Noname manuscript No.
(will be inserted by the editor)

Adaptation and evaluation of the Multisplitting-Newton and Waveform Relaxation methods over distributed volatile environments

Jean-Claude Charr · Raphaël Couturier · David Laiymani

Received: date / Accepted: date

Abstract This paper presents new adaptations of two methods that solve large differential equations systems, to the grid context. The first method is based on the Multisplitting concept and the second on the Waveform Relaxation concept. Their adaptations are implemented according to the asynchronous iteration model which is well suited to volatile architectures that suffer from high latency networks. Many experiments were conducted to evaluate and compare the accuracy and performance of both methods while solving the advection-diffusion problem over heterogeneous, distributed and volatile architectures. The JACEP2P-V2 middleware provided the fault tolerant asynchronous environment, required for these experiments.

Keywords Distributed clusters; Fault tolerance; Multisplitting-Newton; Waveform Relaxation;

1 Introduction

Many natural phenomenas and nuclear reactions (like climate change or nuclear fusion) are represented by differential equations systems. Solving these problems allows scientists to predict numerically the consequences of such reactions without having to physically realize the costly experiments. Thus the use of simulations drastically reduces the experimental cost and eliminates the potential risks that could be encountered while actually executing the reaction. However, most of the time these problems are very large and complex. Therefore, they cannot be solved by a single regular computing unit due to the lack of sufficient memory space or computing power. They can only be

Laboratory of computer sciences, University of Franche-Comté (LIFC)
IUT de Belfort-Montbéliard, Rue Engel Gros, BP 527, 90016 Belfort, France
Tel: +33-3-84587781
Email: {jean-claude.charr,raphael.couturier,david.laiymani}@univ-fcomte.fr

solved by using parallel machines which are very expensive or by using simultaneously many computing resources which form a distributed computing architecture. In the distributed computing literature, we can distinguish three types of distributed architectures:

1. Local cluster: It is composed of many computing units that are located in the same area. They have in general similar specifications and similar configuration and are connected via a local network with low latency and large bandwidth.
2. Distributed clusters: They are composed of many local clusters that are geographically distant from each others. Two computing units from distinct clusters may have different specifications and configurations. Moreover, the communication's latency between two computing units from distinct clusters is usually one hundred times higher than the latency between two computing units in a local network.
3. Global computing: It is composed of many public unused computing units ranging from desktops to PDAs. The main characteristics of this architecture are: the high volatility of the heterogeneous computing units (because the computing units are public and can be turned off or disconnected at any time by their respective owners) and the high latency of communications between computing nodes (because they communicate over Internet that is usually running over a DSL (Digital Subscriber Line) connection speed).

Although we have pointed out explicitly that the global computing architecture is very volatile, the rest of the described architectures are not crash free and at any moment, a computing unit can crash due to hardware or software problems. So, it is recommended to implement a fault tolerant mechanism while using any of the distributed architectures described above. In this paper, we are interested in the grid environment, which encompasses the distributed clusters and global computing architectures which suffers from the heterogeneity and volatility of the computing nodes, and from the high latency of the network interconnecting them.

There are already many parallel methods that efficiently solve large and complex differential equation systems in low latency environments like supercomputers or local clusters (for example PVODE [8]). As shown in [1], these methods are not well adapted for high latency environments because they are fine grained methods where communications between the various computing units are very frequent. This high number of communications between dependent subsystems reduces drastically the performance of the method in high latency environments. The aim of this paper is to present new adaptations of the Multisplitting-Newton [4,11] and the Waveform Relaxation [17,20,13] methods that solve large differential equations systems, to the grid context. Their adaptations are implemented according to the asynchronous iteration model [2] which is well suited to volatile architectures that suffer from high latency networks. Moreover, we provide a comparative study of those two coarse grained methods while solving the advection-diffusion problem over large scale volatile environments. JACEP2P-V2 [10] was used for executing these methods over the volatile and distributed architecture. This framework is dedicated to executing iterative asynchronous numerical methods over such architectures. It offers a decentralized fault tolerant convergence detection mechanism, a decentralized backup procedure and a distributed failure detection and recovery scheme.

The rest of this paper is organized as follows: in the state of the art section, we briefly present the differential equations systems and the PVODE solver which is dedicated to solving large systems of differential equations over parallel architectures. Then, we present the asynchronous iteration model and we show its benefits in volatile and high latency environments. In the third section, we present our adaptations of the Multisplitting-Newton method (MN) and the Waveform Relaxation method (WR) to the grid context. We explain the different steps of each method and illustrate their advantages and drawbacks. In the experiment section, we first compare the solutions of the three methods (PVODE, Multisplitting-Newton and Waveform Relaxation) in order to confirm that they all compute the same solution and to check the accuracy of each method. Then, we present JACEP2P-V2 which is a distributed, asynchronous and fault tolerant platform dedicated for designing parallel iterative asynchronous algorithms and executing them over volatile and high latency architectures. This middleware was used for executing both methods over such environments and comparing their performance while varying the number of crashes, the system's decomposition scheme and the number of sites used. The results of the experiments are discussed at the end of the fourth section. Finally, we end this paper with a conclusion and some perspectives.

2 State of the art

2.1 Differential Equations

As mentioned above, we are interested in solving differential equations which arise from the simulation of physical and natural phenomena. In particular, we focus on solving large and differential equations over grids.
There are two types of differential equations:

- **Ordinary differential equation** ('ODE') is a relation that contains functions of only one independent variable, and one or more of its derivatives with respect to that variable.
  Let y be an unknown function of x.

  $$y : \mathbb{R} \to \mathbb{R}$$

  An ODE of order n involving y has the following explicit form:

  $$F(x, y, y', y'', \ldots, y^{(n-1)}) = y^{(n)}$$

where $y^{'} = dy/dx$ is the first derivative with respect to x, and $y^{(n)} = d^n y/dx^n$ is the nth derivative with respect to x.

An ODE dependent of time is called a non stationary ODE or an initial value problem (IVP). An initial value problem of order n has the following general form:

$$F(t, y, y', y'', \ldots, y^{(n-1)}) = y^{(n)} \quad y(t_0) = y_0$$

where y is a function of t, t_0:initial time and y_0:initial values.

