Available online at www.sciencedirect.com
SCIENCE DIRECT®
Applied Mathematical Modelling 30 (2006) 616–628
APPLIED MATHEMATICAL MODELLING
www.elsevier.com/locate/apm

Synchronous and asynchronous solution of a 3D transport model in a grid computing environment

J.M. Bahi *, R. Couturier, K. Mazouzi, M. Salomon

Laboratoire d'Informatique de l'université de Franche-Comté (LIFC), FRE CNRS 2661, IUT de Belfort-Montbéliard, BP 527 90016 Belfort Cedex, France

Received 8 April 2005; received in revised form 1 June 2005; accepted 14 June 2005
Available online 20 December 2005

Abstract

Numerical simulation is a common approach to understand many phenomena, usually yielding a computationally intensive problem. To overcome insufficient computer capacity and computational speed, a grid computing environment is a suitable approach. In this paper we focus on the development of parallel algorithms to solve a 3D transport model in such a context. The solver is based on the multisplitting Newton method that provides a coarse-grained scheme. Algorithms are implemented using JACE, a grid-enabled Java Asynchronous Computing Environment. This programming environment allows users to design synchronous and asynchronous parallel iterative algorithms as well. Experiments are carried out on a heterogeneous grid environment in which the behaviour of both parallel iterative algorithms is analysed. The results allow us to draw some conclusions about the use of the programming library JACE and the design of parallel iterative algorithms in a grid computing environment.

Keywords: 3D shallow water equations; Multisplitting Newton; Grid computing; Java Asynchronous Computing Environment

1. Introduction

The study of numerous phenomenons rely on numerical simulation, allowing a better understanding of the underlying dynamics. The considered phenomenons range from large scale ones, like earth simulation or phytoplankton dynamics [1], to finer ones, e.g., wood drying [2] or molecular simulation. A preliminary step to numerical simulation is the definition of a computational model. Usually it results in a large nonlinear problem. Its solution can be found using direct algorithms or iterative ones, yielding a hard and intensive computationally numerical problem. We focus on the iterative approach.

Distributed computation is an appealing approach to solve such challenging problems. On the one hand a significant speedup can be obtained on the nonlinear system resolution step, on the other hand memory

* Corresponding author.
E-mail address: bahi@iut-bm.univ-fcomte.fr (J.M. Bahi).
URL: http://info.iut-bm.univ-fcomte.fr (J.M. Bahi).

0307-904X/$ - see front matter
doi:10.1016/j.apm.2005.06.017

demanding simulations can be tackled. In fact, the lack of memory in a single processor computer is an important problem encountered in typical real applications. Furthermore, with the enhancement of the networks it is more and more interesting to use local and distant clusters in order to solve larger problems. This approach allows an easy, and at lower cost, increase of the computational power, making it more effective by networking them together in a computational grid.

Parallel numerical algorithms executed in a distributed environment require usually several inter-processor synchronisations. They are needed to update the data and to start the successive computation steps. In a heterogeneous computing environment with long range communications, synchronisations are an important drawback, often degrading the performances. To overcome this problem, a solution is to use an iterative algorithm and to desynchronise communications and iteration steps. This approach leads to an asynchronous iterative algorithm called AIAC for Asynchronous Iterations with Asynchronous Communications. The convergence of this kind of algorithms is ensured, provided that some hypotheses are fulfilled. Although this may not be always possible, the class of converging AIAC is large enough to be a relevant alternative to the synchronous execution mode. An asynchronous algorithm may outperform its synchronous counterpart due to the suppression of idle times. The price is a delayed convergence with the use of less recent data values. Thus more iteration steps are needed to reach the convergence threshold. Naturally, grid computing environments prefer coarse-grained parallel calculations. In fact, in grid environment the distance among processors and the used network—Internet—result in a much higher price of data transfer. Therefore, asynchronous algorithms may be more robust than the corresponding synchronous version.

In this paper, we simulate the 3D transport of pollutants in shallow water. We present the implementation of a parallel iterative solver and explain its behaviour in a grid computing environment. Both synchronous and asynchronous versions are discussed. The two algorithms are coarse-grained by the use of the parallel multisplitting method [3]. The overall organisation of the paper is as follows. Section 2 introduces grid computing characteristics, describes our target computing platform and presents related work. In Section 3 the considered 3D transport model is summarised. The governing equations and boundary conditions are formulated in Section 3.1. Computational considerations, in particular the way the resulting nonlinear system is solved, are presented in the next subsection. The following section is devoted to the description of JACE, the Java-based grid computing environment. Experimental assessment is done in Section 6 and finally, we conclude in Section 7.

2. Target platform and related work

Roughly speaking, a grid consists of heterogeneous resources (storage, networking and computational resources) distributed over a local, metropolitan or wide-area network, coming and going in a fully dynamic manner [4]. To virtualise resources to solve problems, several key questions have to be addressed, namely security, user interface, workload management, scheduling, data and resources management. The use of heterogeneous workstations (different processors, memory size, etc.) which are not all close together may cause hard problems. The large amount of data to exchange and the possible low bandwidth and high latency delivered by the interconnect technology may also induce low performances. The considered computing platform consists of two clusters: a homogeneous cluster and another heterogeneous one. The machines are neither dedicated, nor volatile. More details are given in Section 6.

