High Performance Computing Using ProActive Environment and The Asynchronous Iteration Model

Raphaël Couturier  David Laiymani  Sébastien Miquée
Laboratoire d’Informatique de Franche-Comté (LIFC)
University of Franche-Comté
IUT de Belfort-Montbéliard, Rue Engel Gros, BP 27, 90016 Belfort, France
Tel.: +33-3-84587781  Fax: +33-3-84587781
Email: {raphael.couturier,david.laiymani,sebastien.miquee}@univ-fcomte.fr

Abstract—This paper presents a new library for the ProActive environment, called AIL-PA (Asynchronous Iterative Library for ProActive). This new library allows to execute programs for solving large scale problems on various architectures. Two models of algorithm can be used: the synchronous iteration model which is efficient on single clusters; the asynchronous iteration model which is more efficient on distributed clusters. Both approaches are tested on both architectures, using Kernel CG of the NAS Parallel Benchmarks on the Grid’5000 platform. These tests also allow us to compare ProActive with AIL-PA and with the Jace programming environment. The results show that the asynchronous iteration model with AIL-PA is more efficient on distributed clusters than the synchronous iteration model. Moreover, these experiments also show that AIL-PA does not involve additional overhead to ProActive.

I. INTRODUCTION

In the past few years, clusters and distributed clusters have replaced super-computers in many applications, e.g. climatic simulation or biological research. In order to efficiently use this massive distributed computation power, numerous numerical algorithms have been elaborated. These algorithms can be broadly classified into two categories:

- **Direct methods**, which give the exact solution of the problem using a finite number of operations (e.g. Cholesky[1], LU[2], etc). However, these methods cannot be applied to all kinds of numerical problems. In general, they are not well adapted to very large problems. Moreover, direct methods are difficult to parallelize.

- **Iterative methods**, that repeat the same instructions until a good approximation of the solution is reached. We show in our work that the algorithm has converged to the solution. Iterative algorithms constitute the only known approach to solving some kinds of problems and they are easier to parallelize than direct methods. Jacobi or Conjugate Gradient[3] algorithms are examples of such iterative methods.

As our aim is to solve very large problems, in the rest of this paper we only focus on iterative methods. Now to parallelize this kind of algorithm, two classes of parallel iterative algorithms can be described:

- **The synchronous iteration model.** In this model, as can be seen on Figure 1, after each iteration (represented by a filled rectangle), a node sends its results to its neighbours and waits for the reception of all dependency messages from its neighbours. This results in large idle times (represented by spaces between each iteration in the figure) and is equivalent to a global synchronisation of nodes after each iteration. These synchronisations can strongly penalise the overall performances of the application particularly in case of large scale platforms with high latency network. Furthermore, if a message is lost, its receiver will wait forever for this message and the application will be blocked. In the same way, if a machine dies, all the computation will be blocked.

- **The asynchronous iteration model.** In this model, as can be seen on Figure 2, after each iteration, a node sends its results to its neighbours and starts immediately the next iteration with the last received data. These data could not be the very last, but data from a precedent iteration. The receiving and the sending mechanisms are asynchronous and nodes do not have to wait for the reception of dependency messages from their neighbours. Consequently, there is no more idle time between two
iterations. Furthermore, this model is tolerant to messages loss and even if a node dies, the rest of the nodes continues the computation, with the last data the dead node sent. Unfortunately, the asynchronous iteration model generally requires more iterations than the synchronous one to converge to the solution.

Now, from an architecture point of view, super-computers are less used, because they are expensive and so is their maintenance. Moreover, when such a machine falls down, no application can be run until it is be alive again. For these reasons, cluster architectures become more and more popular in the scientific community, and particularly distributed clusters. These architectures are composed of many local clusters distributed over many distant sites. Two machines from distinct clusters may have different configurations and specifications. Although these architectures present huge computing resources, they suffer from high latency communications between distinct sites.

Latency is an important problem, especially with synchronous algorithms which provide few overlapping of communications by computations. For these reasons, the asynchronous iteration model is interesting because it provides a natural overlapping and it suppresses all synchronisations.