– **Partial differential equation** ('PDE') is an equation involving functions and their partial derivatives. For example, the wave equation is a PDE and it has the following form:

$$\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} + \frac{\partial^2 \psi}{\partial z^2} = \frac{1}{v^2} \frac{\partial^2 \psi}{\partial t^2}$$

To solve a PDE, we can use the method of lines ("MOL") [19] which approximates the PDE with a large ODE. The basic idea of the MOL is to replace the spatial (boundary value) derivatives in the PDE with algebraic approximations. Once this is done, the spatial derivatives are no longer stated explicitly in terms of the spatial independent variables. Thus, in effect only the initial value variable, typically time in a physical problem, remains. In other words, with only one remaining independent variable, we have a system of ODEs that approximate the original PDE.

2.2 PVODE

PVODE [8] is widely used for solving large systems of ordinary differential equations in parallel environments. It is an extension of the sequential package known as CVODE [13] which has been widely distributed and used. The parallelization of CVODE to PVODE was accomplished through the modification of the vector kernels, allowing them to operate on vectors that have been distributed across processors. The message passing calls between nodes are made through MPI [18]. Although PVODE contains many methods for the resolution of both stiff and non-stiff initial value problems, the standard approach to solve ODE systems is based on three steps:

– Converting the differential equations describing the system into a sequence of non-linear algebraic equations, using the Adams-Moulton Formula [9] to integrate non-stiff problems and the Backward Difference Formula ('BDF') [15] for stiff ones.
– Transforming the non-linear algebraic equations into a sequence of linear problems using a modified Newton method for systems converted by the BDF method or using a functional method for the ones generated by the Adams-Moulton integration.

– Solving the system of linear equations with Gaussian Elimination like methods or iterative ones.

To parallelize this method, the distributed system of linear equations is solved with a parallel solver which requires communicating with neighbors when computing boundaries' values. Thus, as shown in [1,6], this method is fine grained. It synchronizes at each internal step. Therefore, if the implementation of this method is executed over an architecture that suffers from high latency communications, the performance of this method is severely reduced due to the extremely penalizing and frequent synchronizations. Therefore, developing parallel coarse grained resolution methods is essential to solve efficiently large differential equation systems over grids.

2.3 The Asynchronous Iteration Model

Iterative methods are easily and efficiently parallelizable. Indeed, this class of algorithms, can be transformed effortlessly into parallel methods because most of the time, they just exchange data a few times in each iteration. However, common parallelizing models for iterative methods synchronize neighbors at each iteration while exchanging data and synchronize all the nodes while computing a residual value for detecting the global convergence of the parallel iterative method. These synchronizations reduce the overall performance of the parallel application, especially if it is executed on distributed and heterogeneous architectures. An alternative is to use the asynchronous iteration model.

When using an iterative method parallelized according to the asynchronous iteration model, each node does not have to synchronize with its computing neighbors. Indeed, at the end of each iteration (represented by filled rectangles in figure 1) computing nodes send their dependencies data to their neighbors (represented by arrows in figure 1) and begin the next iteration using the last received dependency data. They do not have to wait for the reception of dependency messages from their neighbors like in the synchronous iteration model. Thus, this relative independence eliminates idle time between consecutive iterations. Furthermore, in this model the loss of data messages is tolerated because no nodes wait for fresh dependency messages. They use the last ones they received to compute the next iteration. In this way, two nodes computing the same parallel asynchronous iterative application can execute different iterations at the same time. For example, in figure 1, the first node computes the sixth iteration and begins the seventh while the second node is still executing its fifth iteration. Thus, there is absolutely no synchronization when using the asynchronous iteration model. Thanks to this property, when a computing unit crashes while executing a parallel iterative asynchronous application, the rest of the computing units, executing the same application, continue their tasks and are not affected by this crash. However, this model cannot be applied on all iterative methods and it increases the number of iterations required for an application to converge. In [2], it has been shown that in a grid context, even with more iterations to converge, iterative parallel methods implemented according to the asynchronous iteration model are more efficient than the synchronous ones.

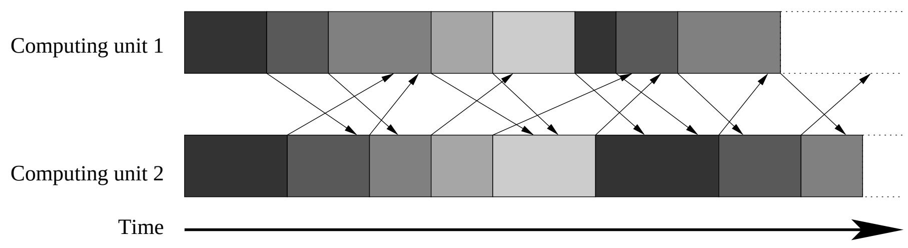

Fig. 1 Two computing units executing a parallel iterative application that is implemented according to the Asynchronous Iteration model.

3 The adaptation of the WR and the MN methods to the grid context

The WR and the MN methods have already been used in the literature to solve large differential equations systems. However, in this section we present how these methods can be adapted to the grid context using the asynchronous iteration model.

3.1 The Waveform Relaxation method with Euler

First of all, we describe the Euler method which can be used with the Waveform Relaxation method to solve nonlinear systems. Then, we present the coupling of the Waveform Relaxation method with the Euler method and their implementation according to the asynchronous iteration model.

3.1.1 The Euler method

The Euler method is a first order numerical sequential procedure for solving ordinary differential equations (ODEs) with a given initial value. Consider the ordinary differential equation:

$$\frac{du(t)}{dt} = f(u(t), t) \tag{1}$$

in which $u(t)$ is the mesh points vector and f is a non linear function which is defined for mesh points and discretized time. We can solve this differential equation by approximating the left hand of the equation, using the explicit Euler method which produces the equation:

$$u(t + H) = H \times f(u(t), t) + u(t) \tag{2}$$

where H is the discrete fixed time step and $u(t+H)$ is the vector u at time $t+H$.

This method can be parallelized according to the synchronous model. However, the resulting parallel method would be fined grained because each subproblem must exchange its boundaries' values with its neighbors at every small discrete step. Thus, the parallel method would not be adapted for heterogeneous, distributed and volatile architectures. An alternative is to couple the Waveform Relaxation method with Euler to develop a parallel iterative algorithm that solves ODEs and that is well suited to such architectures.