Enabling an application for grid computing can be simply done by the use of a grid-enabled version of an appropriate programming library. Obviously, the deployment of the library is complicated by the grid inherent heterogeneity. For example, an MPI application [5,6] may need compilations for specific architectures, using different versions of the library, etc. Java provides useful technology for portable, object-oriented application development, solving most of these problems. As a consequence, several Java MPI clones have been developed (e.g., MPJ [7] or JavaMPI [8]). Nevertheless, these programming environments are not suited for implementing asynchronous parallel iterative algorithms, since they are mono-threaded. This lack is solved by the development of an appropriate (multi-threaded) Java-based grid programming environment called JACE (Java Asynchronous Computing Environment) [9]. A Java-based approach allows to profit from various interesting facilities of this language such as portability on most platforms. Thanks to the multi-threading we can separate computation and communication tasks in different threads. Hence we can implement non-blocking communications, an essential and key component for asynchronous computing. From a practical

point of view, communications are done following the message passing paradigm, based on Java Remote Method Invocation (RMI) [10].

Studying asynchronous algorithms has led to several theoretical works [11,12]. Nevertheless, few algorithms are designed to be executed on a computational grid, in particular, in a grid context where several distant clusters are used. Besides, few experimentations have been conducted in order to analyse the behaviour of applications executed in such a distant context. Of course more work has been done regarding the synchronous mode than regarding its asynchronous counterpart. Compared to [13], this paper presents an experimental analysis of the behaviour of synchronous and asynchronous iterative algorithms when the amount of exchanged data is huge and communications are very important. Moreover, new JACE functionalities are described in the present paper. JACE has been enhanced in order to handle new features: automatic convergence detection, a complete mechanism is included to use clusters behind firewalls, the possibility to analyse program execution (traces) to simplify development. The last important contribution of this paper is the introduction and the experimental analysis of the parallel multisplitting algorithm with overlapping techniques. The influence of the overlapping on convergence is also reported. See [3] for a theoretical study of algorithms with overlapped multisplitting and their relation with Schwarz alternating method.

3. Transport model

Modelling the transport of pollutants in shallow seas is of great interest in environmental issues. The main objective of the simulation is to exhibit, after pollution, long term evolution trends of the considered ecosystem. The results of the simulation are chemical species concentrations in time and space. Ideally, to have a realistic simulation the transport model should be associated with a hydrodynamical model simulating the marine flow. In fact, a transport model has the velocity field as input, and this one should result from a hydrodynamical solver. In this work, we suppose that the velocity field is given, assuming it has a constant value, thus we talk about water and no more seas.

3.1. Mathematical description

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, in which nonlinearity comes only from the bio-chemical interspecies reactions.

A system of 3D advection–diffusion-reaction equations has the following form:

$$\frac{\partial c}{\partial t} + A(c,a) = D(c,d) + R(c,t), \tag{1}$$

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

$$A(c,a) = [\mathbf{J}(c)] \times a^{\mathrm{T}}, \tag{2}$$

$$D(c,d) = [\mathbf{J}(c)] \times d \times \nabla^{\mathrm{T}}, \tag{3}$$

define respectively the advection and diffusion processes ($\mathbf{J}(c)$ denotes the Jacobian of c with respect to (x,y,z)). The local fluid velocities u, v and w of the field $a = (u,v,w)$ and the diffusion coefficients matrix d are supposed to be known in advance. A simulation of pollution evolution in shallow seas is obtained if a would be provided by a hydrodynamical model. The chemical species dynamic transport is defined by both advection and diffusion processes, whereas the term R includes interspecies chemical reactions and emissions or absorption from sources.

Following a common approach, $R(c,t)$ can be expressed using production and loss terms, denoted respectively by P and L:

$$R(c,t) = P(c,t) - L(c,t). \tag{4}$$

Both terms can be further refined:

$$P(c,t) = P_I(c,t) + P_S(t),$$
$$L(c,t) = (L_I(c) + L_S(t)) \times c.$$

On the one hand, the terms P and L indexed by I denote the contributions (emission and absorption) from chemical interspecies reaction, on the other hand contributions from sources are indexed by S.

As we consider a test problem in three spatial dimensions, reduced to two chemical species, the system described by Eq. (1) becomes a system of two PDEs:

$$\begin{pmatrix} \frac{\partial c^1}{\partial t} \\ \frac{\partial c^2}{\partial t} \end{pmatrix} + \begin{pmatrix} \nabla c^1 \times a^T \\ \nabla c^2 \times a^T \end{pmatrix} = \begin{pmatrix} \nabla \cdot ((\nabla c^1) \times d) \\ \nabla \cdot ((\nabla c^2) \times d) \end{pmatrix} + \begin{pmatrix} R^1(c,t) \\ R^2(c,t) \end{pmatrix}, \tag{5}$$

where matrix d satisfies

$$d = \begin{pmatrix} \epsilon(x) & 0 & 0 \\ 0 & \epsilon(y) & 0 \\ 0 & 0 & \epsilon(z) \end{pmatrix}, \tag{6}$$

and the vector function $R(c,t)$ is defined according to

- production terms P_I and P_S

$$P_I(c,t) = \begin{pmatrix} q_4(t)c^2 \\ q_1 c^1 c^3 \end{pmatrix}, \qquad P_S(t) = \begin{pmatrix} 2q_3(t)c^3 \\ 0 \end{pmatrix}, \tag{7}$$

- loss terms L_I and L_S

$$L_I(c) = \begin{pmatrix} 0 & q_2 c^1 \\ q_2 c^2 & 0 \end{pmatrix}, \qquad L_S(t) = \begin{pmatrix} q_1 c^3 & 0 \\ 0 & q_4(t) \end{pmatrix}. \tag{8}$$

Clearly, the coupling of the two PDEs is induced by the reaction term R.

