CRAC: a Grid Environment to Solve Scientific Applications with Asynchronous Iterative Algorithms

Raphaël Couturier, Stéphane Domas

Laboratoire d'Informatique de l'Université de Franche-Comté (LIFC)
IUT de Belfort-Montbéliard
Rue Engel Gros BP 527 90016 Belfort CEDEX France
raphael.couturier,stephane.domas@iut-bm.univ-fcomte.fr

Abstract

This paper presents CRAC, an environment dedicated to design efficient asynchronous iterative algorithms for a grid architecture. Those algorithms are particularly suited for grid architecture since they naturally allow to overlap communications by computations. Each processor computes its iterations freely without any synchronization with its neighbors. All the characteristics of CRAC are described. A real application using four distant clusters, with a total of 120 *processors, shows the interest of this environment and of asynchronous algorithms.*

1. Introduction

MPI is often used to develop scientific applications, running on a local virtual parallel machine. But on a larger scale with distant machines or clusters, MPI may be a bad choice. Indeed, grid computing implies heterogeneity at hardware and software levels. For example, using concurrently two distant clusters implies that the same version of each library, including MPI, is installed. This constraint gets more and more demanding as a large number of computing resources is used, scattered on a large geographical scale. Environments like Globus [12] or ProActive [10] provide a framework in which the application deployment is really easy, as long as these environments are installed on the target machines, which may be difficult. ProActive is written in Java and is based on active objects. Thus, it does not use the classical message passing paradigm. However, it is possible to use message passing and to benefit from the total portability of Java with environments like MPJ [8] or JMPI [14]. Obviously, the performance of Java implementations is lower than codes compiled from C. A factor of ten on scientific applications is quite common. This balance between portability and efficiency seems to be the main point of interest. This can be summarized by the eternal question: how to obtain the best performance at the lowest coding and installation efforts ? As usual, the answer greatly depends on the context. In homogeneous clusters, classical computation libraries often obtain the best possible performance but they are inefficient in a really distributed context. The word "really" is important since this efficiency problem is often skewed by using distant clusters linked by high speed dedicated networks. Obtaining good performance using a collection of distant workstations and clusters, linked by Internet, is far more challenging. To address this problem, we think that the common way of programming scientific applications is a dead-end: we have to adapt the algorithms to the grid context and not the opposite.

Since synchronizing communications (i.e. all-to-all, gather-scatter, ...) are bottlenecks on a widely distributed architecture, they must be avoided. This is possible using **AIACs** (*Asynchronous Iterations-Asynchronous Communications*) algorithms. These algorithms are tolerant to communication deadlines and even message loss. However, the communication semantic is quite special and it cannot be efficiently implemented in MPI.

This is why we have first developed a programming/execution environment called **JACE** (*Java Asynchronous Computing Environment*), dedicated to AIACs algorithms. It is written in Java to achieve a total portability and to insure a simple application deployment. Several experiments based on real scientific problems have been conducted, on different types of architectures. Results presented in [5] [4] [6] clearly show the advantages of AIACs algorithms out of their synchronous version, even sometimes in an homogeneous context. However, as shown in

1-4244-0910-1/07/$20.00 ©2007 IEEE.

[6], an MPI/C code is still faster than a JACE/java bytecode. This is the reason for which we have developed a new environment, inspired by JACE, but written in C++. It is called **CRAC** (*Communication Routines for Asynchronous Computations*). As JACE, it provides not only a set of communication primitives but also a programming/execution framework.

In section 2, we give a comparison between a synchronous and an asynchronous execution. In section 3, we give an overview of CRAC, the motivations to create it and its internals. Finally, section 4 presents a scientific application implemented with CRAC, and the results of our experiments on a grid architecture.