3.1.2 The Waveform Relaxation method coupled with Euler

In the early 1980's, the Waveform Relaxation method was introduced by E. Lelarasmee as an efficient parallel iterative method for solving large sparsely coupled differential equations systems that are generated by the simulation of integrated circuits. Since then, this method has been extended and applied to various other application areas. With the WR approach, the system of equations is decomposed spatially into sets of equations. Each set is solved iteratively by using values from previous iterations: each computing unit integrates its equations on the whole time interval without communicating with its neighbors. At the end of each iteration, each task exchanges with its neighbors its boundaries' values which are used in the evaluation of the next iteration. This procedure is repeated until the solution vector converges to a stable solution.

The convergence of Waveform Relaxation methods is generally slow and often the parallel execution gain is not sufficient to compensate for the slowness of the convergence. However, there are different methods to accelerate this convergence. Among them, the most used schemes are the overlapping [16] and the windowing [21] concepts.

Overlapping: with this technique every node computes a small number of components already allocated to its neighbors, so that each node solves additionally to its own components a percentage of the components computed by its neighbors. This redundancy helps minimizing the error on the extremity points that depend on the unavailable results of the neighbor's frontier components. Figure 2 shows how the division of the components' vector and the data exchanges between the nodes are changed if the overlapping concept is applied: each node has a small percentage of its neighbors components added to its initial components (which is illustrated by doted lines) and each node transfers the components of index equal to $l + 2 * overlap$ for left boundary and $r - 2 * overlap$ for right boundary, with $overlap =$ number of components added to the initial local components vector, r index of the right boundary and l index of the left boundary. The benefits of this concept have been illustrated in [7].

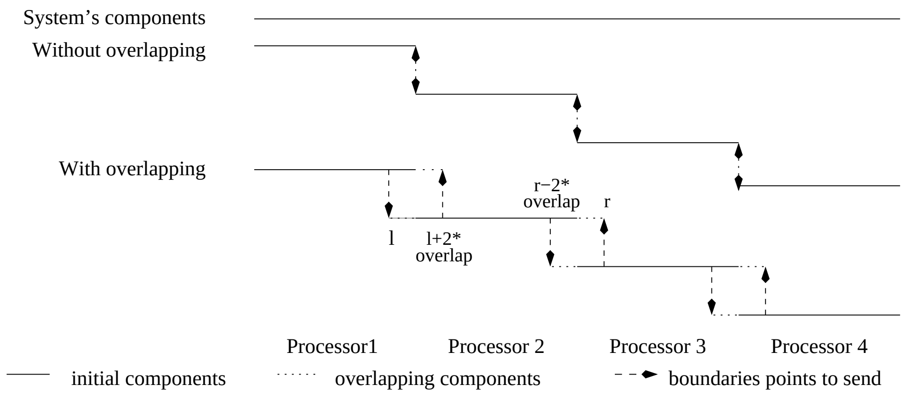

Fig. 2 The decomposition of the system with/without overlapping. The doted lines simulate the overlapping and the arrows simulate the data exchanges between nodes.

Windowing: usually, using the Waveform Relaxation method, we integrate the ODE on the whole time interval at each iteration. This procedure slows down the convergence rate of the method because in contrast with PVODE, every node integrates its equations on a long time interval without communicating with its neighbors, using only the given initial values. A natural solution to this problem is to divide the time interval into windows and iterate on each window until convergence. After each iteration on a window, every node sends its new boundaries' values to its neighbors. Figure 3 displays an ODE system divided between four nodes and the time interval is divided at least into two windows.

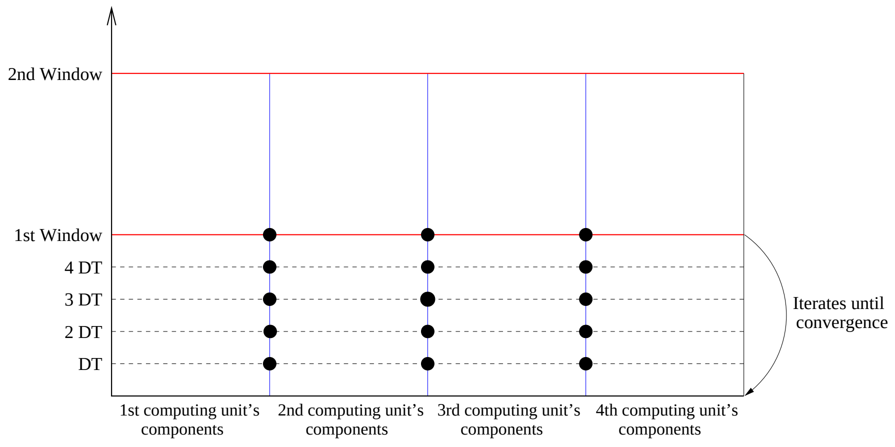
2nd Window
1st Window
4 DT
3 DT
2 DT
DT
Iterates until convergence
1st computing unit's components
2nd computing unit's components
3rd computing unit's components
4th computing unit's components

Fig. 3 The system is equally split into four subsystems and the time interval is divided into several windows where each window contains multiple steps

Algorithm 1 The asynchronous Waveform Relaxation-Euler algorithm

```
1: Split the system's components between the computing nodes
2: Add the overlapped components to the local components
3: uLoc = array containing the local components
4: Set initial values
5: for Each window in the considered time interval do
6:   Copy uLoc into olduLoc
7:   repeat
8:     for Each step of the current window do
9:       Compute the values of the local components using the received boundaries' values
         {in our algorithm using the explicit Euler method}
10:      Store the new boundaries' values in a buffer
11:      Compute the local error
12:    end for
13:    Backup local components every $n$ iterations
14:    Send asynchronously the stored boundaries' values to neighbors
15:    Non blocking reception of boundaries' values from neighbors
16:    Global convergence detection
17:    if Not converged then
18:      Copy olduLoc into uLoc
19:    end if
20:  until Global convergence
21: end for
```

Although both concepts are efficient in accelerating the convergence rate of the WR method, it is very difficult to initially choose the amount of overlapping or the size of a window that gives the optimal results, i.e. a faster convergence. Note that some works have attempted to create an adaptive windowing, but we do not focus on this approach in this paper.

Many sequential resolution methods can be coupled with the WR method to integrate the local subsystem on each time step independently from the other subsystems. In [1] we used the sequential solver found in the CVODE package and we showed that the WR method outperforms PVODE on distributed environments. However, we had some convergence problems when we tried to solve large ODEs using a lot of computing nodes. These problems resulted from the adaptative discretization scheme adopted in CVODE and the heterogeneity of the sub-problems being solved on each node. Indeed, since the boundaries' values change at each window, CVODE is unable to well approximate the solution vector. Therefore, in [6] we used the explicit Euler method to solve this problem. This method proved to be efficient in terms of precision and performance in solving large differential equations when coupled with WR. However, in [6] the WR-Euler method was synchronous, implemented in C and used MPI for exchanging data between the computing nodes. Therefore, it was not well adapted for the grid context (the heterogeneity and volatility of the computing nodes, and the high latency of communications). In this paper, this method is adapted to the grid context by implementing it according to the asynchronous iteration model and over the JACEP2P-V2 platform.