Nowadays, designing of distributed algorithms requires a lot of mechanisms, like the repartition of tasks over all computing machines and the communication management between nodes. So, in order to facilitate the development of applications there exist programming environments which provide some facilities to programmers. In the rest of the document we will focus on two environments:

- **ProActive**[4], which is a middleware for parallel, distributed and concurrent computing in a uniform framework. It provides a set of simple and comprehensive functions to simplify the programming of grid computing applications on clusters or distributed clusters. ProActive is made of standard Java classes, this implies programmers to write a standard Java code. This environment is itself extensible, making the system open for adaptations and optimisations. Unfortunately, writing asynchronous iterative algorithms with ProActive is not easy and may lead to poor performances.

- **Jace**[5], which is a pure Java environment, conceived with a view to provide easy development of asynchronous iterative algorithms.

The aim of this paper is to present a new library, called asynchronous iterative library for ProActive (AIL-PA), designed for the ProActive environment, which aims to provide facilities to write efficient programs based on the asynchronous iteration model.

This paper is organised as follows. Section II explains the motivations of our research. Section III describes more precisely the ProActive environment by giving details about its architecture and its way of working. Section IV presents AIL-PA, which provides to the ProActive environment mechanisms allowing the designing of efficient programs based on the asynchronous iteration model. Finally, section V describes the experiments we have performed with this library, and some comparison results with the Jace environment.

**II. Motivations**

A. **Adding the asynchronous iteration paradigm to the ProActive environment**

As mentioned before, the ProActive environment only allows the development of programs based on the synchronous iteration model. Grid computing is generally used for large problems solving, and in this context, the use of the asynchronous iteration model can be very beneficial, e.g. latencies in network could be overlapped by computing tasks. That is why a library which provides possibility to develop asynchronous iteration algorithms has been created for ProActive.

The fact of adding a new library which provides a set of functions for algorithms implementation allows to add a new way to communicate with the ProActive environment, and is therefore an advantage for it. Here, we also underline that the new library can provide methods for both synchronous and asynchronous iteration models.

B. **Performance comparison between Jace and ProActive**

In addition to the previous point, a performance comparison between the two environments Jace and ProActive could also be established. Indeed, as both environments are built to provide tools to easily develop applications for solving large scale problems, it is interesting to compare them. To be significant, experiments must compare synchronous and asynchronous parts of the two environments with large scale problems on more than 200 computing cores.

**III. The ProActive environment**

ProActive is a complete programming environment, written in the Java language. It was created with the orientation to allow programmers to develop applications with synchronous communications. Its communication model is oriented to synchronous communications, because when two nodes want to exchange data (with the exchange[4, pages 4-5] method) they first have to synchronise each other. So it is hard to implement parallel algorithms based on the asynchronous iteration model. But ProActive environment provides another way to communicate which is based on a pure RMI model (the exchange method also uses the RMI protocol but with a different view of working. When nodes call a remote method, they do not need to wait for the result of this method and could continue their execution, until this result is needed. If the result has arrived, the program can process it and continue its execution. Now if the message has not already arrived, the program waits for it (waiting time is moved in the code) – the result is then called here a “future object” because we consider that it is received even if it is not. This approach provides a little part of asynchronism, only in communications.
A. Presentation

AIL-PA (Asynchronous Iterative Library for ProActive) is a package designed to provide ProActive with a set of methods which allows to develop applications which are based on the asynchronous iteration model.

| AILObject |
| Attributs |
| − messages queues |
| Methods |
| − send message |
| − receive message |

Fig. 4. The AIL-PA extension

AIL-PA is an extension of the “ActiveObject” class of ProActive (which is the main class, the worker class). In fact, AIL-PA is an upper layer of this object by encapsulating it, as shown in Figure 4 (using this library is done by using class MyClass extends AILObject instead of class MyClass implements Serializable in the standard declaration of the class). This solution allows the users to have communication methods accessible directly in the program code and the communication scheme is more simple. As the library provides methods for implementing asynchronous iterative algorithms, it also provides methods for programming applications based on synchronous communications (in some asynchronous iteration model based applications, synchronous communications must be used in some cases). AIL-PA is fully integrated with the ProActive environment and does not need specific configuration to be used. So, it seems to be interesting to get hold of some tools facilitating such a programming.