2. Asynchronism and synchronism

Readers can refer to [2] for a study of asynchronous algorithms on the grid. Figures 1 and 2 present a simple example with three tasks, equally loaded in terms of data to compute. Each iteration (hatched boxes) takes the same time for a given task and the difference between tasks is due to the machines power. At the end of each iteration, each task must send dependencies data (plain/dashed arrows), only to its neighbors in this example. We suppose that for a given peer $\{i, j\}$ of tasks, the dependencies data sent by i always imply the update of the same data set of j. The arrows represent the physical communication time over the network and not the time between the explicit emission and reception of the message by the tasks, which could be longer but not shorter.

In a synchronous execution scheme (Figure 1), the reception of the dependencies is always blocking. After that, the global convergence state must be computed and broadcast to all, generally with a gather-scatter (dotted arrows). This example supposes that task B collects the local states and broadcast the global state.

As we can see, this scheme implies lots of idle times (white boxes) because the fastest machines always wait for the slowest ones to receive all their dependencies. The convergence detection also generates idle times because of the gather-scatter communication. Obviously, the more heterogeneous the machines and the network are, the more idle times there are.

In an asynchronous execution scheme (Figure 2), idle times are completely removed by using non-blocking primitives for all communications (dependencies and convergence detection). Each task begins a new iteration with "old data" if no dependences have been received. For example, A and C immediately begin their second iteration despite the fact that they have received nothing from B. Obviously, this may lead these tasks to momentarily diverge and even the whole computation to loop endlessly, with no convergence. Thus, asynchronism cannot be applied on every iterative algorithm. However, the convergence may be ensured by checking some mathematical properties on the iteration functions. Fortunately these properties are satisfied for a large class of scientific problems such as those described by linear systems involving M-matrices or those modeled by partial differential equations and discretized by the finite difference method (e.g. [9]).

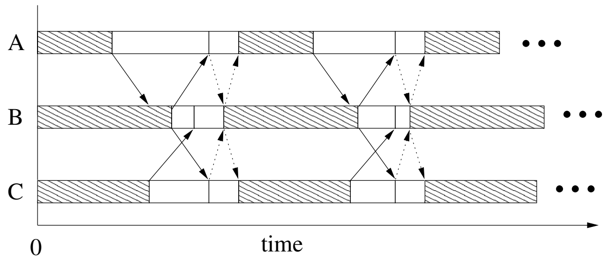

Figure 1. Synchronous iterations

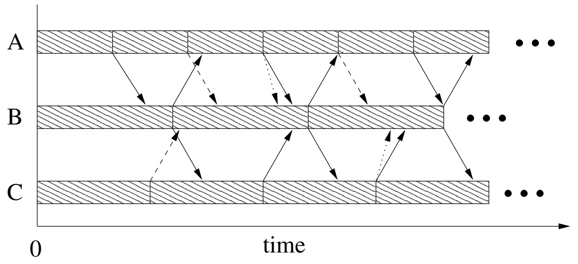

Figure 2. Asynchronous iterations

Since an asynchronous execution very frequently increases the number of iterations to reach convergence compared to a synchronous execution, a convergence acceleration technique is particularly useful. There exists a technique applicable to any asynchronous execution, which can be summarized by: "always update the data with the newest dependencies received". On Figure 2, B receives two dependencies messages from A and C during its second iteration. Convergence may be accelerated if the second message (plain arrow) is consumed by B to update its data, and the first one (dashed arrow) simply discarded. Practically, during a single iteration, if a task receives several dependency messages concerning the same data set to update, the already received message is replaced in memory by the newest. Thus, there is a single occurrence of the same dependency message in memory and it is always the last one received.

After having briefly presented the principles of AIAC algorithm, we are now going to describe CRAC.

3. CRAC

3.1. Genesis

At first glance, it seemed possible to implement asynchronous algorithms with MPI since it provides non-blocking communications. Considering the last section example, it is quite simple for task A to post a reception from B at the beginning of an iteration, and to check if the message arrived at the end. No other receptions are posted until the message is really received. This mechanism is not sufficient for B which may receive several messages from A during the same iteration. More generally, each task must test the existence of an unknown number of messages from the same source, which is quite complicated in MPI. Furthermore, getting the newest of these messages is only possible by explicitly receiving all the messages. This is time consuming and particularly useless. Finally, all these complicated mechanisms must be implemented directly in the application and are hardly reusable for another application.

