Efficient parallelization of a regional ocean model for the western Mediterranean Sea

M Luisa Córdoba[1], Antonio García Dopico[1], M Isabel García[1], Francisco Rosales[1], Jesús Arnaiz[1], Rodolfo Bermejo[2] and Pedro Galán del Sastre[2]

Abstract
This paper focuses on the parallelization of an ocean model applying current multicore processor-based cluster architectures to an irregular computational mesh. The aim is to maximize the efficiency of the computational resources used. To make the best use of the resources offered by these architectures, this parallelization has been addressed at all the hardware levels of modern supercomputers: firstly, exploiting the internal parallelism of the CPU through vectorization; secondly, taking advantage of the multiple cores of each node using OpenMP; and finally, using the cluster nodes to distribute the computational mesh, using MPI for communication within the nodes. The speedup obtained with each parallelization technique as well as the combined overall speedup have been measured for the western Mediterranean Sea for different cluster configurations, achieving a speedup factor of 73.3 using 256 processors. The results also show the efficiency achieved in the different cluster nodes and the advantages obtained by combining OpenMP and MPI versus using only OpenMP or MPI. Finally, the scalability of the model has been analysed by examining computation and communication times as well as the communication and synchronization overhead due to parallelization.

1. Introduction

For an in-depth study of climate change, models are being developed that reproduce the main processes that take place in the five components of the climate system: atmosphere, hydrosphere, geosphere, biosphere and the exchange of mass and energy between them. These models are validated by comparing their results with the observations made in recent decades and the results of other models.

This paper describes a regional climate model, developed by some of the authors (Gallardo-Andrés et al. (2010), with a high temporal and spatial resolution centred on the western Mediterranean basin. This basin is characterized by complex coastlines and strong topographical features such as the Alps, the Pyrenees, the Apennines, the Balkan mountain ranges, the Italian and Hellenic peninsulas and large islands (Balearic islands, Sicily, Sardinia, Corsica, Crete and Cyprus). Together these all contribute to the existence of strong air–sea interactions in the region, with considerable influence on the Mediterranean circulation. This climate model has two main components: an ocean model, MOSLEF, and an atmospheric model, PROMES. As they are coupled by OASIS, a coupler software, they must advance at the same pace. Both are memory-demanding and computationally very expensive, due to the complexity of the simulated system, the huge time intervals involved and their high spatial resolution. This results in unacceptably high simulation execution times which increase even more depending on the temporal and spatial resolution or time interval required. This calls for parallelization and although an atmospheric model using parallelization already exists (Garrido et al. 2009), its accompanying ocean model was still sequential.

Most of the ocean models developed have been parallelized (Sannino et al., 2001; Luong et al., 2004; Fringer et al., 2006; Cowles, 2008; Henshawa and Schwendemanb, 2008; Wang et al., 2010; Tseng and Chien,

[1] Computer Architecture, Universidad Politécnica de Madrid, Spain
[2] Applied Mathematics ETSII, Universidad Politécnica de Madrid, Spain

Corresponding author:
M Luisa Córdoba, Computer Architecture, Universidad Politécnica de Madrid, Spain.
Email: mcordoba@fi.upm.es

2011). Each of these has its own characteristics such as the type of mesh used, how the mesh is divided, how the different submeshes overlap, how the workload is distributed between the processing units, if they are models for distributed or shared memory computers, and the programming model used.

The initial objectives of the parallelization of the ocean model described in this paper were i) to efficiently exploit the characteristics of the most common parallel architectures used in high performance computing: multicore processors and clusters; ii) to demonstrate the viability of parallelization, which was not obvious due to possible adverse effects of data dependencies on final speedup (see Wang et al., 2010); iii) to study its scalability for different architectures; iv) to combine or use different parallelization techniques simultaneously; and most importantly, v) to achieve the highest possible speedup.

For parallelization on multicores the authors have chosen OpenMP (2013) because it introduces few changes in the source code, addressing the parallelization of the computationally most demanding mathematical functions. For parallelization on clusters the authors have chosen MPI (2012) as it is the most commonly used for this environment, addressing parallelization by distributing data. In addition, one of the initial premises was to use both techniques simultaneously as all modern clusters are composed of multicore nodes. All the different versions, sequential, OpenMP, MPI and MPI-OpenMP, share the same source code avoiding typical mistakes when several source codes must be maintained simultaneously.

To evaluate the performance of the parallelized application we have worked with two meshes for the western Mediterranean Sea. The first one is a medium-size mesh that consists of 35 depth layers in the vertical coordinate and a horizontal triangular mesh with 24,495 mesh points: this yields a total of 857,255 mesh points in the three-dimensional mesh that it translates into 6×10^6 unknowns to be calculated every time step. The second mesh is a finer and larger mesh, particularly in the vertical coordinate, consisting of 69 depth layers and 33,275 mesh points in the horizontal triangular mesh: this yields 1.6×10^7 unknowns to be calculated every time step. One year of simulation with the sequential application takes 253.5 hours of CPU time with the medium-size mesh and 4518.6 hours with the large mesh; however, using the parallelized application these CPU times are reduced to 7.6 hours for the medium mesh with 128 processors, and 61.6 hours for the large mesh with 256 processors.

The rest of this paper is organized as follows: Sections 2 and 3 are devoted to the mathematical and numerical ocean model respectively; Section 4 describes the strategies used for parallelization of the ocean model; Section 5 explains the main details of the distributed version; Section 6 shows the results obtained with these parallelizations; Section 7 studies the performance and scalability of the parallelization strategies used in the paper; and finally, Sections 8 and 9 describe the conclusions and future work.

2. The ocean model equations

The ocean is a stratified and slightly compressible Newtonian fluid in a rotating Earth driven by the wind stress and fluxes of heat and freshwater acting on the sea surface. The governing equations of the model, which are known as the primitive equations (PEs) of the ocean circulation, describe the temporal evolution of the flow velocity, temperature and salinity of the seawater plus the equation of state for the density as a function of the temperature, salinity and depth. The equations for the flow velocity are a reformulation of the three-dimensional Navier–Stokes equations under the Boussinesq and hydrostatic approximations; the temperature and salinity are governed by convection–diffusion equations. Following Temam and Ziane 2004) we formulate the PEs for our regional model in a Cartesian coordinate system (O, x, y, z). Let $\mathbf{V}_3$ be the three-dimensional velocity vector of the flow and let $(\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3)$ be an orthonormal vector basis; then we set $\mathbf{V}_3 = \mathbf{u} + W(\mathbf{u})\mathbf{e}_3$, where $\mathbf{u} = u\mathbf{e}_1 + v\mathbf{e}_2$ is the horizontal velocity and $W(\mathbf{u})$ denotes the vertical velocity. Thus, letting D be the ocean domain and Γ_s the ocean surface, we have for $(x, y, z) \in D$ and time $t > 0$:

Momentum equation:

$$\begin{cases} \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial v}{\partial y} + W\frac{\partial u}{\partial z} - fv + \frac{1}{\rho_0}\frac{\partial p}{\partial x} = \nu_H\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + \nu_z\frac{\partial^2 u}{\partial z^2} \\ \frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + W\frac{\partial v}{\partial z} + fu + \frac{1}{\rho_0}\frac{\partial p}{\partial y} = \nu_H\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) + \nu_z\frac{\partial^2 v}{\partial z^2} \\ \int_{-H}^{0}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)dz = 0, \;\; W(\mathbf{u})(x, y, z, t) = \int_{z}^{0}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)dz \end{cases} \tag{1}$$

Conservation equations for temperature, θ_1, and salinity, θ_2:

$$\frac{\partial \theta_i}{\partial t} + u\frac{\partial \theta_i}{\partial x} + v\frac{\partial \theta_i}{\partial y} + W\frac{\partial \theta_i}{\partial z} = A_i\left(\frac{\partial^2 \theta_i}{\partial x^2} + \frac{\partial^2 \theta_i}{\partial y^2}\right) + k_i\frac{\partial^2 \theta_i}{\partial z^2}, \; i = 1, 2 \tag{2}$$

State equation for the density:

$$\rho = \rho(\theta_i, z) \tag{3}$$

At $t = 0$, the velocity, temperature and salinity are prescribed. Next, we explain the meaning of the symbols which appear in the equations: ρ_0 is a constant reference density,

$$p(x, y, z, t) = p_s(x, y, t) + g\int_z^0 \rho dz'$$

g being the modulus of the acceleration of gravity and p_s the surface pressure. Further, $(-fv, fu)$ denote the components of the Coriolis force due to the rotation of the Earth. At mid-latitudes, we can accurately approximate the term f by $f = f_0 + \beta(y - y_0)$ where y_0 is the y-component of the latitude of reference θ_0, $f_0 = 2|W|\sin\theta_0$, $\beta = 2|W|\cos\theta_0/R$, R being the radius of the Earth. The coefficients ν_H, A_i denote the horizontal eddy viscosity coefficients for the horizontal velocity, temperature and salinity respectively, which we assume to be constant, and ν_z, k_i are the vertical eddy viscosity coefficients that are also assumed to be constant.

Boundary conditions:

(1) On the top surface Γ_s $(z = 0)$,

$$\begin{cases} \nu_z \frac{\partial \mathbf{u}}{\partial z} = \tau_w & \text{and} \quad W = 0 \\ \rho_0 C_p k_1 \frac{\partial \theta_1}{\partial z} = Q_T & \text{and} \quad k_2 \frac{\partial \theta_2}{\partial z} = Q_S \end{cases}$$

where τ_w (x, y, t) denotes the wind stress vector, Q_T is the net surface heat flux, C_p is the specific heat capacity and Q_S is the total surface flux due to evaporation, precipitation and river runoff.

