation on four GPUs using OpenMP threads. On the other hand, the parallel synchronous version of MinMin using four GPUs computed exactly the same solution than the sequential MinMin, but improving the execution time in a factor of up to **22.25**× for the largest instances tackled in this work.

The previously commented results demonstrate that the parallel MinMin implementations in GPU introduced in this article are accurate and efficient schedulers for HC systems, which allow tackling large scheduling scenarios in reasonable execution times.

The main line for future work is related with improving the proposed GPU implementations, mainly by studying the management of the memory accessed by the threads. In this way, the computational efficiency of the heuristics on GPU can be further improved, allowing to develop even more efficient parallel implementations. Another line for future works is used this implementations for complement the efficient heuristic local search methods implemented on GPU. We are working on these topics right now.

References

1. S. Ali, H. Siegel, M. Maheswaran, S. Ali, and D. Hensgen. Task execution time modeling for heterogeneous computing systems. In *Proc. of the 9th Heterogeneous Computing Workshop*, page 185, Washington, USA, 2000.
2. F. Berman, G. Fox, and A. Hey. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, Inc., New York, NY, USA, 2003.
3. T. Braun, H. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B. Yao, D. Hensgen, and R. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *J. Parallel Distrib. Comput.*, 61(6):810–837, 2001.
4. H. El-Rewini, T. Lewis, and H. Ali. *Task scheduling in parallel and distributed systems*. Prentice-Hall, Inc., 1994.
5. M. Eshaghian. *Heterogeneous Computing*. Artech House, 1996.
6. R. Fernando, editor. *GPU gems*. Addision-Wesley, Boston, 2004.
7. I. Foster and C. Kesselman. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.
8. R. Freund, V. Sunderam, A. Gottlieb, K. Hwang, and S. Sahni. Special issue on heterogeneous processing. *J. Parallel Distrib. Comput.*, 21(3), 1994.
9. M. Garey and D. Johnson. *Computers and intractability*. Freeman, 1979.

10. O. Ibarra and C. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM*, 24(2):280–289, 1977.
11. D. Kirk and W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.
12. Y. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.
13. J. Leung, L. Kelly, and J. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., 2004.
14. S. Nesmachnow. A cellular multiobjective evolutionary algorithm for efficient heterogeneous computing scheduling. In *EVOLVE 2011, A bridge between Probability, Set Oriented Numerics and Evolutionary Computation*, 2011.
15. nVidia. CUDA website. Available online `http://www.nvidia.com/object/cuda_home.html`, 2010. Accessed on July 2011.
16. J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.
17. M. Theys, T. Braun, H. Siegel, A. Maciejewski, and Y. Kwok. Mapping tasks onto distributed heterogeneous computing systems using a genetic algorithm approach. In *Solutions to parallel and distributed computing problems*, pages 135–178, New York, USA, 2001. Wiley.
18. T. Velte, A. Velte, and R. Elsenpeter. *Cloud Computing, A Practical Approach*. McGraw-Hill, Inc., New York, NY, USA, 2010.

HPCLatAm 2012, pp. 96-111 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

A parallel online GPU scheduler for large heterogeneous computing systems

Santiago Iturriaga[1], Sergio Nesmachnow[1], Francisco Luna[2], and Enrique Alba[2]

[1] Universidad de la República, Uruguay
{siturria,sergion}@fing.edu.uy
[2] Universidad de Málaga, Spain
{flv,eat}@lcc.uma.es

Abstract. This work presents a parallel implementation on GPU for a stochastic local search method to efficiently solve the task scheduling problem in heterogeneous computing environments. The research community has been searching for accurate schedulers for heterogeneous computing systems, able to run in reduced times. The parallel stochastic search proposed in this work is based on simple operators in order to keep the computational complexity as low as possible, thus allowing large scheduling instances to be efficiently tackled. The experimental analysis demonstrates that the parallel stochastic local search method on GPU is able to compute accurate suboptimal schedules in significantly shorter execution times than state-of-the-art schedulers.

Keywords: GPU computing, heterogeneous computing, scheduling.

1 Introduction

Scheduling tasks in current heterogeneous computing (HC) systems challenges researchers to the problem of assigning dozens of thousands of tasks in very short times that should be limited to a few seconds. Indeed, HC systems are becoming larger and larger during the last fifteen years, mainly due to the fast increase of computing power and the rapid development of high-speed networking protocols, but also to the demand of the scientific community that has to address increasingly large problems that require an enormous computing power [7]. Assigning and mapping tasks becomes therefore a critical issue since finding an accurate schedule significantly impacts in both the resource utilization costs and the quality of service (QoS) perceived by the user.

Scheduling problems have been widely studied in operational research [4,12], but most of the classical approaches have faced the task scheduling in homogeneous environments. The ultimate goal of a scheduling problem is to provide an assignment of tasks to resources so that some efficiency criteria is satisfied, usually related to the total execution time for a bunch of tasks (makespan), but frequently also considering other metrics such as the resource utilization and/or the QoS.

The *Heterogeneous Computing Scheduling Problem* (HCSP), in which the resources are different among them, has become important due to the popularization of distributed computing and the growing use of heterogeneous clusters [5,8]. Even the traditional homogeneous scheduling problems are NP-hard [9], so heterogeneity makes this problem even harder, thus allowing exact methods being only useful for solving instances of reduced size.

When dealing with large problem instances, as demanded by the size of current HC systems, heuristic and metaheuristic methods [14,18,19] are the only viable options in practice to compute efficient schedules in reasonable execution times. In this context, the faster the scheduler, the quicker the HC system can allocate new tasks and, as a consequence, the better its utilization degree (and the corresponding income). However, it is very hard, particularly for metaheuristics, to meet the wall-clock time constraint imposed in current heterogeneous cluster computing infrastructures and in grid computing systems. In fact, scheduling dozens of thousands of tasks is even slow for basic low level heuristic methods, which are usually much faster than metaheuristics. This work is focussed on these latter heuristic methods and the use of parallelism to reduce their computational times. The actual scientific contribution therefore lies in the parallelization of a stochastic local search (rPALS [15]) on GPU (Graphic Processing Units) cards. It has been called gPALS. Its main goal is to profit from the computing power of these newly available massively parallel platforms in order to address very large HCSP instances. Indeed, the testbed used is composed of problem instances that range from 8096 to 32768 tasks, and 256 to 1024 machines, respectively. Averaging over 60 different instances, the results have shown that, compared to the state-of-the-art deterministic MinMin heuristic [11], gPALS is able to reach task schedules with a 5% lower makespan more than 11 times faster.

The rest of the manuscript is organized as follows. The next section presents the HCSP formulation and briefly describes the list scheduling heuristics used for initializing the proposed gPALS and in the results comparison. Section 3 introduces the main concepts about GPU computing. The details about the GPU implementation for the stochastic local search method proposed to efficiently solve the HCSP are presented in Section 4. The experimental analysis is described in Section 5, studying the numerical efficacy and the computational efficiency of the proposed method in a number of large-sized HCSP scenarios. Finally, Section 6 presents the main conclusions of the research and formulates the main lines for future work.

2 Heterogeneous computing scheduling

This section introduces the HCSP formulation and presents some considerations about the execution time estimation model used in the problem instances to solve. In addition, the classic deterministic heuristics applied for initializing the proposed gPALS method and the one used as a baseline to compare the gPALS results are described.

HPCLatAm 2012, pp. 96-111 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

2.1 Formal definition

An HC system is a computational platform composed of many computational resources, also called *processors* or *machines*. The scheduling problem in HC considers a set of tasks with variable computing demands to be executed on the system. A task is the atomic unit of workload, so it cannot be divided into smaller chunks, nor interrupted after it is assigned to a machine (i.e., the scheduling problem follows a *non-preemptive* model). The execution times of each task vary from one machine to another, so there will be competition among tasks for using those machines able to execute them in the shortest time.

The most usual objective to minimize in scheduling is the *makespan*, defined as the time spent from the moment when the first task begins its execution to the moment when the last task is completed. However, many other objectives have been considered in scheduling problems [12].

The following formulation presents the mathematical model for the HCSP:

- given an HC system composed of a set of machines $M = \{m_1, m_2, \ldots, m_L\}$ and a collection of tasks $T = \{t_1, t_2, \ldots, t_N\}$ to be executed on the system,
- let there be an *execution time function* $ET : T \times M \rightarrow \mathbf{R}^+$, where $ET(t_i, m_j)$ is the time required to execute the task t_i in the machine m_j,
- the goal of the HCSP is to find an assignment of tasks to machines (a function $f : T^N \rightarrow M^L$) which minimizes the *makespan* metric, defined in Eq. 1.

$$\max_{m_j \in M} \sum_{\substack{t_i \in T: \\ f(t_i)=m_j}} ET(t_i, m_j) \ . \tag{1}$$

Using the 3-field notation from Graham *et al.* [10], the HCSP is denoted $Rm|1|Cmax$.

2.2 Execution time estimation model

In this work, we adopted the *expected time to compute* (ETC) performance estimation model by Ali *et al.* [2], which has been widely used by the research community when facing the HCSP. ETC provides an estimation for the execution time of a collection of tasks in an HC system, taking into account three key properties: machine heterogeneity, task heterogeneity, and consistency.

Machine heterogeneity evaluates the variation of execution times for a given task across the HC resources. A system with similar computing resources has low machine heterogeneity, while high machine heterogeneity represents HC systems with computing resources of different power. *Task heterogeneity* represents the variation of the tasks execution times for a given machine. In a high task heterogeneity scenario, different types of applications are submitted to execution, from simple programs to complex tasks which require large CPU times to be performed. On the other hand, low task heterogeneity models those scenarios when the tasks computational requirements, and thus their execution times, are similar for a given machine.

The ETC model considers a second classification. In a *consistent* ETC scenario, whenever a given machine m_j executes any task t_i faster than other machine m_k, then machine m_j executes all tasks faster than machine m_k. An *inconsistent* ETC scenario lacks of structure among the computing demands of tasks and the computing power of machines, so a given machine m_j may be faster than another machine m_k when executing some tasks, and slower for others. In addition, a third category of *semi-consistent* ETC scenarios is included, to model those inconsistent systems that include a consistent subsystem.

2.3 List scheduling heuristics

Several deterministic heuristics have been proposed for HC scheduling. One of the most used class of such methods is list scheduling heuristics [11]. List scheduling methods work by assigning priorities to tasks based on a particular criteria, sorting the list of tasks in decreasing priority, and assigning each task to a processor, regarding both the task priority and the processor availability.

Variations of two well-known list scheduling heuristics have been used in this work to generate the initial solution for the gPALS method:

- *Minimum Completion Time* (MCT) considers the set of tasks sorted in an arbitrary order. Then, it assigns each task to the machine with the minimum ET for that task.
- *MinMin* greedily picks the task that can be completed the soonest. The method starts with a set U of all *unmapped* tasks, calculates the MCT for each task in U for each machine, and assigns the task with the minimum overall MCT to the machine that executes it faster. The mapped task is removed from U, and the process is repeated until all tasks are mapped.

The MinMin heuristic provides an excellent packing of tasks for HC environments with high level of heterogeneity of both tasks and machines, thus computing better makespan values than other well-known list scheduling heuristics [11]. For this reason, the MinMin results are used in this work as a reference baseline for comparing the results computed with the proposed local search algorithm.

3 GPU computing

GPUs were originally designed to exclusively perform the graphic processing in computers, allowing the Central Process Unit (CPU) to concentrate on the remaining computations. Nowadays, GPUs have a considerably large computing power, provided by hundreds of processing units with reasonable fast clock frequencies. In the last ten years, GPUs have been used as a powerful parallel hardware architecture to achieve efficiency in the execution of applications.

GPU programming and CUDA. The first GPUs used for general-purpose computing were programmed using low-level mechanisms such as the interruption services of the BIOS, or by using graphic APIs such as OpenGL and DirectX [6].

Later, the programs for GPU were developed in assembly language for each card model, and they had very limited portability. So, high-level languages were developed to fully exploit the capabilities of the GPUs. In 2007, NVIDIA introduced CUDA (Compute Unified Device Architecture) [16], a software architecture for managing the GPU as a parallel computing device without requiring to map the data and the computation into a graphic API.

CUDA extends the C language, and it is available since cards of the GeForce 8 Series onwards. Three software layers are used in CUDA to communicate with the GPU (see Fig. 1): a low-level hardware driver that performs the CPU-GPU data communications, a high-level API, and a set of libraries such as CUBLAS for linear algebra and CUFFT for Fourier transforms calculation.

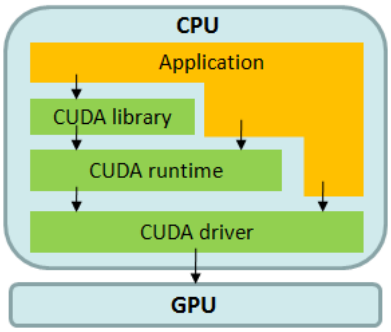


Fig. 1. CUDA architecture.

For the CUDA programmer, the GPU is a computing device which is able to execute a large number of threads in parallel. A specific procedure to be executed many times over different data can be isolated in a GPU-function using many execution threads. The function is compiled using a specific set of instructions and the resulting program (named *kernel*) is loaded in the GPU. The GPU has its own DRAM, and the data are copied from the DRAM of the GPU to the RAM of the host (and viceversa) using optimized calls of the CUDA API.

The CUDA architecture is built around a scalable array of multiprocessors, each one having eight scalar processors, one multithreading unit, and a shared memory chip. The multiprocessors are able to create, manage, and execute parallel threads, with reduced overhead. The threads are grouped into *blocks* (with up to 512 threads), which are executed in a single multiprocessor of the graphic card, and the blocks are grouped in *grids*. Each time that a CUDA program calls a grid to be executed in the GPU, each of the blocks in the grid is numbered and distributed to an available multiprocessor. When a multiprocessor receives a block to be executed, it splits the threads into *warps*, a set of 32 consecutive threads. Each warp executes a single instruction at a time, so the best efficiency is achieved when the 32 threads in the warp executes the same instruction. Otherwise, the warp serializes the threads. When a block finishes its execution, a new block is assigned to the available multiprocessor.

The threads are able to access the data using three memory spaces: a *shared memory* which can be used by the threads in the block; the *local memory* of the thread; and the *global memory* of the GPU. Minimizing the access to the slower memories (the local memory of the thread and the global memory of the GPU) is a very important feature to achieve high efficiency in GPU computing. On the other hand, the shared memory is placed within the GPU chip, thus providing a faster way to store data, such as the registers of each multiprocessor.

4 gPALS: a GPU implementation of a stochastic local search scheduler

This section describes both rPALS, the base algorithm that has been parallelized, and gPALS, its deployment on GPU cards.

4.1 rPALS

The stochastic local search algorithm proposed in this work to efficiently solve the HCSP is based on PALS [1]. The original PALS method is a deterministic local search algorithm originally proposed for the DNA fragment assembly problem.

PALS works on a single solution s, which is iteratively modified by applying a series of movements aimed at locally improving their objective function value $f(s)$. The movement operator performs a modification on two positions i and j in the solution s, while the key step is the calculation of the objective function variation $\Delta f_{(i,j)}$ when applying a certain movement. When the calculation of $\Delta f_{(i,j)}$ can be performed without significantly increasing the computational cost of the algorithm, PALS provides a very efficient search pattern for combinatorial optimization problems.

In this work, a randomized variant of PALS (rPALS), has been used for the HCSP [15]. The aim of the algorithm is to reach accurate schedules in very short times. To do so, from a initial solution computed by a fast scheduling heuristic, rPALS iteratively applies two basic operations that either swap or move randomly chosen tasks allocated to randomly chosen machines, thus avoiding exploring all the possible neighbors. The algorithm has been designed under the paradigm of simplicity; by using simple search operators, the resulting rPALS method is able to scale up in order to face realistic medium-sized HCSP instances.

4.2 gPALS

The emergence of general purpose GPU computing has opened new research lines specially promising in this field of scheduling and planning. Indeed, this new technology will help to address more and more larger problem instances (up to 32768 tasks and 1024 machines in this work) in reasonable wall-clock times which are closer to the actual HC infrastructures. The key issue is to fully exploit the massively parallelism of the GPU cards. This is precisely the main design goal of gPALS, the GPU version of rPALS.

Algorithm 1 presents the pseudo-code of the gPALS algorithm for the HCSP. The method starts by generating an initial schedule s using a list scheduling heuristic (e.g. MCT, pMin-Min/DD, etc.). Then, a search is performed in the GPU in order to find candidate movements to improve the schedule, this search is detailed in Algorithm 2. The GPU search returns the best `GPU_BLOCKS` movements found. The best movement is always applied, the remaining movements are applied in random order as long as they do not modify a machine already modified by a previous movement, otherwise the movement is discarded. Once all the movements are either applied or discarded, the stopping criterion is tested and the algorithm either ends or performs a new iteration.

The movement search on the GPU is organized in blocks; there are `GPU_BLOCKS` blocks, each block having `GPU_THREADS` threads. Each block performs an independent local search in a randomly selected neighbourhood, with the threads in the block collaborating with each other to find the best movement in the assigned neighbourhood. Algorithm 2 presents the movement search performed on the GPU. First, each block deterministically selects a movement type (i.e. `MOVE` or `SWAP`). Then each thread in the block randomly selects the source and destination elements to modify. Each thread evaluates its assigned movement and computes a score for it. After each thread in the block evaluated its assigned movement, the best movement (i.e. the one with the lower score) is selected and returned to Algorithm 1 as the best movement found in the block.

Algorithm 1 gPALS for the HSCP

1: $s \leftarrow$ initialize using a list scheduling heuristic
2: **while** `STOPPING_CONDITION` is not met **do**
3: $m \leftarrow$ Parallel execution of the movement search kernel in s with 128 blocks with 256 threads each {A total of 32768 threads are launched}
4: $s \leftarrow$ Apply the best movement from m
5: $s \leftarrow$ Apply the rest of the movements in m in random order
6: **end while**
7: **return** s

As it can be seen, gPALS requires an initial solution which is iteratively improved (line 1 in Algorithm 1). For this initial solution to be generated, any classical list heuristic could be used in order to provide gPALS with a rather high quality task schedule. Two different versions of gPALS have been devised depending on this heuristic:

- gPALS$_{MCT}$: it uses MCT for generating the initial task schedule.
- gPALS$_{MMDD}$: the *pMin-Min/DD* (or parallel *Min-Min* with domain decomposition) heuristic is adopted. It is a multithreading version of the *Min-Min* algorithm, which performs a task domain decomposition and does not require any synchronization mechanism (see [13] for the details).

A diagram of the parallel model used in gPALS is presented in Fig. 2.

Algorithm 2 Movement search kernel for the GPU.