For all these reasons, MPI was quickly abandoned and we decided to develop a new environment providing the framework to implement iterative algorithms and to execute them synchronously or asynchronously on a widely distributed architecture.

JACE was born from this decision and has largely proved its interest through a lot of experiments, particularly in terms of coding facilities and performances. In our last experiments, we compared the JACE/Java and the MPI/C (MPICH/MADELEINE [13] with PM2) implementation of the same application. The amount of Java code is one third smaller than the C code since there is no need to implement the asynchronism mechanisms. Furthermore, we have obtained an average ratio of 6 between the Java and C execution time, even though it is common to have 10 for scientific applications. However, this ratio is often considered too big in respect of the higher coding effort needed in C. This is why we chose to develop a C++ environment, based on the same principles as JACE, and adding some optimized primitives and mechanisms that take into account the architecture of the grid.

3.2. Architecture

CRAC is based on the classical MPI triplet: daemon, application, spawner. The daemon is launched on each machine constituting the Virtual Distributed Machine (*VDM*). The user develops its application and launches it with the spawner on the desired machines. However, the similarity with MPI nearly stops here. Even if the CRAC programming interface uses the message passing paradigm, the semantic of communications is completely different and several primitives do not exist in MPI. Furthermore, the internals of CRAC are based on multithreading and even the application is a thread. Finally, the virtual distributed machine relies on a hierarchical view of the network in order to reach machines with private IPs and to limit the bandwidth use on slow links.

The following items present the different components of CRAC, from the VDM to the programming interface.

- Virtual Distributed Machine (VDM): the efficiency problem of distributed executions partly comes from "low" bandwidth on links between distant geographical sites. In this case, a primitive like a gather-scatter that does not take care of the network architecture may be totally inefficient if all messages must take the slowest link of the architecture. Assuming that machines can be gathered in "sites", which have good bandwidth, and that sites are linked by "low" bandwidth, all global communications may be optimized to take account of this organization. This is the case when the architecture is composed of clusters linked by the Internet. Unfortunately, cluster machines often have private IPs and can be reached only through a frontal machine. To get round this problem, the frontal may relay messages.

MPI does not take into account the network architecture but CRAC does. Thus we can give the following definition of "site" as a pool of machines that can directly connect to one another. This notion is not necessarily geographical but this may sometimes be the case. For example, if the machines of two distant clusters can freely interconnect, it is better to separate them in two sites if the bandwidth between the clusters is low. If it is similar to the bandwidth inside clusters, they can be gathered in the same site.

Within a site, four types of machines are possible: master, supermaster, slave, frontal. The last type may be applied to any of the three first. For example, a machine may be a slave frontal. This characterization allows to optimize the management of the VDM (starting/stopping the daemons, spawning, ...), and to reach machines with private IPs. Here is the definition of the types.

- frontal: a machine that can relay messages from outside the site to the private IP machines of the site. It can also relay messages to another site if a machine cannot send outside the site.

- slave: a machine with no particular role.

- master: a machine that collects informations from the slaves of the site and relays them to the supermaster, or that relays informations from the supermaster to the slaves.

- supermaster: a machine that collects/sends informations from/to the masters. Obviously, the supermaster is a master but is unique.