(2) At the bottom of the ocean $(z = -H(x, y))$ and on the lateral solid boundaries,

$$u = v = W = 0, \ \frac{\partial \theta_1}{\partial \mathbf{n}_1} = \frac{\partial \theta_2}{\partial \mathbf{n}_2} = 0$$

where $\mathbf{n}_i = A_i(n_1\mathbf{e}_1 + n_2\mathbf{e}_2) + k_i n_3 \mathbf{e}_3$ and (n_1, n_2, n_3) is the outward unit normal vector.

3. Numerical method

The main difficulties of equations (1) to (3) are the non-linear convection terms $\mathbf{u} \cdot \nabla_H \mathbf{u} + W(\mathbf{u})\partial \mathbf{u}/\partial z$ and the non-local character of the divergence constraint. We propose, in the framework of finite elements, a modified Lagrange–Galerkin method introduced in Bermejo and Saavedra (2012) and Bermejo et al.(2012), to integrate the nonlinear convection terms, combined with a fractional step projection method to deal with the divergence constraint. Let

$$\frac{DA}{Dt} = \frac{\partial A}{\partial t} + u\frac{\partial A}{\partial x} + v\frac{\partial A}{\partial y} + W\frac{\partial A}{\partial z}$$

be the total derivative of A, where A may represent either the velocity vector or the temperature and the salinity. The Lagrange–Galerkin method approximates DA/Dt at time instant t_{n+1} by the formula

$$\frac{DA(\mathbf{x}, t)}{Dt}\Big|_{t=t_{n+1}} \simeq \frac{A(\mathbf{x}, t_{n+1}) - A(\mathbf{X}(\mathbf{x}, t_{n+1}; t_n), t_n)}{\Delta t} \tag{4}$$

where $\mathbf{X}(\mathbf{x}, t_{n+1}; t_n)$ denotes the position at time t_n of a particle that, moving with the flow velocity $\mathbf{V}_3$, will reach the point $\mathbf{x}$ at the time instant t_{n+1}. $\mathbf{X}(\mathbf{x}, t_{n+1}; t_n)$ is called the departure point at time t_n associated with the point $\mathbf{x}$. We must note that $\mathbf{X}(\mathbf{x}, t_{n+1}; t)$ is the solution of the initial value problem

$$\frac{d\mathbf{X}(\mathbf{x}, t_{n+1}; t)}{dt} = \mathbf{V}_3(\mathbf{X}(\mathbf{x}, t_{n+1}; t), t), \ \ \mathbf{X}(\mathbf{x}, t_{n+1}; t_{n+1}) = \mathbf{x} \tag{5}$$

Next, making use of the approximation (4) in (2) we approximate the temperature and salinity at time t_{n+1} by the equations

$$\frac{\theta_i^{n+1} - \theta_i^n(\mathbf{X}^{n,n+1}(\mathbf{x}))}{\Delta t} = A_i\left(\frac{\partial^2 \theta_i^{n+1}}{\partial x^2} + \frac{\partial^2 \theta_i^{n+1}}{\partial y^2}\right) + k_i \frac{\partial^2 \theta_i^{n+1}}{\partial z^2} \tag{6}$$

and the boundary conditions $B(\theta_i^{n+1})$ on ∂D. Projection methods were proposed by Chorin (1968) and Temam (1969) in the late 1960s to integrate the Navier–Stokes equations; see Guermond et al. (2006) for a recent overview of projection methods for incompressible flows. Thus, following this methodology we calculate the numerical solution to (1) by the following two-step procedure.

(1) *Viscous step.*

Given $\mathbf{u}^n$, $\mathbf{u}^{n-1}$, W^n, W^{n-1} and p_s^n, calculate $\widehat{\mathbf{u}}^{n+1} = (\widehat{u}^{n+1}, \widehat{v}^{n+1})$ by solving the system

$$\begin{cases} \frac{\widehat{\mathbf{u}}^{n+1}}{\Delta t} - \nu_H\left(\frac{\partial^2 \widehat{\mathbf{u}}^{n+1}}{\partial x^2} + \frac{\partial^2 \widehat{\mathbf{u}}^{n+1}}{\partial y^2}\right) - \nu_z \frac{\partial^2 \widehat{\mathbf{u}}^{n+1}}{\partial z^2} - f\widehat{\mathbf{u}}^{\perp n+1} = \\ \frac{\mathbf{u}^n(\mathbf{X}(\mathbf{x}, t_{n+1}; t_n))}{\Delta t} - \frac{1}{\rho_0}\nabla_H p_s^n - \frac{g}{\rho_0}\nabla_H \int_z^0 \rho^{n+1}\, dz', \\ B\widehat{\mathbf{u}}^{n+1} \text{in } \partial D \end{cases} \tag{7a}$$

2) *Poisson equation step.*

For all $n \geq 1$, calculate p_s^{n+1} by solving in Γ_s

$$\begin{cases} \nabla_H \cdot (H(x, y)\nabla_H(p_s^{n+1} - p_s^n)) = \frac{\rho_0}{\Delta t}\nabla_H \cdot \int_{-H(x,y)}^0 \widehat{\mathbf{u}}^{n+1}\, dz \\ H(x, y) \cdot \nabla_H(p_s^{n+1} - p_s^n) \cdot \mathbf{n}|_{\partial \Gamma_s} = 0 \end{cases} \tag{7b}$$

The velocity $\mathbf{u}^{n+1}$ is calculated by

$$\mathbf{u}^{n+1} = \widehat{\mathbf{u}}^{n+1} - \frac{\Delta t}{\rho_0}\nabla_H(p_s^{n+1} - p_s^n) \tag{7c}$$

where $\nabla_H = (\partial/\partial x, \partial/\partial y)$ is the gradient operator and $\nabla_H \cdot = (\partial/\partial x + \partial/\partial y)$ is the divergence operator. The vertical velocity $W^{n+1}(\mathbf{u})$ is calculated by substitution of $\mathbf{u}^{n+1}$ into (1).

Remark 1 Note that if $(p_s^{n+1} - p_s^n)$ is a solution of (7b) so is $(p_s^{n+1} - p_s^n) + K$, where K is an arbitrary constant, so that, in order to uniquely determine the solution $p_s^{n+1} - p_s^n$ we require that $\int_M (p_s^{n+1} - p_s^n) dx\, dy = 0$.

For space discretization of equations (6) to (7c) we use finite elements because this method can easily work with variable meshes that allow a better representation of the irregular coastline and bottom topography as well as the regions of strong variation of the flow. The finite element mesh is composed of tetrahedra and we represent the velocity, temperature and salinity by piecewise quadratic polynomials so that their values are calculated at the vertices and mid-points of the edges of the tetrahedra, whereas the vertical component of the velocity and the density are represented by piecewise linear polynomials and calculated at the vertices. The surface pressure, represented by piecewise linear polynomials, is calculated at the triangular faces of the tetrahedra intersecting the upper surface. The finite element formulation yields the following system of linear equations.

(1) *Temperature and salinity* $\theta_i^{n+1} = (\theta_1^{n+1}, \ldots, \theta_{CN}^{n+1})$:

$$(M + \Delta t(A_i S_1 + k_i S_2))\theta_i^{n+1} = R_i^{n+1} \tag{8}$$

(2) *Velocity* $\widehat{\mathbf{U}}^{n+1} = (\widehat{U}^{n+1}, \widehat{V}^{n+1}) := \left(\left(\widehat{U}_1^{n+1}, \ldots, \widehat{U}_{CN}^{n+1}\right)^{\mathrm{T}}, \left(\widehat{V}_1^{n+1},\right.\right.$
$\left.\left.\ldots, \widehat{V}_{CN}^{n+1}\right)^{\mathrm{T}}\right)$

$$\begin{pmatrix} M+\Delta t(\nu_H S_1+\nu_z S_2) & \Delta t F \\ -\Delta t F & M+\Delta t(\nu_H S_1+\nu_z S_2) \end{pmatrix} \begin{pmatrix} \widehat{U}^{n+1} \\ \widehat{V}^{n+1} \end{pmatrix} = \begin{pmatrix} R_U^{n+1} \\ R_V^{n+1} \end{pmatrix} \tag{9}$$

(3) *Pressure increment* $Q_s^{n+1} = P_s^{n+1} - P_s^n := (p_{s1}^{n+1}, \ldots, p_{sNP}^{n+1})^{\mathrm{T}} - (p_{s1}^n, \ldots, p_{sNP}^n)^{\mathrm{T}}$

$$AQ_s^{n+1} = D_1^{n+1} \Rightarrow P_S^{n+1} = Q_s^{n+1} + P_s^n \tag{10}$$

(4) *Velocity* $\mathbf{U}^{n+1} = (U^{n+1}, V^{n+1}) := \left((U_1^{n+1}, \ldots, U_{CN}^{n+1})^{\mathrm{T}}, (V_1^{n+1}, \ldots, V_{CN}^{n+1})^{\mathrm{T}} \right)$

$$\begin{pmatrix} M & 0 \\ 0 & M \end{pmatrix} \begin{pmatrix} U^{n+1} \\ V^{n+1} \end{pmatrix} = \begin{pmatrix} M & 0 \\ 0 & M \end{pmatrix} \begin{pmatrix} \widehat{U}^{n+1} \\ \widehat{V}^{n+1} \end{pmatrix} + \begin{pmatrix} G_U^{n+1} \\ G_V^{n+1} \end{pmatrix} \tag{11}$$

(5) *Vertical component of the velocity* $W^{n+1} := (W_1^{n+1}, \ldots, W_{LN}^{n+1})^{\mathrm{T}}$

$$M_L W^{n+1} = D_2^{n+1} \tag{12}$$

M, S_1, S_2, F and A are all symmetric and sparse square matrices, and M_L is a diagonal matrix which is obtained by row sum of the elements of the linear mass matrix. We solve system (9) by a preconditioned Bi-CGSTAB method with a diagonal preconditioner because it is a well conditioned non-symmetric system so that this preconditioner may yield an efficient parallel solver. The rest of the systems are symmetric and solved by a preconditioned Conjugate Gradien (CG) method with diagonal preconditioner system (11) since the matrix is also very well conditioned, and by the incomplete Choleski conjugate gradient system (10); the solution of the latter system is calculated by a single processor.