Algorithm 1 illustrates the main steps of the asynchronous Waveform Relaxation method coupled with Euler. Once the system has been initialized, the time interval is decomposed into windows. For each window, an iterative procedure is applied: the window is decomposed into fixed small time steps DT. Each computing unit integrates the system on each DT using the Euler method. Then the new boundaries' values are stored in a buffer. After integrating the system on the whole window, the boundaries' values, stored in the buffer at each DT, are asynchronously exchanged between neighbors. Then, each computing unit integrates the system again on the same time window while using the last received dependency data. This procedure is repeated until the system converges to the solution. Once the convergence is reached, in conformity with the chosen threshold, it begins integrating the system on the next time window. The asynchronous communications, decentralized backups and global convergence detection mechanisms are not detailed in this algorithm because they are implemented in JACEP2P-V2 which is described in section 4.2.1. The work published in [5] proves that the convergence conditions of the asynchronous iteration model are met in the Waveform Relaxation method.

3.2 The Multisplitting-Newton method

In this section, we first of all describe the Newton method which could be used in conjunction with the Multisplitting method to solve nonlinear systems. Then, we present the Multisplitting-Newton method which is a parallel iterative method, compatible with the asynchronous iteration model.

3.2.1 The Newton method

To solve Equation (1), we can also use an implicit time integration method (like implicit backward Euler) that transforms the system into:

$$\frac{u(t+H)-u(t)}{H}=f(u(t+H),t+H), \tag{3}$$

where H is a fixed time-step.

The main differences between the explicit and the implicit methods is that the explicit methods calculate the state of a system at a later time from the state of the system at the current time (like in Equation 2), while implicit methods find a solution by solving an equation involving both the current state of the system and the later one (like in Equation 3). It is clear that implicit methods require more computation than explicit methods (solving the equation), and they can be much harder to implement. However, implicit methods are often used because many problems arising in real life are stiff, for which the use of an explicit method requires impractically small time steps to keep the error in the result bounded. For such problems, to achieve a given accuracy, it takes much less computational time to use an implicit method with larger time steps. That said, whether one should use an explicit or implicit method depends upon the problem to be solved.

The solution to the nonlinear system (3) is computed using the standard Newton method. This approach leads to an iterative scheme, given an initial approximation u^0:

$$J \times d^{k+1} = -F(u^k) \tag{4}$$

where J is the Jacobian matrix of $F(u^k)$ (the Jacobian matrix is the matrix of all first-order partial derivatives of a vector-valued function with $J_{ij} = \frac{\partial F_i(u)}{\partial u_j}$), $d^{k+1} = \delta u(t+H)^{k+1} = u(t+H)^{k+1} - u(t+H)^k$ and $F(u^k) = F(u(t+H)^k, u(t), t) = u(t) + H \times f(u(t+H)^k, t+H) - u(t+H)$. For more information, the reader can refer to [3].

Solving the equation (4) is equivalent to finding the solution of a linear system at each iteration. One can notice that the Jacobian matrix is sparse. In practice, the quasi-Newton method is preferred. It consists in computing the Jacobian matrix only at the first iteration of a given time step in order to reduce the execution time, since this part is often very time consuming. However, the quasi-Newton may require a slightly higher number of iterations than the Newton method to converge.
From a parallel point of view, two approaches are possible. The first one consists in using a parallel sparse linear solver. In this case, a synchronization is required at each Newton iteration and unless using an asynchronous sparse linear solver, synchronizations are required between each iterations of the solving process. The alternative is the Multisplitting-Newton method which is described below.

3.2.2 The Multisplitting-Newton method

The asynchronous Multisplitting-Newton method has some similarities with block decomposition techniques. The principle is to split the initial domain into several sub-domains in order to assign one of them to each computing unit involved in the parallel computation. In our case, the Multisplitting-Newton algorithm allows us to solve equation (4) in parallel.

The Multisplitting-Newton's decomposition is illustrated in figure 4. In fact, the Jacobian matrix is split into blocks and δu and F are decomposed in a compatible manner and are respectively called $dLoc$ and $FLoc$ (because each part is a local one assigned to a computing unit). Dependencies on the left and the right, illustrated by parts with 0 in the figure, can be ignored since those parts of the Jacobian are taken into account in the right hand side. This solution has the advantage of ignoring large parts of the Jacobian matrix which simplifies its implementation.

Algorithm 2 summarizes the main ideas of the Multisplitting-Newton algorithm implemented according to the asynchronous iteration model. After the

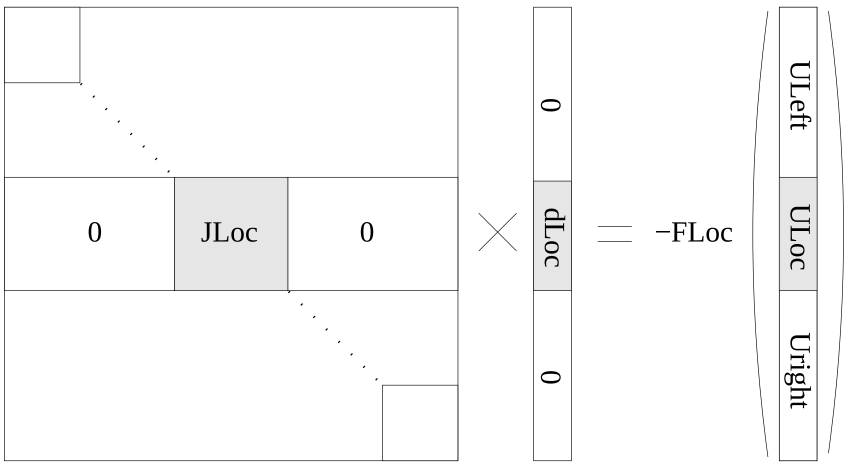

Fig. 4 the decomposition of the Jacobian matrix, vector solution and function in the Multisplitting-Newton method.