1: **shared** $M \leftarrow \emptyset$ {shared across all threads in the block}
2: $s \leftarrow$ Current schedule
3: movement $\leftarrow$ Choose which movement to perform
4: **if** movement is `TASK_SWAP` **then**
5: $t_x, t_y \leftarrow$ Choose two random tasks ($t_x \neq t_y$)
6: $m \leftarrow$ Swap t_x with t_y
7: **else if** movement is `TASK_MOVE` **then**
8: $t_x \leftarrow$ Choose a random task
9: $m_y \leftarrow$ Choose a random machine
10: $m \leftarrow$ Move t_x to m_y
11: **end if**
12: **if** makespan of the schedule s increases after the movement m **then**
13: $score \leftarrow \infty$
14: **else**
15: $ct_x \leftarrow$ Compute time of the machine m_x to which t_x is assigned
16: $ct'_x \leftarrow$ Compute time of the machine m_x after applying the movement
17: $ct_y \leftarrow$ Compute time of the machine m_y (when swapping, the machine to which t_y is assigned)
18: $ct'_y \leftarrow$ Compute time of the machine m_y after applying the movement
19: $score \leftarrow (ct'_x - max(ct_x, ct_y)) + (ct'_y - max(ct_x, ct_y))$
20: **end if**
21: $M \leftarrow M \cup m$
22: **synchronize()** {threads in the block}
23: $m_{best} \leftarrow$ Parallel reduce M to find best movement in the block
24: **return** m_{best}

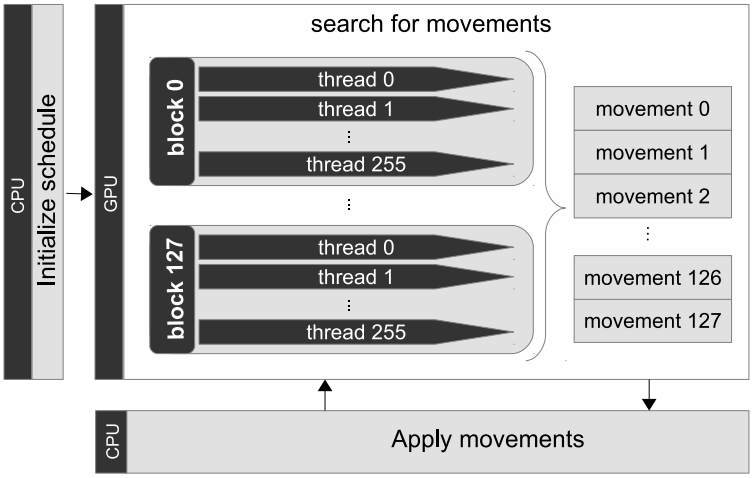


Fig. 2. Parallel model applied in gPALS.

5 Experimental analysis

This section introduces the set of HCSP instances and the computational platform used to evaluate the proposed LS algorithm. After that, the experiments conducted to determine the best values for the randomized PALS parameters are presented. Finally, the results obtained when solving realistic HCSP instances are analyzed in detail.

5.1 HCSP instances

To evaluate the proposed gPALS method, a specific set of 60 HCSP instances was used. These instances were randomly generated following the *range based* methodology proposed by Ali *et al.* [2], and they were previously employed to evaluate a cellular genetic algorithm scheduler for HC systems in the work by Pinel et al. [17].

The HCSP instances solved in this article model realistic large-sized HC infrastructures. Three problem dimensions were studied in the experimental analysis of gPALS: (tasks×machines) 8192×256, 16384×512, and 32768×1024. This dimensions are far more larger than the ones usually tackled in the related literature. For each problem dimension considered, 20 instances were used, follow the parametrization values suggested by Braun *et al.* [3].

5.2 Development and execution platform

The rPALS algorithm was implemented in C, using the standard `stdlib` library and compiled with gcc 4.1.2. The experimental analysis was performed in a Dell PowerEdge with QuadCore Xeon E5430 processor at 2.66 GHz, 8 GB RAM, and CentOS Linux (platform website: `http://www.fing.edu.uy/cluster`).

5.3 Results and discussion

This section is aimed at presenting the experimental results obtained. It has been structured in two separate subsections so as to analyze, firstly, the quality of the tasks schedules reached by MinMin, gPALS$_{MCT}$, and gPALS$_{MMDD}$ in terms of their makespan and, secondly, the parallel performance of the proposed approaches, gPALS$_{MCT}$ and gPALS$_{MMDD}$, with respect to MinMin.

Numerical efficiency. Table 1 reports the makespan reached by MinMin and the two versions of gPALS for the three set of instances with increasing size. Before going into details, we want to make clear the experimental conditions. Whereas MinMin has been let to execute until it schedules all the tasks, i.e., until a full solution is built (it is a constructive heuristic), both gPALS$_{MCT}$ and gPALS$_{MMDD}$ stop when 30 seconds of GPU computation have elapsed (the loading time of the instance is not considered here). The cells with the best makespan are marked with a gray background.

HPCLatAm 2012, pp. 96-111 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

Table 1. Makespan of the three algorithms for 60 HCSP instances, 20 for each dimension —8192×256, 16384×512, and 32768×1024.

	8192×256			16384×512			32768×1024		
	MinMin	gPALS$_{MCT}$	gPALS$_{MMDD}$	MinMin	gPALS$_{MCT}$	gPALS$_{MMDD}$	MinMin	gPALS$_{MCT}$	gPALS$_{MMDD}$
1	1845.2	1835.4	1710.4	1934.1	1886.1	1777.0	1996.2	1927.3	1831.3
2	1889.9	1863.2	1740.0	1940.5	1889.9	1781.2	1979.0	1918.0	1824.3
3	1894.3	1831.0	1716.0	1949.0	1895.7	1782.1	1980.7	1919.7	1821.6
4	1890.1	1866.4	1743.0	1922.0	1887.1	1778.4	1982.8	1929.3	1830.6
5	1859.6	1843.7	1724.8	1904.1	1867.7	1757.1	1971.9	1918.5	1818.4
6	1863.4	1829.0	1715.4	1901.7	1885.7	1772.6	1973.4	1912.0	1816.3
7	1897.3	1862.3	1736.9	1945.5	1898.1	1787.6	1991.0	1926.0	1829.0
8	1874.5	1852.6	1738.4	1903.8	1878.6	1768.5	1991.8	1922.5	1822.0
9	1871.5	1853.3	1730.9	1937.3	1894.7	1782.4	1994.8	1929.5	1832.2
10	1865.9	1867.2	1742.5	1935.7	1877.6	1769.9	1997.1	1922.1	1822.9
11	1840.7	1823.8	1711.9	1937.4	1899.2	1786.7	1975.8	1914.7	1816.2
12	1867.3	1836.7	1724.2	1916.2	1871.6	1762.8	1974.2	1912.2	1813.5
13	1895.4	1867.5	1744.6	1911.8	1884.8	1771.6	1978.6	1915.2	1818.6
14	1884.8	1841.8	1725.4	1927.6	1898.7	1784.0	1988.1	1923.3	1824.3
15	1851.0	1828.2	1710.8	1944.4	1901.0	1787.5	1972.1	1915.8	1819.3
16	1846.3	1837.2	1724.1	1939.5	1886.5	1777.9	1979.3	1913.3	1814.2
17	1874.7	1818.1	1707.8	1933.6	1878.4	1764.9	1991.8	1916.7	1814.9
18	1862.8	1856.5	1736.7	1929.5	1887.0	1776.4	1986.8	1922.5	1825.5
19	1892.5	1853.4	1732.7	1910.8	1880.6	1765.6	1975.2	1914.5	1814.2
20	1869.0	1853.8	1731.2	1941.0	1891.6	1781.0	1991.7	1919.9	1819.2

The experimental results in Table 1 clearly point out that gPALS$_{MMDD}$ is the algorithm that reached the task schedules that most reduces the makespan for all the instances addressed. This occurs consistently for the three instances sizes, i.e., 8192×256, 16384×512, and 32768×1024. Averaging over all the instances of the same size, gPALS$_{MMDD}$ improves the makespan computed by MinMin in 7.72%, 7.91%, and 8.18%, respectively. It is important to note the relevance of these values, given the experimental conditions. Though slightly, these average values show that, the larger the instances, the better the improvement, and this has been achieved by keeping the same computation time, i.e., 30 seconds. That is, for search spaces very much larger (both the number of tasks and machines is doubled), our approach is able to improve even more MinMin, which requires in turn very much longer execution times (see the next section). To a lesser extent, the same claims hold for gPALS$_{MCT}$: the average improvements are also increasing with the instance size, but only 1.37%, 2.13%, and 3.24%, respectively.

In order to better support our claims, Fig. 3 displays the evolution of the makespan of a typical 8192×256 instance in terms of (a) the iterations and (b) the execution time of gPALS$_{MCT}$ and gPALS$_{MMDD}$ in a typical execution, respectively. The makespan obtained by Min-Min is also included as a baseline for the comparison. These subfigures shows a very interesting fact. For gPALS, the more accurate the initial solution (in this case, that computed by MCT), the earlier the stagnation in a local minimum. Indeed, it can be seen that pMin-

HPCLatAm 2012, pp. 96-111 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

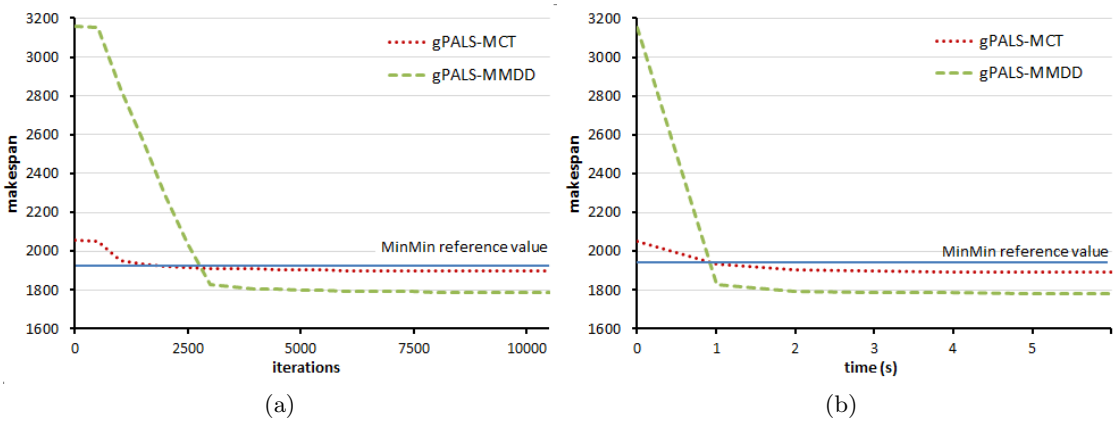


Fig. 3. Evolution of the makespan during a typical execution of the three compared algorithms for a 8192×256 instance with respect to (a) the iterations of gPALS and (b) its wall-clock time.

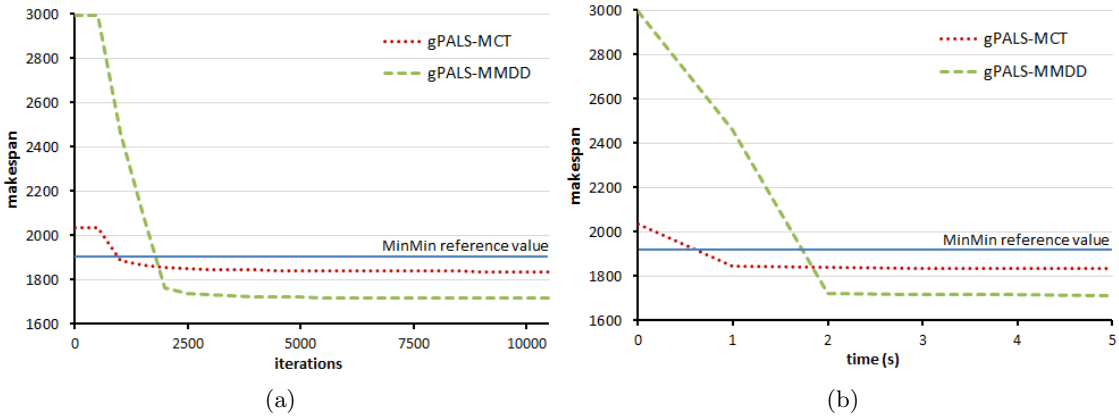


Fig. 4. Evolution of the makespan during a typical execution of the three compared algorithms for a 16384×512 instance with respect to (a) the iterations of gPALS and (b) its wall-clock time.

MinDD reaches a task schedule with much higher (worse) makespan, and then gPALS is able to iteratively move and swap tasks between machines that allow the makespan to be continuously reduced up to the iteration 2500. With respect to Min-Min, gPALS$_{MCT}$ requires around 1000 generations to reach a lower makespan, whereas gPALS$_{MMDD}$ is around iteration 2000. If we now turn to analyze the evolution with respect the execution time, the picture changes. The first remark here is that the two gPALS versions outperform MinMin after just one single second of computation, clearly showing their suitability for addressing this large instances of the HCSP problem. The second remark raises when comparing gPALS$_{MCT}$ and gPALS$_{MMDD}$: the latter also requires just one second to reach a more accurate task schedule than the former.

Figure 4 presents the evolution of makespan values with respect to both the iterations of gPALS and the execution time, but for a representative 16384×512-sized instance. All the previous claims hold as well with the only difference in subfigure (b), in which now the generation of the initial solution for gPALS$_{MMDD}$ with the pMin-MinDD heuristic takes longer and delays outperforming both MinMin and gPALS$_{MCT}$ about one second. The same behavior was detected for the other HCSP instances in the benchmark set solved in this article.

Table 2. Wall-clock of the three algorithms (in seconds) for 60 HCSP instances, 20 for each dimension —8192×256, 16384×512, and 32768×1024.

	8192×256			16384×512			32768×1024		
	MinMin	gPALS$_{MCT}$	gPALS$_{MMDD}$	MinMin	gPALS$_{MCT}$	gPALS$_{MMDD}$	MinMin	gPALS$_{MCT}$	gPALS$_{MMDD}$
1	15.0	11.6	10.6	110.7	9.7	16.7	839.8	21.1	112.6
2	15.0	9.8	9.8	110.6	9.9	17.0	838.8	20.5	115.8
3	14.8	9.7	8.5	110.6	9.7	17.6	842.5	18.9	98.5
4	14.9	10.4	9.7	110.7	11.4	19.1	841.7	20.9	111.3
5	14.9	8.1	8.4	111.2	13.1	17.1	845.3	22.2	105.7
6	14.9	7.9	8.9	111.4	12.0	17.9	834.6	21.9	110.9
7	14.9	8.4	9.0	111.3	11.7	19.0	837.4	20.6	105.5
8	15.3	8.0	8.4	111.3	12.4	17.1	843.2	19.6	112.3
9	15.0	8.6	8.3	111.1	10.0	18.1	838.4	19.6	105.0
10	15.0	11.8	8.2	110.7	12.4	18.9	839.3	21.4	122.2
11	14.8	8.8	8.6	110.8	13.1	19.8	840.4	20.6	109.6
12	14.9	7.9	8.2	110.9	12.6	19.8	838.7	19.1	100.3
13	14.9	7.7	8.2	110.7	13.4	19.9	843.1	20.3	110.3
14	14.9	7.5	8.2	110.8	13.5	20.2	841.8	19.9	112.1
15	14.9	7.7	8.2	110.5	13.4	19.1	844.0	20.8	107.4
16	15.1	8.7	8.2	110.8	12.7	19.9	834.9	22.0	111.2
17	15.0	7.5	8.2	111.3	13.3	19.9	837.6	21.1	106.1
18	14.6	9.0	8.3	111.3	13.2	19.7	842.7	20.5	108.3
19	14.9	7.5	8.2	111.2	13.0	20.2	838.8	19.3	125.0
20	14.9	8.1	8.2	111.1	13.3	19.1	840.0	19.9	99.3

Parallel performance. We have already provided the reader with some hints about the main features of the computational times of the three algorithms, but we now want to detail them in a separate experimentation. Table 2 includes the wall-clock time of MinMin, gPALS$_{MCT}$ and gPALS$_{MMDD}$ for the 60 HSCP instances with increasing size considered in this work. The experimental conditions for the two gPALS methods have changed: they stop when they reach a task schedule with a lower makespan than that of MinMin (in the previous section, the stopping condition was to reach 30 seconds of GPU computation).

The first clear claim is that the two gPALS versions are always faster than MinMin to achieve an task schedule with the same makespan. The truly interesting point here is that, the larger the instance, the higher the reduction in the

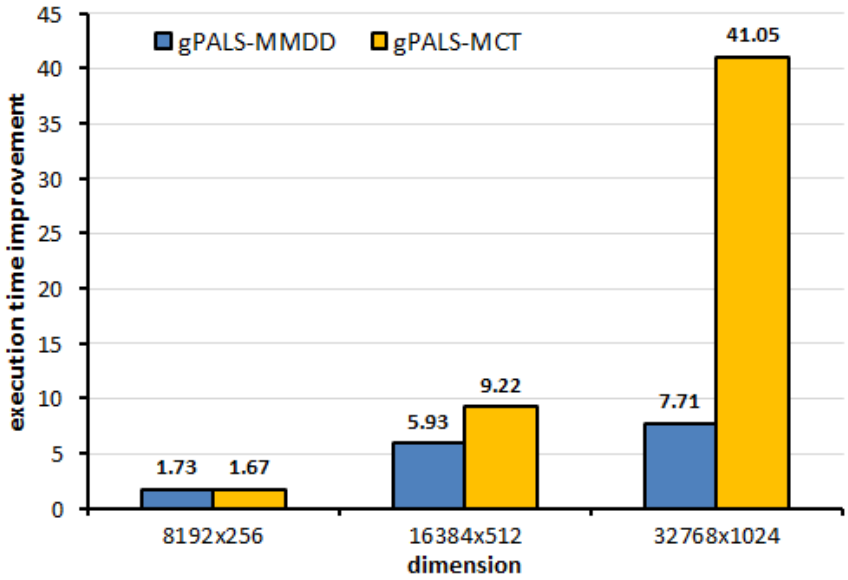


Fig. 5. Average execution time improvements of gPALS$_{MCT}$ and gPALS$_{MMDD}$ with respect to MinMin.

execution times. Indeed, MinMin requires roughly 15, 110, and 840 seconds to build a solution for 8192×256, 16384×512, and 32768×1024 instances, respectively, whereas gPALS$_{MCT}$ and gPALS$_{MMDD}$ need 8, 12, and 20 seconds, and 9, 18, and 110 seconds, respectively. These differences can be clearly seen in Fig. 5, which displays the average execution time improvements over all the instances of the same size reached by gPALS$_{MCT}$ and gPALS$_{MMDD}$. The execution time improvement refers to the reduction in the execution time of an algorithm that runs in a parallel computing platform (in our case the GPU) with respect another one that executes sequentially, i.e., $\frac{t_{CPU}}{t_{GPU}}$. For the smaller instance considered in this work, the two gPALS approaches perform the same, with execution time improvements of 1.73 and 1.67. However, as long as the size of instance increases, MinMin requires more time to complete, i.e., it does not scale well, whereas our approaches do scale properly, specially gPALS$_{MCT}$, which has been able to reach an execution time improvement of 40.1 for the largest instance. We would like to dive a little bit more on the results of the two gPALS versions and explain why the execution time improvements of gPALS$_{MCT}$ is much higher. Obviously, it has to do with the computational time of the initial task schedule by the heuristic. MCT is a extremely fast method whose translation to the GPU does not make sense because little benefits would be obtained. On the other hand, pMinMinDD is much heavier and, even ported to the GPU, takes longer to build a solution, what reduces its execution time improvements.