B. Components of the library

The communication layer used is the RMI protocol, which was modified by ProActive. This solution was chosen in order to fit with the environment and not to overload the library with new communication methods. But the conception is general enough to allow the addition of new communication protocols, like Socket or NIO for example. In the library, there are four main parts:

- The Message class. This part defines the skeleton of a message to be transmitted. A message is an entity which is composed of data and a tag which describes the related iteration, a priority or a step number. By default, ProActive can just send tables of integers or bytes. This part allows to send more kinds of messages: integers, doubles or booleans.

As can be seen on Figure 3, the architecture of ProActive is based on the deployment of one or more Java Virtual Machine(s) on each node (so several virtual machines can be found on one physical machine). All tasks communicate with each other by using the RMI protocol or exchange data via the exchange method. In order to find the address of the destination node, some methods are used to retrieve a “group” which contains a workers list. As workers are known, we can call remote methods directly on the distant worker, or exchange data directly.

The deployment of the ProActive architecture is an important point and its configuration is based on a XML file, which describes the nodes list, the definitions of different environment variables (like ProActive root path, libraries path, etc), the parameters of the Java virtual machine... Each parameter and/or definition given in this file will be sent to every worker which configures its environment with these values.

ProActive environment provides a lot of functionalities which are essential when a large scale environment, like distributed clusters, is used. The “object migration” can be mentioned here. Indeed, ProActive allows monitoring and retrieving the state of a physical machine, and if the machine falls down rapidly or if its charge is too high, workers (which are considered as objects) can migrate to another machine. The monitoring of the machine can be done by the programmer in the code, in a separated thread, but users can also do it by using IC2D (a plugin part of ProActive), which provides a visual representation of the architecture. From this plugin, user can migrate a worker from a node to another.

So, ProActive is a complete environment providing a lot of methods to easily program many kinds of algorithms. Unfortunately, according to the previous point, ProActive needs a lot of memory to be executed, because a lot of libraries have to be loaded. Another drawback is that it is difficult to implement algorithms based on the asynchronous iteration model. Indeed, this algorithmic model implies a rigorous multi-threading programming and a specific management of the communications buffers.
This allows to exchange more kinds of data and does not need to convert data to integers or bytes. A drawback of this method is that it implies that the library needs more memory to be executed.

- **The MessageList class.** This class manages the messages queues in which messages are stored waiting for a retrieve request from the program. Two solutions could be implemented to manage these queues: using one shared queue, or using multiple queues. The second method was chosen because in a large scale architecture, with a lot of workers, searching in an unique queue may be costly; it implies to search a message from a sender with a specific tag in a large messages list, and this operation could be long if there is a lot of data exchanges. With the chosen solution, there are several queues per worker, as can be seen on Figure 5 – if the architecture implies 5 workers, each of them has 5 queues (because one is reserved for the convergence messages).

![Fig. 5. The MessageList](image)

So, the search for a message from a sender is very fast because it just has to take the relative queue, and then the search is only focused on the tag. Furthermore, when the architecture contains few workers, no fall of performance is observed.

- **The AILObject class.** This class is the main class, which is extended by the user’s class. It is an extension of a ProActive ActiveObject. It benefits from all the advantages it gives like the object migration for example. This part provides all methods for sending and receiving messages (reception of messages could be synchronous or asynchronous). Each receiving method, when called by the program, creates a separated thread to search for the requested message in the MessageList object. In synchronous mode, the thread searches and waits until the requested message arrives, but in asynchronous mode if the message has not arrived, the method returns an empty message and the program should work with old data. It also contains a “MessageList” object which manages messages queues. Finally, the AILObject class provides as a separate thread (an ActiveObject) a class which is in charge of the convergence detection of the application.