The scheme to calculate the solution $(\mathbf{u}^{n+1}, W^{n+1}(\mathbf{u}), p_s^{n+1}, \theta_i^{nn+1})$ at time instant t_{n+1} is the following:

(1) Generate the grid and calculate the matrices of the systems once and for all at time instant $t = 0$.
(2) For $n = 1, 2, \ldots, N$ do:
  (2.1) For each vertex $\mathbf{x}_i$ of the tetrahedra calculate the departure point $\mathbf{X}(\mathbf{x}_i, t_{n+1}; t_n)$ by numerically solving the ordinary differential equations (5).
  (2.2) Calculate the right-hand side of the systems (8) to (12).
  (2.3) Solve system (8).
  (2.4) Calculate ρ^{n+1} by the equation of state.
  (2.5) Solve the systems (9)–(12).

Several remarks are in order.

Remark 2 The Lagrange-Galerkin (LG) method used in the model yields integrals of the form

$$\int_{\overline{K}^{n,n+1}} \phi_j(\mathbf{X}(\mathbf{x}, t_{n+1}; t_n))\phi_i(\mathbf{X}(\mathbf{x}, t_{n+1}; t_n)) \tag{13}$$

that have to be approximated with high accuracy for the method to maintain the theoretical stability and convergence properties studied in Bermejo and Saavedra (2012) and Bermejo et al. (2012). Here $\overline{K}^{n,n+1}$ is the tetrahedron whose vertices are the points $\mathbf{X}(\mathbf{x}_i, t_{n+1}; t_n)$ and $\{\phi_i\}$ is the set of global basis functions of the finite element space. The important point (in terms of CPU time savings) is that the number of departure points to be calculated now is M, M being the number of interior mesh points, instead of $NE \times NCP$, which is the number of the departure points to be calculated in the conventional LG as presented in Douglas and Russell (1982) and Pironneau(1982). Here, NE is the number of elements in the mesh and NCP is the number of quadrature points per element. The integrals on $\overline{K}^{n,n+1}$ are approximated by high order quadrature rules in the usual way of finite element technology. It can be proved that our LG-projection method is unconditionally stable in the L^2-norm.

Remark 3 The calculation of the departure points $\mathbf{X}(\mathbf{x}_i,t_{n+1}; t_n)$ is done by the second-order implicit scheme of González Gutiérrez LM and Bermejo (2005) using the algorithm of Allievi and Bermejo (1997) to search the points in the fixed mesh. The model can also use the Runge–Kutta methods of order two and four of Xiu and Karniadakis (2001).

Remark 4 An accurate calculation of the integrals (13) yields a very small error $\int_D A^n(\mathbf{X}(\mathbf{x}, t_{n+1}, t_n) - \int_D A^n(\mathbf{x})$, in other words, the method has good global conservation properties; nevertheless, we can improve the conservation property, in other words, $\int_D A^n(\mathbf{X}(\mathbf{x}, t_{n+1}, t_n) = \int_D A^n(\mathbf{x})$ by adapting the conservative quasi-monotone semi-Lagrangian scheme of Bermejo and Conde (2002) to the LG finite element framework.

Remark 5 Since the LG-projection method is unconditionally stable in the L^2-norm, the model supports CFL numbers greater than one. This has an influence on the width of the halo region because it has to contain the backtracked elements $\{\overline{K}^{n,n+1}\}$; see Malevsky and Thomas (1997). In fact, one may have two halo regions, namely, one halo for the calculation of the integrals

$$\int_{\overline{K}^{n,n+1}} \phi_j(\mathbf{X}(\mathbf{x}, t_{n+1}; t_n))\phi_i(\mathbf{X}(\mathbf{x}, t_{n+1}; t_n))$$

which depends on the CFL number, and another halo for the solution of the elliptic problems. In the actual implementation of our model both halo regions are the same.

4. Parallelization strategy

The strategy chosen (Crainic and Toulouse, 2010) for parallelizing the sequential ocean model was to obtain maximum speedup while safeguarding the portability of the parallel code as well as maintaining consistency with the results obtained with the sequential version. This implied efficient use of the resources of current cluster architectures based on shared memory nodes (multicore processors). In these systems, two levels of parallelism can be exploited: parallelism between the nodes of the cluster and parallelism between the multiple cores of a node. To exploit these levels of parallelism, a previous exhaustive study of the sequential application code was performed to find the main sources of parallelism, emphasizing its two main components, the processing task and the data to be processed. This resulted in two possible and compatible approaches:

- Data decomposition. In this approach, the data to be processed are decomposed so that each parallel task processes a portion of the data. This solution allows exploitation of the inter-node and intra-node parallelism available in modern high performance computers, that is, distributing the data to be processed among the cluster nodes and exploiting the multiple cores available in each node.
- Functional decomposition. In this approach, the focus is on the processing task rather than on the data to be processed. The main idea is to break down the processing task into multiple subtasks performing a portion of the overall work to be executed in parallel.

The next step is to combine these approaches, as suggested by other authors (Rabenseifner et al., 2009), to obtain good performance in current hybrid architectures. This involves employing a hybrid programming model, that is, using OpenMP for parallelization inside each node and MPI for passing messages between nodes.

4.1 Data decomposition: Inter-node parallelization

The data used to model the Mediterranean Sea consist of a three-dimensional mesh with thousands of points on the horizontal and several vertical levels. Unstructured grids are also used in Fringer et al. (2006) and Cowles (2008) while Sannino et al.(2001), Henshawa and Schwendemanb (2008), Wang et al.(2010) and Tseng and Chien (2011) use regular meshes and Luong et al. (2004) use a multiblock grid, with rectangular and curvilinear grids. The advantage of irregular meshes with respect to regular meshes is their variable resolution. This allows refining of the mesh in selected regions in order to solve small-scale dynamics and strong topographical gradients. It also enables a better representation of the topographical features of the ocean basin, such as the Strait of Gibraltar and the Strait of Messina, offering high resolution for specific areas. Some regular meshes try to solve this problem by using an unstructured mesh refinement to locally increase the resolution (Henshawa and Schwendemanb, 2008).

A domain decomposition method has been used to distribute the simulation processing tasks among available processors, that is, computing each variable for each mesh point at each time step. Thus, the Mediterranean computational mesh has been horizontally decomposed into a set of smaller submeshes which are distributed among the cluster nodes. This is the classical approach to parallelize ocean models on clusters (Sannino et al., 2001; Luong et al., 2004; Fringer et al., 2006; Cowles, 2008; Wang et al., 2010; Tseng and Chien, 2011), and it is also used for other similar problems such as reservoir simulations (Shuttleworth et al., 2009), although in our case the use of an unstructured mesh makes it more difficult to distribute the mesh among the cluster and to achieve good load balancing between all the cluster nodes.

Each cluster node runs a copy of the processing task code on its allocated submesh, while all the numerical algorithms used are almost the same as in the sequential version. These algorithms compute the values of each mesh point taking into account the values of their neighbouring points. Thus, to obtain the same results as with the sequential version, the processor responsible for each submesh needs to be able to consult points from neighbouring submeshes. Therefore an overlapping domain decomposition method has been used: a data slice has been added at the boundaries of each submesh, corresponding to neighbouring submeshes, in such a way that all the submeshes overlap. An important difference with respect to the classical approach is that each submesh is updated in an almost independent way, in other words, the initial linear equation system is substituted by N linear equation subsystems, one for each submesh, using almost the same numerical algorithms in each node as in the sequential version but applied to its submesh.

In order to guarantee a balanced workload between the cluster nodes, a separate application has been implemented that decomposes the overall computational mesh taking into account the number of points in the horizontal model domain and the number of requested processors, so that each cluster node has a similar amount of work to do at each simulation time step. Figure 1 displays a possible decomposition of the Mediterranean basin into 32 submeshes.

This parallelization strategy has been implemented using a message-passing programming model, through the MPI library, as this model fits naturally into the distributed-memory target architecture, which is effectively a cluster of processors. The choice of MPI guarantees the portability of the parallel code. The details of this distributed implementation are shown in Section 5.

4.2 Data decomposition: Intra-node parallelization

The study of the processing task has been performed using the profiling tool Callgrind (Valgrind, 2012), and the profile data visualization tool KCachegrind (KCachegrind, 2013), used for sequential and multi-threaded applications performance analysis. These tools provide information about the functions called during application execution and display it as a call graph, including data about the caller–callee relationship between functions, the number of calls and the cost of each function.