HPCLatAm 2012, pp. 96-111 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

6 Conclusions

This work has presented gPALS, a GPU implementation of a randomized local search procedure for addressing large instances of the scheduling problem in HC systems. The aim of the algorithm is to reach accurate schedules in very short times for large HCSP instances using the parallel computing resources available in GPUs. To do so, from a initial solution computed by either the MCT heuristic or a parallel version of the MinMin heuristic, two variants of gPALS were implemented in GPU by iteratively applying two basic operations that either swap or move randomly chosen tasks allocated to randomly chosen machines. The key is that the variation in the makespan of these operations can be computed efficiently and, consequently, several millions operations can be performed in few seconds.

The experimental analysis performed using a testbed with 60 large HCSP instances of three increasing dimensions (up to 32768 tasks and 1024 machines) compared the two proposed versions of gPALS against Min-Min, one of the best state-of-the-art list scheduling heuristics for HC environments. The experimental results demonstrate that the proposed gPALS implementations are able of compute better makespan values than MinMin in all the 60 studied instances.

The gPALS$_{MMDD}$ variant is the best method between the two GPU implementations, obtaining significant improvements (up to 8.18%) with respect to MinMin. These reductions in the makespan obtained by gPALS$_{MMDD}$ have taken a wall-clock time of 30 seconds, which represent a factor of almost 8× in the computational efficiency with respect to the MinMin scheduler. On the other hand, gPALS$_{MCT}$ computed schedules significantly faster than both gPALS$_{MMDD}$ and MinMin, achieving execution time improvements up to 41.05 with respect to MinMin. The solution computed by gPALS$_{MCT}$ improves upon the ones computed using MinMin, but they have lower quality (i.e., larger makespan values) than those found by the gPALS$_{MMDD}$ implementation. Regarding the execution time comparison, both gPALS implementations are able to improve over the MinMin makespan result in only a few seconds of execution time (without counting the time spent in computing the initial solution).

The previously commented results have demonstrated that the new gPALS$_{MMDD}$ algorithm is an accurate and very efficient scheduler for the HCSP instances tackled in this article.

Two main lines are proposed for future work: improve the efficacy of the search in gPALS, and also to enhance the computational efficiency of the proposed optimization method. Regarding the first issue, we propose to analyze carefully the landscape of the HCSP, in order to design specialized basic operations that further improve the efficacy of the search in gPALS, by avoiding to explore non-promising regions of the search space. On the other hand, in order to be able to address even larger HCSP instances in shorter times, we also plan to engineer a more efficient version of gPALS by employing domain-decomposition parallel computing techniques in GPU.

Acknowledgments. The work of S. Iturriaga and S. Nesmachnow has been partially supported by ANII and PEDECIBA, Uruguay. The work of F. Luna and E. Alba has been partially funded by the Spanish Ministry of Science and Innovation and FEDER under contracts TIN2008-06491-C04-01 (the MSTAR project) and TIN2011-28194 (the roadME project), and by the Andalusian Government under contract P07-TIC-03044 (the DIRICOM project).

References

1. E. Alba and G. Luque. A new local search algorithm for the DNA fragment assembly problem. In C. Cotta and J. van Hemert, editors, *Proceedings of 7th European Conference on Evolutionary Computation in Combinatorial Optimization*, volume 4446 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2007.
2. S. Ali, H. Siegel, M. Maheswaran, S. Ali, and D. Hensgen. Task execution time modeling for heterogeneous computing systems. In *Proceedings of the 9th Heterogeneous Computing Workshop*, page 185, Washington, DC, USA, 2000. IEEE Computer Society.
3. T. Braun, H. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B. Yao, D. Hensgen, and R. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, 2001.
4. H. El-Rewini, T. Lewis, and H. Ali. *Task scheduling in parallel and distributed systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.
5. M. Eshaghian. *Heterogeneous Computing*. Artech House, Norwood, MA, USA, 1996.
6. R. Fernando, editor. *GPU gems*. Addision-Wesley, Boston, 2004.
7. I. Foster and C. Kesselman. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.
8. R. Freund, V. Sunderam, A. Gottlieb, K. Hwang, and S. Sahni. Special issue on heterogeneous processing. *Journal of Parallel and Distributed Computing*, 21(3), 1994.
9. M. Garey and D. Johnson. *Computers and intractability*. Freeman, 1979.
10. R. Graham, J. Lawler, E. Lenstra, and A. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Ann of Discrete Mathematics*, 5:287–326, 1979.
11. Y. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computer Surveys*, 31(4):406–471, 1999.
12. J. Leung, L. Kelly, and J. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., Boca Raton, FL, USA, 2004.
13. S. Nesmachnow and M. Canabé. GPU implementations of scheduling heuristics for heterogeneous computing environments. In *Proceedings of the XVII Congreso Argentino de Ciencias de la Computación*, 2011.
14. S. Nesmachnow, H. Cancela, and E. Alba. A parallel micro evolutionary algorithm for heterogeneous computing and grid scheduling. *Applied Soft Computing*, 12(2):626–639, 2012.
15. S. Nesmachnow, F. Luna, and E. Alba. An efficient stochastic local search for heterogeneous computing scheduling. In *15th International Workshop on Nature Inspired Distributed Computing*, 2012.

16. nVidia. CUDA website. Available online `http://www.nvidia.com/object/cuda_home.html`, 2010. Accessed on July 2011.
17. F. Pinel, B. Dorronsoro, and P. Bouvry. Solving very large instances of the scheduling of independent tasks problem on the GPU. *Journal of Parallel and Distributed Computing*, 2012. In press, DOI:10.1016/j.jpdc.2012.02.018.
18. F. Pinel, J. Pecero, P. Bouvry, and S. U. Khan. A two-phase heuristic for the scheduling of independent tasks on computational grids. In *2011 International Conference on High Performance Computing and Simulation (HPCS)*, pages 471 – 477, 2011.
19. G. Ritchie and J. Levine. A fast, effective local search for scheduling independent jobs in heterogeneous computing environments. In *Proceedings of the 22nd Workshop of the UK Planning and Scheduling Special Interest Group*, 2003.

HPCLatAm 2012, pp. 112-118 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

Biclustering of very large datasets with GPU tecnology using CUDA

Javier Arnedo-Fdez, Igor Zwir*, and Rocío Romero-Zaliz

Dpt. of Computer Science and Artificial Intelligence
University of Granada
Spain
{arnedo,igor,rocio}@decsai.ugr.es

Abstract. In this work we report our first research steps on using GPUs to accelerate biclustering of very large data sets, which are common in real world applications such as biomedical and biotechnological. The bicluster problem is NP-hard, thus, finding an optimal solution could be time consuming, especially when dealing with large data sets. We present a GPU-accelerated implementation of the biclustering probabilistic move-based algorithm called FLOC, which can efficiently and accurately approximate biclusters with low mean squared residues without the impact of random interference. Results show that when the size of the dataset increases, the GP-GPU version of FLOC solves the biclustering problem much faster than the CPU FLOC version running on a single CPU core.

Keywords: Data-mining, Bioinformatics, Biclustering, Parallel algorithm, GPU, CUDA, GP-GPU.

1 Introduction

There has been a substantial interest in scientific and engineering computing community to speed up the CPU-intensive tasks on graphical processing units (GPUs) with the development of General Purpose GPU (GP-GPU) systems, since GPUs have a very large memory bandwidth and computational power. Cluster analysis is a widely used technique for grouping a set of objects into classes of similar objects and commonly used in many fields such as data mining, pattern recognition and bioinformatics [1, 2] and a suitable application for the GPUs intensive power of calculation. A special case of clustering is *biclustering* where there is a simultaneously grouping of rows and columns to uncover submatrices of a given data matrix that optimize a desired objective function [3].

There are many biological applications of biclustering algorithms mainly focused on DNA microarray studies and ranges from gene sample classification, genetic pathways identifcation, gene co-regulation study, transcriptional regulatory modules identification, biomarkers discovery, drug design, single nucleotide

* This work is supported by University of Granada - GREIB.PT.2011.20 - GREIB.AL.2011.06.

HPCLatAm 2012, pp. 112-118 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

polymorphism (SNP) analysis, and genetic interactions identification [4]. Even-though microarray studies are beign replaced by newer and sophisticated methods, like next generation sequencing (NGS), biclustering techniques are still a useful tool for their analysis [5, 6].

Although there is a need to build faster biclustering algorithms that can deal with very large datasets, there is, to our best knowledge, only one biclustering GP-GPU implementation published [7].

2 Background

2.1 Parallel architectures

While conventional CPU clusters still dominate the High-Performance Computing (HPC) market, GPUs are gaining popularity as cost-effective HPC accelerators. GPUs provide a huge amount of fine-grain parallelism, since thousands of threads may be running concurrently [7].

GP-GPU systems have become increasingly popular in recent years as a means of delivering large computational power to the desktop market. Such systems consist of a host CPU with the GPU connected through a PCI-Express link. GPUs support high computational rates (in terms of floating point operations per second) and have a high bandwidth to memory on the GPU board. This makes such systems ideal for throughput oriented applications [8].

Special libraries and packages were developed for building GP-GPU system, like the API extension the C programming language called CUDA [9] (Compute Unified Device Architecture) for NVIDIA cards and OpenCL [10]. In this work we will use CUDA to code normal C functions and run them on the GPU's stream processors, thus taking advantage of a GPU's ability to operate on large matrices in parallel, while still making use of the CPU when appropriate.

2.2 Biclustering

In gene expression analysis a bicluster is defined as a submatrix spanned by a set of genes and a set of samples. Alternatively, a bicluster may be defined as the corresponding gene and sample subsets [11].

The concept of *bicluster* was introduced by Cheng and Church [12] to capture the coherence of a subset of genes and a subset of conditions. Unlike previous methods that treat similarity as a function of pairs of genes or pairs of conditions, the bicluster model measures coherence within the subset of genes and condition. The coherence score is defined as a symmetric function of genes and conditions involved and thereby the biclustering is a process of simultaneous grouping of genes and conditions. The so called mean squared residue was employed and applied to expression data transformed by a logarithm and augmented by the additive inverse. While the mean squared residue represents the variance of the selected genes and conditions with respect to the coherence, the goal of biclustering is to find biclusters with low mean squared residue [13]. It has been proven

HPCLatAm 2012, pp. 112-118 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

that the problem of finding biclusters satisfying these criteria is NP-hard in general. Therefore, a set of heuristic algorithms were designed by Cheng and Church [12] to either find one bicluster or a set of biclusters which consist of iterations of masking null values and discovered biclusters, coarse and fine node deletion, node addition, and the inclusion of inverted data.

2.3 FLOC

Cheng and Church original heuristics suffer from some serious drawback that produce the masking of null values and discovered biclusters with random numbers that may result in the phenomenon of *random interference* which in turn impacts the discovery of high quality biclusters. To address this issue and to further accelerate the biclustering process, a probabilistic move-based algorithm called FLOC [13] was developed for generalizing the model of bicluster to incorporate null values that can discover a set of possibly overlapping biclusters simultaneously.

The data is represented in the form of a matrix where the rows correspond to the genes and the columns correspond to the conditions. The FLOC biclustering algorithm starts from a set of seeds (initial biclusters) and carries out an iterative process to improve the overall quality of the biclustering. At each iteration, each row and column is moved among biclusters to produce a better biclustering in terms of lower mean squared residues. The best biclustering obtained during each iteration will serve as the initial biclustering for the next iteration. The algorithm terminates when the current iteration fails to improve the overall biclustering quality [13].

3 GP-GPU implementation of FLOC

The FLOC algorithm is implemented inside a Bioconductor package called BicARE [14]. The complexity of this FLOC algorithm implementation is $O((N + M)^2 \times k \times p)$, where N and M are the number of rows and columns of the original data matrix D, while k is the number of the biclusters to search for and p is the number of iterations till termination [13]. Although the FLOC biclustering method is faster than the original Cheng and Church approach, when N and M are large the FLOC algorithm performance is quite slow for real world applications where the number of genes and conditions can be very high.

To accelerate the FLOC algorithm we decided to implement a GPU version of the original algorithm based on the CUDA programming model from NVIDIA [9].

To detect which function or piece of code were the most time consuming, we perfomed a profiling of the FLOC algorithm. The funtion which calculates of the *residue* of a bicluster (see Definitions 1, 2, 3) is called several times during the execution of the algorithm, each time performing $O((M + N) \times k)$ operations.

Definition 1. *The residue of an entry d_{ij} of data matrix D in a bicluster (I, J), where $I \subseteq \{1, \ldots, M\}$ subset of genes, $J \subseteq \{1, \ldots, N\}$ subset of condition, is*

HPCLatAm 2012, pp. 112-118 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

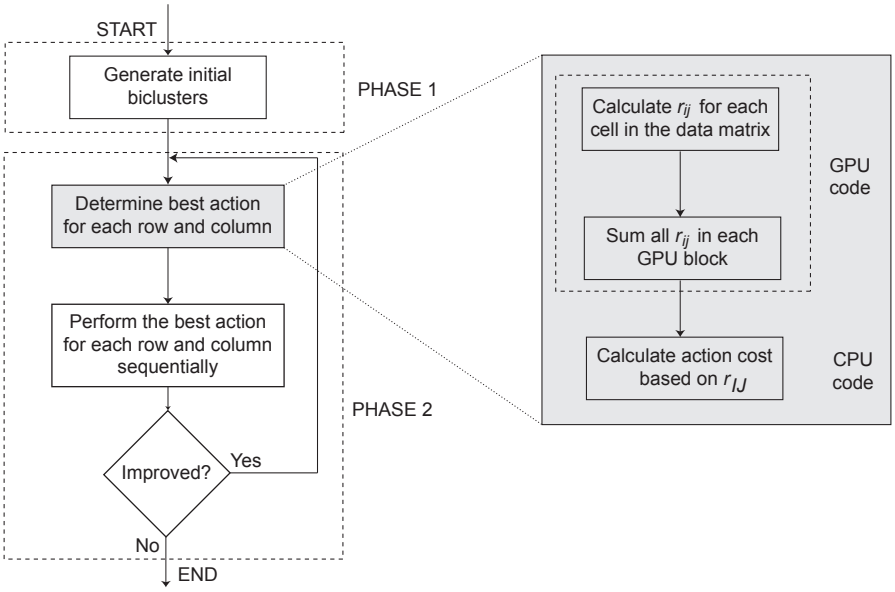


Fig. 1. GP-GPU FLOC flowchart. In gray we show the section of the flowchart that is parallelized using GP-GPU.

$r_{ij} = d_{ij} - d_{iJ} + d_{IJ}$ *if* d_{ij} *is specified in the bicluster, else* $r_{ij} = 0$. d_{iJ} *stands for the sum of all* d_{ij} *in* J, *while* d_{IJ} *stands for the sum of all* d_{ij} *for all* I *and* J.

Definition 2. *The volume* v_{IJ} *of a bicluster* (I, J), *where* $I \subseteq \{1, \ldots, M\}$ *subset of genes,* $J \subseteq \{1, \ldots, N\}$ *subset of condition, is defined as the number of specified entries* d_{ij} *such that* $i \in I$ *and* $j \in J$.

Definition 3. *The residue of a bicluster* (I, J) *is* $r_{I,J} = \frac{\sum_{i \in I, j \ in J} r_{ij}^2}{v_{IJ}}$, *where* $I \subseteq \{1, \ldots, M\}$ *subset of genes,* $J \subseteq \{1, \ldots, N\}$ *subset of conditions,* r_{ij} *is the residue of the entry* d_{ij} *and* v_{ij} *is the volume of the bicluster.*

To accelerate the calculus of the residue fuction we created a function to be executed in each core of a GPU producing the calculation for a specific cell of the data matrix and its posterior sum. Each cell (i, j) of the data matrix calculates r_{ij}^2. Afterwards, the sum of all cell residues are performed in the same GPU device for each block independently, avoiding the overhead of transmitting all data from device to CPU. Finally, in the CPU the sum of every block is calculated and divided by the volume of the bicluster v_{IJ} (Figure 1).

4 Experiments and Results

Several experiments were performed to analyze the performance of our GPU-FLOC implementation.

HPCLatAm 2012, pp. 112-118 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

First, we wanted to see if the size of the data matrices used for biclustering was actually an issue as we thought it would be. Therefore, we fixed all FLOC parameters but the size of the data matrices and run the original and improved FLOC algorithms. We created randomized matrices for sizes 10×10, 50×50, 100×100, 200×200, 300×300, 500×500, 1000×1000 and 2000×2000. For each size we created 5 different random matrices, obtaining 40 matrices in total. Figure 2 shows the results obtained using this synthetic dataset. Each point in the plot represents the mean time spent in the biclustering calculation for each matrix using different sizes. We can infer from this experiment that for small matrices (Figure 2(a)) the CPU FLOC version is faster, but when the size of the dataset is above aproximately 250×250 the GPU-FLOC version is much more efficient. All the previous experiments using the GPU-FLOC algorithm were run using 256 threads in each block of a GPU device.

(a) CPU FLOC

Matrix size	Time consumption (s)	
	Mean	*Standard Deviation*
10×10	0.01	1.79e-03
50×50	1.32	2.43e-02
100×100	10.06	4.47e-03
200×200	79.10	2.53e-01
300×300	266.34	6.79e-01
500×500	1233.60	1.41e+00
1000×1000	9859.20	8.47e+00
2000×2000	80152.90	1.56e+02

(b) GP-GPU FLOC

Matrix size	Time consumption (s)	
	Mean	*Standard Deviation*
10×10	2.40	6.98e-02
50×50	15.36	1.23e-01
100×100	29.47	2.18e-01
200×200	100.88	2.54e-01
300×300	170.40	4.34e-01
500×500	483.94	8.48e-01
1000×1000	2050.00	3.15e+00
2000×2000	10449.70	1.47e+01

Table 1. Statistics for the performed experiments.