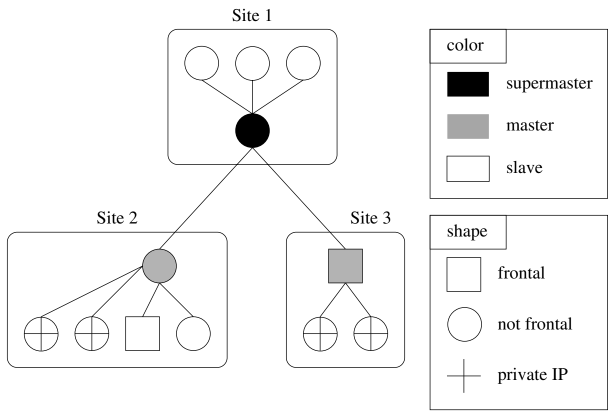

Figure 3. An example of VDM

The VDM is defined via an XML file, which is a perfect language to describe its hierarchical organization. This file is passed as an argument to a booter (like lamboot) that launches the daemons on each machine of the VDM. Then, a TCP connection is created between each master and the supermaster, and between each slave and its own master. This hierarchy allows to limit the bandwidth use between the sites. For example, when tasks are spawned for an execution, the supermaster sends the configuration of the execution (the machines used and their number) to all masters, which relay the information to their own slaves.

In order to limit the number of connections between tasks, the convergence detection mechanism also uses this hierarchy. Thus, even if a master runs no task, its daemon is in charge of collecting the local convergence state of each task running in the site.

Figure 3 shows an example of VDM with 3 sites. The lines represent the TCP connections that constitute the hierarchical network used for convergence detection and for management (essentially launching and stopping tasks). In sites 2 and 3, two slaves have private IPs. Thus, it is mandatory for a machine of this site to be a frontal. It may be the master itself as in site 3 or simply another slave, as in site 2. It can be noticed that there are no connections between masters and that the supermaster may also have slaves, as in site 1.

During the execution of an application, a task may communicate data to a task on another machine. The hierarchical network is never used for that. Instead, a new TCP connection is created between the machines running the two tasks the first time they want to communicate (see just below).

- Daemon: a CRAC daemon is launched on each machine of the VDM. During an execution, its main use is to send and receive messages for the local tasks. If the machine is a frontal, the daemon may also relay messages to tasks hosted by another daemon. These operations are executed by two threads.

- the **Sender** thread: each time it awakes, it checks in the **outgoing queue** the presence of messages to send. If no socket exists to the destination machine, the Sender tries to connect and to retrieve a new socket, dedicated to send application data to that destination. Even if the destination machine hosts several tasks, a single socket is used.

  However the destination machine may have a private IP. In this case, the Sender tries to connect to the frontal machine of the destination site. Each message will be sent to the frontal, which will relay the data to the real destination.

  In order to optimize the global communication time, each message is composed of a header followed by packets and is not sent in one chunk. As each packet has a fixed destination, the Sender does a loop on the destinations of the packets: it sends a packet to one destination after another. Obviously, if a new message to an existing destination is inserted in the outgoing queue, it must wait for the end of the emission of the current one. But if the new message is for a new destination, it can be sent immediately. This process is a kind of pipeline, which greatly reduces the time needed by the last message inserted in the queue to arrive completely at its destination.

- the **Receiver** thread: it uses a polling mechanism to passively detect connection demands and the incoming of data on existing sockets. In the last case, the Receiver uses the header to determine the destination task. If this task is not running on the machine, it means that the message must be relayed and it is directly put in the outgoing queue to be sent by the Sender. If the machine hosts the destination task, the Receiver retrieves the source task from the header and a slot of the **incoming queue**, associated to that task, is used to store the data.

  The slot allocation policy is the following. The Receiver always checks if a slot with the same message characteristics {source,destination,tag} exists. If this is the case, existing data are overlapped by the ones to come, else a new slot is created. This overlapping is particularly useful to accelerate convergence in asynchronous executions (see section 2). The slot is freed when the task retrieves its data.

  Taking the example of Figure 2, the Receiver of B would create a slot for the first message coming from A. At the end of its first iteration, B retrieves the data of the slot. During its second iteration, the Receiver creates a slot for the second message from A but uses the same slot to store the data of the third message. Thus, B is insured to always have the latest data sent by A.

It must be noticed that this policy works perfectly well for synchronous executions. Indeed, for a given triplet {source,destination,tag}, a single message can be sent/received during the same iteration. Thus, there cannot be lost data because of overlapping messages.