Algorithm 2 The asynchronous Multisplitting-Newton algorithm

```
1: oldu = *Array containing vector u at the previous iteration*
2: uLoc = *Array containing local components*
3: -FLoc = *Local part of the function used to approximate the ODE*
4: JLoc = *Local part of the Jacobian matrix*
5: dLoc = *Local part of $\delta u$, the solution of the linear system obtained with Newton*
6: Initialization of variables, especially oldu and uLoc
7: for each step of the considered time interval do
8:   repeat
9:     Computation of the boundaries' conditions if processor is concerned
10:    Computation of the Jacobian matrix JLoc at the first iteration with -FLoc, uLoc
       and oldu
11:    Computation of FLoc with uLoc and oldu
12:    dLoc=LinearSolver(JLoc,-FLoc)
13:    uLoc=uLoc+dLoc
14:    Backup uLoc every $n$ iterations
15:    Send asynchronously the boundaries' values to neighbors
16:    Non blocking reception of the boundaries' values from neighbors
17:    Global convergence detection
18:  until Global convergence
19:  Copy uLoc into oldu
20: end for
```

initialization part, the main loop is executed until the considered simulation time is reached. For each time step, the algorithm iterates on the Newton process to compute the solution of the ordinary differential equation. At each iteration the computing process applies, if necessary, the boundaries' conditions. This algorithm uses the quasi-Newton method, therefore it only computes the Jacobian matrix at the first iteration. Afterward, $FLoc$ is computed using both arrays $uLoc$ and $oldu$. The following step allows the algorithm to solve the local linear system composed of the Jacobian matrix and $FLoc$. The solution is used to set up the value of the vector *uLoc*. As explained previously, the asynchronously exchanged vector between neighbors is not δu but u. That is why, in the algorithm, the vector u at the previous iteration (*oldu*) is not a local vector, since it contains values computed by some neighbors. Once the convergence is reached, in conformity with the chosen threshold, values of *uLoc* are copied into *oldu* at the right location in order to begin the integration on the next time step. The asynchronous communications, decentralized backups and global convergence detection mechanisms are not detailed in this algorithm because they are implemented in JACEP2P-V2 which is described in section 4.2.1.

4 Experiments

To compare the precision of the WR-Euler and Multisplitting-Newton methods to the standard PVODE and to evaluate the performance of both methods in a grid context, we have applied these methods on the the transport problem. This problem models the transport of pollutants in shallow seas. The main objective of the simulation is to exhibit the long term evolution trends of the considered ecosystem after pollution. The results of the simulation are chemical species concentrations in time and space. Transport processes of pollutants, salinity, and so on, combined with their bio-chemical interactions can be mathematically formulated as a system of advection-diffusion-reaction equations. It follows an initial boundary value problem for a nonlinear system of PDEs. A system of 2D advection-diffusion-reaction equations has the following form:

$$\frac{\partial c}{\partial t} + A\left(c,a\right) = D\left(c,d\right) + R\left(c,t\right), \tag{5}$$

where c denotes the vector of unknown species' concentrations of length m, and the two vectors

$$A\left(c,a\right) = \left[\mathbf{J}\left(c\right)\right] \times a^T, \tag{6}$$
$$D\left(c,d\right) = \left[\mathbf{J}\left(c\right)\right] \times d \times \nabla^T, \tag{7}$$

define respectively the advection and diffusion processes ($\mathbf{J}\left(c\right)$ denotes the Jacobian of c with respect to (x,y)). The local fluid velocities u and v of the field $a = (u,v)$ and the diffusion coefficients matrix d are supposed to be known in advance. The chemical species dynamic transport is defined by both advection and diffusion processes, whereas the term R includes inter-species chemical reactions and emissions or absorption from sources. For more information concerning the components and the discretization of the system the reader can refer, for example to [3] in which a third dimension is added to the problem, however, in this paper the 2D variant of the problem is solved.

4.1 Precision

Since the solvers in PVODE are implemented in the C programming language, we have used this language to develop the solution for this problem according to the three resolution methods. The computing processes exchanged data using the message passing interface LAM MPI. The precision tests were conducted on the laboratory's local cluster. It is composed of 20 homogeneous computing units. Each one is equipped with a 3.0Ghz Pentium 4 processor and 1GB RAM. The nodes in this cluster communicate via a 1GB fast local network. To test the pertinence of the solutions given by the Multisplitting-Newton and the Waveform Relaxation-Euler methods and since we do not know the exact solution of the problem, we considered the solution obtained by PVODE as the reference solution for this problem. Then, we have evaluated the relative difference between the solutions of each couple of methods. The relative difference (r) is computed as follows:

$$r = \frac{max(|v_i - v_i^{'}|)}{max_{j=0,...,n}(v_j, v_j^{'})} \quad i = 0, ..., n$$

where $v^{'}$ and v are the two solution vectors computed by the two resolution methods that are being compared (WR-Euler method or the Multisplitting-Newton method or PVODE).

Table 1 presents the relative differences obtained when the solutions, given by the different methods, are compared. We have executed the PVODE method using two thresholds: For the first one (respectively second one) the required precision was equal to 10^{-4} (respectively 10^{-10}). We use the max norm to detect the convergence of the iterative process, for a 10^{-10} precision the max difference between the solution vectors of two successive iterations should be inferior to 10^{-10}. Thus, the method using the lower threshold should give better results. For the WR-Euler method and the Multisplitting-Newton method, the respective required precisions were 10^{-11} and 10^{-12}. We have used different threshold in order to have the same accuracy in the different resolution methods. The first set of relative differences is obtained by comparing the final solution vectors for a simulation over the time interval $[0, 200s]$. These experiments show that the solutions obtained by the WR-Euler method and the Multisplitting-Newton method are very close to the solution computed by the PVODE method. Indeed, the relative difference between the three solutions is less than 0.1%. Moreover, if we compare the solution vectors of these methods with PVODE's solution that is computed with high precision, we notice that the relative difference between the three solutions is less than 0.01%. Therefore, the solutions computed with the two coarse grained methods are more accurate than those computed with PVODE at normal precision. So, we can consider that they are relatively correct.

The second set of relative differences is obtained by comparing the final solution vectors of the different methods for a simulation over the time interval

$[0, 1000s]$. We have performed these experiments to discover how much the accuracy of the results is reduced when simulating over long time intervals. Using the relative differences presented in table 1, we notice that the results are 10 times less accurate than those obtained over a small time interval. This reduction of precision is due, to the small errors (like rounding errors) that accumulate over time. These problems are very common in the numerical computing field, even the PVODE solver suffers from them.