As far as transport is concerned, in this work vertical advection (dimension z) is neglected, and the velocity field vector a is supposed to have a constant value. Hence, a is given by

$$a = (u,v,w) = (-V,-V,0), \tag{9}$$

with $V = 10^{-3}$. Regarding diffusion coefficients, horizontally they are positive constants, whereas the vertical ones vary. Thus we have a matrix d of the form

$$d = \begin{pmatrix} K_h & 0 & 0 \\ 0 & K_h & 0 \\ 0 & 0 & K_v(z) \end{pmatrix}, \tag{10}$$

where $K_h = 4.0 \times 10^{-6}$ and $K_v(z) = 10^{-8} \times \exp(\frac{z}{5})$.

For the reaction term (apart from the two unknown concentrations of the contaminants, i.e., c^1 and c^2), the different quantities in Eqs. (5)–(8) are chosen as follows:

- $c^3 = 3.7 \times 10^{16}$, $q_1 = 1.63 \times 10^{-16}$ and $q_2 = 4.66 \times 10^{-16}$,
- $q_3(t)$, $q_4(t)$ are chosen according to ($j = 3,4$),

$$q_j(t) = \exp\left[\frac{-a_j}{\sin(\omega t)}\right] \quad \text{if } \sin(\omega t) > 0,$$
$$q_j(t) = 0 \quad \text{if } \sin(\omega t) \leqslant 0,$$

using the following parameters: $\omega = \pi/43{,}200$, $a_3 = 22.62$ and $a_4 = 7.601$.

4. Multisplitting-Newton method

In this section, we present the Newton method and its multisplitting approach.

4.1. Newton method

After discretisation of Eq. (5) with an Euler scheme in time and centred second-order finite difference schemes in space, we obtain a nonlinear system of ordinary differential equations (ODEs):

$$\frac{\mathrm{d}U(t)}{\mathrm{d}t} = f(U(t), t), \tag{11}$$

in which $U(t)$ is the mesh points vector and f is a nonlinear function.

As we use an implicit time integration, the previous ODEs system becomes:

$$\frac{U(t) - U(t - \delta t)}{\delta t} = f(U(t), t), \tag{12}$$

where δt is a fixed time-step. We can rewrite it as follows:

$$g(U(t), U(t - \delta t), t) = U(t - \delta t) - U(t) + \delta t f(U(t), t), \tag{13}$$

and so the problem consists in solving $g(U(t), U(t - \delta t), t) = 0$. For commodity reason, we use an equivalent form

$$g(U(t), C(t - \delta t)) = 0, \tag{14}$$

where C is a constant vector at time $t - \delta t$.

The solution to the nonlinear system described by Eq. (14) is computed using the standard Newton method. This approach leads to an iterative scheme, given an initial approximation $U^0(t)$:

$$U^{k+1}(t) = U^k(t) - J^{-1}(U^k(t)) g(U^k(t), C(t - \delta t)), \tag{15}$$

where $J(U^k(t)) = J(k, t)$ is the Jacobian matrix of vector $g(U^k(t), C(t - \delta t))$. This equation can also be formulated in the following way:

$$-J(k, t)(U^{k+1} - U^k) = g(U^k(t), C(t - \delta t)) \iff U^{k+1}(t) = F(U^k(t)). \tag{16}$$

Solving the last equation is equivalent to find the solution of a linear system at each iteration. One can notice that the Jacobian matrix is sparse. In practical case, the quasi-Newton method is preferred. It consists in computing the Jacobian matrix only at the first iteration in order to reduce the execution time, since this part is often very time consuming. It results that the number of iterations may be slightly higher than with the Newton method.

In parallel, two approaches are possible. The first one consists in using a sparse linear solver. In this case, a synchronisation is required at each Newton iteration. In addition, unless using an asynchronous sparse linear solver, all other solvers require several synchronisations. The alternative is called multisplitting-Newton, we describe it below.

4.2. The multisplitting-Newton method

This method has some similarities with block decomposition techniques. The principle is to split the initial domain in several sub-domains in order to assign one of them to each processor involved in the parallel computation. In our case, the multisplitting Newton algorithms allow us to solve Eq. (16) in parallel. In this equation, if we replace $-J(k, t)$, $(U^{k+1} - U^k)$ and $g(U^k(t), C(t - \delta t))$ respectively by $-J$, δU and g for commodity reason, then we need to solve

$$-J \times \delta U = g, \tag{17}$$

at each iteration.