A summary of the results obtained for a two-step simulation of the ocean model is shown in Figure 2. This graph shows the functions consuming more than 10% of the total execution time. As can be seen Matrix_Vector_Product is the most time-consuming function (83.85% of the total execution time) and therefore efficient parallelization of this function has been taken into account to improve overall

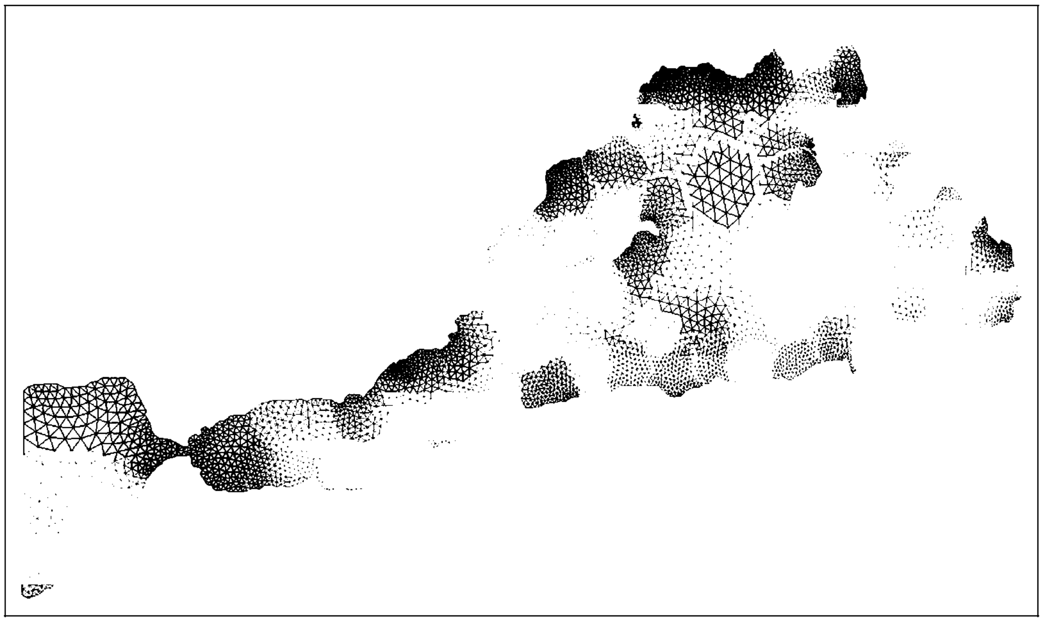


Figure 1. Decomposition of the western Mediterranean basin into 32 submeshes.

execution time. This function is employed in a special version of the conjugate gradient method, and consists of a loop that processes all the elements, one by one, used in the ocean model and accumulating the obtained results in an array representing the data domain. Mathematical libraries that support BLAS (basic linear algebra subprograms) cannot be used to implement this CG method as the ocean grid is supported by a data structure representing the vicinity of each node, instead of a traditional two-dimensional matrix. The advantage of using this data structure is that each element can be solved individually. This avoids clogging up memory with huge matrices.

To deal with the parallelization of this function, a loop parallelization approach has been implemented, following a previous in-depth analysis of possible loop-carried dependencies as well as the data used on each iteration. As there are no loop-carried dependencies, loop-level parallelism has been exploited using a shared memory programming model, with OpenMP as the programming interface. Parallelization was achieved using OpenMP directives to distribute the loop iterations among the threads, where the number of threads corresponds to the number of CPUs available on each cluster node. Regarding the data used in the loop, the data structure where values are accumulated on each iteration is shared. Therefore, auxiliary private data structures have been defined for each thread to avoid race conditions, and a critical section has been declared where all the auxiliary data are accumulated in the shared data structure.

This parallelization approach provides two advantages. The first is that the use of OpenMP compiler directives avoids the need to maintain separate sequential and parallel code versions, and also provides different scheduling strategies for parallel loops and a portable code. The second advantage is that it exploits the current multicore processor characteristics, that is, multithreading and a shared memory model that allows sharing of the common data used on each loop iteration with a minimum latency cost (sharing cache at L2 and L3 levels). The results obtained in terms of speedup and scalability are shown in Section 7.

4.3 Functional decomposition

This approach has been analysed studying the sequential dependencies between the different computations performed in each simulation step, trying to break them down into multiple subtasks that could be executed in pipeline mode. The time consumed by these computations has also been measured, resulting in an unbalanced solution. This is mainly due to the velocity computation which employs most of the simulation time (92%). Therefore this approach has been discarded.

5. Distributed implementation

The scheme to calculate the mathematical solution to the physical model was described at the end of the Section 3. The skeleton of the procedure to advance the simulation one time step is the following:

1. Calculate the convective terms using the method of characteristics, equations (5).
2. Calculate the temperature and the salinity, conservation equations (8).
3. Calculate the density, state equation (3).
4. Calculate an approximation of the horizontal velocity components, viscous step equations (9).
5. Calculate the pressure increment, Poisson equation (10). As previously stated in Remark 1 (p. 6), the overall pressure for the full mesh should be conserved.
6. Calculate the final horizontal velocity components, equations (11).

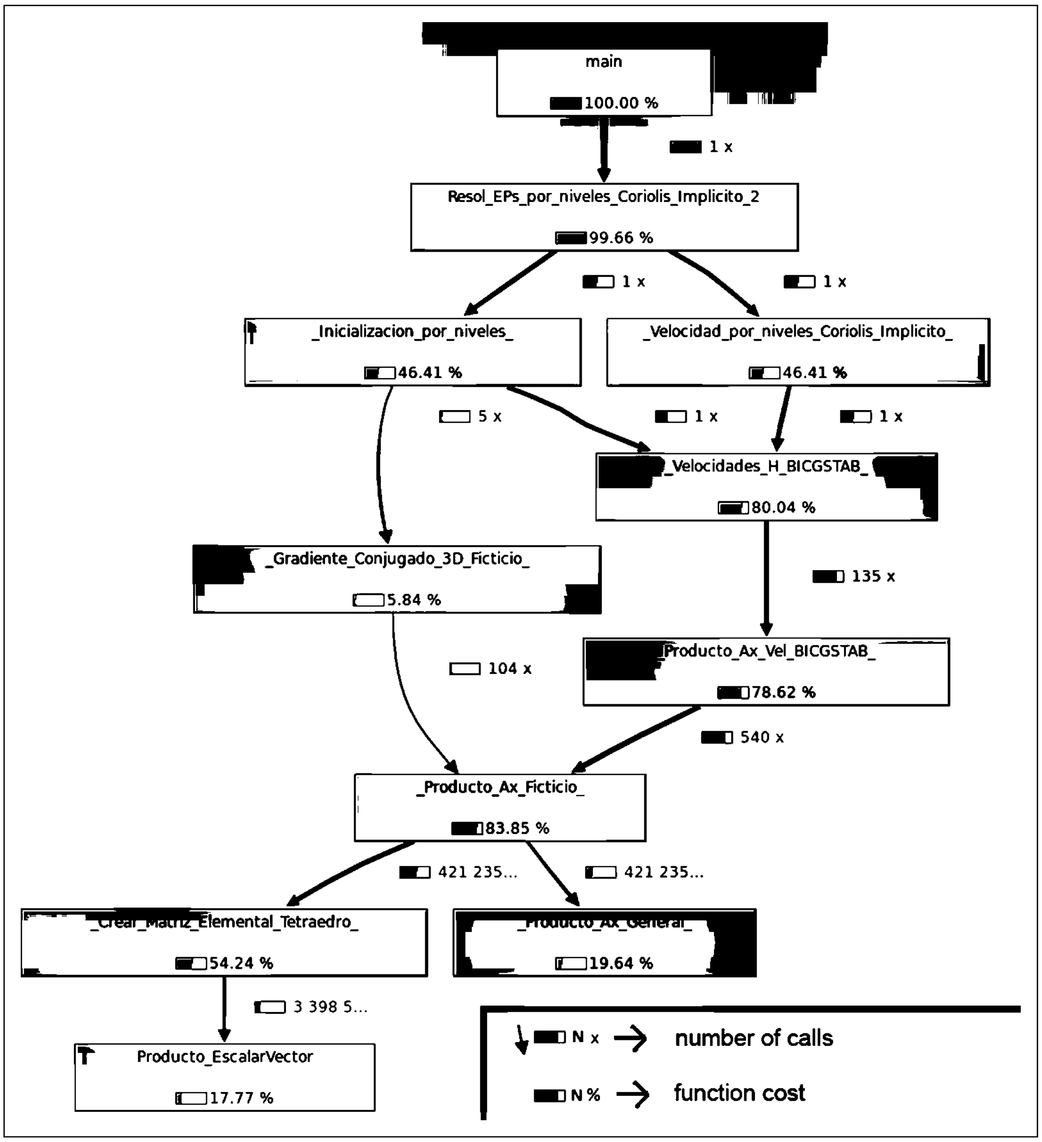


Figure 2. Sequential ocean model application call-graph view.

7. Calculate the vertical velocity component, equation (12).

To achieve maximum scalability this initial sequential implementation has been adapted to execute on clusters of multicore processors through data decomposition. The original input data of the full mesh are preprocessed and divided into as many submeshes as needed. Then the classic master–slave approach has been applied. A special process called 'master' coordinates the calculations performed by a set of worker processes called 'slaves', each slave being responsible for one of the submeshes the data was divided into. There are several reasons for choosing this approach rather than other more decentralized parallel implementation approaches (Crainic and Toulouse, 2010):

- The master obtains a global state at the end of each simulation step, providing a suitable point to easily implement an application-snapshot mechanism. This allows the execution point to be recovered in case of system failure which is something that could happen on long-term executions of days or weeks.
- The one time-step simulation procedure is adapted to the slaves reusing most of the original code but applying it only to a subset of the original data. Code for data exchanges between the slaves and the master is required at specific points, but collective communication mechanisms are used to introduce as little delay as possible.
- There is at least one fundamental calculation in each simulation step that needs to be solved globally for the entire mesh as one unit, the calculation of the pressure increment. The proposed model provides a simple solution.
- Good scalability. It scales well as long as the computation performed by the slaves compensates for the communication needed between the master and the slaves in order to distribute the input data, coordinate their work and collect the output data.

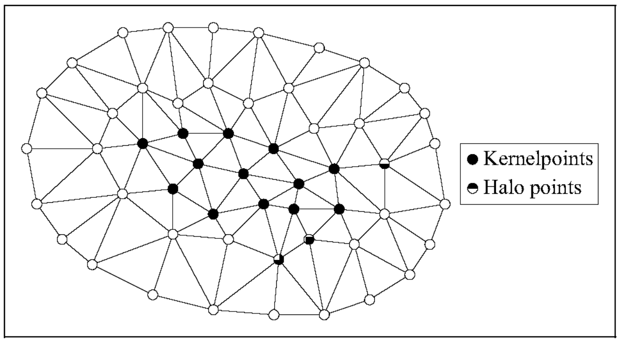


Figure 3. Points used by a slave to calculate its submesh: kernel points, belonging to that submesh, and halo points, width 1, which are copies of points belonging to neighbouring submeshes.

The final performance of the simulation largely depends on the slaves working over a set of well balanced submeshes, because a synchronization barrier is needed at the end of each simulation step and therefore the overall performance depends on the slowest slave. This is the main goal of the mesh division preprocessing, which is also a computationally intensive application, though fortunately its resulting division (see example in Figure 1) is computed only once for each mesh and then reused many times. This is why it was designed as an independent application.