- **The Convergence class.** This part of the library is essential to applications based on iterative algorithms because it provides mechanisms for the convergence detection; for now, it provides a centralised convergence detection. The problem with the global convergence detection is only related to iterative algorithms which execute the same block of instructions until the “residual vector” is inferior to a requested precision ($\epsilon$). In a parallel application, this mechanism is decomposed into two phases: local and global convergence detection. When the residue of a subproblem, executed by a node is inferior to $\epsilon$, we say that the iterative method has converged to the solution on this node. The parallel iterative method converges globally when all nodes converge locally at the same time. With this part, two main methods can be used, as the application uses synchronous or asynchronous iteration paradigm. In the first one, the method returns the state of the global convergence only when it has received all local convergences from all workers. In the second one, with the asynchronous iteration model, some problems appear, because workers do not wait for last data to continue their job. When a worker sends its local convergence, the convergence manager (this part) stores the evolution of the convergence of this worker. After a finite number of the same convergence states (this number can be changed – by default it is fixed to five), the manager updates the convergence state of the worker, computes the global convergence and sends it to the worker. This protection is used for avoiding false convergence detections, when workers oscillate around the convergence threshold. Moreover, this class is an extension of a ProActive ActiveObject and benefits from all the advantages it gives.

Finally, Figure 6 presents the whole architecture of AIL-PA, with dependencies between components. On the figure, a plain arrow represent an association between two classes and an empty arrow represents an extension of a class.

![Fig. 6. Architecture of AIL-PA](image)
V. EXPERIMENTS

After having presented the library, we describe the experiments we have conducted and their components.

A. The Jace environment

Jace (Java Asynchronous Computing Environment) is a pure Java programming environment, especially designed for implementing asynchronous iterative algorithms (it also allows programming applications based on the synchronous iteration model).

The architecture of Jace is divided into three parts, which are represented on Figure 7.

- The daemons. Daemons (represented by filled ovals on the figure) are executed on each node of the architecture. They control the whole environment by initialising “workers” (which are tasks executing the program), managing machines, collecting results, etc. Each daemon is divided into two parts:
  - a set of JaceRuntime objects, which provides methods which are accessible from every local object of the program. This component manages communication and localisation of nodes and tasks in the Jace virtual machine.
  - a set of JaceServer objects, which provides methods which are executed by a separate thread and are accessible from every distant node (via RMI calls). These methods process requests from other nodes and transmit distant calls to the concerned local component (the worker or the daemon’s part).

- The spawner. It is an entity which allows us to execute an application. It takes in charge a list of parameters, which are the number of tasks to execute, the parameters of the application, the participant daemons list.

- The workers. As mentioned before, workers are created by daemons following a request of the spawner. This entity is composed of two layers:
  - the application layer allows the execution of tasks. In order to benefit from multi-core processors machines, it allows daemons to execute multiple tasks.
  - the communication layer is composed of two threads (“sender” for sending messages and “receiver” for the reception) and a set of messages queues. It provides three protocols which are RMI (Remote Method Invocation), Socket and NIO (New Input/Output).

The Jace environment offers interesting features like the possibility to implement both synchronous and asynchronous iterative algorithms and the possibility of using three different communication protocols. But it also has some drawbacks like the fact that it is not fault tolerant, and the lack of object migration for load balancing purpose.

B. The NAS Parallel Benchmark Kernel CG

We used the “Kernel CG” of the NAS Parallel Benchmarks (NPB) [6] to test the ProActive environment with and without AIL-PA versus the Jace environment. This benchmark is designed to be used on large architectures, because it tests communications over latency networks, by processing unstructured matrix vector multiplication. In this benchmark, a Conjugate Gradient is used to compute an approximation to the smallest eigenvalue of large, sparse and symmetric positive definite matrix, by the inverse power method. In our tests, the whole matrix contains nonzero values, in order to stress more communications. As the Conjugate Gradient method cannot be executed with the asynchronous iteration model we have replaced it by another method called the multisplitting method. This latter supports the asynchronous iterative model.