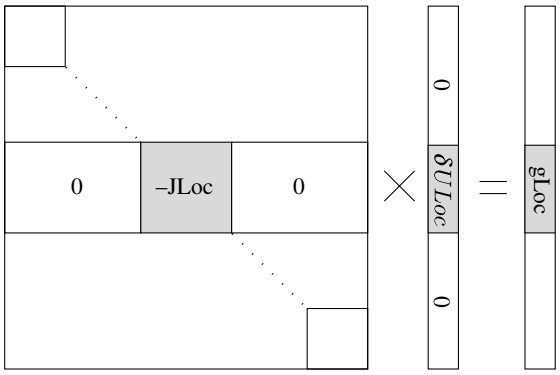

Fig. 1. Jacobian matrix decomposition, vector solution and function.

Multisplitting Newton decomposition is illustrated in Fig. 1. In fact, the Jacobian matrix is split into blocks and δU, g are decomposed in a compatible manner and are respectively called $\delta ULoc$ and $gLoc$ because each part is a local one assigned to a processor. 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. In this case, processors exchange parts of vector U instead of vector δU. This solution has the advantage to ignore large parts of the Jacobian matrix which simplifies the implementation part. Proofs that this scheme converge to the solution can be found in [3].

Algorithm 1 summarises the main ideas of the multisplitting Newton algorithm. After the initialisation part, the main loop is executed until the considered simulation time is reached. For each time step, the algorithm iterates on the Newton process. At each iteration a processor applies, if necessary, the boundary conditions. The next step is the Jacobian matrix computation, one can notice that this calculus is only achieved at the first iteration. Thus, this algorithm uses the quasi-Newton method. Then, *gLoc* is computed using both arrays *UCurrentLoc* and *UOld*. As explained previously, the exchanged vector between processors is not δU but U. That is why, in the algorithm, the vector U at the previous iteration (*UOld*) is not a local vector, since it contains values computed by some neighbour processors. The following step allows the algorithm to solve the local linear system composed of the Jacobian matrix and *gLoc*. The solution is used to set up the value of the vector *UCurrentLoc*.

The next statement concerns the communication part. According to the synchronous or the asynchronous version, two main features differ: the communication management and the convergence detection. In order to clearly understand the differences, we briefly recall the definition we introduced in [14]. A fully synchronous algorithm is called SISC (Synchronous Iterations with Synchronous Communications). If communications are asynchronous, then the algorithm is called SIAC (Synchronous Iterations with Asynchronous Communications). With the asynchronous version, communications are necessarily asynchronous. Thus this algorithm is called AIAC (Asynchronous Iterations with Asynchronous Communications). To present a unified view of the different execution modes, we dropped the data receptions in Algorithm 1. In synchronous modes (SISC and SIAC), blocking receptions are always performed with blocking communications. Conversely, AIAC implementation requires receptions that do not block the computation thread (user task). Using JACE, a communication library presented in the next section, we take advantage of its non-blocking receptions. So, neglecting this difference, each version of the algorithm uses exactly the same communication scheme [13]. Moreover, a processor does not send the same part of its vector *UCurrentLoc* to each of its neighbours. Consequently, the function *PartOf* determines which part of the data must be sent according to the considered processor. The received data are stored in *UOld*. The convergence detection phase can be achieved using a centralised algorithm as described in [14] or a decentralised one as presented in [15].

Once the convergence is reached, in conformity with the chosen threshold, values of *UCurrentLoc* are copied into *UOld* at the right location. For each time step the quasi-Newton method is applied, resulting in the computation of the solution of the ordinary differential equation. For more references on multisplitting techniques see, e.g., [3,16,17] and the references therein.

Algorithm 1 (*Multisplitting-Newton algorithm*)

NbProcs = *Number of processors*
MyRank = *Processor rank*
UOld = *Array containing vector U at the previous iteration*
UCurrentLoc = *Array containing local subvector U for the current iteration*
gLoc = *Local part of the function used to approximate the ODE*
−JLoc = *Local part of the Jacobian matrix*
$\delta ULoc$ = *Local part of* δU, *the solution of the linear system obtained with Newton*
DependsOnMe[NbProcs] = *Array of depending processors*
Initialisation of variables, especially UOld and UCurrentLoc
for each step of the considered time interval **do**
  repeat
    Computation of boundary conditions if processor is concerned
    Computation of the Jacobian matrix −JLoc at the first iteration with gLoc, UCurrentLoc and UOld
    Computation of gLoc with UCurrentLoc and UOld
    $\delta ULoc$ = LinearSolver(−JLoc,gLoc)
    UCurrentLoc = YCurrentLoc + $\delta ULoc$
    for i = 0 to NbProcs-1 **do**
      if i! = MyRank and DependsOnMe[i] **then**
        Send(PartOf(UCurrentLoc,i),i)
      end if
    end for
    Convergence detection
  until Convergence of the Newton process
  Copy UCurrentLoc into UOld
endfor

An interesting point of this algorithm is that it allows to overlap some problem unknowns. In this case, two processors compute the same unknowns. Consequently the time of an iteration is longer but the number of iterations to reach the convergence could be reduced. This point is highlighted in Section 6. Fig. 2 illustrates an overlapping with a two dimensional discretisation for the sake of simplicity. On the right of this figure, the processor P sends its dependencies to the processor P' and reciprocally. These processors share no components. On the left part of the figure, they share some components. These last ones are represented by hashed rectangles. Processor P uses the dependencies of P' and P' uses those of P.