The mesh division algorithm used adapts the k-means (MacQueen, 1967) and Kernighan–Lin (Kernighan and Lin, 1970) algorithms to work on irregular meshes with non-uniform point density. As in the case of other authors (Rivera et al., 2010) our algorithm uses the Kernighan–Lin for local refinement trying to simultaneously balance the number of mesh points in each submesh and to minimize the boundaries between the submeshes, in other words, balancing the computational requirements for each submesh and minimizing the communication needed. Other models use a different strategy to balance the workload, such as that in Henshawa and Schwendemanb (2008) which uses a bin packing algorithm, the best-fit decreasing bin packing algorithm, or Luong et al. (2004), which uses OpenMP to balance the workload in a dual-level parallel code with MPI. Other authors rely on libraries, such as Cowles (2008) who uses the METIS library (METIS, 2013), or Fringer et al. (2006) who use the parallel version, ParMETIS library (ParMETIS, 2013).

Aggregating the simulation results of each submesh does not yield the same result as simulating the original full mesh, in other words, completely separate calculations for each submesh are not possible. As shown in Figure 3 every slave also needs to consult and compute points around its submesh, known as ghost points (Luong et al., 2004; Henshawa and Schwendemanb, 2008; Tseng and Chien, 2011) or halo points (Sannino et al., 2001; Cowles, 2008; Wang et al., 2010). This implies not only more redundant computation in the slaves, as can be seen in the following algorithm description, but also the need to exchange some data between neighbouring slaves through the master at every time step.

The width of the halo around each submesh is a fundamental parameter (Wang et al., 2010) for simulation accuracy but, at the same time, has a significant impact on the amount of both computation and communication required in the slaves, as shown in Section 6.2. To improve load balancing and overall application performance a further refinement of the mesh division algorithm was introduced, taking into consideration the number of ghost points.

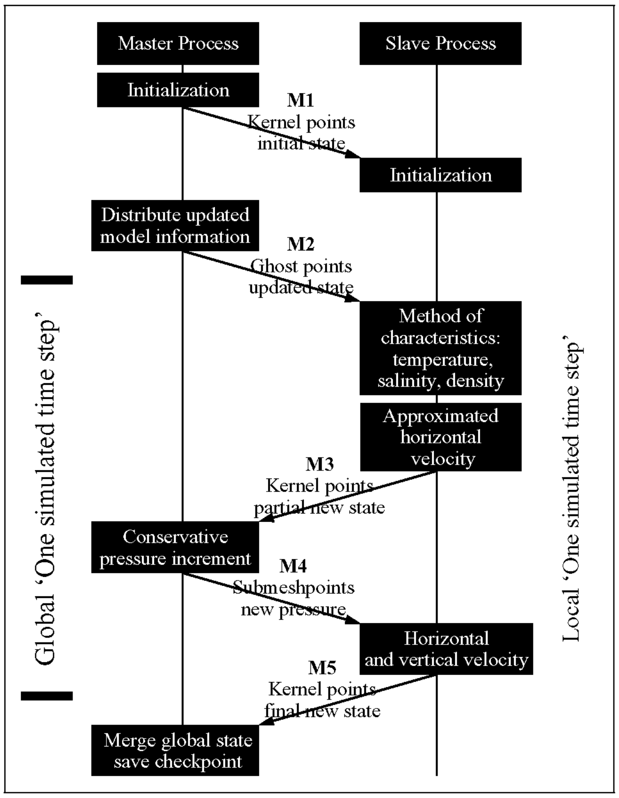

Master Process
Slave Process
Initialization
M1
Kernel points
initial state
Initialization
Distribute updated model information
M2
Ghost points
updated state
Method of characteristics: temperature, salinity, density
Approximated horizontal velocity
M3
Kernel points
partial new state
Conservative pressure increment
M4
Submeshpoints
new pressure
Horizontal and vertical velocity
M5
Kernel points
final new state
Merge global state save checkpoint
Global 'One simulated time step'
Local 'One simulated time step'

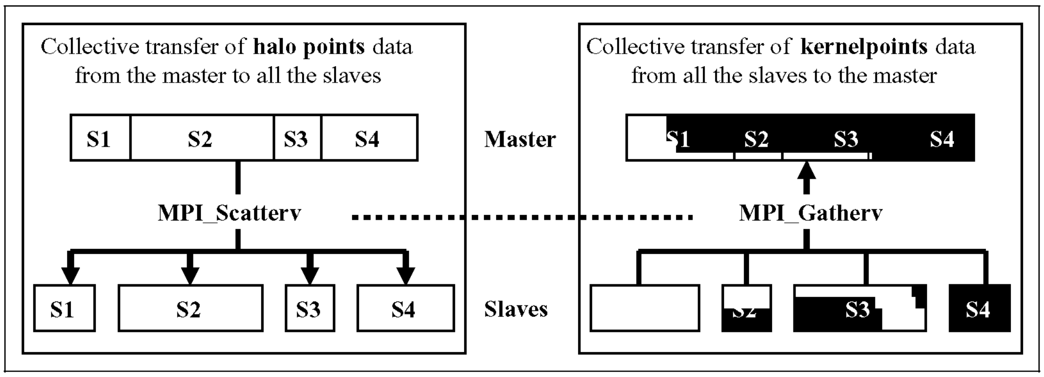
Figure 4. Master–slave system and intercommunication.

The following diagram (Figure 4) shows the interrelation between the master and each slave in terms of algorithm steps and communication. M1 to M5 are the messages exchanged with each other.

The basic algorithms followed by the master and the slaves are the following.

Master algorithm

1. Application initialization. Send message M1 to each slave with its submesh description and the corresponding kernel points data.
2. Send message M2 to each slave with updated information for its submesh ghost points corresponding to the overlapping areas.
3. The pressure increment should be computed centralized at the master and messages M3 and M4 are exchanged.
4. Wait to receive message M5 from each slave with the new state of its kernel points, and merge them to compose the state of the global mesh. Here a recovery snapshot is periodically saved.
5. Go back to step 2.



Collective transfer of halo points data from the master to all the slaves
S1
S2
S3
S4
MPI_Scatterv
S1
S2
S3
S4
Master
Slaves
Collective transfer of kernelpoints data from all the slaves to the master
S1
S2
S3
S4
MPI_Gatherv
S3
S4

Figure 5. MPI functions used to transfer the halo points of all the submeshes, S1, . . . , S4, from the master to the corresponding slaves, and the kernel points of all submeshes from the slaves to the master.

Slave algorithm

1. Wait to receive message M1 from the master with its assigned submesh and the corresponding initial data.
2. Wait to receive message M2 from the master with updated model information about the ghost points around its submesh.
3. Compute one time step of the simulation applying the original sequential procedure to its submesh, including the ghost points.
4. The pressure increment should be computed centralized at the master and messages M3 and M4 are exchanged.
5. Send message M5 to the master with the new model state corresponding to its kernel points.
6. Go back to step 2.

The master–slave system has been implemented using the MPI standard. Thus, the application can be executed both on distributed and shared memory cluster architectures. The first MPI process plays the role of master. It performs the whole initialization and then executes the coordination loop using the collective transfer operations MPI_Scatterv and MPI_Gatherv to efficiently communicate with the slaves (see Figure 5).

This kind of distributed parallelization is compatible with other levels of parallelism within the multiple cores or processors in each cluster node (with OpenMP or using SIMD operations on those cores). The resulting speedup obtained with all these parallelization strategies is shown in the following section.

6. Methodology and viability

6.1 Methodology

The performance analysis of the different parallelization techniques used in this paper for both shared and distributed memory computers has been carried out on the supercomputer Magerit at the CesViMa Supercomputing Center, whose main characteristics are:

- 1036 eServer BladeCenter JS20 nodes, each one with two PowerPC processors at 2.2 GHz (8.8 GFlops) and 4 GB of RAM. There were up to 256 nodes available for our experiments.
- 168 eServer BladeCenter JS21 nodes, each one with four PowerPC processors at 2.3 GHz (9.2 GFlops) and 8 GB of RAM.
- The nodes are connected by a high performance optical fibre Myrinet. Gigabit auxiliary nets are also provided for control and management.
- 256 hard disks of 750 GB organized as a fault-tolerant distributed file system (GPFS).
- SLURM/Moab job queue systems that guarantee exclusive processor assignment to each job.

All the processing nodes operate independently and with the same software configuration.

For the viability and precision study, simulations with 100 steps were executed. As the behaviour of the application after initialization was quite similar for all the simulation steps, the number of steps was reduced from 100 to 12 to evaluate the speedup, scalability and efficiency of the parallel version, in order to reduce the huge number of hours of simulation needed, as every simulation was executed several times to avoid spurious data.

We tested the scalability by working with two meshes, as mentioned in Section 1. In the first mesh the number of unknowns to be calculated at every time step was 6×10^6, whereas in the second mesh this number was 1.6×10^7. Each data set was executed for 24 hours of simulated time, corresponding to 12 simulation steps. Although the simulated time is short, it is sufficient taking into account the required number of repetitions of all experiments. The MPI performance tests have been carried out using the MPICH implementation and the xlc (IBM C) compiler. Execution time, speedup and efficiency have been analysed in our experiments, increasing the number of processors involved in the ocean model simulation from 1 to 256.