The daemon also creates the **Converger** thread that is in charge of collecting and updating information about the convergence, using the hierarchical network of the VDM. It implies that the supermaster has more information to collect than masters, and masters more than slaves. Thus, the working of this thread depends on the machine type but whatever the case, its final goal is to provide the global convergence state to the tasks.

- Task: the application task is a thread that executes within the daemon context. Thus, the task can directly access message queues (incoming and outgoing). This is not the case for MPI, in which a task is a process and must communicate (with an Unix socket or shared memory) with the daemon to send/receive data.

As CRAC is an object environment, the **Task** class is defined as a thread, containing all primitives of the programming interface and the classical attributes of a task (identifier, number of task in the daemon and in the VDM, ...). CRAC also declares (as an include file) the **UserTask** class which inherits from **Task**. This class contains a `run()` method that must be defined by the user in a C++ file, which is compiled as a shared library. When a task is launched on a machine via the spawner, the daemon dynamically loads its code and creates a new thread object containing this code. The thread is started and its `run()` method automatically called, as in Java.

- Spawner: the CRAC spawner is a classical MPI spawner, except that it uses an XML file to specify which and how many tasks are launched on which machine. The access path of the code of each task must be given for each machine. Thus, it is possible to have an MIMD execution. It is also possible to pass arguments to each task. For now, the spawn is only static and tasks cannot be added during an execution.

- Programming interface: it is defined in the Task class. It provides the classical primitives to implement message passing codes but some have special semantic and some are dedicated to iterative algorithms. Here are four characteristic examples that greatly differ from MPI.

- `CRACSend()`: the emission of a message is never blocking. This routine simply copies the data in a slot of the outgoing queue. Thus, the buffer containing the data can be immediately reused. The slot allocation policy is identical to that of the incoming queue: a new slot may be created or an existing slot chosen and its data overlapped.
- `CRACRecv()`: the reception may be blocking or not, depending on a parameter of this function. In MPI, the non-blocking reception returns an identifier that allows to test and to wait for the total reception of the message. In CRAC, it is like a test/receive. If the message is in the incoming queue, the buffer passed to `CRACRecv()` is filled and it is left empty if no message arrived. This semantic is dedicated to an asynchronous execution for which it must be possible to begin another iteration without new data being received.
- `CRACConvergence()`: it may be blocking or not, depending on a parameter of this function. In both cases, it takes a boolean as a parameter, which is the local convergence state. It returns the global convergence state as a boolean. Obviously, this routine must be used in blocking mode for a synchronous execution. For a description of its working in asynchronous mode, one can refer to [4].
- tags : each message must be marked by an integer value defined by the user. As mentioned above and in the Receiver description, there is an automatic replacement if a message with the same triplet {source,destination,tag} is already present in the queues. Thus, the user must assign the same tag to the messages that are used to update the same data set of the destination task. Obviously, the same tag must never be used for messages updating different data sets.

In the next section, we present an application that allowed us to examine the behavior of CRAC in a grid context.

4. Experimentations: Advection-Diffusion problem

In order to analyze the behavior of CRAC, we conducted experiments on the GRID'5000 architecture. We have chosen a problem based on an advection-diffusion equation which is modeled by a PDE (*Partial Differential Equation*).

4.1. The problem and the method used to solve it

In this problem, we compute the evolutions of the concentrations of two chemical species in a two dimension do-main. This problem corresponds to an advection-diffusion system with two species. It is solved by using a discretization of the space on a two-dimensional grid (x, z).

The evolutions of the concentration species are given by

$$\frac{\partial c^i}{\partial t} = K_h \frac{\partial^2 c^i}{\partial x^2} + V \frac{\partial c^i}{\partial x} + \frac{\partial}{\partial z} K_v(z) \frac{\partial c^i}{\partial z} + R^i(c^1, c^2, t) \quad (1)$$

where c^i $(i = 1, 2)$ denotes the concentration of the chemical species, K_h, V and K_v respectively denotes the horizontal diffusion coefficient, the velocity and the vertical diffusion coefficient. R^i represents the reaction of the chemical species [11]:

$$\begin{array}{l} R^1(c^1, c^2, t) = -q_1 c^1 c^3 - q_2 c^1 c^2 + 2q_3(t) c^3 + q_4(t) c^2 \\ R^2(c^1, c^2, t) = q_1 c^1 c^3 - q_2 c^1 c^2 + q_4(t) c^2 \end{array} \quad (2)$$

with

$$\begin{array}{lcllcl} K_h & = & 4.0 \times 10^{-6} & V & = & 10^{-3} \\ K_v(z) & = & 10^{-8} e^{\frac{z}{5}} & c^3 & = & 3.7 \times 10^{16} \\ q_1 & = & 1.63 \times 10^{-16} & q_2 & = & 4.66 \times 10^{-16} \\ q_j(t) & = & e^{-a_j / sin(\omega t)} & \multicolumn{3}{l}{\text{for } sin(\omega t) > 0} \\ q_j(t) & = & 0 & \multicolumn{3}{l}{\text{otherwise}} \end{array} \quad (3)$$

and $j = 3, 4$, $\omega = \pi/43200$, $a_3 = 22.62$ and $a_4 = 7.601$.

The initial conditions are the following

$$\begin{array}{l} c^1(x, z, 0) = 10^6 \alpha(x) \beta(z) \\ c^2(x, z, 0) = 10^{12} \alpha(x) \beta(z) \end{array} \quad (4)$$

with

$$\begin{array}{l} \alpha(x) = 1 - (0.1x - 1)^2 + (0.1x - 1)^4/2 \\ \beta(z) = 1 - (0.1z - 1)^2 + (0.1z - 4)^4/2 \end{array} \quad (5)$$

The discretization using an implicit Euleur scheme along x and z allows us to rewrite the system of PDEs in Equation 1 in a system of ODEs (*Ordinary Differential Equations*) of the form

$$\frac{dy(t)}{dt} = f(y(t), t) \quad \text{with} \quad y = (c^1, c^2) \quad (6)$$

where the vector $y(t)$ is a vector of n elements.

In a sequential execution, the Newton algorithm is used to solve such a system of ODEs, i.e. to find an approximation of the vector which contains the unkowns of the system at each time step t. Solving this equation in parallel is possible in at least two different ways.

The first one, and probably the best known and most commonly used one, consists in using the Newton iteration in parallel. At each Newton iteration a large linear system need to be solved and a parallel linear solver is used for that. Synchronous linear solvers often require several synchronizations whereas asynchronous linear ones do not require any synchronization at all. Nevertheless, each Newton iteration requires a synchronization. So, whatever the parallel solver used, synchronizations are unavoidable.

The second solution consists in using the multisplitting method which can be used in an asynchronous mode. Practically speaking, it leads to an algorithm which does not require any synchronization to solve Equation 6 for a given t. The principle is the following: the non linear function is not considered in its totality on each processor. For convenience, it is decomposed into as many parts as there are computing tasks. Each task manages the computation of the unknowns corresponding to its part of the function. Thus, each task solves a part of the non linear function using the sequential Newton Algorithm, mentioned above. Obviously, several iterations are required for the algorithm to converge to the initial problem solution. As said previously, an asynchronous execution supposes that the computation of the next iteration begins without waiting for data from neighbor tasks. If data are always available, the computation quickly converges towards the solution but if some are missing, the process converges more slowly. Obviously, the convergence rate depends on a lot of parameters such as the machines power, the network bandwidth, and the problem itself. For more details on the multisplitting Newton method (implementation details and convergence results), interested readers are invited to read [3, 7, 15] and the references therein.

To summarize, the benefit of using the asynchronous multisplitting Newton method is that only one synchronization is required at each time step to compute the next time step, whereas all other methods require much more synchronizations that may be penalizing in a grid computing context with distant clusters.