Time interval	Method	WR-Euler	MN	PVODE	PVODE High precision
0 to 200s	WR-Euler	0	4.27×10^{-5}	1.01×10^{-4}	2.32×10^{-5}
	MN	4.27×10^{-5}	0	1.23×10^{-4}	3.89×10^{-5}
	PVODE	1.01×10^{-4}	1.23×10^{-4}	0	1.04×10^{-4}
0 to 1000s	WR-Euler	0	3.31×10^{-4}	2.8×10^{-3}	3.09×10^{-4}
	MN	3.31×10^{-4}	0	2.79×10^{-3}	1.79×10^{-4}
	PVODE	2.8×10^{-3}	2.79×10^{-3}	0	2.78×10^{-3}

Table 1 The relative differences obtained when comparing the solutions computed by the different methods: Multisplitting-Newton (MN), Waveform Relaxation coupled with Euler (WR-Euler) and PVODE.

4.2 Performance

Since we are interested in solving large differential equations in high latency, volatile and heterogeneous environments, we have implemented the Multisplitting-Newton method and the Waveform Relaxation method according to the asynchronous iteration model. This model makes these methods more suitable to such architectures. However, executing this type of methods over such architectures, requires having a distributed platform capable of fulfilling all the functionalities of the asynchronous iteration model and able to resist the potential crash of any computing unit. Therefore, we have used the JACEP2P-V2 framework which is presented in the next paragraph and we have implemented the WR and MN methods in Java.

4.2.1 JACEP2P-V2

JACEP2P-V2 is a parallel, fault tolerant and multi-threaded platform capable of executing parallel, asynchronous and iterative algorithms over volatile and heterogeneous architectures. This middleware is implemented using Sun Microsystems' Java programming language in order to make it platform independent and thus able to run on heterogeneous computing units. Figure 5 illustrates the architecture of JACEP2P-V2 that is composed of three groups of entities:

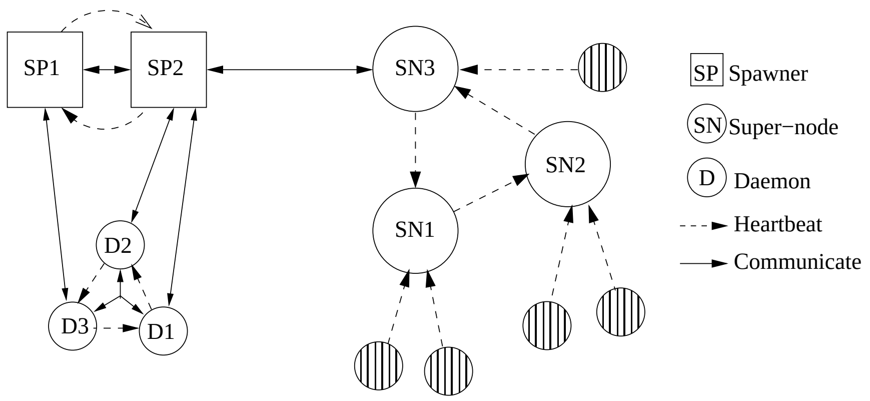

Fig. 5 JACEP2P-V2's architecture and different components.

– **Super-nodes**. They are represented by a big circle in figure 5. They form a ring network and store the identifiers (IP addresses) of all the computing nodes that are connected to the platform (and which are not executing any application). These identifiers are stored in a data structure called "register".
– **Spawners**. When a user wants to execute a parallel application, he launches a spawner (represented by a square in figure 5) with the required parameters like the number of computing nodes N necessary to execute the application. The spawner contacts a super-node to reserve the N computing nodes plus some extra nodes in order to transform them into spawners for fault tolerance reasons. When the spawner receives the register from the super-node, it transforms the extra nodes into spawners and stores the identifiers of the rest of the computing nodes in its own register. Once the extra nodes are transformed into spawners, they form a ring network and they receive the register containing the identifiers of the computing nodes. Then each spawner becomes responsible for a subgroup of computing nodes, starts the tasks on the computing nodes under its command and sends a specific register to them. So each computing node receives a specific register that only contains the identifiers of the computing nodes it interacts with and that depends on the problem being solved.
– **Daemons**. They are the computing nodes (represented in figure 5 by a hashed small circle if they are free and by a white small circle if they are executing an application). Once launched, they connect to a super-node and wait for a task to execute. Once they begin executing an application, they form a ring network and each daemon can only communicate with the daemons that are identified in its register.

To tackle the specificities of the asynchronous iteration model, JACEP2P-V2 has an asynchronous message passing mechanism. At the end of each iteration, the daemon stores the messages that are meant to be sent to neighboring nodes in a 'Send Buffer' and retrieves the new messages, received while executing the last iteration, from the ‘Reception Buffer’. Then, it begins computing the next iteration. In parallel, the ‘Sender thread’ is activated and it asynchronously sends the messages stored in the ‘Send Buffer’ to their respective destinations. When a daemon receives a message, the ‘Reception thread’ stores it in the ‘Reception Buffer’ without affecting the computing thread. This mechanism allows each daemon to send and receive data messages from other daemons without any synchronization with neighbors and without stopping the computing thread.

The fault tolerance mechanism implemented in JACEP2P-V2 is based on the decentralized checkpointing concept for the daemons and the duplication concept for spawners and super-nodes. Each daemon saves its state on its neighboring daemons according to the ‘Round Robin’ strategy. The frequency of these backups is defined by the user. On the other hand, the spawner and the super-node are duplicated on many computing units. All three types of nodes form ring networks where each component regularly heartbeat the next node. If a node crashes, the next node detects the absence of heartbeat messages and signals to the spawner that the previous node is dead. Thus, it triggers the restoring mechanism where the task of the dead node is assigned to a new and work free node. If the dead node is a daemon, the new node replaces the dead one and retrieves the last backup from the backup neighbors. Then it continues the task from that last checkpoint. Otherwise, the new node will be a new duplication like the rest of the spawners or super-nodes.
For more details on the architecture and functionalities of JACEP2P-V2, the reader can refer to [10].