In the next section we present the main features of the programming environment used to implement the previous algorithm.

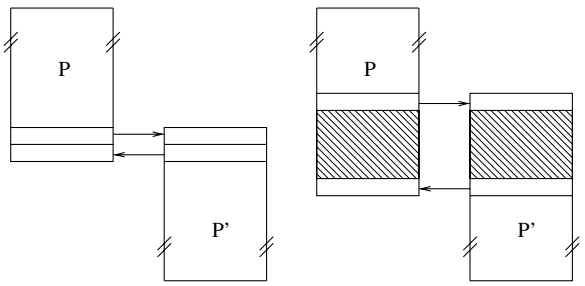

Fig. 2. Non-overlapping and overlapping cases.

5. Java Asynchronous Computing Environment

Basically, the idea that leads to the development of a Java Asynchronous Computing Environment, called JACE, was to design a grid-enabled programming and execution environment allowing a simple and efficient implementation of asynchronous algorithms. Obviously, this environment can also run synchronous algorithms. Roughly speaking, JACE builds a virtual parallel machine by connecting a set of heterogeneous and distant computers. It schedules tasks, executes them and returns results to the user. Since the description of Jace in [9], the following new features have been included:

- *Easy access to machines*
  The virtual parallel machine created by JACE is composed by a set of multiple machines scattered on several distant sites. A machine can be secured by a firewall, JACE ensures its localisation and a transparent access.
- *Task migration*
  A task can migrate from a machine to another one without interrupting computation. Users can easily implement their load-balancing algorithms.
- *Distributed convergence detection*
  JACE is designed to manage the convergence control. More precisely, distributed algorithms controlling the convergence are implemented, the user controls convergence by calling JACE functions.

As in classical message passing library like PVM [18], JACE is composed of three components: daemon, task and spawner.

5.1. Daemon

This component is the core of the library, it provides communications, task localisation and execution. It runs on all hosts taking part in the computation. Fig. 3 shows the internal architecture of the daemon. Threads are represented by wavelets and messages queues by open boxes. To communicate with other daemons, each daemon implements a lightweight RMI server. This server proposes methods to perform communications and configurations between daemons. The method can be remotely invoked by other daemons and are completely transparent to the user.

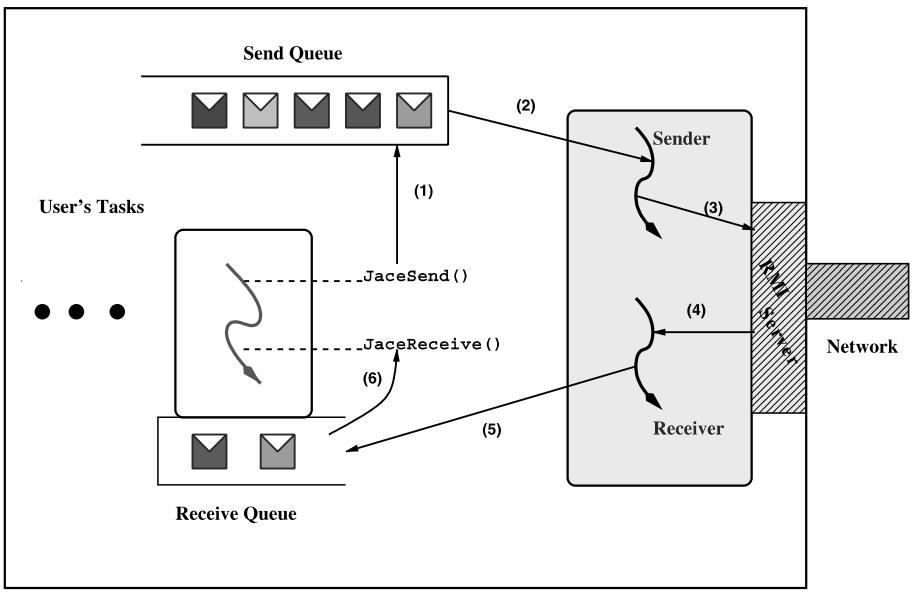

Fig. 3. JACE daemon architecture.

Communications between tasks are ensured by the daemon. JACE uses a combination of message passing and RMI methods to implement communications. The message flow is illustrated in Fig. 3. As it can be seen, each sent or received message passes through a number of stages on the daemon, namely (1)–(3) in the former case and (4)–(6) in the latter one, before it finally reaches the destination task. The *Send Queue* stores messages from local tasks to remote ones. To send a message a user task uses a simple queue access method called `JaceSend()` (step (1)). The *sender* thread probes in the send queue (step (2)) and tries to send existing messages to the remote tasks using RMI calls (step (3)) to the daemon where the remote tasks are located. Each local task has its own *Receive Queue*. The *receiver* thread has in charge to store the incoming messages (step (4)) in the appropriate queue (step (5)). Finally, every user task accesses directly to its receive messages through the primitive `JaceReceive()` offered by the programming environment (step (6)).

In the asynchronous execution mode, when a user task or receiver thread tries to add a message, respectively in the send and receive queue, it always checks if there already exists a message with the same tag and source or destination. If such a message occurs, this one is replaced by its newest version. We can note that a tag allows a programmer to differentiate two messages from the same source or destination, containing different types of information.