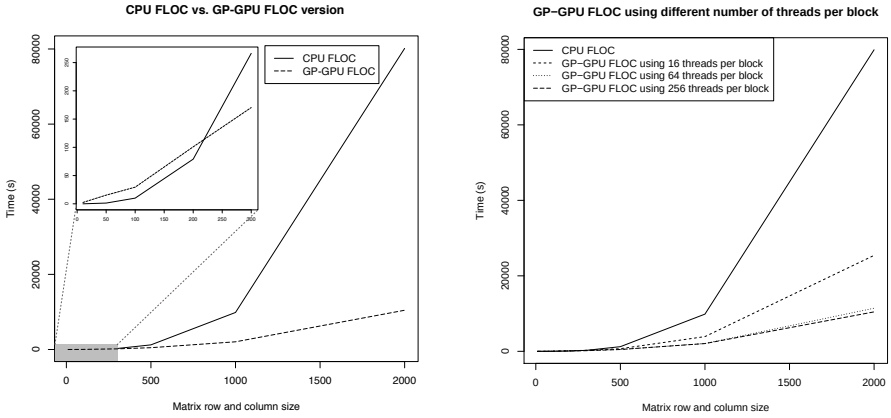
Second, we wanted to test which thread and block configuration was the best option and whether changing these parameters made a substantial difference in performance. The experimental framework used fixed all FLOC parameters but the number of threads per block. Figure 2(b) shows the results. As we expected, when the number of threads per block increases the performace of the GPU implementation is faster, but when the number of threads is quite high the gain is negligible. Nevertheless, using the slowest configuration of the GP-GPU FLOC (i.e., 16 threads per block), its performance is much better than the CPU FLOC algorithm.

All experiments were run in an Intel i7 980 machine with 16 GB of RAM and Gainward GeForce GTX 480 video cards with 1.5 GB of RAM each.

5 Discussion

Preliminary results show that the use of GPU acceleration can substantially improve the performance of biclustering methods. This improvement will help

HPCLatAm 2012, pp. 112-118 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina



(a) CPU FLOC vs. GP-GPU FLOC version.

(b) GP-GPU FLOC using different number of threads per block.

Fig. 2. CPU FLOC vs. GP-GPU FLOC version. Matrix size ranges from 10×10 to 2000×2000. The number of biclusters searched were 10 and 50 iterations were performed, for both versions.

bioinformatic software to cope with the large amount of data that NGS technology is providing.

Memory transfers from the host CPU to the GPU devices over the PCI-Express bus is the main issue when programming GPU applications. The bandwidth of PCI-Express is much lower compared to the on-board memory bandwidth. This can then become the bottleneck of the system, especially if large amounts of data need to be transferred over the bus [9]. It is therefore critical to minimize the total data that is sent back and forth from CPU to GPU memory. Also, there is a limit in the number of threads per block and blocks per grid that has to be considered. Not all algorithms are suitable for GPU acceleration, whatsmore a wrong implementation can cause the GP-GPU algorithm to be even slower than the CPU version.

Future work will be devoted to test all possible GPU parameter's configuration including the use of more than one GPU, and to compare them with other parallel architectures like MPI or PVM [15, 16].

References

1. Yildirim, A.A., Ozdoğan, C.: Parallel wavelet-based clustering algorithm on gpus using cuda. Procedia Computer Science **3**(0) (2011) 396 – 400
2. Petros, X., Nikita, B., Neng, F., Panos M, P. In: Biclustering: Algorithms and Application in Data Mining. John Wiley & Sons, Inc. (2010)
3. Busygin, S.: Biclustering in data mining. Computers & Operations Research **35**(9) (2008) 2964–2987

HPCLatAm 2012, pp. 112-118 (full paper)
ISSN 2422-5207
Session: GPU computing
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

4. Liu, L., Wei, D., Li, Y.: Handbook of Research on Computational and Systems Biology: Interdisciplinary Applications. Igi Global (2011)
5. Wang, J., Tan, A., Tian, T.: Next Generation Microarray Bioinformatics: Methods and Protocols. Methods in Molecular Biology. Springer Verlag (2011)
6. Huang, Q., Wu, L.Y., Qu, J.B., Zhang, X.S.: Analyzing time-course gene expression data using profile-state hidden Markov model. In: IEEE International Conference on Systems Biology. (2011)
7. Mejía-Roa, E., García, C., Gómez, J., Prieto-Matías, M., Nogales-Cadenas, R., Tirado, F., Pascual-Montano, A.D.: Biclustering and classification analysis in gene expression using nonnegative matrix factorization on multi-gpu systems. In: 11th International Conference on Intelligent Systems Design and Applications. (2011)
8. Satish, N., Sundaram, N., Keutzer, K.: Optimizing the use of gpu memory in applications with large data sets. In: HiPC. (2009) 408–418
9. Kirk, D.B., Hwu, W.m.W.: Programming Massively Parallel Processors: A Hands-on Approach. 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2010)
10. Khronos OpenCL Working Group: The OpenCL Specification, version 1.0.29. (8 December 2008)
11. Tanay, A., Sharan, R., Shamir, R.: Biclustering Algorithms: A Survey. Handbook of Computational Molecular Biology (2004)
12. Cheng, Y., Church, G.M.: Biclustering of expression data. Proceedings / ... International Conference on Intelligent Systems for Molecular Biology ; ISMB. International Conference on Intelligent Systems for Molecular Biology **8** (2000) 93–103
13. Yang, J., Wang, H., Wang, W., Yu, P., Ibm, U., Chapel, U., Ibm, H., Watson, T.J., Watson, T.J.: Enhanced biclustering on expression data. In: Proc. of 3rd IEEE Symposium on BioInformatics and BioEngineering (BIBE03. (2003) 321–327
14. Gestraud, P., Brito, I., Barillot, E.: Bicare: Biclustering analysis and results exploration (2010)
15. Pacheco, P.S.: Parallel programming with MPI. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996)
16. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V.S.: PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing. Scientific and engineering computation. (1994)

HPCLatAm 2012, pp. 119-132 (full paper)
ISSN 2422-5207
Session: Applications
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

Optimizing Latency in Beowulf Clusters

Rafael Garabato[1], Andrés More[1 2], and Victor Rosales[1]

[1] Argentina Software Design Center (ASDC - Intel Córdoba)
[2] Instituto Universitario Aeronáutico (IUA)

Abstract. This paper discusses how to decrease and stabilize network latency in a Beowulf system. Having low latency is particularly important to reduce execution time of High Performance Computing applications. Optimization opportunities are identified and analyzed over the different system components that are integrated in compute nodes, including device drivers, operating system services and kernel parameters.

This work contributes with a systematic approach to optimize communication latency, provided with a detailed checklist and procedure. Performance impacts are shown through the figures of benchmarks and mpiBLAST as a real-world application. We found that after several straightforward optimizations on default configuration Gigabit Ethernet latency was reduced from about 50 μs of communication latency. Using different techniques, it is possible to get as low as nearly 20 μs.

1 Introduction

1.1 Beowulf Clusters

Instead of purchasing an expensive and high-end symmetric multiprocessing (SMP) system, most scientists today choose to interconnect multiple regular-size commodity systems as a means to scale computing performance and gain the ability to resolve bigger problems without requiring heavy investments [1] [2] [3].

The key driving factor is cost, hence out-of-the-box hardware components are used together with open source software to build those systems. In the specific case of academia, open source software provides the possibility to make software stack modifications, therefore enabling innovation and broadening their adoption.

Clusters are nearly ubiquitous at the Top500 ranking listing most powerful computer systems worldwide, clustered systems represent more than 80% of the list (Figure 1).

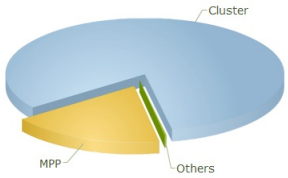


Fig. 1. Top500 List by Architecture (as of November 2011)

As the cheapest network fabrics are the ones being distributed on-board by system manufacturers, Ethernet is the preferred communication network in Beowulf clusters. At the moment Gigabit Ethernet is included integrated on most hardware.

1.2 Latency

Latency itself can be measured at different levels, in particular communication latency is a performance metric representing the time it takes for information to flow from one compute node into another. It then becomes not only important to understand how to measure the latency of the cluster but also to understand how this latency affects the performance of High Performance applications [4].

In the case of latency-sensitive applications, messaging needs to be highly optimized and even be executed over special-purpose hardware. For instance latency directly affects the synchronization speed of concurrent jobs in distributed applications, impacting their total execution time.

1.3 Related Work

There are extensive work on how to reduce communication latency [5] [6]. However, this work contributes not with a single component but with a system wide point of view.

The top supercomputers in the world report latencies that commodity systems cannot achieve (Figure 2). They utilize specially built network hardware, where the cost factor is increased to get lower latency.

System	Latency	Description
HP BL280cG65	0.49 μsec	Best Latency
Fujitsu K Computer	6.69 μsec	Top system

Fig. 2. Latency at the HPCC ranking

High performance network technology (like InfiniBand [7]) is used in cases were Ethernet cannot meet the required latency (see reference values in Figure

3). Some proprietary network fabrics are built together with supercomputers when they are designed from scratch.

Latency	Technology
30-125 μsec	1Gb Ethernet
5-30 μsec	10Gb Ethernet

Fig. 3. System Level Ethernet Latency

1.4 Problem Statement

The time it takes to transmit on a network can be calculated as the required time a message information is assembled and dissembled plus the time needed to transmit message payload. Equation 1 shows the relation between these startup plus throughput components for the transmission of n bytes.

$$t(n) = \alpha + \beta \times n \tag{1}$$

In the hypothetical case where *zero bytes* are transmitted, we can get the minimum possible latency on the system (Equation 2). The value of α is also known as the theoretical or zero-bytes latency.

$$t(0) = \alpha \tag{2}$$

It is worth noticing that α is not the only player in the equation, $1/\beta$ is called network bandwidth, the maximum transfer rate that can be achieved. β is the component that affects the overall time as a function of the package size.

2 Benchmarking Latency

There are different benchmarks used to measure communication latency.

2.1 Intel MPI Benchmarks

The Intel MPI Benchmarks (IMB) are a set of timing utilities targeting most important Message Passing Interface (MPI) [8] functions. The suite covers the different versions of the MPI standard, and the most used utility is Ping Pong.

IMB Ping Pong performs a single message transfer exercise between two active MPI processes (Figure 4). The action can be run multiple times using varying message lengths, timings are averaged to avoid measurement errors.

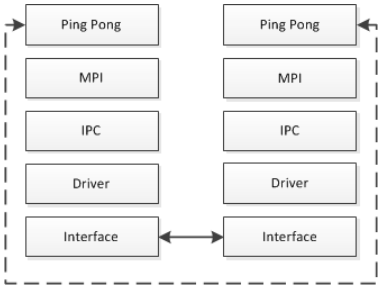


Fig. 4. IMB Ping Pong Communication

Using only MPI basic routines, a package is sent (`MPI_SEND`) from a host system and received (`MPI_RECV`) on a remote one (Figure 5) and the time is reported as half the time in μs for an X long bytes (`MPI_BYTE`) package to complete a round trip.

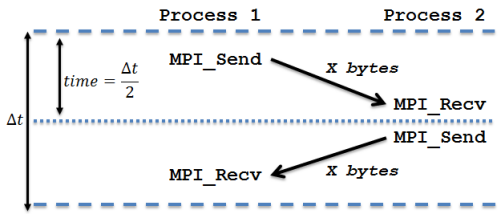


Fig. 5. IMB Ping Pong Benchmark

As described by the time formula at Equation 1, different measures of transmission time are obtained depending on the package size. To get the minimum latency an empty package is used.

2.2 Other Benchmarks

There are other relevant HPC benchmarks that are usually used to exercise clusters: HPL and HPCC. These exercise the system from an application level, integrating all components performance for a common goal.

It is worth mentioning that there are other methods that work at a lower level of abstraction, for instance using Netperf [11] or by following RFC 2544 [12] techniques. However these last two measure latency at network protocol and device level respectively.

High Performance Linpack High Performance Linpack is a portable benchmark for distributed-memory systems doing pure matrix multiplication [9]. It

provides a testing and timing tool to quantify cluster performance. It requires MPI and BLAS supporting libraries.

High Performance Computing Challenge Benchmarks The HPC Challenge benchmark suite [10] packages 7 benchmarks:

HPL: measures floating point by computing a system of linear equations.
DGEMM: measures the floating point rate of execution of double precision real matrix-matrix multiplication.
STREAM: measures sustainable memory bandwidth.
PTRANS: computes a distributed parallel matrix transpose
RandomAccess: measures random updates of shared distributed memory
FFT: double precision complex one-dimensional discrete Fourier transform.
b_eff: measures both communication latency and bandwidth

HPL, DGEMM, STREAM, FFT run in parallel in all nodes, so they can be used to check if cluster nodes are performing similarly. PTRANS, RandomAccess and b_eff exercise the system cluster wide. It is expected that latency optimizations impact their results differently.

3 Methods

Given a simplified system view of a cluster, there are multiple compute nodes that together run the application. An application uses software such as libraries that interface with the operating system to reach hardware resources through device drivers. This work analyzes the following components:

Ethernet Drivers: interrupt moderation capabilities
System Services: interrupt balancing and packet-based firewall
Kernel Settings: low latency extensions on network protocols

Further work to optimize performance is always possible; only the most relevant optimizations were considered according to gathered experience over more than 5 years on the engineering of volume HPC solutions.

3.1 Drivers

As any other piece of software, device drivers implement algorithms which, depending on different factors, may introduce latency. Drivers may even expose hardware functionalities or configurations that could change the device latency to better support the Beowulf usage scenario.

Interrupt Moderation Interrupt moderation is a technique to reduce CPU interrupts by caching them and servicing multiple ones at once [13]. Although it make sense for general purpose systems, this introduces extra latency, so Ethernet drivers should not moderate interruptions when running in HPC clusters.

To turn off Interrupt Moderation on Intel network drivers add the following line on each node of the cluster and reload the network driver kernel module. Refer to documentation [15] for more details.

```
# echo "options e1000e InterruptThrottleRate=0" > /etc/modprobe.conf
# modprobe -r e1000e && modprobe e1000e
```

For maintenance reasons some Linux distributions do not include the configuration capability detailed above. In those cases, the following command can be used to get the same results.

```
# ethtool eth0 rx-usecs
```

There is no portable approach to query kernel modules configurations in all Linux kernel versions, so configuration files should be used as a reference.

3.2 Services

Interrupt Balancing Some system services may directly affect network latency. For instance *irqbalance* job is to distribute interrupt requests (IRQs) among processors (and even between each processor cores) on a *Symmetric Multi-Processing* (SMP) system. Migrating IRQs to be served from one CPU to another is a time consuming task that although balance the load it may affect overall latency.

The main objective of having such a service is to balance between power-savings and optimal performance. The task it performs is to dynamically distribute workload evenly across CPUs and their computing cores. The job is done by properly configuring the IO-ACPI chipset that maps interruptions to cores.

An ideal setup will assign all interrupts to the cores of a same CPU, also assigning storage and network interrupts to cores near the same cache domain. However this implies processing and routing the interrupts before running them, which has the consequence of adding a short delay on their processing.

Turning off the *irqbalance* service will help then to decrease network latency. In a Red Hat compatible system this can be done as follows:

```
# service irqbalance stop
# chkconfig irqbalance off
$ service irqbalance status
```

Firewall As compute nodes are generally isolated on a private network reachable only through the head node, the firewall may not even be required. The system firewall needs to review each package received before continuing with the execution. This overhead increases the latency as incoming and outgoing packet fields are inspected during communication.

Linux-based systems have a firewall in its kernel that can be controlled throughout a user-space application called *iptables*. This application runs in the system as a service, therefore the system's service mechanisms has to be used to stop it.

```
# service iptables stop
# chkconfig iptables stop
$ lsmod | grep iptables
```

3.3 Kernel Parameters

The Linux Transport Control Protocol (TCP) stack makes decisions by default that favors higher throughput as opposed to low latency. The Linux TCP stack implementation has different packet lists to handle incoming data, the PreQueue can be disabled so network packets will go directly into the Receive queue. In Red Hat compatible systems this can be done with the command:

```
# echo 1 > /proc/sys/net/ipv4/tcp_low_latency
$ sysctl -a | grep tcp_low_latency
```

There are others parameters that can be analyzed [14], but the impact they cause are too application specific to be included on a general optimization study.

4 Optimization Impact

4.1 IMB Ping Pong

Using IMB Ping Pong as workload, the following results (Figure 6) reflect how the different optimizations impact communication latency. The actual figures on average and deviation are shown below at Figure 7.

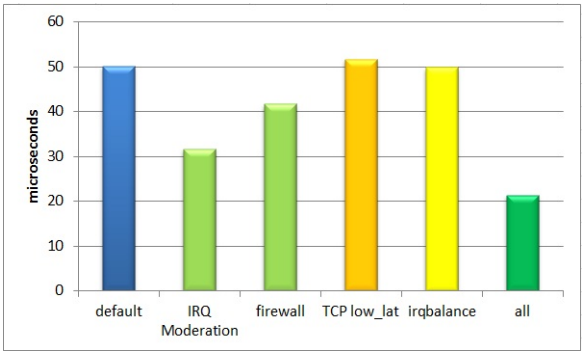


Fig. 6. Comparison of Optimizations

Optimization	$\tilde{x}$ (σ^2)	Impact
Default	50.03 (4.31)	N/A
IRQ Moderation	31.63 (0.83)	36.79%
Firewall	41.62 (8.90)	16.82 %
TCP LL	51.59 (8.22)	-3.11%
IRQ Balance	49.72 (9.68)	0.62 %
Combined	21.31 (2.09)	57.40 %

Fig. 7. IMB Ping Pong Optimization Results

The principal cause of overhead in communication latency is then IRQ moderation. Another important contributor is the packet firewall service. We found that the low latency extension for TCP was actually slightly increasing the IMB Ping Pong reported latency. In the case of the IRQ balance service, the impact is only minimal.

Optimizations impact vary, and not surprisingly they are not accumulative when combining them all. At a glance, it is possible to optimize the average latency in nearly 54%, nearly halving result deviations.

4.2 High Performance Linpack

A cluster-wide HPL running over MPI reported results as shown in Figure 8. The problem size was customized to `Ns:37326 NBs:168 Ps:15 Qs:16` for a quick but still representative execution with a controlled deviation.

Optimization	Wall-time	Gflops
Default	00:20:46	0.02921
Optimized	00:09:03	0.07216

Fig. 8. HPL Results

As we can see on the results, the actual synchronization cycle done by the algorithm heavily relies on having low latency. The linear system is partitioned in smaller problem blocks which are distributed over a grid of processes which may be on different compute nodes. The distribution of matrix pieces is done using a binary tree among compute nodes with several rolling phases between them. The required time was then reduced 56%, and the gathered performance was increased almost 2.5 times.

4.3 HPCC

Figures 9 and 10 show HPCC results obtained with a default and optimized Beowulf cluster. As we can see on the results, the overall execution time is directly affected with a 29% reduction. The performance figures differ across packaged benchmarks as they measure system characteristics that are affected by latency in diverse ways.