With the multisplitting algorithm, the $A$ matrix is split into horizontal rectangle parts, as Figure 8 shows. Each of these parts is affected to a processor – so the size of data depends on the matrix size but also on the number of participant nodes. In this way, a processor is in charge of computing its $X_{Sub}$ part by solving the following subsystem:

$A_{Sub} \times X_{Sub} = B_{Sub} - D_{Left} \times X_{Left} - D_{Right} \times X_{Right}$

After solving $X_{Sub}$, the result must be sent to other processors which depend on it.

![Data decomposition for the multisplitting method implementation](image)
The multisplitting method can be decomposed into four phases:

1) **Data decomposition.** In this phase, data are allocated to each processor assuming the decomposition exposed on figure 8. Then, each processor iterates until converge on the following.

2) **Computation.** To begin with, each processor computes \( B_{\text{Loc}} = B_{\text{Sub}} - D_{\text{eprLeft}} \times X_{\text{Left}} - D_{\text{eprRight}} \times X_{\text{Right}} \). Then, it solves \( A_{\text{Sub}} \times X_{\text{Sub}} = B_{\text{Loc}} \) by using a sequential version of the Conjugate Gradient method.

3) **Data exchange.** Each processor sends its \( X_{\text{Sub}} \) part to its neighbours. Here, the neighbourhood is closely related to the density of the \( A \) matrix. Clearly, a dense matrix implies an all-to-all communication scheme while a matrix with a low bandwidth reduces the density of the communication scheme.

4) **Convergence detection** Each processor computes its local convergence and sends it to a server node. When this one detects that each processor has converged, it stops the whole computation process.

It can be pointed out here that it is possible to modify the data decomposition in order to obtain non-disjoint rectangle matrices. This property of multisplitting methods, called overlapping, can bring significant improvements to convergence speed, but it is not the aim of this paper. More details about this method can be found in [7].

In our benchmark, the solver part of the multisplitting method is the Conjugate Gradient. Its implementation is multi-threaded, so it benefits from multi-core processors.

**C. The Grid’5000 platform**

The platform used for our tests, called Grid5000[8], is a French nationwide experimental set of clusters which provides a configurable and controllable instrument. We can find many clusters with different kinds of computers with various specifications and software.

Clusters are spread over 9 sites, as can be seen on Figure 9, and the computing power represents more than 5000 computing cores interconnected by the “Renater” network. This network is the national network for research and education; it provides a large bandwidth with low latency.

**D. Experiments on local cluster**

The first experiments set has been realised on a local cluster from a Grid’5000 site. From this cluster, 50 computers where used. Those provide 4 computing cores (so it represents 200 computing nodes) with 8 Go of memory, and are interconnected by a 10Gb/s local network.

In these tests, the NAS Kernel CG problem was used, with a large square matrix, of dimension 1 500 000. For these tests, the number of main iterations was fixed at 30 and all environments used the RMI protocol.

<table>
<thead>
<tr>
<th>Environments</th>
<th>Execution Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIL-PA with Synchronous model</td>
<td>758.65</td>
</tr>
<tr>
<td>AIL-PA with Asynchronous model</td>
<td>821.36</td>
</tr>
<tr>
<td>ProActive (Synchronous model)</td>
<td>765.08</td>
</tr>
<tr>
<td>Jace with Synchronous model</td>
<td>601.68</td>
</tr>
<tr>
<td>Jace with Asynchronous model</td>
<td>683.32</td>
</tr>
</tbody>
</table>

**TABLE I**

**EXPERIMENTS ON LOCAL CLUSTER OF NAS PROBLEM OF SIZE 1 500 000**

Table I shows the average results of 10 executions of our benchmarks, on a single cluster. First, we can focus on the synchronous iteration model comparison. Between AIL-PA in synchronous mode and ProActive, we note that both environments are similar. The new library is equivalent to ProActive and there is no loss of performance and no added overhead. Now when comparing it with the Jace environment, we note that Jace is more efficient than ProActive and AIL-PA. This could be explained by the fact that Jace is a lighter environment and needs less threads, so processors have less computation to manage.