The daemon is started on all desired machines. An XML file containing information on the different machines is given as parameter to start the daemons. As soon as all daemons are started JACE is ready to execute tasks.

5.2. Task

A JACE application is a set of cooperating sequential tasks. As shown in Fig. 3, all tasks run as Java threads in the daemon process. Thus, multiple tasks are allowed to be executed on the same host and can share the system resources. When a task is spawned, an identification number (task ID) is assigned to it, this number is an integer whose value ranges from 0 to $p - 1$, with p being the global number of task in the JACE application. Thanks to the JACE daemon, task location is hidden to the users.

To write an application using the functionalities offered by the JACE programming environment, the user simply needs to extend the JACE **Task** base class and define a *run*( ) method containing its program code. In fact, the Task class may be considered as a programming interface.

5.3. Spawner

When all daemons are started, computations begin by launching the user task in the desired nodes. This functionality is supplied by the JACE *spawner* program. It supports only static spawning. It means that tasks cannot be spawned and deleted during program execution. The number of tasks to be created must be specified. Following the SPMD[1] parallel programming paradigm, all tasks execute the same code, since they are instances of the same Java class. The task mapping is done by JACE, no user interaction is requested. The program code can be located on a valid uniform resource location, a local host or on the Internet. After termination of the user program, the daemon returns the results and cleans the current session by calling a Java garbage collection.[2]

6. Experiments

6.1. Hardware environment

Our experimental context consists of two distinct clusters located within the same University. The first cluster is homogeneous, it means that all the nodes have the same architecture and operating system. In fact, it is composed of 16 Pentium IV processors running at 3 GHz, each node having 1 Gb of memory, and Linux as

[1] Single Program Multiple Data.

[2] Garbage collection is the process of figuring out how to detect when an object is no longer referenced, and how to make that unused memory available for future allocations.

operating system. These machines are connected via high-speed switches, hence it ensures the ability to reach high-performance bandwidths. The second cluster is also composed of 16 machines, but in this case, they are heterogeneous: 9 Pentium IV 2.4 GHz and 7 Pentium IV 3 GHz. The network interconnects for these nodes are standard 100 Mb Ethernet.

To estimate the throughput that can be achieved on a link we use iperf tools. The resulting bandwidth measures are as follows:

- the achievable bandwidth between a pair of machines located in the homogeneous cluster is on average equal to 583 Mb/s;
- a link between machines of the heterogeneous cluster has an average bandwidth of 94 Mb/s;
- finally, the bandwidth of inter-cluster connections is about 92 Mb/s.

These conditions are clearly not very representative of grid computing (distant cluster context). To have an experimental framework in accordance with grid computing environment, we downgrade the network performances of the homogeneous cluster. This can be easily done owing to the recent Linux kernel version which support QoS tools. It can be noticed that this feature is not supported by the machines in the heterogeneous cluster.

6.2. Simulation parameters

The initial concentrations of the two chemical species are prescribed:

$$c^1(x,y,z,t=0) = 10^6 \times \alpha(x) \times \beta(y) \times \gamma(z),$$
$$c^2(x,y,z,t=0) = 10^{12} \times \alpha(x) \times \beta(y) \times \gamma(z),$$

where α, β and γ are functions of the form

$$\alpha(x) = 1 - (0.1 \times x - 1)^2 + (0.1 \times x - 1)^4/2,$$
$$\beta(y) = 1 - (0.1 \times y - 1)^2 + (0.1 \times y - 1)^4/2,$$
$$\gamma(z) = 1 - (0.1 \times z - 1)^2 + (0.1 \times z - 1)^4/2.$$

The simulation time interval is equal to 1200 s. To solve the linear system obtained by the multisplitting-Newton algorithm we use MTJ,[3] a Java package designed to implement a sparse linear solver using several iterative methods. For performances reasons we have decided to use GMRES. Each result in the following represents the average of 10 executions and is expressed in seconds, considering the execution time of the slowest processor.

6.3. Results and discussion

Let us first note that, we have only implemented SISC and AIAC versions. SIAC algorithms are interesting when data can be sent on the fly. In our case, due to the solver this is not possible: data are available only at the end of a Newton iteration. In all the experimentations we have conducted we have chosen equal grid size in all the directions.

In order to appreciate the scalability of synchronous and asynchronous versions, we report the execution times of both versions on the first cluster composed of 16 homogeneous machines and 1 Gb/s network. In this case, experiments were performed without overlapping. Fig. 4 shows that the synchronous version is faster than the asynchronous one. The context is clearly favourable to synchronous execution since machines are homogeneous with a high bandwidth network. In addition, with few processors, the synchronisations do not penalise the synchronous version whereas the asynchronous one is sensible to communication delays. With more processors, the difference between both versions decreases.