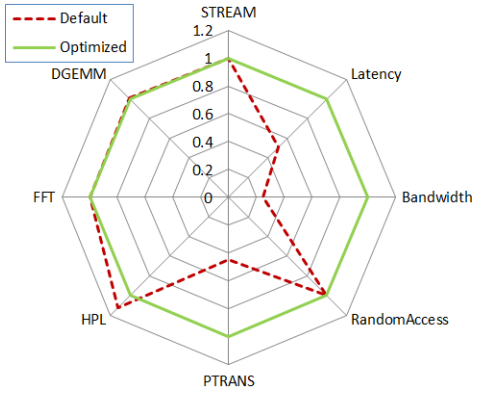


Fig. 9. HPCC Performance Results (higher is better)

Optimization	Wall-time
Default	00:10:32
Optimized	00:07:27

Fig. 10. HPCC Timing Results

Local benchmarks like STREAM, DGEMM and HPL are not greatly affected, as they obviously do not need communication between compute nodes. However, the actual latency, bandwidth and PTRANS benchmark are impacted as expected due they communication dependency.

4.4 mpiBLAST

In order to double check if any of the optimization have hidden side effects and the real impact on the execution of a full-fledge HPC application, a real-world code was exercised. mpiBLAST [16] is an open source tool that implements DNA-related algorithms to find regions of similarity between biological sequences.

Figure 11 shows the actual averaged figures after multiple runs. Results got with a default and optimized system on a fixed workload for mpiBLAST. The required time to process the problem was reduced by 11% with the previous 42% improvement as measured by IMB Ping Pong.

Optimization	Wall-time
Default	534.33 seconds
Optimized	475.00 seconds

Fig. 11. mpiBLAST Results

This shows that the results of a synthetic benchmark like IMB Ping Pong can not be used directly to extrapolate figures, they are virtually the limit to what can be achieved by an actual application.

4.5 Testbed

The experiments done as part of this work were done over 32 nodes with the following bill of materials (Figure 12).

Component	Description
Server Board	Intel(R) S5000PAL
CPU	Intel(R) Xeon(R) X5355 @ 2.66GHz
Ethernet controller	Intel(R) 80003ES2LAN (Copper) (rev 01)
RAM Memory	4 GB DDR2 FB 667 MHz
Operating System	Red Hat Enterprise 5.5 (Tikanga)
Network Driver	Intel(R) PRO/1000 1.2.20-NAPI
Ethernet Switch	Hewlett Packard HPJ4904A

Fig. 12. Compute Node Hardware and Software

HPCLatAm 2012, pp. 119-132 (full paper)
ISSN 2422-5207
Session: Applications
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

5 Optimization Procedure

Figure 13 summarizes the complete optimization procedure. It is basically a sequence of steps involving checking and reconfiguring Ethernet drivers and system services if required. Enabling TCP extensions for low latency is not included due their negative consequences.

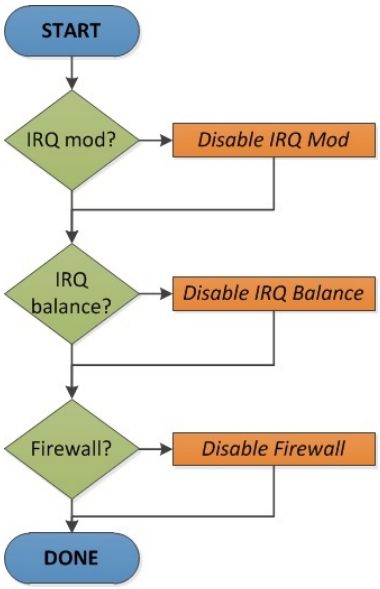


Fig. 13. Latency Optimization Procedure

5.1 Detailed Steps

The steps below include the purpose and an example of the actual command to execute as required on Red Hat compatible systems. The pdsh[3] parallel shell is used to reach compute nodes at once.

Questions (1) helps to dimension the required work to optimize driver configuration to properly support network devices. Questions (2) helps to understand what's needed to properly configure system services.

1. Interrupt Moderation on Ethernet Driver
   (a) Is the installed driver version the latest and greatest?

```
$ /sbin/modinfo -F version e1000e
1.2.20-NAPI
```

[3] `http://sourceforge.net/projects/pdsh`

(b) Is the same version installed across all compute nodes?

```
$ pdsh -N -a '/sbin/modinfo -F version e1000e' | uniq
1.2.20-NAPI
```

(c) Are interrupt moderation settings in HPC mode?

```
# pdsh -N -a 'grep "e1000e" /etc/modprobe.conf' | uniq
options e1000e InterruptThrottleRate=0
```

2. System Services

(a) Is the firewall disabled?

```
# pdsh -N -a 'service iptables status' | uniq
Firewall is stopped.
```

(b) Is the firewall disabled at startup?

```
# pdsh -N -a 'chkconfig iptables --list'
irqbalance 0:off 1:off 2:off 3:off 4:off 5:off 6:off
```

(c) Was the system rebooted after stopping firewall services?

```
$ uptime
 15:42:29 up 18:49, 4 users, load average: 0.09, 0.08, 0.09
```

(d) Is the IRQ balancing service disabled?

```
# pdsh -N -a 'service irqbalance status' | uniq
irqbalance is stopped
```

(e) Is IRQ balancing daemon disabled at startup?

```
# pdsh -N -a 'chkconfig irqbalance --list' | uniq
irqbalance 0:off 1:off 2:off 3:off 4:off 5:off 6:off
```

Once gathered all the information required to known if optimizations can be applied, the following list can be used to apply configuration changes. Between each change a complete cycle of measurement should be done. This include contrasting old and new latency average plus their deviation using at least IMB Ping Pong.

Disable IRQ Moderation

```
# pdsh -a 'echo "options e1000e InterruptThrottleRate=0" >> \
/etc/modprobe.conf'
# modprobe -r e1000e; modprobe e1000e
```

Disable IRQ Balancer

```
# pdsh -a 'service irqbalance stop'
# pdsh -a 'chkconfig irqbalance off'
```

Disable Firewall

```
# pdsh -a 'service iptables stop'
# pdsh -a 'chkconfig iptables off'
```

HPCLatAm 2012, pp. 119-132 (full paper)
ISSN 2422-5207
Session: Applications
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

6 Conclusion

This work shows that by only changing default configurations the latency of a Beowulf system can be easily optimized, directly affecting the execution time of High Performance Computing applications. As a quick reference, an out-of-the-box system using Gigabit Ethernet has around 50 μs of communication latency. Using different techniques, it is possible to get as low as nearly 20 μs.

After introducing some background theory and supporting tools, this work analyzed and exercised different methods to measure latency (IMB, HPL and HPCC benchmarks). This work also contrasted those methods and provided insights on how they should be executed and their results analyzed.

We identified which specific items have higher impact over latency metrics (interrupt moderation and system services), using de-facto benchmarks and a real-world application such as mpiBLAST.

6.1 Future Work

Running a wider range of real-world computational problems will help to understand the impact in different workloads. A characterization of the impact according to the application domain, profiling information or computational kernel might be useful to offer as a reference.

There are virtually endless opportunities to continue with the research on latency optimization opportunities; among them components like BIOS, firmware, networking switches and routers. An interesting opportunity are the RX/TX parameters of Ethernet drivers that control the quantity of packet descriptors used during communication.

Another option is to implement an MPI trace analysis tool to estimate the impact of having an optimized low latency environment. At the moment there are several tools to depict communication traces (Jumpshot[4], Intel's ITAC[5]), but they do not provide a simulation of what would happen while running over a different network environment. Having this approximation can be useful to decide if it is worth to purchase specialized hardware or not.

Acknowledgments

The authors would like to thanks the Argentina Cluster Engineering team at the Argentina Software Design Center (ASDC Intel) for their contributions.

References

1. T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer, *Beowulf: A parallel workstation for scientific computation*, 1995.

[4] `http://www.mcs.anl.gov/research/projects/perfvis/software/viewers`
[5] `http://software.intel.com/en-us/articles/intel-trace-analyzer`

2. John Salmon, Christopher Stein, Thomas Sterling, *Scaling of Beowulf-class Distributed Systems*, Proceeding of the 1998 ACM/IEEE SC98 Conference, 1998.
3. William Gropp, Ewing Lusk and Thomas Sterling, *Beowulf Cluster Computing with Linux, Second Edition*, 2003.
4. Qlogic, *Introduction to Ethernet Latency* 2011
5. Steen Larsen, Parthasarathy Sarangam, Ram Huggahalli, *Architectural Breakdown of End-to-End Latency in a TCP/IP Network*, 2007.
6. Annie P. Foong, Thomas R. Huff, Herbert H. Hum, Jaidev P. Patwardhant, Greg J. Regnier, *TCP Performance Re-Visited*, 2003.
7. Infiniband Trade Association, *Infiniband Architecture Specification Release 1.2.1*, Jan 2008
8. Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, 2009.
9. A. Petitet, R. C. Whaley, J. Dongarra and A. Cleary, *HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*, 2008.
10. Jack J. Dongarra, Piotr Luszczek, *Overview of the HPC Challenge Benchmark Suite*, 2006.
11. R. Jones, *Netperf*, `http://www.netperf.org`, 2007.
12. S. Bradner and J. McQuaid, *IEEE RFC2544: Benchmarking Methodology for Network Interconnect Devices*, 1999
13. Intel Corporation, *Interrupt Moderation Using Intel Gigabit Ethernet Controllers Application Note*, April 2007.
14. Intel Corporation, *Assigning Interrupts to Processor Cores using an Intel(R) 82575/82576 or 82598/82599 Ethernet Controller*, September 2009.
15. Intel Corporation, *Improving Measured Latency in Linux for Intel(R) 82575/82576 or 82598/82599 Ethernet Controllers* 2009.
16. H. Lin, P. Balaji, R. Poole, C. Sosa, X. Ma, W. Feng, *Massively Parallel Genomic Sequence Search on the Blue Gene/P Architecture* IEEE/ACM SC2008, November 2008.

HPCLatAm 2012, pp. 133-148 (full paper)
ISSN 2422-5207
Session: Applications
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

SMCV: a Methodology for Detecting Transient Faults in Multicore Clusters

Diego Montezanti[1,3], Fernando Emmanuel Frati[1,3], Dolores Rexachs[2], Emilio Luque[2], Marcelo Naiouf[1] and Armando De Giusti[1,3]

[1]Instituto de Investigación en Informática LIDI, Facultad de Informática, UNLP
{dmontezanti,fefrati,mnaiouf,degiusti}@lidi.info.unlp.edu.ar
[2]Departamento de Arquitectura de Computadores y Sistemas Operativos, UAB
{dolores.rexachs,emilio.luque}@uab.es
[3]Consejo Nacional de Investigaciones Científicas y Técnicas

Abstract. The challenge of improving the performance of current processors is achieved by increasing the integration scale. This carries a growing vulnerability to transient faults, which increase their impact on multicore clusters running large scientific parallel applications. The requirement for enhancing the reliability of these systems, coupled with the high cost of rerunning the application from the beginning, create the motivation for having specific software strategies for the target systems. This paper introduces SMCV, which is a fully distributed technique that provides fault detection for message-passing parallel applications, by validating the contents of the messages to be sent, preventing the transmission of errors to other processes and leveraging the intrinsic hardware redundancy of the multicore. SMCV achieves a wide robustness against transient faults with a reduced overhead, and accomplishes a trade-off between moderate detection latency and low additional workload.

Keywords: transient fault, silent data corruption, multicore cluster, parallel scientific application, soft error detection, message content validation, reliability.

1 Introduction

The challenge of improving the computation performance of current processors has been achieved by increasing integration scale, which implies that the number of transistors within chips is growing. Additionally, the increment of the operation frequency has caused a raise in the internal operation temperature. These factors, added to a decrease in input power, cause processors to be more vulnerable to transient faults [14,17].

A transient fault is the consequence of interference from the environment that affects some hardware component in the computer. This can be caused by electromagnetic radiation, overheating, or input power variations, and can temporarily invert one or several bits of the affected hardware element (single bit-flip or multiple bit-flip) [2].

The way in which each transient fault occurs is unique; any given transient fault does not occur exactly the same never again throughout the lifespan of the system. These faults are short-lived and do not affect the regular operation of the system, although they can result in the incorrect execution of an application. Physically, they can be located anywhere in the hardware of the system; in this context, the faults that affect processor registers and logics are critical, since other parts of the system, such as memories, storage devices and buses, have built-in mechanisms (such as ECCs[1] or parity bits) capable of detecting and correcting this type of faults [1].

From the perspective of the program being run, the fault can alter the status of a hardware component that contains important information for the application. Depending on the time and specific location of the fault, it can affect application behavior or results and, therefore, system reliability [3].

The impact of transient faults becomes more significant in the context of HPC. Even if the mean time between faults (MTBF) in a commercial processor is of the order of one every two years, in the case of a supercomputer with hundreds or thousands of processors that cooperate to solve a task, the MTBF decreases as the number of processors increases. Since the year 2000, error reports due to large transient faults in large computers or server groups have become more frequent [1,20]. This situation is worse with the advent of multicore architectures, which incorporate a great degree of parallelism at hardware level. Also, the impact of the faults becomes more significant in the case of longer applications, given the high cost of relaunching execution from the beginning. These factors justify the need for a set of strategies to improve the reliability of high-performance computation systems. In this way, the first step is detecting the faults that affect application results but are not intercepted by the operating system and, therefore, do not cause the application to be aborted.

Traditionally, the existing proposals for providing transient fault tolerance have been divided into those that tackle the problem from a hardware standpoint, and those that do so from an application perspective.

Hardware-based techniques [8,9,11,13] aim to protect the various elements in the processor by adding additional logics to provide redundancy. These are most widely used in critical environments, such as flight systems or high-availability servers, where the consequences of a transient fault can be disastrous.

Hardware-redundancy-based techniques, however, are inefficient in general purpose computers. The cost of designing and verifying redundant hardware is high, and the environmental conditions in which the processors are used and processor ageing are the main causes for faults that cannot be predicted during the development stage. On the other hand, in many applications (audio or video on demand), the consequences of a fault are not as severe, so there is no critical need to add thorough fault-tolerance mechanisms [21].

The compromise between the achieved reliability and the resources involved makes software-redundancy-based strategies [19] to be the most appropriate for general purpose computational systems. The basic idea for detecting faults, called DMR[2],

[1] ECC: Error Correcting Code
[2] DMR: Dual Modular Redundancy

consists in duplicating application computation. Both replicas operate over the same input data and compare their outputs [8,11]. These techniques are characterized by their low cost and flexibility, allowing various configuration options to adapt to specific application needs [4].

An important aspect of detection lies in the validation interval. If results are compared only at the end, the fault that affects the application is detected with little additional workload, but the cost of relaunching the application from the beginning is high, especially in the case of large parallel applications. On the other end, if partial results are validated frequently, a high workload is introduced but the cost of re-executing the application from the last consistent state is lower than in the previous case. Therefore, a compromise must be reached between the detection interval and the additional workload introduced.

There are numerous proposals for detection, based on duplication, designed for serial programs, whose purpose is ensuring execution reliability. From this standpoint, a parallel application can be viewed as a set of sequential processes that have to be protected from the consequences of transient faults by means of the set of adopted techniques.

In this context, SMCV (Sent Message Content Validation) is presented, which is a proposal specifically designed for the detection of transient faults in scientific, message-passing parallel applications that execute on the nodes of a multicore cluster. SMCV uses software techniques that leverage the intrinsic redundancy existing in multicores, replicating each process of the parallel application in a core of the same processor. The detection is performed by validating the contents of the messages to be sent using a moderate validation interval and adding a reduced additional workload and a low overhead with respect to execution time. SMCV is a distributed strategy that improves the reliability of the system (formed by the cluster and the parallel application), isolating the error produced in the context of an application process and preventing it from propagating to the others. The end goal is to ensure that the applications that finish do so with correct results.

The rest of this paper is organized as follows: in Section 2, the theoretical context related to transient faults and their consequences in message-passing parallel applications is reviewed. In Section 3, related work is discussed. Section 4 describes this work´s proposal and explains the choices made. In Section 5, the methodology proposed is described in detail. Section 6 discusses the initial experimental validation. In Section 7, future lines of work are described, and Section 8 presents the conclusions.

2 Background

2.1 Soft Errors. Classification.

The errors (external manifestations of an inconsistent internal status) produced by transient faults are called soft errors. While transient faults affect system hardware, soft errors can be observed from the perspective of program execution.

Figure 1 shows the classification of the possible consequences of transient faults [24].

HPCLatAm 2012, pp. 133-148 (full paper)
ISSN 2422-5207
Session: Applications
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

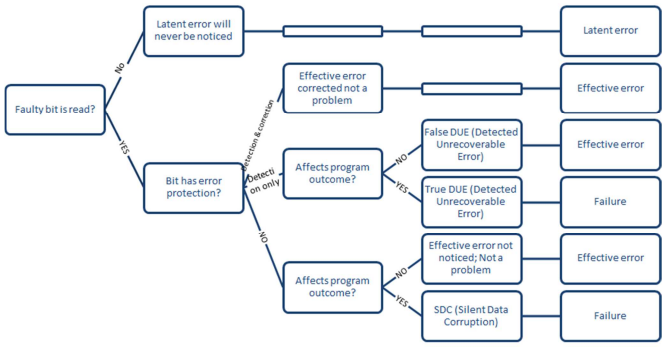


Fig. 1. Classification of possible outcomes of a transient fault (adapted from [24])

The soft error rate (SER) of a system is given by [18]:

$$SER = DUE + SDC + LF \quad (1)$$

A Detected Unrecoverable Error (DUE) is a detected error that has no possibility of recovery. DUEs are a consequence of faults that cause abnormal conditions that are detectable on some intermediate software layer level (e.g. Operating System, communication library). Normally, they cause the abrupt stop of the application. For instance, an attempt to access an illegal memory address (segmentation fault) or an attempt to run an instruction that is not allowed (e.g. zero division).

A Silent Data Corruption (SDC) is the alteration of data during the execution of a program that does not cause a condition that is detectable by system software. Its effects are silently propagated through the execution and cause the final results to be incorrect. From a hardware point of view, this is caused by the inversion of one or several bits of a processor´s register being used by the application, causing the program to generate faulty results.

A Latent Fault (LF) is a fault that corrupts data that are not read or used by the application so, despite the fault effectively happening, it does not propagate through the execution and has no impact on the results.

As a consequence, it is important that strategies are developed to intercept SDCs, which are the most dangerous type of faults that can occur from the point of view of reliability, because the program appears to be running correctly but, upon conclusion, its output will be corrupted.

2.2 Transient Faults in Message Passing Parallel Applications

The occurrence of a transient fault that causes an SDC in a core that is running one of the processes of a message-passing parallel application can have two different consequences:

$$SDC = TDC + FSC \quad (2)$$

A Transmitted Data Corruption (TDC) is an error in which the fault affects data that are part of the contents of a message that has to be passed. If undetected, the corruption is propagated to other processes of the parallel application.