Second, when using the asynchronous iteration model we point out that, as tests were run on a single cluster, this model is penalised by its nature. Results confirm that the asynchronous iteration model is not an efficient choice on such an architecture. This problem comes from the fact that the asynchronous model needs more iterations to converge to the solution, so it takes more time.

**E. Experiments on distributed clusters**

The second experiments set has been realised on distant clusters from two distant sites of Grid’5000. The architecture
used contained 110 computers, with this repartition: 32 quad-core computers on a cluster (128 computing nodes) with 8 Go of memory; 78 quad-core computers, from another cluster from a distant site (312 computing nodes) with 2 Go of memory; so it represents a total of 440 cores. Inside each cluster, local computers are interconnected by an efficient network at 10 Gb/s, and the two clusters are interconnected by a 10 Gb/s network. In this architecture, latency appears due to the distance between the two sites and the fact that many people are using the network at the same time.

In these tests, the NAS Kernel CG problem was also used, as in the previous experiments. For these tests, the number of main iterations was fixed at 10, due to the amount of time each iteration takes, and all environments used the RMI protocol during these tests.

<table>
<thead>
<tr>
<th>Environments</th>
<th>Execution Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIL-PA with Synchronous model</td>
<td>4310.24</td>
</tr>
<tr>
<td>AIL-PA with Asynchronous model</td>
<td>3259.27</td>
</tr>
<tr>
<td>ProActive (with Synchronous model)</td>
<td>—</td>
</tr>
<tr>
<td>Jace with Synchronous model</td>
<td>3923.68</td>
</tr>
<tr>
<td>Jace with Asynchronous model</td>
<td>2861.14</td>
</tr>
</tbody>
</table>

**TABLE II**

**Experiments on distributed clusters of NAS problem of size 1 500 000**

Table II shows the average results of 10 executions of our benchmarks, on the distributed clusters. Unfortunately, the first thing we can point out is that ProActive, with the exchange method of the Exchanger class, does not work on distributed clusters. So, we cannot compare the performances of our new library with the standard ProActive message passing one.

Now, with the synchronous iteration model, we can note that again the Jace environment is faster than AIL-PA. This fact is not surprising since Jace is lighter than AIL-PA, and by extension lighter than ProActive, because it provides less functionalities.

Now comparing the synchronous iteration model and the asynchronous one, we can note that the asynchronous iteration model is more efficient on such an architecture both with Jace and AIL-PA. This comes from latency in network interconnecting the two sites. We can say that with the use of this kind of architecture, AIL-PA provides to the ProActive environment a better efficiency with suitable algorithms.

**VI. Conclusion**

In this paper, we have presented a new library for the ProActive environment: AIL-PA. This library allows the ProActive environment to use easily the asynchronous iteration model. This new functionality is interesting since this model has proven to be efficient on distributed clusters.

We have shown, in our different experiments, that the synchronous iteration model is more efficient on single clusters, because of low latency in network. But, due to limitations of this architecture in term of total available memory, when very large problems should be solved, distributed clusters should be used. Using this kind of architecture leads to have latency in communications due to the distance between different sites and the use, by many users, of the interconnecting network. In this case, the asynchronous iteration model is more efficient than the synchronous one. Moreover, our experiments have shown that AIL-PA does not involved additional overhead to ProActive.

AIL-PA allows ProActive users to easily implement programs based on the asynchronous iteration model. Our experiments show that our library allows programs to be more efficient in asynchronous mode on distributed clusters. Moreover, as the library provides synchronous mechanisms, we show that it also provides a good alternative to synchronous communications with the ProActive environment.

Our future work concerns adding new communications protocols, like Socket or NIO, to our AIL-PA library. This would enhance the performances registered by AIL-PA, since Socket communications are faster than RMI.

**REFERENCES**