4.2.2 Experimental results

The performance tests have been conducted on Grid’5000 [14], the French national grid. This platform is currently composed of more than 6200 heterogeneous cores that are distributed over 9 sites in France. Most of those sites have a Gigabit Ethernet Network for local machines. Links between the different sites range from 2.5 Gbps up to 10Gbps. Two sets of experiments have been realized. In the first one, we only used one site with homogeneous computing units: we used the cluster Grelon located in Nancy. Each computing unit was equipped with two dualcores Intel Xeon 5110 1.6Ghz and 2GB RAM. We used 100 nodes to solve a problem containing 16,000,000 components on the time interval $[0, 200s]$. The optimal overlap values were used in these experiments. To simulate a volatile environment, we used a shell script that randomly kills each 60 seconds a daemon that is executing a task. Then, a new daemon is launched on that computing unit. This new daemon is connected to the platform and ready to execute a new task. Since the dead daemons are relaunched after a failure, only few (about 5) extra daemons are reserved to ensure the fault tolerance of the application. Some of the parameters for these experiments (number of components, number of machines, number of crashes per minute, etc.) were chosen randomly or by following our intuition because we do not have the resources nor the time to test and verify each and every possible parameter. Moreover, in the asynchronous iteration model, the execution process is not deterministic, therefore it is not easy to justify the selection of some parameters based on the results of the experiments.

		Method	
Decomposition	Status	Multisplitting-Newton	WR-Euler
10×10	without crashes	21m30s	3m24s
	with crashes	21m56s	3m32s
100×1	without crashes	12m5s	3m4s
	with crashes	12m16s	3m13s
1×100	without crashes	17m9s	4m56s
	with crashes	17m30s	5m5s

Table 2 Execution time taken with JACEP2P-V2 to integrate the system on the simulated time interval [0,200s] using 100 computing units located on one site and while killing a random computing node each 60 seconds.

Table 2 presents the execution times taken for solving the problem described above using the Multisplitting-Newton method and the WR-Euler method. Since PVODE is not fault tolerant, it cannot be used over volatile architectures to compare its performance with other methods. However, PVODE has already been compared to the WR method in [6] and experiments showed that the WR method is more adapted to heterogeneous and distributed architectures than PVODE. On the other hand, since PVODE is well optimized for low latency architectures like homogeneous local clusters, it outperforms both asynchronous fault tolerant methods executed over a grid.

Table 2 shows the different execution times taken while varying the problem's decomposition scheme or the volatility of the computing units. The 10×10 decomposition means that the system has been vertically decomposed into 10 subsystems and each subsystem is horizontally decomposed into 10 smaller subsystems. In the same way, the 100×1 (respectively 1×100) decomposition means that the system is horizontally (respectively vertically) decomposed into 100 subsystems. The results show that the WR-Euler method outperforms the Multisplitting-Newton method and solves the problem in a smaller time period. Although, the Multisplitting-Newton method can integrate the system on a larger time steps than the WR-Euler method (we used a time step equal to 10 seconds for the Multisplitting-Newton method and equal to 0.1 second for the the WR-Euler method), the iterative Multisplitting-Newton method requires solving a linear system at each iteration which takes an important amount of time. In our implementation of this method, we used the GMRES method for solving the linear system on each computing unit. GMRES is a widely used linear solver and when experimentally compared with other iterative linear solvers, it proved to be the most efficient. We used a GMRES method implemented in the Matrix Toolkits for Java package (MTJ) [12] which can benefits from multicores machines because it is multi-threaded. Moreover, the Multisplitting-Newton method requires more iterations than the WR-Euler method to converge to the solution. On the other hand, the WR-Euler method computes directly the solution at each iteration using the Euler formula and the implementation of the windowing concept in this method has drastically accelerated its convergence and reduced the execution time it takes. All these reasons made the WR-Euler method a faster resolution method than the Multisplitting-Newton method for solving complex ODEs. However, we can notice that if we decompose the system in just one dimension, (1×100 or 100×1), the execution time taken by the Multisplitting method is considerably reduced. Indeed, when the system is decomposed in one dimension rather than two, each subsystem has two or less boundaries rather than four. Thus it might be less dependent of data received from neighbors which allows the system to converge in fewer iterations. In the same way, we notice that if the system is decomposed only horizontally, it is solved faster than when decomposed vertically. This is related directly to the problem being solved. Indeed, in this problem the values of the components aligned horizontally vary more than those aligned vertically. So if the system is only decomposed vertically, each subsystem must executes a lot of iterations to converge to the solution because the values of the boundaries components are evolving at each time step. Moreover, we also notice that both methods resisted to the computing units' crashes which demonstrates the benefits of the asynchronous iteration model and the efficiency of the JACEP2P-V2 platform for detecting the crashes and replacing the dead daemons. Finally, the effect of these crashes is negligible since the execution times are almost unaffected. Therefore, these methods are well adapted to volatile environments.

In the second set of tests, we used computing units located on two distant sites in order to have a higher latency in the communications between nodes, especially between nodes from distinct sites. Half of the computing units were located in Nancy's site. The architecture of the computing units on this site was described in the previous set of experiments. The second half of the computing units were located in Rennes' site. We used some computing units from the Paraquad cluster. Each one was equipped with two dualcores Intel Xeon 5148 LV 2.33Ghz and 4GB RAM. This heterogeneous architecture represents a real distributed cluster environment. We used 100 computing units distributed over these two sites to solve the same problem described above over the same time interval. Moreover, the same shell script was used to simulate the volatility of the computing units.

Table 3 presents the execution times taken for solving the problem described above using the Multisplitting-Newton and the WR-Euler methods. It also shows the different execution times taken while varying the problem's decomposition scheme or the volatility of the computing units. The results of this set of experiments show that the WR-Euler method outperforms again

		Method	
Decomposition	Status	Multisplitting-Newton	WR-Euler
10×10	without crashes	23m2s	3m24s
	with crashes	23m29s	3m40s
100×1	without crashes	11m33s	3m1s
	with crashes	11m51s	3m32s
1×100	without crashes	16m53s	4m44s
	with crashes	17m25s	4m57s

Table 3 Execution time taken with JACEP2P-V2 to integrate the system on the simulated time interval [0,200s] using 100 computing units located on two distant sites and while killing a random computing node each 60 seconds.