[3] http://mtj.dev.java.net/smt.html

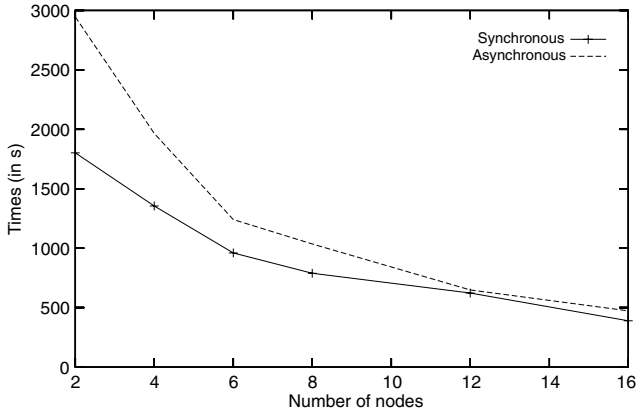

Fig. 4. Problem scalability in a local homogeneous cluster with a $60 \times 60 \times 60$ mesh.

In the following we report experimentations that highlight the behaviour in a distant grid computing context. In this case, we have used 32 machines and we have decomposed the domain problem in 4 in both x-dimension and y-dimension, whereas z-dimension is divided in 2.

Figs. 5 and 6 respectively illustrate the multisplitting-Newton algorithm execution times of synchronous and asynchronous version in function of the problem size. In fact, these figures only report experiments with

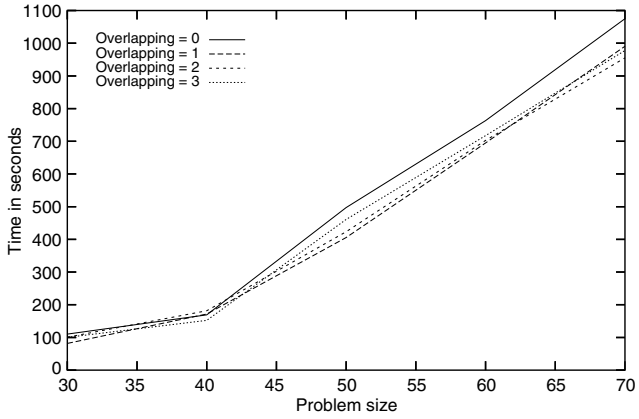

Fig. 5. Synchronous execution—overlapping influence—32 heterogeneous processors.

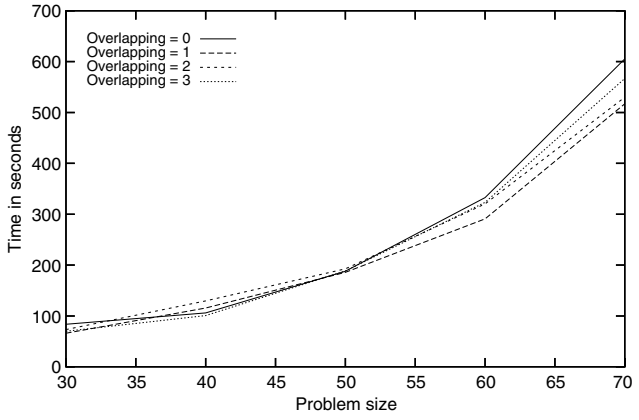

Fig. 6. Asynchronous execution—overlapping influence—32 heterogeneous processors.

Table 1
Iterations number comparison (with and without overlapping): synchronous vs. asynchronous version

Overlapping parameter	0					3				
Size	30	40	50	60	70	30	40	50	60	70
Synchronous	15	18	20	22	25	10	12	13	15	18
Asynchronous										
Min.	86	83	81	76	73	48	42	36	34	32
Max.	166	150	134	118	110	90	66	52	50	46

equal grid size in all the directions. For example, a problem whose size is equal to 60 means that there are 60 data points in each dimension, resulting in $60 \times 60 \times 60$ mesh points. Network performances of the homogeneous clusters have been limited to 1 Mb/s in order to simulate far distant cluster connection.

Several comments can be made upon these figures. First, we can observe that small problems (with size ranging from 30 to 40) result in similar execution times for both synchronous and asynchronous execution modes. This can be easily explained since, in the synchronous context, iterations are performed slowly, due to important idle times compared to the computation times whereas, in the opposite, the asynchronous version performs a larger number of iterations. Many of them are very interesting from a computational point of view, as no data updates occur during the previous iteration.

Larger problems (size above 40) reveal the relevance of asynchronism: the synchronous version is clearly slower than its asynchronous counterpart. The experiments point out proportionately faster executions of the asynchronous version than of the synchronous one, besides the synchronous/asynchronous computation times ratio is near to 2. In fact, in the synchronous framework faster processors wait for slower ones at each iteration and communications slow down iterations because they are blocking. Conversely, in the asynchronous version, processors perform their computation freely without waiting for any other neighbours and communications can be overlapped by computations since they are performed by different threads.

In Figs. 5 and 6 we also report the execution times of both algorithms with different overlapping parameters. It appears that the greater the overlapping parameter is, the smaller the number of iterations to reach the convergence is, and also the more time consuming an iteration is. In these figures, when the overlapping parameter is set to one, it signifies that the size in one dimension has been increased at most twice, one in each direction if the considered processor has two neighbours. Let us consider a problem with a $60 \times 60 \times 60$ domain size; with the chosen decomposition, a processor is in charge of a sub-domain of size $15 \times 15 \times 30$ mesh points. Furthermore, z-dimension has been decomposed only into 2, hence no processor has two neighbours in this dimension. So, without overlapping the size of the linear subsystem mapped to each processor is $15 \times 15 \times 30 \times 2 = 13{,}500$, because there are two chemical species in the problem. If the overlapping parameter is set to 3, the size of the linear subsystem becomes $(15 + 6) \times (15 + 6) \times (30 + 3) \times 2 = 29{,}106$, i.e., more than twice larger. Considering both versions, according to the problem size, the best overlapping size varies from 1 to 3.