On the other hand, in the case of a Final Status Corruption (FSC), the fault affects data that are not part of the contents of the message, but is propagated locally during the execution of the affected process, corrupting its final state. In this case, the behavior is similar to that of a sequential process.

Since a parallel application consists in the collaboration among multiple processes to perform a task, its success is based on communicating the local computation results obtained by each process to the others. Therefore, all faults that cause a TDC have a high impact on the end results. On the other hand, the faults that cause an FSC are related to the centralized part of the computation, and can therefore be detected by comparing the end results. Following this line, it follows that, if the task is divided among a larger number of processes, there will be a larger number of messages and a consequent growth in the TDC portion.

In this context, SMCV proposes a detection scheme that is focused on those faults that cause TDCs, and adds a final stage for comparing results to ensure system reliability. The solution proposed is discussed in Sections 4 and 5.

3 Related Work

Fault Tolerance (FT) involves three phases: detection, protection and recovery. One of the ideas most commonly used for detecting faults, proposed by Rotenberg [23], is duplicating the execution of a process hosted in a given core, using another core that works as redundancy. Both replicas operate on the same input data, compare their partial results every given period of time, and only one of them writes to memory or sends a message to another process [7,8,9,10,11].

Among the proposals that are based on software redundancy, code duplication, with several variants, has been the idea most widely adopted in the field of transient fault detection. SRT (*Simultaneous & Redundant Threading*) [5] is a first approximation to this, which consists in simultaneously running two replicas of a program as separate threads, dynamically scheduling hardware resources between them, and providing detection through input duplication and output comparison. In [6] CRT (*Chip-level Redundant Threading*) is proposed, which is the application of this technique to CMP environments. SRTR (SRT *with Recovery*) [7] proposes improvements to the detection mechanism and provides recovery through reexecution in the pipeline. CRTR (CRT *with Recovery)* [8] improves detection by separating execution from threads to mask the communication latency between cores, and it applies the recovery mechanisms proposed in [7] for a CMP environment. In [9], DDMR (*Dynamic* DMR) is proposed, a technique in which the cores that run the application in redundant mode are dynamically associated to prevent defective cores from affecting reliability, dealing with processing asymmetries and improving scalability. It introduces the possibility of configuring the system to operate in redundant mode or using

the cores separately for processing. All these solutions involve some modification to system hardware.

In [4], the *Mixed Mode Multicore* model is proposed, which allows running the applications that require reliability in redundant mode and, for applications that require high performance, avoiding this penalty, thus providing flexibility through configuration settings.

In [12], the proposal is obtaining a reduced version of the application by removing inefficient computation and computation related to predictable control flow. The full application and its reduced version are run in separate threads, providing redundancy and advance results that speed up the execution of the application. The authors in [11] propose selecting a core to carry out monitoring tasks over the processes that are run in the other cores, cyclically verifying their states. As an alternative, more than one core can be used for diagnosis operations, and the coverage level in case of faults can be configured, as well as the maximum overhead allowed. Thus, there is no need to produce a full replica of the program.

Among the solutions that are purely based on software, PLR [21] proposes the creation of a set of redundant processes for each application, being transparent to it. The implementation allows the Operating System to intelligently manage available hardware resources. This technique is designed for sequential programs.

In the context of these options, SMCV proposes a detection solution that is specific to message-passing parallel applications, not requiring any hardware modifications and leveraging the redundant resources that already exist in the multicore environment.

4 Work Hypothesis. Proposed Solution

In this section we present the rationale for SMCV. First, the usefulness of validating message contents is explained, and the features provided by the methodology are mentioned. Then, the leverage of redundant hardware resources by SMCV to increase system reliability is described.

4.1 Validating Contents of Sent Messages

The detection methodology proposed in this paper is essentially based on the hypothesis that, in a system formed by a multicore cluster that is running a message-passing parallel application, most of the significant computation (understood as that which impacts application results) will be part of the content of a message that is sent to other application process at some point during execution. Faults can corrupt data, flags, addresses or instruction code. However, if the corrupted value is significant for the results of the application, this situation will eventually be reflected on message incorrectness. Thus, of the total faults that can cause SDC, most will belong to the TDC category. Therefore, to detect faults that corrupt important data, the contents of the messages should be monitored. As regards the sequential phase, during which there are no communications, the end results are verified to ensure reliability.

SMCV is a detection strategy based on validating the contents of the messages to be sent. Each application process is duplicated, and both replicas compare all the fields that form message contents before sending; the message is sent only if the comparison is successful.

This technique allows detecting all faults that cause TDCs; from the point of view of the parallel application, SMCV ensures that any fault that affects the state of a process is not propagated to other process of the application, which confines the effects of the fault to the local process. Faced with an error, SMCV currently notifies the application and produces a safe stop. If a final comparison of the results is added to detect faults in the serial portion, SMCV ensures system reliability and, therefore, that the results of any application that finishes execution are correct.

Message contents are validated before sending the message. Thus, only one of the replicas effectively sends the message, which means that no additional network bandwidth is consumed. Taking into account that current networks have protocols that ensure reliable communications, there is no need to verify the contents of the messages upon reception (which would involve the transmission of two messages).

SMCV provides the following features:

- Each process and its replica are locally validated. The strategy is distributed in each application process. It is decentralized.
- It prevents the propagation of errors among application processes. Also, it detects errors in the serial part of the application by checking the end results.
- It introduces a low overhead in execution time, since only one comparison is added for each byte of each outgoing communication and the end result (it should be noted that the cost of comparison is lower than that of communication).
- A conservative detection strategy, designed for sequential programs, consists in duplicating application computation; to protect program outputs, each memory write operation is checked before being written [8]. Compared with this type of alternatives, SMCV involves a reduced work overload. In this sense, it can be said that it is a lightweight technique.
- When a fault is detected, the application is stopped, allowing relaunching the execution. There is no need to wait for the incorrect stop to re-execute, so SMCV narrows error latency. This carries a gain in reliability, but also in time, which becomes particularly significant in scientific applications that can run for several days.
- SMCV increases system reliability, understood as the number of times the application ends correctly, because it is able to detect faults that cause TDC.
- It achieves a trade-off between detection latency, additional workload and involved resources. SMCV allows latency in detection, since no verification is carried out when the corrupt value is first used. This postpones detection until the time when the altered data are part of the contents of the message. However, this implies a lower additional workload than validating each write operation (which produces low latency with high workload), and better leverages the resources than an only final comparison (which involves duplicating all computation to detect only at the

end, producing high latency with low workload). The less frequent communication between processes, the higher latency and the lower workload.

4.2 Leveraging Redundant Hardware Resources

Hardware manufacturer's trend is to add more cores to processors. However, many applications do not take advantage of all computation resources efficiently. On the other hand, the increase in the amount of transient faults goes hand in hand with the rise in the number of processing cores. As a consequence, the focus is no longer only processor performance, but factors such as reliability and availability have become more relevant. Therefore, the use of cores to carry out tasks related to fault tolerance has advantages both as regards to leveraging these resources as well as adding a beneficial feature for the system.

In this context, SMCV takes advantage of the intrinsic redundancy existing in multicores, using CMP cores to locate the replicas of the processes that perform useful computation for the application. The output to main memory is the critical aspect for selecting the cores that will be used to detect the faults that occur in the others. SMCV tries to exploit the memory hierarchy of the CMP, so that the redundancy of the computation that is executed in any given core is placed in another core with which some level of cache is shared. Thus, many comparisons will be resolved at LLC[3], minimizing main memory access.

5 Proposed Methodology Description

As already explained, SMCV is a software-centric strategy that can detect transient faults in a multicore cluster on which a message-passing scientific parallel application is being run. Upon detection of a fault, a user report is issued and the application is aborted, thus increasing system reliability.

Figure 2 shows an outline of the proposed detection methodology. Each process in the parallel application is run in a core of the CMP, and the computation it carries out is internally duplicated in a thread, which in turn is executed in a core that shares some cache level with the core running the original process. Thus, there is no need to access the main memory, taking benefit from the hierarchy to solve comparisons.

Each process is run concurrently with its replica, which means that a synchronization mechanism is required. When a communication is to be performed (point-to-point or collective), it temporarily stops execution and waits for its replica to reach the same point. Once there, all fields from the message to be sent are compared, byte to byte, to validate that the contents calculated by both replicas are the same. Only if this proves true, one of the replicas sends the message, ensuring that no corrupt data are propagated to other processes.

[3] LLC: Last Level Cache

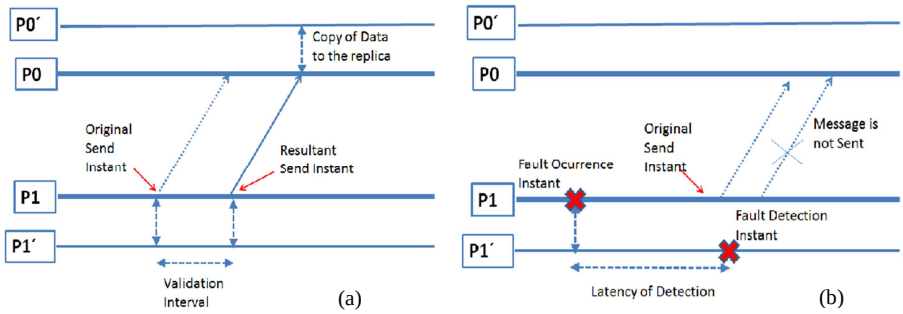


Fig. 2. SMCV methodology. (a) Proposed detection outline. (b) Behavior in presence of faults.

The recipient(s) of the messages stop upon reception and remain on hold. Once received, it copies the contents of the message to its replica (also using memory hierarchy) and both replicas continue with their computation. Assuming that network errors are detected and corrected at the network layer, the validated message reaches its destination uncorrupted. By comparing the message before sending it, the message can be sent only once. Were it be compared on reception, two copies of the message would have to be sent through the network, which would be detrimental to bandwidth use and network fault vulnerability.

Finally, when application execution finishes, the obtained results are checked once to detect faults that may have occurred after communications ended, during the serial part of the application.

5.1 Characterizing SMCV's Additional Workload

Additional workload is related to computing amount added by the fault detection strategy. This metric is useful to compare this methodology with other options. To have an approach, a conservative strategy based on the validation of memory write operations, similar to those used in sequential applications, has been analyzed. In this case, parallel application processes are also duplicated in threads as described, but the results of all write operations are validated (as opposed to validating only the contents of the messages sent). This strategy can detect all faults, but with a significant increase in computation amount.

The work overload W_{WV} introduced by the write validation technique is given by:

$$W_{WV} = (S + M.k).(C_{sync} + C_{comp}) \quad (3)$$

In Equation (3), S represents the number of write operations performed by the application, excluding those corresponding to the messages it sends. It is assumed that the application sends M messages of k elements (average) each. C_{sync} and C_{comp} represent the costs of a synchronization operation and a comparison operation, respectively. The first factor in Equation (3) is therefore the total number of write operations performed by the application. If all write operations are validated, each will involve a synchronization operation and a comparison operation.

On the other hand, the workload added by message validation, W_{MV} is given by:

$$W_{MV} = M \,.\, (C_{sync} + k \,.\, C_{comp}) \quad (4)$$

In the case of message validation, for each message there is an only synchronization operation and *k* comparisons (one for each element in the message).

The relation between the workload introduced by SMCV and a strategy that validates all write operations will then be given by:

$$\frac{W_{MV}}{W_{WV}} = \frac{M \,.\, C_{sync} + M \,.\, k \,.\, C_{comp}}{S \,.\, (C_{sync} + C_{comp}) + M \,.\, k \,.\, C_{sync} + M \,.\, k \,.\, C_{comp}} \quad (5)$$

The quotient of Equation (5) is always a number lower than 1, which means that the additional computation overload for validating messages is lower than that for validating all write operations.

The analysis was carried out for one of the processes that communicate all its results. In the case of a process that performs serial computing, the overload for comparing the end results is added, but this is the same in both techniques. Therefore, this analysis is sufficiently general and representative of various situations.

It can be concluded that SMCV is a lightweight strategy that adds a reduced workload versus more conservative strategies that will detect faults that have no impact on the results of the application.

6 Initial Experimental Validation

The SMCV methodology has been assessed to determine its detection efficacy and the overhead introduced regarding to execution time. The results obtained are shown in this section.

6.1 Testing SMCV's Effectiveness

Tests were run with the detection tool to test its efficacy. The application used for the tests was a parallel matrix multiplication (C = A * B), programmed following the Master/Worker paradigm with 4 processes (the Master and 3 Workers), with the Master also taking part of the computation of the C matrix [22]. The Master process divides matrix A among all Workers and sends each one the chunk assigned to it, keeping a chunk for itself to participate in the calculation of the resulting matrix. Then, the Master sends each Worker a copy of the entire matrix B. After this, all processes compute their corresponding chunk of matrix C and, in the final stage, send the Master the part that they have calculated. The Master builds matrix C from what the Workers sent and its own computation. All messages used are non-blocking. The communications library used is OpenMPI.

All the experiments were run on a cluster with 16 blades, each one having 2 Quad Core Intel Xeon 5405 2GHz processors, 12 MB of L2 cache and 2 GB of main

memory. For this first test, an only blade was used, with the 4 processes and their replicas mapped to the 8 cores of the blade.

To implement SMCV, a part of an MPI communication primitive's library was developed, with the added functionality of fault detection by comparison upon sending, message contents duplication upon reception, and concurrency control between replicas. The Pthreads library was used for creating the replicas, and replica synchronization was done with semaphores.

The SMCV strategy was applied to the described application, replicating each of its processes in a thread as explained in Section 5 (for this, the source code of the application is required). The experiment consisted in injecting faults at various points of the application by means of a debugging tool. To do this, a breakpoint is inserted at a certain point of the execution of one of the application processes; the value of a variable is modified, and computation is resumed, so that the consequence of the fault at the end of the execution can be analyzed (this technique simulates a real fault in a processor register, since for data corruption to become apparent, it must be observable as a difference between the memory states of the replicas).

Even though a transient fault can randomly occur at any point during execution, significant processing time points were selected for the simulated injection, both for the Master and the Workers.

The strategy was capable of detecting all faults that affected message contents (TDC), as expected, notifying and aborting the application so that the corruption was not able to propagate. Thus, all Workers processing is protected. On the other hand, the faults that occurred in the data kept by the Master for local computation, and those that were produced after the partial results from all Workers had been collected by the Master in the last stage (corresponding to the FSC portion) were detected while comparing the end results.

6.2 Overhead Measurements

The overhead is a metric of the incidence of the detection tool on system performance, in the absence of faults. The overhead can be determined as the extra execution time implied by adding the SMCV strategy to the original application, on the architecture described above. The time added by SMCV is a consequence of the duplication of each process, the synchronization between replicas, the comparison carried out before each message is sent, the duplication of the messages received, and the final verification of the results.

Experiments were carried out by applying the SMCV methodology to the matrix multiplication application with 2, 4 and 8 processes (including the Master), with square matrix sizes of 512, 1024, 2048, 4096 and 8192 elements. The mapping between processes and processors was made in a way that ensures the same conditions of execution with and without the SMCV strategy, in order to directly compare the execution times. Up to 4 processes, an only blade was used, running application processes and its replicas using the all 8 cores. In the case of 8 processes, two blades were used, each of them running 4 processes of the application (without SMCV) and 4 processes and its replicas (8 cores) with SMCV.

Each experiment was run five times, and the results were averaged to improve stability. The standardized results, with respect to the execution time of the application with no fault detection, are shown in Table 1 and Figure 3.

	Procesos		
Tam (N)	2	4	8
512	0,87%	14,24%	55,11%
1024	0,01%	1,63%	21,40%
2048	0,39%	1,61%	10,05%
4096	-0,14%	0,91%	4,74%
8192	0,17%	0,92%	2,45%

Table 1. Overhead measurements

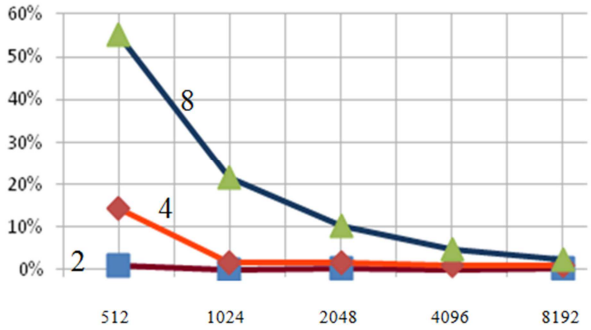


Fig. 3. SMCV's overhead in execution time

As it can be observed, the overhead decreases as the size of the problem grows up. This is because, with larger matrixes, the application spends more time computing. However, for any given number of processes, the number of messages remains constant. Therefore, synchronization, comparison and message contents duplication times are overshadowed by processing time. On the other hand, small matrixes require a short computation time and therefore all communication-related detection activities become more relevant.

Similarly, it can be seen that, for any given matrix size, overhead increases with the number of processes. This is explained by the fact that the number of messages (and therefore, synchronizations, verifications and copies) increases with the number of processes.

The case of 2 processes was the one that presented a wider dispersion between different repetitions of the experiment. A factor of randomness is present; inclusive, in the case of N = 4096, the incorporation of SMCV appears to perform better than the original application. However, with the precision of the obtained measurements, differences below 1 %, which occur in all cases, are considered negligible.

Based on the experiments carried out, it can be concluded that, when the size of the problem increases but the number of processes remains constant, the overhead is sig-

nificantly low. This would mean that, in real applications with high performance requirements, handling large amounts of data, similar overheads can be obtained.

7 Future Work

This work is part of a more extensive proposal whose purpose is providing transient fault tolerance for systems formed by scientific, message-passing parallel applications that are run on multicore cluster architectures.

Fault tolerance includes the phases of detection, protection, and recovery. In the context of permanent faults, the existing techniques most widely used are checkpointing and event log for protection, and rollback-recovery [24]. The proposal consists in integrating the transient fault detection methodology to the protection and recovery strategies available for permanent faults to provide transient fault tolerance. This means that there is no need of using triple modular redundancy (TMR) [16] with voting mechanisms to detect and recover from a transient fault. Also, since transient faults do not require system reconfiguration, recovery can be achieved by re-executing the same core of the failed process.

In the road towards achieving this goal, the following lines are open:

1. Perfecting the detection strategy:

- Expanding the experimental validation. A test that is more thorough than the one carried out so far requires the use of the methodology with standard applications. In the next stage, NAS benchmarks will be used, which are widely used in the scientific environment to measure the performance of parallel machines because they are representative of the type of computation most frequently made. These benchmarks respond to other parallel programming paradigms, and also have the advantage of providing self-verification functions of the results, which is useful for validating the detection strategy. In this sense, the integration with fault injection tools is desirable, to improve validation capabilities by means of extensive random fault injection campaigns. The overhead obtained with these applications will be measured.
- Achieving transparency for the application. At the current development level, SMCV's duplication process, based on threads, requires minor changes in the application code (and recompiling) to support the location of the replica in shared-memory with the original process and the use of the communications library with extended functionality. To obtain this transparency, replication must be implemented at the level of processes rather than threads.
- Optimizing the methodology to improve the trade-off between reliability, overhead, additional workload, detection latency (related to the recovery cost) and resource utilization. A detailed characterization will allow suggesting new ways of improving performance, considering the possibility of configuring the robustness level based on application coverage needs or maximum overhead permitted [6,11,13].