the Multisplitting-Newton. In fact, the results are very similar to those of the first experiment. Since both methods are coarse gained and implemented according to the asynchronous iteration model, they are almost immune to the high latency of the communications and to the heterogeneity of the computing units. Indeed, the computing units do not have to synchronize at each iteration and they do not have to wait for the reception of data messages from their neighbors to compute the next iteration. So, there is no idle times between iterations and fast computing units do not have to wait for slower ones. Therefore, both methods are well adapted for high latency and heterogeneous environments. It is important to point out that the connection between the two sites has a large bandwidth and if it was smaller, we predict that the performance of the WR-Euler method would be drastically reduced because the data messages exchanged between neighbors when using the WR-Euler method are a lot bigger than those in the Multisplitting-Newton method. For example, if a subsystem has 100 boundaries components with its right-hand side neighbor and is using the Multisplitting-Newton method to solve its local task, each message sent by this node to its right-hand side neighbor is about $100 \times 8 = 800 Bytes$ (each component is a double that requires $8 Bytes$ of storage space). Usually this method requires around 60 iterations to integrate on one time step, so the total size of the messages sent to that neighbor is around $60 \times 800 = 48 KBytes$. On the other hand, if using the WR-Euler method and a window is composed of 100 discrete time steps, each message is about $100 \times 800 = 80 kBytes$. Usually this method requires around 15 iterations to integrate on one window, so the total size of the messages sent to the right-hand side neighbor is around $15 \times 80000 = 1.2 MBytes$ which is 25 times bigger than that of the Multisplitting-Newton method.

5 Conclusion and perspectives

In this paper we have presented two coarse grained methods for solving differential equation systems over volatile, heterogeneous and high latency architectures. Those methods were implemented according to the asynchronous iteration model and executed over the fault tolerant and parallel JACEP2P-V2

platform. We have compared the results of both methods with those of the efficient PVODE method. That comparison proved the pertinence of the results obtained by either method. Then, two set of experiments were undertaken to compare the performance of both methods. Those experiments showed that the WR-Euler method solves the problem faster than the Multisplitting-Newton method due to the absence of expensive linear solvers in the first one and to its fast convergence rate.
In our opinion, many applications discretized with a finite difference scheme and that are compatible with the asynchronous iteration model, should benefit from this work, if they are solved in a grid context. However, we point out that real-life simulations require optimized implementations and tuning, with careful choice and tuning of the resolution methods which are beyond the scope of this paper.
In future works, we plan to apply this work to various problems to confirm this statement. Moreover, for now we compared the WR and MN methods over distributed clusters connected with large bandwidth network, it would be interesting to compare the performance of these methods over architectures with smaller bandwidth such as the global computing architecture and to evaluate the effect of smaller bandwidth on the performance of the WR-Euler method.

Acknowledgement. The authors wish to express their gratitude to Grid'5000 community and staff members for their support and useful advices while using the platform. We also sincerely wish to thank the reviewers for their constructive comments.

References

1. J. Bahi, J.-C. Charr, R. Couturier, and D. Laiymani. A parallel algorithm to solve large stiff ode systems on grid systems. *HETEROPAR'07, IEEE Cluster*, pages 534–541, 2007.
2. J. Bahi, S. Contassot-Vivier, and R. Couturier. Parallel iterative algorithms: from sequential to grid computing. *Chapman & Hall/CRC Numerical Analysis & Scient Comp. Series*, 2007.
3. J. Bahi, R. Couturier, K. Mazouzi, and M. Salomon. Synchronous and asynchronous solution of a 3D transport model in a grid computing environment. *Elsevier Applied Mathematical Modelling*, 30(7):616–628, 2006.
4. J. Bahi, J.-C. Miellou, and K. Rhofir. Asynchronous multisplitting methods for nonlinear fixed point problems. *Springer Numerical Algorithms*, 15:315–345, 1997.
5. J. M. Bahi, K. Rhofir, and J.-C. Miellou. Parallel solution of linear DAEs by multisplitting waveform relaxation methods. *Linear Algebra and its Applications*, 332–334(1):181–196, August 2001.
6. Jacques Bahi, Jean-Claude Charr, Raphael Couturier, and David Laiymani. A parallel algorithm to solve large stiff ode systems on grid systems. *The International Journal of High Performance Computing Applications, (IJHPCA)*, 23:140–151, 2009.
7. Kevin Burrage, Carolyn Dyke, and Bert Pohl. On the performance of parallel waveform relaxations for differential systems. *Elsevier Applied Numerical Mathematics: Transactions of IMACS*, 20(1–2):39–55, February 1996.
8. G. Byrne, D. George, and A. C. Hindmars. Pvode, an ode solver for parallel computers. *Sage International Journal of High Performance Computing Applications*, 13(4):354–365, 1999.

9. G. Byrne and A. Hindmarsh. A polyalgorithm for the numerical solution of ordinary differential equations. *ACM Trans. on Math. Soft.*, 1:71–96, 1975.
10. J.-C. Charr, R. Couturier, and D. Laiymani. JACEP2P-V2: A fully decentralized and fault tolerant environment for executing parallel iterative asynchronous applications on volatile distributed architectures. In *Future Generation Computer Systems*, volume In press. Elsevier, 2010.
11. Raphaël Couturier, Christophe Denis, and Fabienne Jézéquel. GREMLINS: a large sparse linear solver for grid environment. *Parallel Computing*, 34(6-8):380–391, 2008.
12. Matrix Toolkits for Java. http://www.ressim.berlios.de/.
13. M. J. Gander. A waveform relaxation algorithm with overlapping splitting for reaction diffusion equations. *Wiley Numerical Linear Algebra with Applications*, 6:125–145, 1998.
14. grid'5000. http://www.grid5000.fr.
15. K. Jackson and R. Sacks-Davis. An alternative implementation of variable step-size multistep formulas for stiff odes. *ACM Trans. on Math. Soft.*, 6:295–318, 1980.
16. R. Jeltsch and B. Pohl. Waveform relaxation with overlapping splittings. *Siam Journal of Science Computing*, 16(1), 1995.
17. E. Lelarasmee, A. Ruehli, and A. Sangiovanni-Vincentelli. The wavefrom relaxation method for time-domain analysis of large scale integrated circuits. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, CAD-1:131–145, 1982.
18. MPI Forum. MPI: A message passing interface. In *Proceedings of Supercomputing '93*, pages 878–883, Portland, OR, November 1993. IEEE CS Press.
19. W. E. Schiesser. *The Numerical Method of Lines.* Academic Press, 1991.
20. J. White, F. Odeh, A. Ruehli, and A. S. Vincentelli. Waveform relaxation: Theory and practice. *Trans. of Soc. for Computer Simulation*, 2:95–133, 1985.
21. H. Zhang. A note on windowing for the waveform relaxation. Technical report, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center Hampton, April 94.