4.2. Results on GRID'5000

Experimentations have been conducted on the GRID'5000 architecture. Currently, the GRID'5000 platform is composed of an average of 1000 bi-processors that are located in 9 sites in France: Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia-Antipolis, Toulouse. Most of those sites have a Gigabit Ethernet Network for local machines. Links between the different sites ranges from 2.5 Gbps up to 10Gbps. Most processors are AMD Opteron. For more details on the GRID'5000 architecture, interested readers are invited to visit the website: www.grid5000.fr.

In Tables 1 and 2, we report the result of experiments on the advection-diffusion problem discretized as described previously, using the multisplitting method. In order to solve sequentially the sparse linear system, we have used the MUMPS software [1] which is a direct sparse linear solver. In the following experiments, we have used 120 ma-

discretization step : 360 s				
problem size	synchronous		asynchronous	
	exec. time (s)	# of iter.	exec. time (s)	# of iter.
1400×1000	47.8	252	25.9	264-290
2100×1500	123.8	429	80.6	452-496
2800×2000	271.7	626	190.7	710-832
4200×3000	981.3	984	668.8	1108-1274

Table 1. Experimentations on the advection-diffusion problem with the GRID'5000 architecture : 360 s discretization steps.

discretization step : 720 s				
problem size	synchronous		asynchronous	
	exec. time (s)	# of iter.	exec. time (s)	# of iter.
1400×1000	75.5	393	39.4	401-437
2100×1500	242.1	696	184.8	712-846
2800×2000	431.9	964	299.0	1042-1169
4200×3000	1368.9	1523	1046.7	1691-1864

Table 2. Experimentations on the advection-diffusion problem with the GRID'5000 architecture: 720 s discretization steps.

chines scattered in 4 sites of GRID'5000. Nodes are approximately similar, with a computing power ranging from AMD Opteron 2Ghz to AMD Opteron 2.25Ghz.

In the following, each result is the mean of 10 executions. In order to compare the behavior of the application we have chosen two discretization steps: 360 seconds and 720. For each value, reported execution times have been achieved for 2 time steps. Different sizes of problems have been examined in order to analyze the behavior of CRAC with a variable ratio between computation and communication time. For example, a problem size of 4200×3000 means that, because of the two chemical species, the global matrix has $2 \times 4200 \times 3000 = 25,200,000$ rows and columns, with 10 non-null elements on each row. It can be noticed that the multisplitting method allows the overlapping of some components, which may decrease the number of iterations. In our experiments, we have chosen an overlapping size equal to 20 for each dimension.

The study of Table 1 and 2 reveals that the asynchronous version of the algorithm is always faster than the synchronous one. This phenomenon is due to the fact that in the synchronous case, all tasks are synchronized at each iteration of the multisplitting method. When the problem

discretization step	problem size	exec. time ratio
360	1400×1000	1.85
	2100×1500	1.53
	2800×2000	1.42
	4200×3000	1.47
720	1400×1000	1.92
	2100×1500	1.31
	2800×2000	1.44
	4200×3000	1.31

Table 3. Ratio between synchronous and asynchronous execution times.

size increases, the ratio of computation over communication time increases too, and the difference between the synchronous and the asynchronous execution times decreases. It is clearly shown in the last column of Table 3, which gives the synchronous execution time divided by the asynchronous one. This fact was commonly observed in all our studies of asynchronous algorithms. For each version of the algorithm, Tables 1 and 2 also report the number of iterations required to reach the convergence. In the asynchronous case, this number varies from one execution to another, and from one processor to another. That is why we report an interval which corresponds to the minimum and the maximum number of iterations of the different executions. Without considering the mode of execution of the algorithm, the larger the size of the discretization step, the more iteration are required to reach the convergence.

>From a programmer point of view, CRAC is easy to use to develop an asynchronous version of a synchronous algorithm using the multisplitting method. By extension, any synchronous algorithm designed to be asynchronous as well, will be easy to implement. In fact, only the initialization of the program is different by specifying the execution mode, synchronous or not.

It should be noticed that we do not claim that any synchronous algorithm can be transformed into an asynchronous one. Only some algorithms for which the convergence has been studied can be run in an asynchronous mode.