2. Providing full tolerance to SDC, restoring the system to its state previous to the fault:

In a following stage, the distributed detection strategy (already optimized) will be integrated with fault tolerance architectures oriented to permanent faults. The goal is obtaining a system capable of tolerating both permanent and transient faults. In this sense, integration with RADIC [15] will be attempted; RADIC is a transparent, scalable, flexible, and fully distributed architecture that provides fault tolerance through non-reliable elements and can recover after a permanent fault in a node. The aim is to leverage the methodology provided by RADIC for permanent faults (the rollback recovery mechanism, with non-coordinated checkpoints and message logs), and add transient fault tolerance. The resulting system will have to be tested to determine the reliability obtained, transparency for the application, resource utilization, overhead in absence of faults, and degradation in presence of faults.

8 Conclusions

In this paper, SMCV is presented, which is a transient fault detection methodology, purely implemented through software and specifically designed for scientific, message-passing parallel applications that are run on multicore clusters. Under the premise that in this type of applications, all information that is relevant for the end results is transmitted among the processes that are part of it, the SMCV strategy is based on validating the contents of the messages to be sent and comparing the end results to achieve a compromise between a high level of robustness against faults and the introduction of a low execution time overhead, consequence of the non-detection of the faults that would normally not affect the results. Also, it introduces a reduced additional workload versus the more conservative strategies that validate all write-to-memory operations, similar to the ones used in sequential applications.

References

1. Mukherjee, S. S., Emer, J., Reinhardt, S. K.: The Soft Error Problem: An Architectural Perspective. HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, 243 – 247 (2005)
2. Wang, N. J., Quek, J., Rafacz, T. M., Patel, S. J.: Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks, 61 – 70 (2004)
3. Mukherjee, S. S.: Architecture Design for Soft Errors. Morgan Kaufmann (2008)
4. Lesiak, A., Gawkowski, P., Sosnowski, J.: Error Recovery Problems. Dependability of Computer Systems, 2007. DepCoS-RELCOMEX '07. 2nd International Conference on, 270 – 277 (2007)
5. Shivakumar, P., Kistler, M., Keckler, S. W., Burger, D., Alvisi, L.: Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks, 389 – 398 (2002)

6. Wells, P. M., Chacraborty K. Sohi G. S.: Mixed-Mode Multicore Reliability. ASPLOS 2009. SESSION: Reliable systems II, 169 – 180 (2009)
7. Reinhardt, S. K., Mukherjee S. S.: Transient Fault Detection via Simultaneous Multithreading. Proceedings of the 27th annual International Symposium on Computer Architecture, Vancouver, British Columbia, Canada, 25 – 36 (2000)
8. Kontz M., Reinhardt S. K., Mukherjee S. S.: Detailed Design and Evaluation of Redundant Multithreading Alternatives. Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02). Anchorage, Alaska, 99 – 110 (2002)
9. Vijaykumar T. N., Pomeranz, I. Cheng, K.: Transient-Fault Recovery using Simultaneous Multithreading. Proceedings of the 29th Annual International Symposium on Computer Architecture, Anchorage, Alaska. Session 3: Safety and Reliability, 87 – 98 (2002)
10. Gomaa M., Scarbrough C., Vijaykumar T. N., Pomeranz, I.: Transient-Fault Recovery for chip Multiprocessors. Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA ´03), San Diego, California, 98 – 109 (2003)
11. Golander A., Weiss S., Ronen R.: Synchronizing Redundant Cores in a Dynamic DMR Multicore Architecture. IEEE Transactions on Circuits and Systems II: Express Briefs Volume 56, Issue 6, 474 – 478 (2009)
12. Sundaramoorthy K., Purser Z., Rotenberg E.: Slipstream Processor: Improving both Performance and Fault-tolerance. ACM SIGPLAN Notices Volume 35, Issue 11, 257 – 268 (2000)
13. Barr A. H., Pomaranski K. G., Shidla D. J.: United States Patent Application Publication US 2005/0102565 A1: Fault Tolerant Multicore Microprocessing (2005)
14. Gramacho, J., Rexachs del Rosario, D., Luque, E.: A Methodology to Calculate a Program´s Robustness against Transient Faults. PDPTA 2011, 645 – 651 (2011)
15. Santos, G., Duarte, A., Rexachs del Rosario, D., Luque, E.: Providing Non-stop Service for Message-Passing Based Parallel Applications with RADIC. Euro-Par 2008, 58 – 67 (2008)
16. Mathur, F., Avizienis, A.: Reliability analysis and architecture of a hybrid-redundant digital system: generalized triple modular redundancy with self-repair. AFIPS '70 (Spring) Proceedings of the May 5-7, 1970, Spring Joint Computer Conference (1970)
17. Mukherjee, S.; Weaver, C.; Emer, J.; Reinhardt, S., Austin, T.: A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. MICRO-36.Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 29 – 40 (2003)
18. Weaver, C., Emer, J., Mukherjee, S. S., Reinhardt, S. K.: Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor, ACM SIGARCH Computer Architecture News, Volume 32, Issue 2, page 264 (2004)
19. Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., August, D. I.: SWIFT: Software Implemented Fault Tolerance, in Proceedings of the international symposium on Code generation and optimization, Washington DC, USA, 243–254 2005
20. Bronevetsky, G., Supinski, B.: Soft error vulnerability of iterative linear algebra methods. ICS '08: Proceedings of the 22nd annual international conference on Supercomputing. New York, NY, USA: ACM, 155 – 164 (2008)
21. Shye, A., Blomstedt, J., Moseley, T., Reddi, V. J., Connors, D. A.: PLR: A software approach to transient fault tolerance for multicore architectures, Dependable and Secure Computing, IEEE Transactions on, Volume 6, Issue 2, 135 – 148 (2009)
22. Leibovich F., Gallo S., De Giusti L., Chichizola F., Naiouf M., De Giusti A.: Comparación de paradigmas de programación paralela en cluster de multicores: Pasaje de mensajes

e híbrido. Un caso de estudio. Proceedings of XVII Congreso Argentino de Ciencias de la Computación (CACIC 2011), 241 – 250 (2011)
23. Rotenberg E.: AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. Proceedings of the 29th Annual International Symposium on Fault- Tolerant Computing, 84 – 91 (1999)
24. Rexachs, D., Luque, E.: High Availiability for Parallel Computers. JCS&T Vol. 10 No. 3, 110 – 116 (2010).

HPCLatAm 2012, pp. 149-161 (full paper)
ISSN 2422-5207
Session: Applications
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

Evolutionary Statistical System for applying in Forest Fire Spread Prediction*

Germán Bianchini, Miguel Mendez-Garabetti and Paola Caymes-Scutari

Laboratorio de Investigación en Cómputo Paralelo/Distribuido (LICPaD)
Departamento de Ingeniería en Sistemas de Información, Facultad Regional Mendoza
- Universidad Tecnológica Nacional. (M5502AJE) Mendoza, Argentina

Abstract. Several propagation models have been developed to predict forest fire behaviour. They can be grouped into empirical, semi-empirical, and physical models. These models can be used to develop simulators and tools for preventing and fighting forest fires. Nevertheless, in many cases the models present a series of limitations related to the need for a large number of input parameters. Furthermore, such parameters often have some degree of uncertainty due to the impossibility of measuring all of them in real time. Therefore, they have to be estimated from indirect measurements, which negatively impacts on the output of the model. In this paper we present a method which combines Statistical Analysis with Parallel Evolutionary Algorithms (taking advantage of the computational power provided by High Performance Computing) to improve the quality of model's output.

1 Introduction

Different propagation models have been developed to predict fire behaviour. They can be classified into empirical, semi-empirical, and physical models [8]. The probable fire behaviour is predicted in empirical models from average conditions and accumulated knowledge obtained from laboratory and outdoor experimental fire or from historical fires. Semi-empirical (semi-physical or laboratory models) are those models based on a global energy balance and on the assumption that the energy transferred to the unburned fuel is proportional to the energy released by the combustion of the fuel; one of the most important among these models is the pioneering work of Rothermel (1972 and 1983) [21, 22]. Finally, physical (theoretical or analytical) models are based on physical principles and have the potential to accurately predict the parameters of interest over a broader range of input variables than empirically based models do. These models can be used to develop simulators and tools for preventing and fighting forest fires. Some old and current examples are Behave-Plus [1], FARSITE [9], FIREMAP [2], FireStation [15], WRF-Fire [16], XFire [14], etc.

According to Fons [10] the relevant factors that affect the rate of spread and shape of a forest fire front are the fuel type (type of vegetation), humidity, wind

* This work has been supported by Conicet under project PIP 11220090100709, by UTN under project UTN1194 and by ANPCyT under project PICT PRH-00242.

HPCLatAm 2012, pp. 149-161 (full paper)
ISSN 2422-5207
Session: Applications
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

speed and direction, forest topography (slope and natural barriers), and fuel continuity (vegetation thickness). Therefore, models require a set of input parameters, including vegetation type, moisture contents, wind conditions, and so on, and they provide the evolution of the fire line in the successive simulation steps. However, the result obtained after the direct application of a simulator (known as Classical Prediction and explained in Section 2) usually differs from reality because of the difficulty of providing accurate input values to the model. Given this uncertainty, we propose an alternative method, that tries to determine the possible fire behaviour based on Statistical Analysis [19] and Parallel Evolutionary Algorithms (PEAs) [18] as optimization method. This method corresponds to an improvement of a previous methodology based on Statistical Analysis and High Performance Computing, which has been modified by the combination with Evolutionary Algorithms to improve the prediction level and reduce the execution time.

Clearly, the simulation of the spread of forest fires is a challenge from the computational point of view, given the complexity of the models involved, the need for efficient numerical methods and resource management for results. In this context, the method presented in this paper is an important tool for the prevention and prediction of forest fires, as it provides more complete information about the potential fire behaviour. This is a general method which could be applied on different propagation models (e.g. floods, snow avalanches, landslides, etc.), but here we only present its application to forest fire prediction.

In the remaining sections of this paper we describe the direct use of a simulator in section 2 (known as Classical Prediction); section 3 shows the predecessor of the current method (Statistical System for Forest Fire Management - S^2F^2M [4, 5]) and section 4 describes the new methodology, implemented in a system called Evolutionary Statistical System (ESS) [3]. In section 5 we compare both methods using a set of real cases of forest fires and also we comment on the obtained results related to the execution time and the speed-up obtained when we work on a cluster computer. Finally, we present the main conclusions.

2 Classical Prediction

Classical Prediction approach is depicted in Fig. 1. In this scheme, FS corresponds to the underlying fire simulator, which will be seen as a black box. RFL0 is the real fire line at time t_0 (initial fire front), whereas RFL1 corresponds to the real fire line at t_1. If the prediction process works, after executing FS (which should be fed with the corresponding input parameters and RFL0) the predicted fire line at time t_1 (PFL) should coincide with the real fire line (RFL1).

As we mentioned previously, models require static parameters (information about topography), parameters that can change very slowly (type of vegetation), parameters that can change frequently (moisture content), and parameters that are completely dynamic (like wind conditions). The simulator will not work properly without this set of parameters. The precision of these parameters is a very important point in prediction of the behaviour, and in many cases it is

HPCLatAm 2012, pp. 149-161 (full paper)
ISSN 2422-5207
Session: Applications
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

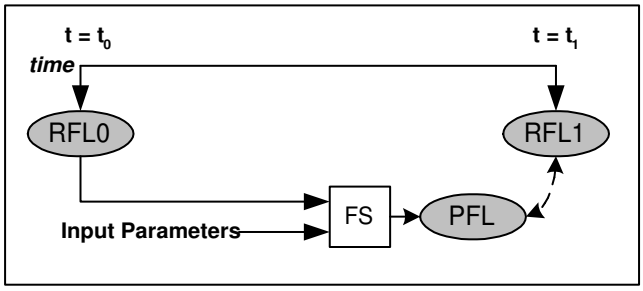


Fig. 1. Diagram of Classical Prediction of forest fire propagation (FS: Fire Simulator; PFL: Predicted Fire Line; RFLX: Real Fire Line on time X)

impossible to carry out some types of measurements, particularly in a real fire situation.

Generally, the obtained prediction using this approach does not match the reality. One reason for the discrepancy between real and simulated propagation stems from the difficulty of feeding the model with accurate input values. Uncertainties in the input variables can have a substantial impact on the result errors and should be considered.

In this context, the prediction of the fire line behaviour cannot be considered to be reliable for two reasons: on the one hand, the difficulties in making an accurate estimate of the parameters and, on the other hand, the resulting prediction is based on a single simulation, which does not constitute a reasonable basis for making a decision given the uncertainty of the parameters.

3 Statistical Method for Uncertainty Reduction

The statistical method for uncertainty reduction [4–6] has been the result of the combination of various research projects. This method has as its bases the concepts of statistical analysis and distributed computing. Basically, the method finds a pattern of behaviour of the model without performing a specific analysis of each scenario (where a particular setting of the input parameter values defines an individual scenario). All the possible scenarios are discretely generated considering a certain domain by a factorial experiment [19] and the model is evaluated with each set of values. The results are combined to determine the trend in the behaviour of the model, adjusting to the current observation of it. The pattern found is then taken to predict the next step.

This method requires a large number of operations, and therefore is very time demanding. For this reason, we applied a parallel computing scheme for its implementation. Because of this, we used multiple computational resources working in parallel to reduce the time. Keeping in mind the nature of the problem, we applied a Master-Worker paradigm [12, 17], because the problem we face can be divided into multiple partitions and the same calculations can be applied over

HPCLatAm 2012, pp. 149-161 (full paper)
ISSN 2422-5207
Session: Applications
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

each data subset. Therefore, we face a problem that can be solved using domain decomposition: a main processor can calculate each combination of parameters and send them to a set of Workers. These Workers carry out the simulation in parallel, taking into account several combinations of parameters, and return the partial results to the Master, which aggregates all these individual results at each iteration. Also, the Master process is responsible for the statistical stage and it is in charge of the remaining prediction technique.

A scheme of a whole prediction system is presented in Fig. 2. As we can see, the process of prediction needs a calibration stage at the beginning (time period that goes from t_0 to t_1) to firstly obtain a K_{ign} value (Key Ignition value) to start up the prediction chain. For every i from 1 to n, both the prediction operation for time t_i and the calibration stage to obtain the K_{ign} to be used in time t_{i+1} will overlap at time t_i. This situation is the one depicted in Fig. 2. As can be observed, the output generated by the **SS** box (Statistical Stage) is used for a double purpose. On the one hand, the probability maps are used as an input of the **SK** box (Search K_{ign}) to search for the current K_{ign}, which will be used at the next prediction time. In this stage, a Fitness Function (**FF**) is used to evaluate the probability map. On the other hand, the output of **SS** box enters the Fire Prediction box (**FP**), which will be in charge of generating the prediction map taking into account the K_{ign} evaluated at previous time. This process will be repeated during the execution as the system is fed with new information about the fire situation.

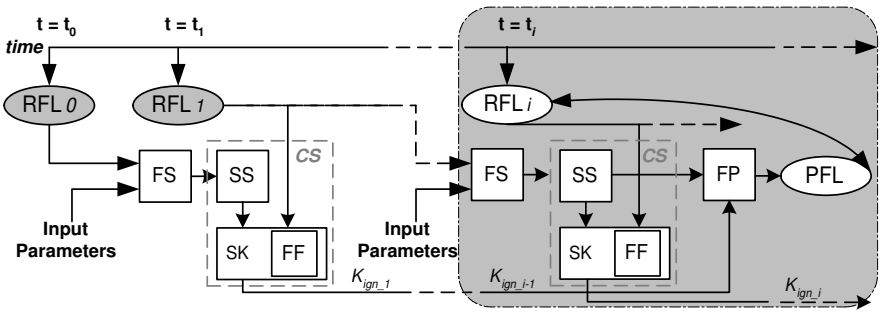


Fig. 2. Detailed diagram of S^2F^2M (FS: Fire Simulator; CS: Calibration Stage; SS: Statistical Stage; SK: Search Kign Stage; FF: Fitness Function; FP: Fire Prediction; PFL: Predicted Fire Line; RFLX: Real Fire Line at time X)

Although the statistical method can be used to solve various Grand Challenge Problems, as a case of study, the method has been applied on a behavioural model of forest fire propagation. As a result, we developed a system called Statistical System for Forest Fire Management (S^2F^2M), which is the product of the combination of applying the proposed method with the simulator fireSim (fireSim is the implementation of a fire behaviour simulator based on the Rothermel model

HPCLatAm 2012, pp. 149-161 (full paper)
ISSN 2422-5207
Session: Applications
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

[21] and implemented with the library fireLib [7]). For a detailed description of the method, we suggest the reader to consult [4, 5].

4 Evolutionary Statistical System

The improvement and modification of the statistical method discussed in the previous section has resulted in a new method that combines the strength of three components: uncertainty reduction, evolutionary algorithms and parallelism, that is why the new method has been called Evolutionary Statistical System [3]. The improvement of the method is related to the introduction of features of PEAs in the calibration step of the statistical method. As we seen in the previous section, the statistical phase of the methodology includes all the results of a series of cases that arise as some combination of the possible resulting values (within valid ranges) of the parameters that exhibit uncertainty. Clearly, there is a certain percentage of cases that do not contribute significant values to the global result, whether they are now redundant, or because they are too far from reality (and thus, could be considered as negative cases that ultimately degrade the result provided by the method). To avoid this problem is that we have decided to apply the Evolutionary Algorithms (EAs), whose basis is explained in more details in the next section.

4.1 Evolutionary Algorithms

Evolutionary algorithms (EAs) mimic the concept of natural biological evolution: they operate on a population of potential solutions applying the principle of survival of the fittest [11]. In each iteration EAs create a new set of approaches through a process of selecting individuals according to the level of fitness for the problem domain (through the fitness function that quantifies this feature) and perform a recombination of them using operators that mimic natural genetics. This process leads to the evolution in the population of individuals that have best adapted to the environment just as happens in natural adaptation.

The EAs model natural processes such as selection, crossover, mutation, migration, locality and the notion of neighbourhood, working on populations of individuals rather than on unique solutions. Thus, the search can be performed in parallel, thus providing a number of potential solutions instead of one. This scheme is known as Parallel Evolutionary Algorithms (PEAs). According to the amount of populations involved in the algorithm, the treatment and the operators PEAs can be classified in three broad groups: Unique Population and Parallel Evaluation, Unique Population and Overlapped Neighbourhoods, and Multiple Populations and Migration. In this work, we consider the first group.