To highlight the difference between the synchronous and asynchronous execution modes, we summarise in Table 1 the number of iterations needed to reach convergence. Little variation with the time steps can be observed in the synchronous context. Thus, we have chosen the most frequent one in this case. For the asynchronous mode, the number of iterations varies from an execution to another and also in function of the communication delays and the computation power. So, we put the minimum and maximum number of iterations observed during the experimentations. It can also be seen that the overlapping reduces the number of iterations in both synchronous and asynchronous contexts. It should be noticed that when the problem size increases, the ratio between computation and communication increases too. Hence, the number of iterations in asynchronous case comes nearer to the synchronous one.

7. Conclusion

In the present paper parallel synchronous and asynchronous iterative algorithms have been discussed in order to solve a 3D transport model in a grid environment. The principles of parallel iterative algorithms

are described in details. Both algorithms are based on parallel multisplitting iterative Newton method. This method allows us to design coarse-grained algorithms that provide an efficient scalability. Thanks to the multisplitting method and especially to the overlapping technique, the number of iterations may be reduced and therefore, the execution times may decrease. The asynchronous version is derived from the synchronous one by changing some features (communications, convergence detection, etc.). Nevertheless the global structure of the algorithm is maintained.

To build efficient algorithms dedicated to grid environments, different important issues must be fulfilled. First of all, frequent synchronisations must be avoided, i.e., coarse grained algorithms are preferred. In the same way, decentralised processes are better suited for grid context where machines can be scattered all around the world. Asynchronism offers a relevant way to overlap communications by computations and, by the way, limit the influence of bad network conditions. A grid-enabled programming environment, called JACE, is presented. It is designed to provide a complete framework to develop synchronous and asynchronous algorithms. It supports several interesting features to build grid applications, notably the feature pointed out in this work. We strongly encourage the scientific community to use it.

The experiments presented in this work led us to deduce that the asynchronous version is more efficient than the synchronous one in distant grid computing environment, a context where network parameters are not ideal and architecture is heterogeneous.

References

[1] N.N. Pham Thi, J. Huisman, B. Sommeijer, Simulation of 3d phytoplankton dynamics: competition in light-limited environments, J. Comput. Appl. Math. 174 (2005) 57–77.
[2] S.L. Truscott, I.W. Turner, A heterogeneous three-dimensional computational model for wood drying, Appl. Math. Modell. 29 (4) (2005) 381–410.
[3] J.M. Bahi, J.-C. Miellou, K. Rhofir, Asynchronous multisplitting methods for nonlinear fixed point problems, Numer. Algorithms 15 (3–4) (1997) 315–345.
[4] I. Foster, C. Kesselman, The Grid: Blueprint for a New Computing Infrastructure, second ed., Morgan Kaufmann, 2004.
[5] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message Passing Interface, MIT Press, 1994.
[6] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, M. Snir, MPI: The Complete Reference (vol. 2: The MPI-2 Extensions), MIT Press, 1998.
[7] M. Baker, B. Carpenter, New MPJ: a proposed Java message-passing API and environment for high performance computing, in: International Workshop on Java for Parallel and Distributed Computing, May 2000.
[8] S. Mintechev, JavaMPI Java interface for MPI implementation. Technical report, University of Westminster, 1997.
[9] J.M. Bahi, S. Domas, K. Mazouzi, Jace: a java environment for distributed asynchronous iterative computations, in: 12-th Euromicro Conference on Parallel, Distributed and Network based Processing, PDP'04, IEEE Computer Society Press, 2004, pp. 350–357.
[10] T.B. Downing, Java RMI: Remote Method Invocation, IDG Books, 1998.
[11] M. El Tarazi, Some convergence results for asynchronous algorithms, Numer. Math. 39 (1982) 325–340.
[12] D.P. Bertsekas, J.N. Tsitsiklis, Parallel and Distributed Computation: Numerical Methods, Prentice Hall, Englewood Cliffs, NJ, 1989.
[13] J.M. Bahi, R. Couturier, P. Vuillemin, Asynchronous iterative algorithms for computational science on the grid: three case studies, in: VECPAR' 2004, Lecture Notes in Computer Science, vol. 3402, Springer Verlag, 2005, pp. 302–314.
[14] J.M. Bahi, S. Contassot-Vivier, R. Couturier, Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms, IEEE Trans. Parallel Distrib. Syst. 16 (4) (2005) 289–299.
[15] J.M. Bahi, S. Contassot-Vivier, R. Couturier, F. Vernier, A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms, IEEE Trans. Parallel Distrib. Syst. 16 (1) (2005) 4–13.
[16] J. Arnal, V. Migallon, J. Penadés, Nonstationary parallel Newton iterative methods for nonlinear problems, in: VECPAR'2000, Lecture Notes in Computer Science, vol. 1981, 2001, pp. 380–394.
[17] A. Frommer, D. Szyld, On asynchronous iterations, J. Comput. Appl. Math. 23 (2000) 201–216.
[18] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing, The MIT Press, 1994.