All the previous remarks could be formulated with any similar programming environment allowing to develop synchronous and asynchronous algorithms. However, CRAC is, to the best of our knowledge, the first environment dedicated to implement efficient asynchronous algorithms in C. In the past, we tried to implement the same problem with a multithreaded version of MPI. As in our works with JACE, CRAC allows the reduction of the amount of code by a third, which represent all the lines needed to explicitly manage the asynchronism. Furthermore, our experiments on a cluster of 20 machines have shown that the difference of the execution time is less than 1% between the CRAC and the multithreaded MPI version. Thus, the overhead of CRAC is negligible.

5. Conclusion and perspectives

In this paper, we have shown that CRAC is very useful to design asynchronous algorithms. Currently, the scientific community does not really know this kind of algorithm and think that this is only adapted to experts. We sincerely hope that CRAC would reverse this trend by providing a programming environment dedicated to easily implement and to efficiently run asynchronous algorithms.

We plan to use CRAC to solve large sparse linear problems with the multisplitting method. There are some similarities with the multisplitting method for nonlinear problems. Larger experimentations with an important number of distant clusters and processors (more than 1000) should more than ever show the interest of using asynchronous algorithms to efficiently solve scientific applications in a grid computing context.

References

[1] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods in Appl. Mech. Eng.*, 184:501–520, 2000.

[2] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Evaluation of the asynchronous iterative algorithms in the context of distant heterogeneous clusters. *Parallel Computing*, 31(5):439–461, 2005.

[3] J. M. Bahi and R. Couturier. *Asynchronous multisplitting methods for linear and nonlinear systems*, chapter 6, pages 147–181. Gakkotosho, Tokyo Japan, 2006. ISBN 4-7625-0434-3.

[4] J. M. Bahi, S. Domas, and K. Mazouzi. Combination of java and asynchronism for the grid : a comparative study based on a parallel power method. In *18th IEEE and ACM Int. Conf. on Parallel and Distributed Processing Symposium, IPDPS 2004*, pages 158a, 8 pages, 2004.

[5] J. M. Bahi, S. Domas, and K. Mazouzi. Jace : a java environment for distributed asynchronous iterative computations. In *12th Euromicro Conference on Parallel, Distributed and Network based Processing, PDP'04*, pages 350–357, 2004.

[6] J. M. Bahi, S. Domas, and K. Mazouzi. More on JACE: New functionalities, new experiments. In *IPDPS'2006, 20th IEEE and ACM Int. Symposium on Parallel and Distributed Processing Symposium*, pages 231–239, 2006.

[7] J. M. Bahi, J.-C. Miellou, and K. Rhofir. Asynchronous multisplitting methods for nonlinear fixed point problems. *Numerical Algorithms*, 15(3,4):315–345, 1997.

[8] M. Baker and B. Carpenter. MPJ: A proposed java message passing api and environment for high performance computing. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 552–559, London, UK, 2000. Springer-Verlag.

[9] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, Englewood Cliffs NJ, 1989.

[10] D. Caromel and al. ProactivePDC : Java library for parallel, distributed, and concurrent computing. http://www-sop.inria.fr/oasis/ProActive/.

[11] A. C. Hindmarsh and R. Serban. Example program for cvode. http://www.llnl.gov/CASC/sundials/.

[12] S. T. I. Foster, C. Kesselman. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal Supercomputer Applications*, 15:200–222, 2001.

[13] G. Mercier. MPICH-Madeleine III : An MPI Implementation for Heterogeneous Clusters of Clusters. http://dept-info.labri.u-bordeaux.fr/ mercier/mpi.html.

[14] S. Morin, I. Koren, and C. Krishna. Jmpi: Implementing the message passing standard in java. In *Int. Conf. on Parallel and Distributed Processing Symposium, IPDPS 2002*, 2002.

[15] D. B. Szyld and J.-J. Xu. Convergence of some asynchronous nonlinear multisplitting methods. *Numerical Algorithms*, 25:347–361, 2000.