In each generation the fitness of each individual in the population is evaluated in parallel. Multiple individuals are stochastically selected from the current population (depending on fitness), and modified (by recombination or by random mutation) to form a new population. The fitness is defined in terms of the genetic representation and measures of quality of the solution represented.

The execution of the PEA may finalize by various criteria. One method is to finish after a predetermined number of iterations. Another way is to check whether the measure of population quality has improved or not after a certain number of generations. Another is to finish when all individuals are identical, which can only happen when not using the mutation.

Evolutionary algorithms are a powerful tool for solving different kinds of problems [20]. However, sometimes this type of methodology iterates for a long time and does not converge or converges to a local optimum. This is one of the reasons why it is interesting combine the use of evolutionary methods with parallel computing. However, given the use of evolutionary algorithms in optimization problems, where they have found very good results, we propose the application of this methodology in combination with statistical methods, as discussed in the following section.

4.2 Methodology of the Evolutionary Statistical System

The Evolutionary Statistical System (ESS), classified as Data-Driven methods with Multiple Overlapping Solution, is an improvement of the S^2F^2M method previously commented. It combines the original uncertainty reduction method implemented in S^2F^2M with the advantages that offer the Parallel Evolutionary Algorithms (PEAs), dealing with a population of scenarios relevant to the study. ESS, like its predecessor, is based on statistics, mainly on the concept of factorial experiment [19], where the combination of several factors (input parameters) defines a scenario. In this case, each scenario is represented by an individual in a population of possible solutions.

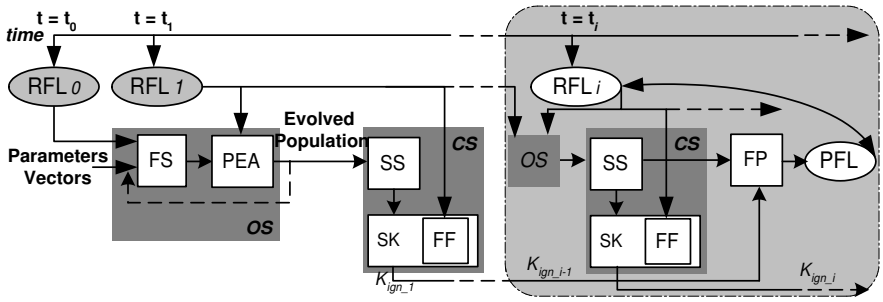


Fig. 3. Diagram of ESS (FS: Fire Simulator; PEA: Parallel Evolutionary Algorithm; OS: Optimization stage; SS: Statistical System; SK: Search K_{ign}; FF: Fitness Function; CS: Calibration stage; FP: Fire Prediction; PFL: Predicted Fire Line, RFLX: Real Fire Line on time X)

A scheme of ESS is presented in Fig. 3. As can be observed, the system is divided in two general stages: an Optimization Stage (**OS**) that implements the

parallel evolutionary algorithm (**PEA** box), and the Calibration Stage (**CS**) that is in charge of the statistical method. **OS** iterates until the population reaches a certain level of quality. For each individual **FS** and the fitness are calculated in parallel. Then, every individual will be included in the Statistical System (**SS** box). Similarly to S^2F^2M, the output of **SS** (a probability map) has a double purpose. On the one hand, the probability maps are used as the input of the **SK** box (Search K_{ign}) to search for the current K_{ign} (a key number used to make a prediction), which will be used at the next prediction time. In this stage, a Fitness Function (**FF**) is used to evaluate the probability map. On the other hand, the output of **SS** box enters the Fire Prediction box (**FP**). **FP** will be in charge of generating the prediction map taking into account the K_{ign} evaluated at previous time. All this process will be repeated during the execution as the system is fed with new information about the fire situation.

The architecture of the ESS is based on the Master-Worker paradigm [12, 17]: In each iteration the Master distributes an individual per Worker; the simulation of the model and the evaluation of fitness function are applied over each individual (tasks carried out by the Workers), returning the results to the Master. This process is repeated until every individual in the population is treated. Finally the Master evolves the population, aggregates the partial results and makes the prediction for each time step.

5 Experimental Results

This section compares the results obtained after applying the original statistical method (S^2F^2M) and the Evolutionary Statistical System (ESS) described in this paper. To that end we have used four cases of controlled burns. They were made in the field (Fig. 4), particularly in a hill of Serra de Lousã (Gestosa, Portugal). The burns were part of the SPREAD project [23]. These experiments were very useful to collect experimental data, to support the development of new concepts and models, and to validate existing methods or models in various fields of fire management. We have not included the results of the classical method because the values obtained are low and do not contribute information to this work. In addition, previous studies have shown that the values obtained by applying the statistical method surpasses the quality of the prediction achieved by the classic approach [5].

Along the progress of burning, discrete steps were defined to represent the progress of the fire front. Therefore, we consider various time instants t_0, t_1, t_2... etc. In Table 1 can be appreciated the characteristics (size and slope) of the land used for each experiment. In order to gather as much information as possible about the fire-spread behaviour, a camera recorded the complete evolution of the fires. The videos obtained were analyzed and several images were extracted every certain period of time. From the images, the corresponding fire contours were obtained and converted into a suitable format so they could be interpreted by the methods.

HPCLatAm 2012, pp. 149-161 (full paper)
ISSN 2422-5207
Session: Applications
E. Mocskos & S. Nesmachnow (Eds.)
July 25 - August 3, 2012
Buenos Aires, Argentina

Fig. 4. Real fire during the burns in the Gestosa area.

Table 1. Dimensions and slopes of the plots used in experiments.

Experiment	Width (m)	Length (m)	Slope (°)
1	58	50	21
2	89	91	21
3	95	123	21
4	20	30	6

In experiments 1 and 2 the cell size was 1 m^2, and in experiments 3 and 4 the cell size was 0.333 m^2. The remaining parameters such as wind conditions and moisture content were variable.

5.1 The fitness function

It is necessary to define a criterion to compare the prediction resulting from each method with the real situation. To evaluate the system response we have defined a fitness function. Since the simulator uses an approximation based on cells, the fitness function is defined as a quotient. The following equation shows the expression:

$$Fitness = \frac{(\#cells \bigcap - \#IgnitionCells)}{(\#cells \bigcup - \#IgnitionCells)}$$

where $\#cells \bigcap$ represents the number of cells in the intersection between the simulation results and the real map, $\#cells \bigcup$ is the number of cells in the union of the simulation results and the real situation, and $\#IgnitionCells$ represents the number of burned cells before starting the simulation.

A fitness value equal to one corresponds to the perfect prediction because it means that the predicted area is equal to the real burned area. On the other hand, a fitness equal to zero indicates the maximum error because, in this case, our experiment did not coincide with reality at all.

5.2 Comparison

According to the information already known about the experiments and the models of Rothermel [21] for some of the parameters, certain ranges have been specified (in particular those parameters that exhibit uncertainty). A part of this information has been measured during the experiment, and the remainder has been taken from standard values used by BehavePlus [1].

The experiments 1, 3 and 4 belong to cases of fires started at the base of the field through pyrotechnic devices in a linear way. Meanwhile, in experiment 2, the fire originated in a single point. After execution of the methods, the fitness values found are shown in Figure 5. We can see that in all four cases, ESS performs better compared to the original version of the method. However, at certain times, the values found may be similar or even slightly lower than the results of the original method (this happens in Experiment 2 at minute 7.5 and in Experiment 3 at minute 12).

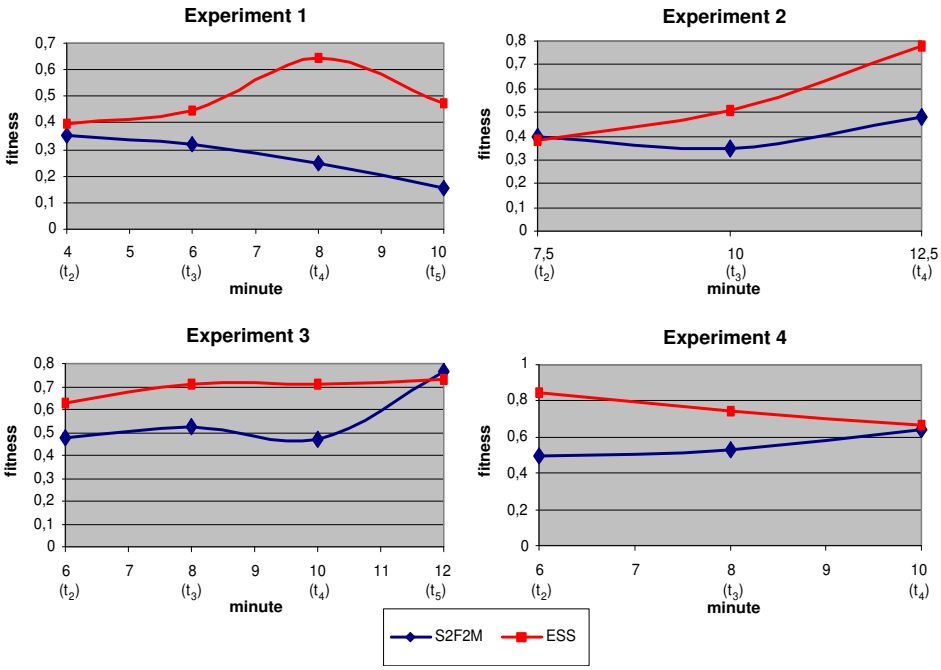


Fig. 5. Fitness comparison between the S^2F^2M and ESS for four experiments.

It is important to emphasize that for calibration and prediction purposes these methods need one real fire line more than the classical prediction, so we cannot provide suggestions at the first time t_1, i.e., along the first step of these methods it is only possible to apply the calibration stage whose result will be used in t_2. Thus, from t_2 to t_n, every step t_i of the methods executes both **CS**

and **FP**, basing their **FP** in the K_{ign} provided by the previous step t_{i-1} (see Fig. 2 and 3). This is the reason why the figure shows the results from t_2 considered in each experiment.

Another important point to highlight is to mention that the shown values for ESS are the average of ten executions. In the case of S^2F^2M, this is not necessary because it gives a deterministic output.

5.3 Parallelism and Speedup

The results were obtained by executing both systems on a LINUX cluster (12 processors AMD64 2G RAM and Gigabit Ethernet 1000 Mbps) under an MPI environment [13]. The performance gain has been analyzed using the measure known as Speedup [12], which is defined as the ratio of the time taken to solve a problem on a single processing element to the time required to solve the same problem on a parallel computer with p identical processing elements.

The numbers of processors used were 1, 2, 4, 6, 8, 10 and 12 (although the graph of speedup is usually designed with p equal to successive powers of 2). Figure 6 shows the values obtained as an average of all experiments.

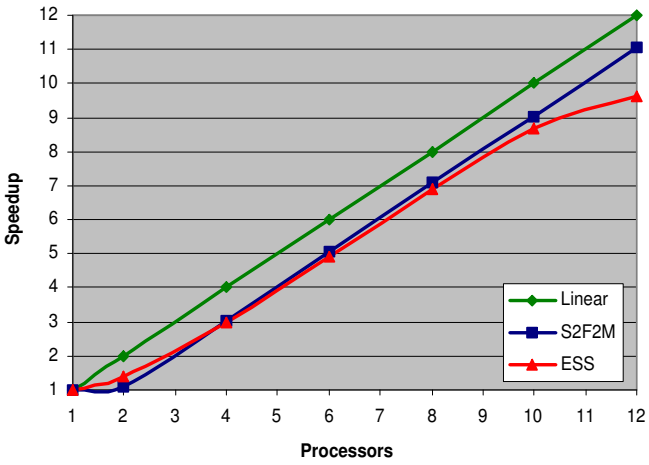


Fig. 6. Speedup for the methods.

The continuous line represents the linear case (or ideal Speedup). As we can see, both methods have a speedup relatively good (S^2F^2M a bit better than ESS). For the purposes of a fair comparison, in both cases were performed the same number of simulations. Thus, in addition to the graph, the execution times are also similar (ESS takes on average 10% less execution time). However, in actual executions, ESS usually takes even less time because in principle, the number

of iterations depends on when it finds individuals who meet the expected fitness, and this usually happens before in ESS that in S^2F^2M (remember that S^2F^2M is deterministic and exhaustive method, while ESS is a not deterministic one). For instance, for Experiment 4, ESS can take around 35 *min* to find individuals with fitness equal to 0.85, or it can spend 140 *min* looking for individuals with fitness equal to 0.95. In conclusion, there is a trade-off between time and quality, and depends on the user to configure certain parameters to emphasize either the time restriction or the expected quality.

6 Conclusions

In this work, a method is described, which represents a major enhancement compared to previous methodologies. As we have seen, the techniques that combine high performance computing with statistical methods have excellent ability to solve or reduce the problem of uncertainty in input parameters. For this reason, it is of great interest the ongoing research on this subject, so as to optimize and evolve on the approaches and methods already developed to maximize the results achieved. Then, from S^2F^2M we have arrived at the concept of Evolutionary Statistical System (ESS). To do this, we combined the power of the statistical calculation with capabilities provided by parallel evolutionary algorithms, achieving results that actually improve the original methodology S^2F^2M based solely on statistical calculation and high performance computing. Both methods have been described throughout the present work. They correspond to methods to reduce uncertainty in the input parameters, in this case applied to the prediction of forest fires spread. Given the costs, risks and obvious difficulties for design multiple fires in real plots to obtain reliable data for experimentation and validation of the methods, the experiments were conducted on four real fires considering different instants of time in each case. In addition to significantly improve the accuracy of the prediction quality of the classical method, one of the most important features of both methods is that they are general enough to be used on different models (floods, avalanches, etc.). Thus, the combination of evolutionary computation, parallelism and uncertainty reduction is a promising option for tackling various Grand Challenge Problems, as in this case it is the prediction of forest fire behaviour.

In this first approach of ESS, we decide apply parallelism only in the evaluation of the individuals, with the goal of gradually increase the degree of parallelism to compare the results offered by each alternative of PEAs. Further study should focus on the analysis and tuning of the method to obtain the best possible results and compare it with other methods.

References

1. Andrews, P.L., Bevins, C.D., Seli, R.C.: BehavePlus fire modeling system, version 2.0: User's Guide. Gen. Tech. Rep. RMRS-GTR-106WWW. Ogden, UT: Dept. of Agriculture, Forest Service, Rocky Mountain Research Station (2003) pp. 132

2. Ball, G.L., Guertin, D.P.: FIREMAP, in Nodvin, S. C. and Waldrop, T. A., Fire and the Environment: Ecological and Cultural Perspectives: Proceedings of an International Symposium. Knoxville, TN. USDA Forest Service, Southeastern Forest Experiment Station, Asheville, NC. General Technical Report SE-69 (1991) 215–218
3. Bianchini, G., Caymes Scutari, P.: Uncertainty Reduction Method Based on Statistics and Parallel Evolutionary Algorithms. Proceedings of High-Performance Computing Symposium - 40 JAIIO (HPC 2011, ISSN: 1851-9326) (2011) 1–4
4. Bianchini, G., Denham, M., Cortés, A., Margalef, T., Luque, E. (2010): Wildland Fire Growth Prediction Method Based on Multiple Overlapping Solution, Journal of Computational Science, **1** Issue 4 (2010) 229–237
5. Bianchini, G., Cortés, A., Margalef, T., Luque, E.: Improved prediction methods for Wildfires using High Performance Computing: A comparison. LNCS **3991**, Part I (2006) 539–546
6. Bianchini, G., Denham, M., Cortés, A., Margalef, T., Luque E.: Improving forest-fire prediction by applying a statistical approach. Forest Ecology and Management. (**234**, supplement 1) (2006) S210
7. Bevins, C. D.: FireLib User Manual & Technical Reference. (2004) http://www.fire.org (Accessed on May 2012)
8. Bodrožić, L., Stipanicev, C., Šeric, M.: Forest fires spread modeling using cellular automata approach. Modern trends in control. (eds V. Hladky, J. Paralic, J. Vaščak). (2006) 23–33 (Košice, Slovakia: equilibria)
9. Finney M.A.: FARSITE: Fire Area Simulator-model development and evaluation. Res. Pap. RMRS-RP-4, Ogden, UT: U.S. Department of Agriculture, Forest Service, Rocky Mountain Research Station. (1998) pp. 47
10. Fons W.: Analysis of fire spread in light forest fuels, J. Agric. Res. **72** (1946) 93–121
11. Goldberg, D.E.: Genetic and evolutionary algorithms. Come of age. Communications of the ACM, **37** (3) (1994) 113–119
12. Grama A., Gupta A., Karypis G., Kumar V.: Introduction to Parallel Computing. Second Edition. Pearson (2003)
13. Gropp W., Lusk E., Skjellum A.: Using MPI - Portable Parallel Programming with the Message-Passing Interface. Second edition. The MIT Press (1999)
14. Jorba, J., Margalef, T., Luque, E., Campos da Silva, J., Viegas, D.X.: Parallel Approach to the Simulation of Forest Fire Propagation. Proc. 13 Inter. Symposium "Informatik fur den Umweltshutz" der Gesellshaft Fur Informatik (GI) (1999) 68–81
15. Lopes, A.M.G., Cruz, M.G., Viegas, D.X.: FireStation - An integrated software system for the numerical simulation of wind field and fire spread on complex topography. Environmental Modelling & Software, **17** (3) (2002) 269–285
16. Mandel, J., Beezley, J.D., Kochanski, A.K.: Coupled atmosphere-wildland fire modeling with WRF 3.3 and SFIRE 2011, Geoscientific Model Development (GMD) **4** (2011) 591–610
17. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Addison-Wesley (2005)
18. Michalewics, Z.: Genetic Algorithms + Data Structures = Evolution Programs. Third, Revised and Extended Edition. Springer (1999)
19. Montgomery, D.C., Runger G.C.: Probabilidad y Estadística aplicada a la Ingeniería. Limusa Wiley (2002)
20. Nelson, K.M.: Applications of evolutionary algorithms in mechanical engineering. (1997) http://digitool.fcla.edu/dtl_publish/34/12514.html (Accessed on May 2012)
21. Rothermel, R. C.: A mathematical model for predicting fire spread in wildland fuels, Res. Pap. INT-115, US Dept. of Agric., Forest Service, Intermountain Forest and Range Experiment Station. (Ogden, UT.) (1972)

22. Rothermel, R. C.: How to predict the spread and intensity of forest fire and range fires. Gen. Tech. Rep. INT-143, US Dept. of Agric., Forest Service, Intermountain Forest and Range Experiment Station. (Ogden, UT.) (1983)
23. Viegas, D.X. (Coordinator) et al., Project Spread - Forest Fire Spread Prevention and Mitigation (2004) http://www.algosystems.gr/spread/ (Accessed on May 2012)