6.2 Viability

It was expected that execution time would improve by using a higher number of processors and a smaller submesh

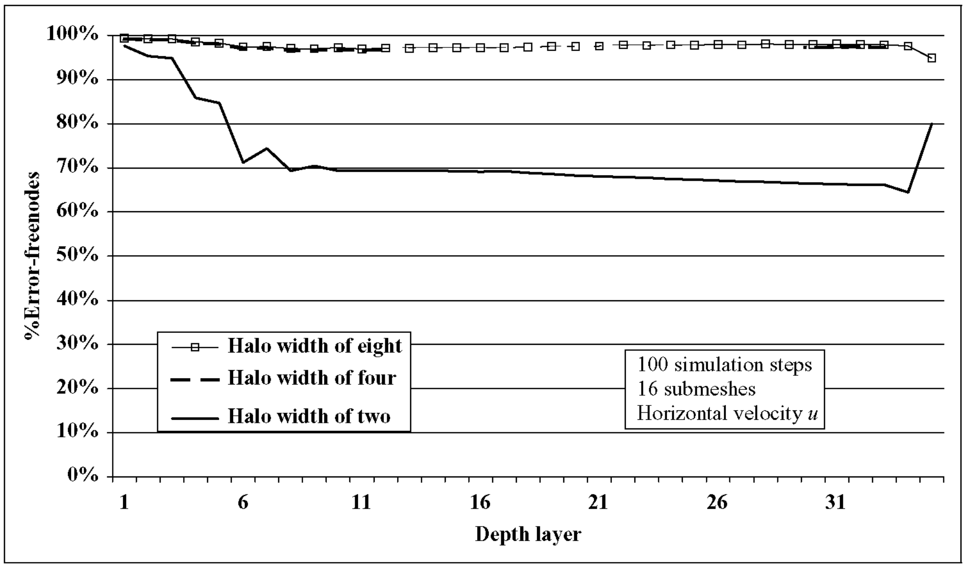


Figure 6. Percentage of error-free points for horizontal velocity *u* and different halo widths (two, four and eight), with 16 submeshes and after 100 simulation steps.

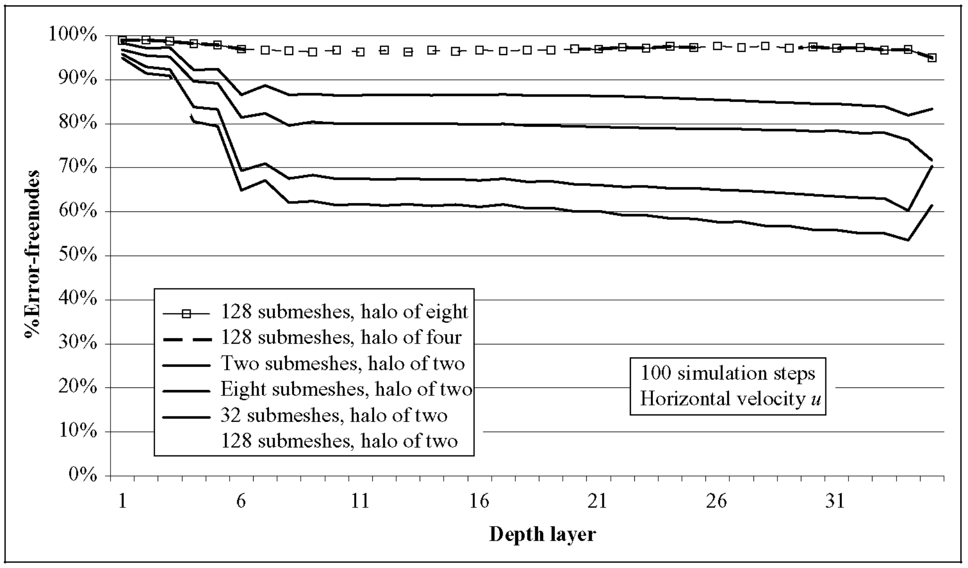


Figure 7. Precision of results versus number of submeshes (submesh size) shows that the halo width is the most important factor.

size. On the other hand, reducing the submesh size is also a potential source of errors in the results. Correct results were ensured by using enough ghost points around each submesh to obtain the same results as with the sequential version. However, introducing more ghost points also increases execution time as they need to be updated at every simulation step. Experiments with different ghost strip widths (halo widths) were performed and the influence of the submesh size was also studied. Figure 6 shows the percentage of error-free points (relative error $< 1\%$) obtained at each depth level for calculating the horizontal component of velocity u for a halo width of two, four and eight rows of points, 16 submeshes and 100 simulation steps. Tests in this case consisted of more extended simulations to allow the detection of cumulative precision errors. We tested the horizontal velocity u because it has been demonstrated to be the most sensitive to errors. It can be seen from this figure that a halo width of two is clearly insufficient, as a significant part of the results includes errors, whereas a width of four avoids this problem. A halo of eight rows shows exactly the same behaviour as one with four, but the execution time is higher (see Figure 8).

Figure 7 shows the influence of the number of submeshes on the precision of the results. For a halo width of two rows of points around the submesh, the percentage of error-free points decreases considerably as the number of submeshes increases. For a width of four, the results are almost free of error, even for a high number of small

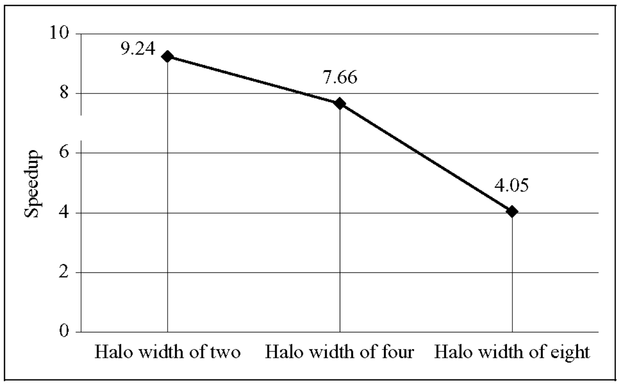


Figure 8. Speedup obtained for different halo widths, using the medium-size mesh with 16 submeshes and after 100 simulation steps.

partitions (128 partitions). Results in this case are clearly better than the best partition configuration using a ghost area width of two rows. Thus, halo width is more important for correct results than submesh size or number of partitions. It can also be concluded that the minimum halo width needed is four. Using this halo width, the simulation results are the same as those obtained with the sequential application, independent of the number or size of the submeshes.

It should be noted that as the halo width increases, the computation and synchronization needed for every submesh are also higher, and therefore the speedup decreases (see Figure 8). Therefore, halo widths above four do not improve the quality of the results, but considerably decrease speedup.

A further conclusion is that ghost nodes are relevant for effective load balancing, and have therefore been taken into account in the mesh division algorithm as it was seen in Section 5.

7. Performance and scalability

Performance results are presented considering speedup as the performance gain achieved by the parallelization, that is, the ratio of sequential execution time over the parallel execution time. This classical definition of speedup differs from other definitions which consider the serial time as the execution time of the parallel application running on just one processor of the parallel system. Efficiency is considered as the ratio of speedup to the number of processors, and measures the fraction of time during which a processor is usefully employed.

To analyse the speedup, several experiments varying the number of processors have been conducted for the two parallelization strategies described in Section 4. First, the speedup with shared memory multiprocessors was tested using OpenMP, with configurations of two and four cores. Then, the speedup with distributed memory was analysed with configurations of 16, 32, 128 and 256 processor architectures. One processor played the master role, and the others were working in parallel with each submesh (7, 15, 31, 127 and 255) as slave processors. And finally, the speedup obtained combining both shared memory and distributed memory strategies was examined.

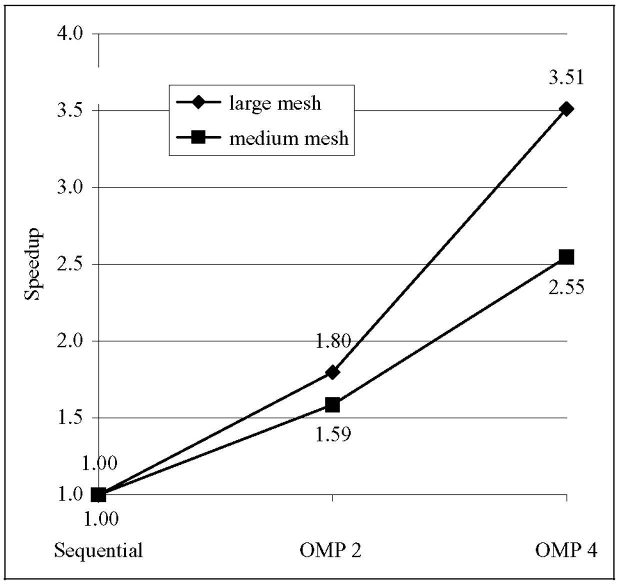


Figure 9. Speedup reached with OpenMP for two and four processors after 12 simulation steps.

7.1 Speedup

OpenMP. In Figure 9 the speedup obtained through parallelization with shared memory multiprocessors can be seen for different numbers of processors. This figure shows the good performance obtained with this parallelization strategy, as a speedup of 3.51 was reached using just four processors for the large-size mesh. A lower speedup was reached with the medium-size mesh, 2.55, due to the lack of parallelism.

MPI. Figure 10 shows the speedup obtained with distributed memory multiprocessors using MPI, as the number of processors increases from 1 to 256. This version offers good scalability even up to 256 processors, reaching a speedup of 53.09 for the large-size mesh. The speedup obtained with the medium-size mesh is lower, 25.3, as the computation time for each simulation step is shorter. The limiting factor in this case is communication, as information exchanges take almost the same time for both meshes. Also, the performance does not scale at the same rate as the number of processors, due to the following factors: communication overheads; some code regions such as the pressure computation are naturally sequential; higher proportion of ghost points with respect to kernel points at each submesh and hence increased computational requirements; workload balancing becomes more difficult with decreasing submesh size; and the partitions produced by the division algorithm are less uniform as the number of submeshes increases, resulting in a higher workload. Therefore, there is no need for the computational power offered by 256 processors when dealing with small data submeshes.

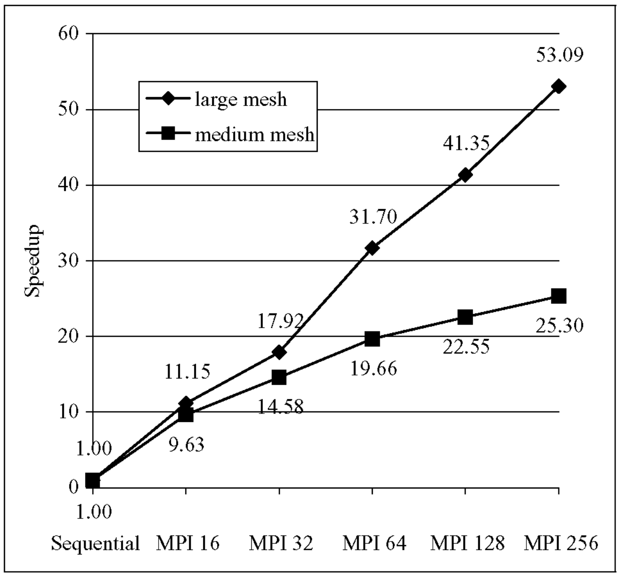


Figure 10. Speedup obtained using MPI for 16 to 256 processing nodes (15 to 255 submeshes) after 12 simulation steps.

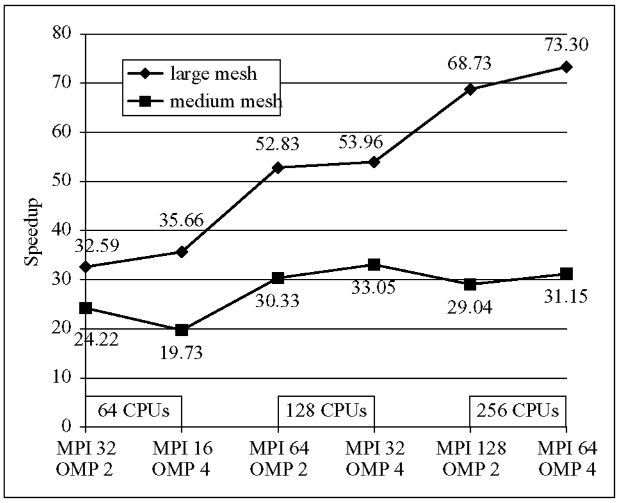


Figure 11. Speedup using OpenMP and MPI simultaneously.

Combining OpenMP and MPI. Figure 11 shows the overall speedup obtained combining the two parallelization techniques for different cluster configurations, achieving a maximum speedup of 73.3 with 256 processors. This hybrid parallelization strategy offers the best performance as it exploits parallelism at different hardware levels and enables more efficient use of the available resources in a cluster using OpenMP for the processors in one node and MPI for the different cluster processing nodes.

Using a large number of submeshes limits the speedup due to the low computation times needed for such small submeshes as the parallelization overheads increase with the number of submeshes. Using only MPI, with 256 processors over 256 nodes, yields a speedup of 53.09 for the large mesh (see Figure 10) whereas when using a smaller number of submeshes (64) the speedup reaches up to 73.3 for the same mesh and number of processors, 256 processors over 64 nodes. A smaller number of submeshes means lower communication costs and thus lower global computational requirements, and a better ratio of kernel points/ghost points and therefore less redundant computation and better workload balance.

Finally, Figure 12 shows the overall performance obtained in the Magerit supercomputer with different cluster configurations, considering all the parallelization techniques proposed: both OpenMP and MPI alone and combined.

These global results show that the performance can be considerably improved as the number of processors increases, and therefore the ocean model simulation should scale well when finer spatial and temporal resolution models are used. The medium mesh seems to reach the best performance executing with 32 processing nodes and four execution threads at each processing node, reaching a speedup of 33.05. The graph suggests further potential speedup for the large-size mesh if more than the 256 available processors are used.

7.2 Efficiency

As the number of processors increases, efficiency decreases as expected, but within levels that still allow high scalability, as can be seen in Figure 13. Performance results show that combining OpenMP and MPI not only yields better speedup, but also improves efficiency. This hybrid approach therefore seems to be the best choice to execute the simulation application. Figure 13 shows how, for the same number of processors, different parallel configurations yield different levels of efficiency. For 64 processors the best parallel option is MPI and OpenMP with two threads when the medium mesh is used, but MPI and OpenMP with four threads is better for the large-size mesh. For 128 and 256 processors, using MPI and OpenMP with four threads is the best solution.

7.3 Scalability of the master–slave model

The master–slave model used for the distributed memory approach has some synchronization and communication overheads that could limit scalability. As stated in Section 5 the master processor collects partial results from the slaves, composes the global mesh and distributes results among slave processors. Therefore, the performance of the master is critical for global application performance, as computations are performed sequentially, and the slave processors have to wait. Figure 14 shows the average master processor utilization versus average communication and waiting times for 128 processors (127 submeshes). The master utilization is only 28.19% and there are no bottlenecks. Its rate of increase is smooth and there is still room for further increase.

8. Conclusions

In this paper we have described and evaluated the parallelization of an ocean model with an irregular computational mesh for current cluster architectures based on multicore

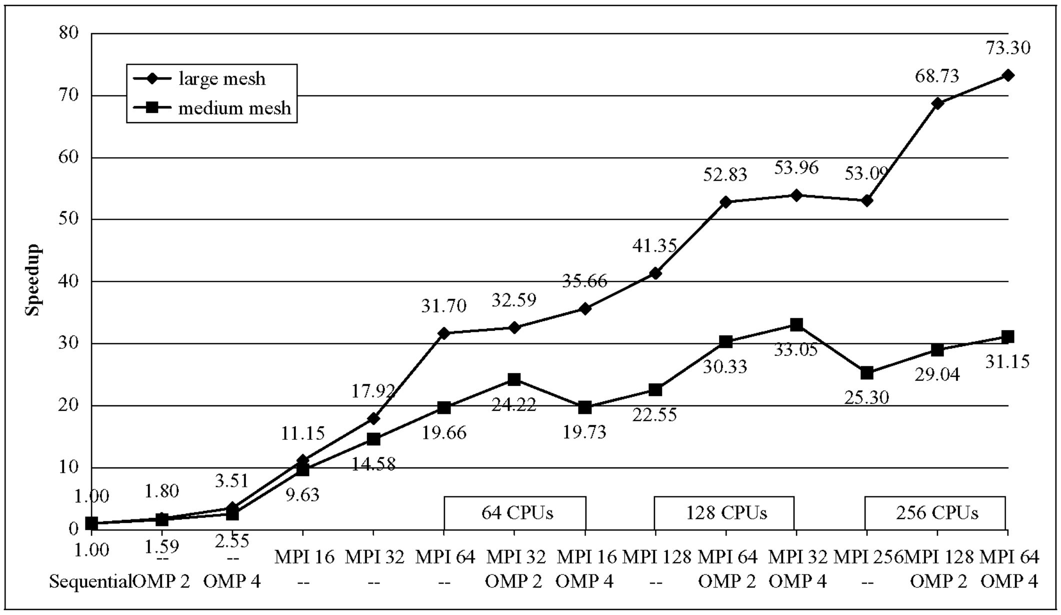


Figure 12. Global speedup obtained using only OpenMP, only MPI and combining both paradigms.

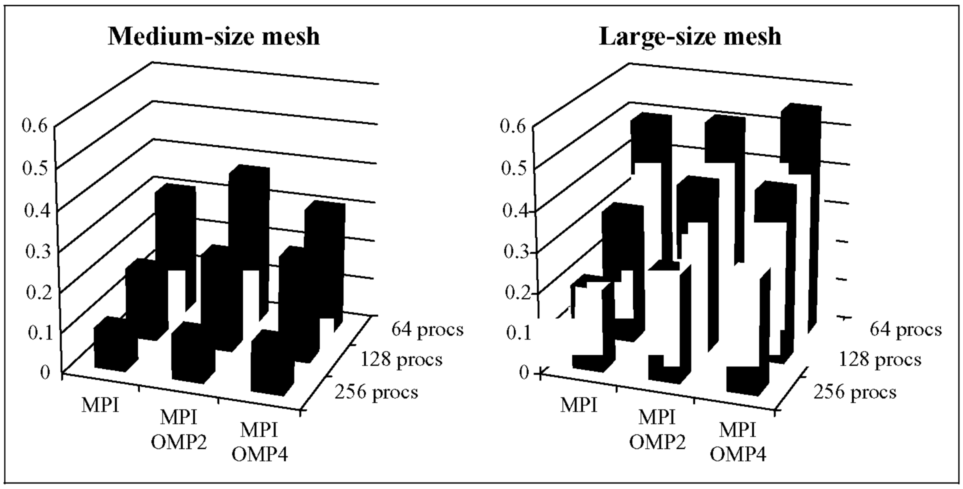


Figure 13. Efficiency obtained for the two meshes using only MPI and combining MPI and OpenMP, with different numbers of processors.

processors. This parallelization simultaneously exploits all the hardware levels of a cluster: intra-processor-level parallelism using vectorization; the shared memory level on each cluster node using OpenMP; and the distributed memory level between nodes using MPI. These techniques can easily be adapted to suit different architectures, using OpenMP to exploit the multiple cores of a computer or MPI for clusters or a combination of both.

The speedup obtained is 3.51 using OpenMP with four processors (Figure 9). Using MPI the speedup is 53.09 with 256 processors (Figure 10). The maximum speedup obtained, by combining MPI and OpenMP, is 73.3 with 256 processors, using the large-size mesh, as can be seen in Figure 11. These figures show that the parallelization is very scalable and the limiting factor for reaching the maximum speedup is the number of processors, in other words, a higher speedup could be obtained using additional processors. The best speedup is obtained by using MPI and OpenMP together because the number of messages between MPI processes needed to compute the model is lower with OpenMP, and it also reduces the number of submeshes needed, thus improving load balancing. Moreover it reduces the ratio of halo points versus kernel points, avoiding redundant computation.

The high speedup obtained offers scope for improving both the model and the simulations, by increasing the temporal and/or spatial resolution of the model to improve its reliability and by performing much longer simulations to analyse more realistic scenarios.

9. Future work

The first task at hand is to couple the new parallel ocean model with the existing parallel atmosphere model, using a

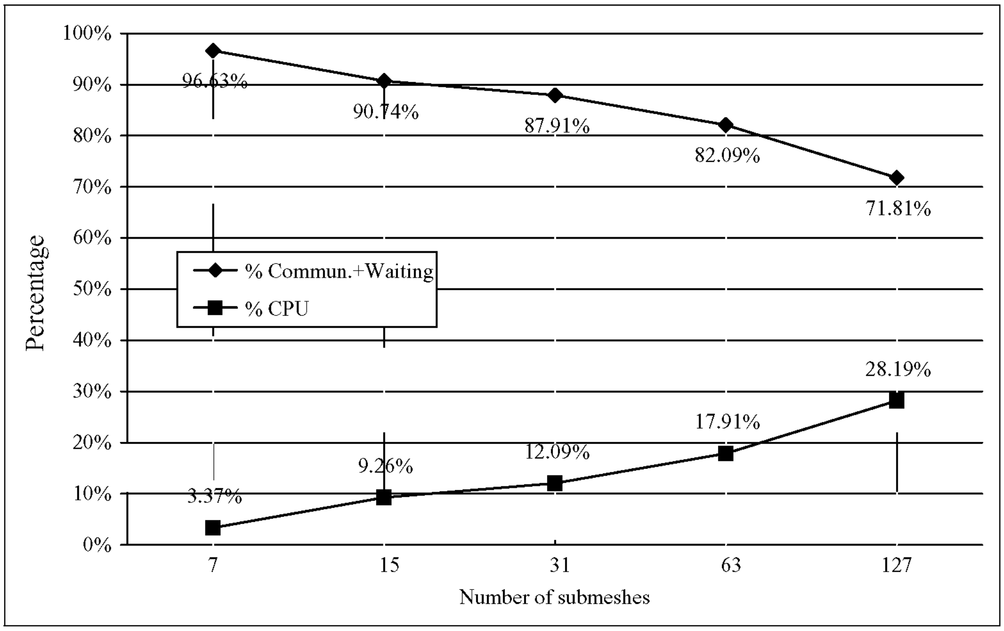


Figure 14. Percentage CPU and communication time at master processor.

higher temporal and spatial resolution in order to exploit the new performance of the parallel ocean model.

The authors are at present working on the complete Mediterranean basin model, and extending the parallelization performance analysis to the complete basin is also planned.

The future scope for improvement of the proposed parallelization includes several options. A major improvement would be to replace the current central algorithm for computing the pressure with a distributed one, thereby avoiding synchronization between each time step and the resulting loss of speedup.

Another possible improvement is to use an N-workers strategy instead of the master–slave approach. This offers the advantage of removing the master as the indispensable node for sending and receiving messages, thus making the application more scalable. Currently the master is not a bottleneck, but in the future, with an increased number of processors, it could become a bottleneck and in this way reduce the performance.

Another alternative to be explored is the use of parallel mathematical libraries that support BLAS in order to solve the linear equation systems by implementing typical algorithms from the literature, such as the conjugate gradient, to increase the portability and perhaps the performance. In this way all the code of the Matrix_Vector_Product function could be replaced by a parallel and optimized library, such as ACML (AMD Core Math Library), ATLAS (Automatically Tuned Linear Algebra Software, from Sourceforge), MKL (Intel Math Kernel Library) or the Performance Library (from Oracle), adapting the data structures to the chosen library.

Finally, an interesting alternative to explore is the use of graphics processing units (GPUs) to execute the more expensive computations, as GPUs offer enormous computing power at quite a reasonable price. The use of GPUs would offer very high performance for certain matrix calculations required by the model. This could be achieved by implementing certain particularly time-consuming routines at GPU level. This solution could also be integrated into the current solution, combining MPI, OpenMP and GPUs.

Acknowledgements

The authors thankfully acknowledge the computer resources, technical expertise and assistance provided by the Centro de Supercomputación y Visualización de Madrid (CeSViMa)

Funding

This work has been developed thanks to the financial support of the Spanish Ministry of Science and Education through the I+D national project CGL2007-66440-C04-04/CLI.

References

Allievi A and Bermejo R (1997) A generalized particle search-locate algorithm for arbitrary grids. *Journal of Computational Physics* 132: 157–166.

Bermejo R and Conde J (2002) A conservative quasi-monotone semi-Lagrangian scheme. *Monthly Weather Review* 130: 423–430.

Bermejo R and Saavedra L (2012) Modified Lagrange–Galerkin methods of first and second order in time for convection–diffusion problem. *Numerical Mathematics* 120: 601–638.

Bermejo R, Galán del Sastre P and Saavedra L (2012) A second order in time modified Lagrange-Galerkin-finite element method for the incompressible Navier-Stokes equations. *SIAM Journal on Numerical Analysis* 50: 3084–3109.

Chorin AJ (1968) Numerical solution of the Navier-Stokes equations. *Mathematics of Computation* 22: 745–762.

Cowles GW (2008) Parallelization of the FVCOM coastal ocean model. *International Journal of High Performance Computing Applications* 22(2): 177–193.

Crainic TG and Toulouse M (2010) Parallel meta-heuristics. In: M Gendreau and J-Y Potvin (eds) *Handbook of Metaheuristics (International Series in Operations Research & Management Science*, vol. 146). New York: Springer, pp. 497–541.

Douglas J and Russell TF (1982) Numerical methods for convection-dominated diffusion problems based on combining the method of characteristics with finite element or finite difference procedures. *SIAM Journal on Numerical Analysis* 19: 871–885.

Fringer OB, Gerritsen M and Street RL (2006) An unstructured-grid, finite-volume, nonhydrostatic, parallel coastal ocean simulator. *Ocean Modelling* 14: 139–173.

Gallardo-Andrés C, Galán del Sastre P and Bermejo R (2010) PROMES-MOSLEF: An atmosphere-ocean coupled regional model. Coupling and preliminary results over the Mediterranean basin. In: *4th HYMeX workshop*.

Garrido JE, Arias E, Cazorla D, et al. (2009) PROMESPAR: A parallel implementation of the regional atmospheric model PROMES. In: *Proceedings of the world congress on engineering*, London, UK, 1–3 July.

González Gutiérrez LM and Bermejo R (2005) A semi-Lagrangian level set method for incompressible Navier-Stokes equations with free surface. *International Journal for Numerical Methods in Fluids* 49: 1111–1146.

Guermond JL, Minev P and Shen J (2006) An overview of projection methods for incompressible flows. *Computer Methods in Applied Mechanics and Engineering* 195: 611–645.

Henshawa WD and Schwendemanb DW (2008) Parallel computation of three-dimensional flows using overlapping grids with adaptive mesh refinement. *Journal of Computational Physics* 227(16): 7469–7502.

KCachegrind (2013) Profile data visualization. Available at: http://kcachegrind.sourceforge.net (accessed 7 November 2013).

Kernighan BW and Lin S (1970) An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal* 49: 291–307.

Luong P, Breshears CP and Ly LN (2004) Application of multi-block grid and dual-level parallelism in coastal ocean circulation modeling. *Journal of Scientific Computing* 20(2): 257–275.

MacQueen JB (1967) Some methods for classification and analysis of multivariate observations. In: *Proceedings of 5th Berkeley symposium on mathematical statistics and probability*, pp. 281–297.

Malevsky AV and Thomas SJ (1997) Parallel algorithms for semi-Lagrangian advection. *International Journal for Numerical Methods in Fluids* 25: 455–473.

METIS (2013) Serial graph partitioning and fill-reducing matrix ordering. Available at: http://glaros.dtc.umn.edu/gkhome/metis/metis/overview (accessed 7 November 2013).

MPI (2012) MPI forum. Available at: http://www.mpi-forum.org/ (accessed 7 November 2013).

OpenMP (2013) OpenMP specifications. Available at: http://openmp.org/wp/openmp-specifications/ (accessed 7 November 2013).

ParMETIS (2013) ParMETIS – Parallel graph partitioning and fill-reducing matrix ordering. Available at: http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

Pironneau O (1982) On the transport-diffusion algorithm and its applications to the Navier-Stokes equations. *Numerische Mathematik* 38: 309–332.

Rabenseifner R, Hager G and Jost G (2009) Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: *17th Euromicro international conference on parallel, distributed and network-based processing*.

Rivera CA, Henichea M, Glowinskib R, et al. (2010) Parallel finite element simulations of incompressible viscous fluid flow by domain decomposition with Lagrange multipliers. *Journal of Computational Physics* 229(13): 5123–5143.

Sannino G, Artale V and Lanucara P (2001) An hybrid OpenMP-MPI parallelization of the Princeton ocean model. In: *Parallel computing: Advances and current issues*, pp. 222–229.

Shuttleworth R, Maliassov S and Zhou H (2009) Partitioners for parallelizing reservoir simulations. In: *Society of Petroleum Engineers reservoir simulation symposium*, The Woodlands, Texas, 2–4 February.

Temam R (1969) Sur l'approximation de la solution des équations de Navier-Stokes par la method des pas fractionnaires (II). *Archive for Rational Mechanics and Analysis* 33: 377–385.

Temam R and Ziane M (2004) Some mathematical problems in geophysical fluid dynamics. In: S Friedlander and D Serre (eds). *Handbook of Mathematical Fluid Dynamics*. North-Holland: Elsevier.

Tseng Y and Chien M (2011) Parallel domain-decomposed Taiwan multi-scale community ocean model. *Computers & Fluids* 45(1): 77–83.

Valgrind (2012) Instrumentation framework for building dynamic analysis tools. Available at: http://www.valgrind.org/ (accessed 7 November 2013).

Wang G, Qiao F and Xia C (2010) Parallelization of a coupled wave-circulation model and its application. *Ocean Dynamics* 60(2): 331–339.

Xiu D and Karniadakis GE (2001) A semi-Lagrangian high order method for Navier-Stokes equations. *Journal of Computational Physics* 172: 658–684.