High Performance Computing with RPC programming style

Georges Bosilca, Gille Fedak, Olivier Richard and Franck Cappello
Laboratoire de Recherche en Informatique
Université Paris-Sud, 95405 ORSAY Cedex
fci@lri.fr

May 17, 2000

Abstract

This paper investigates the performance of the RPC model of an original parallel environment called OVM for Out-of-order execution parallel Virtual Machine. OVM system is designed to reach high performance for parallel applications. OVM is based on a client-server architecture. Its design follows two main principles: RPC programming style and dynamic load balancing. This paper mainly address the performance issues related to RPC programming style. We test the performance of an OVM implementation on top of Myrinet for regular as well as irregular parallel applications. The contribution is three fold. First, we provide some key performance results of OVM. Second we show that RPC programming style can approach the performance of traditional SPMD programming style at least for limited number of nodes. To demonstrate this, we compare OVM implementation and original MPI implementation for 3 kernels of the NAS 2.3 benchmark. Finally we illustrate the programming and performance capabilities of OVM for irregular parallel applications examining a large real world application called AIRES.

Keywords: High performance parallel computing, RPC based communications, NAS 2.3 benchmark, Irregular parallel applications, performance evaluation

1 Introduction

RPC are being used, since their introduction on 1984, mainly in the context of Client Server systems across LAN, MAN and WAN. Current extensions of RPC include object management mechanisms and oriented object programming style (Remote Method Invocation in JAVA, CORBA RPC). Few experiments have been done in the context of high performance computing with this RPC programming style. Several research projects like Active messages, SHRIMP and Fast Messages provide either a fast communication system with a RPC like API or a performance optimized implementation of traditional RPC. These systems have been designed in the perspective of fast communication supports or for improving the performance of traditional client server applications. None of them address the issue of RPC programming style performance in the context of high performance parallel computing.

The Out-of-order execution parallel Virtual Machine (OVM) system is designed to reach high performance with client-server architecture. RPC programming style is used on the client side for invoking services on server side. The RPC approach of OVM drastically diverges from traditional RPC implementation and its current extensions. OVM only conserves the general programming style and leaves other functionalities like data marshaling and service binding. To go further, OVM does not address object oriented programming. So mechanisms used in CORBA or JAVA RMI like IDL and code load at execution time do not corresponds to OVM architecture philosophy. Conversely, OVM extends RPC paradigm for high performance parallel

computing. OVM is built on a specific set of assumptions. Although some of these assumptions may be relaxed on future OVM implementations, they govern the first development direction of OVM.

OVM architecture has two main principles: dynamic load balancing and a RPC programming style. This paper addresses the issues related to high performance RPC. So dynamic load balancing feature of OVM is mostly hidden in this study. To test the performance of OVM RPC programming style, we compare it with a traditional programming approach for high performance regular applications. We also present OVM parallelization and performance capabilities for an irregular parallel application. Althrough this application requires load balancing, we use a very simple load balancing scheme in order to highlight the performance of OVM RPC programming style.

The first section of this paper presents OVM principles, design and performance. We focus on decisions related to high performance RPCs. Section 3 compares OVM with MPI for the NAS NPB 2.3 benchmark. We consider the original MPI implementation and a specific OVM implementation based on the serial version of the NAS. Section 4 presents the parallelization and the performance of a large application called AIRES using OVM. The last section presents related work. Finally, we conclude.

2 Out-of-order execution parallel Virtual Machine

OVM is an original parallel environment for high performance parallel computing. It features an API for programming applications starting from scratch or from existing sequential version. It also provides a runtime support managing multiple computing resources.

Main target for OVM is high performance parallel applications. The general architecture of OVM is based on two main features: a RPC style programming interface and a build-in dynamic load balancing mechanisms.

Like some other client-server programming environments (Cilc, PM2, Atapascan), OVM is based on a set of assumptions related to high performance platforms. Unlike these environments, OVM does not consider the multithread programming paradigm.

The specific set of assumptions that governs OVM design is the following: a) OVM should run on high performance architecture. Myrinet clusters or parallel computers like IBM SP, Compaq SC and SGI Origin are appropriate platforms for OVM. We could call OVM as a SAN oriented client server system. b) a particular important point is that OVM relies on fast interconnexion network and user level protocol. Applications will not take advantage of the some performance optimizations of RPC mechanism on slow networks because OVM uses a very lightweight RPC mechanism. c) OVM considers homogeneous platforms. Processors can run at different speed but they must be of the same type to avoid data marshaling. All servers have same implementations of the services used by the application. Memory size on all servers should be identical and known at compile time to avoid dynamic server selection from memory size. d) OVM considers static platforms in term of number of servers. The number of servers should be the same from the start to the end of the application. There is neither server discovery mechanism nor service recovery in case of server failure or disconnection. e) OVM intents to provide RPC programming style for regular and irregular parallel applications. It uses a three-third client server architecture: client, broker, server [1]. Unlike CORBA or DCOM, this architecture does not address object oriented mechanisms. Conversely, OVM addresses load balancing but only the broker is aware and responsible of this feature. Neither the client nor the servers manages load balancing or even is aware of the broker decisions.

Figure 2 shows the general organization of OVM.

2.1 Client side

Client interacts with OVM using a low level API. Programmer makes requests to OVM annotating a sequential program with directives. Directive annotations concerns the OVM internal

operations (**create**, **kill**, **put**, **get** and **move**) and functions to execute remotely. A preprocessor translates programmer directives in API function calls. In the translation process, the preprocessor prepares requests to the broker. Programmer may also directly use API functions although this utilization is not recommended because real applications usually require very large number of API function calls.

The following program shows how annotations are used in a C program to request a remote execution of a matrix product.

```
{
float a[100];
/* Declaration and affectation of Matrix A, B, C and integer N */
//ovm create A,100  /* create an array called A with 100 entries in the server side */
//ovm put A,a        /* send the value of the a array to instantiate A array */
...
/* Requests remote execution of the matrix product */
//ovm req(bserv,dgemm,N,A,B,C)
...
}
```

As the first program interaction with OVM, the client requests to create an array A on the server side. The broker selects the server where the array is allocated. A is a reference to a contiguous memory region (100 words) on the server. The second step consists in the transfer of the value of the client local array a to the remote array A. Matrix element distribution (column first or row first) in memory regions is not relevant for OVM which conserves the words order during the transfer. The last step is the request for remote execution of the **dgemm** function with parameters **N,A,B,C**.

Programmer must select the parts of his program to execute remotely. The process of function selection for remote execution follows the framework shown in figure 1.

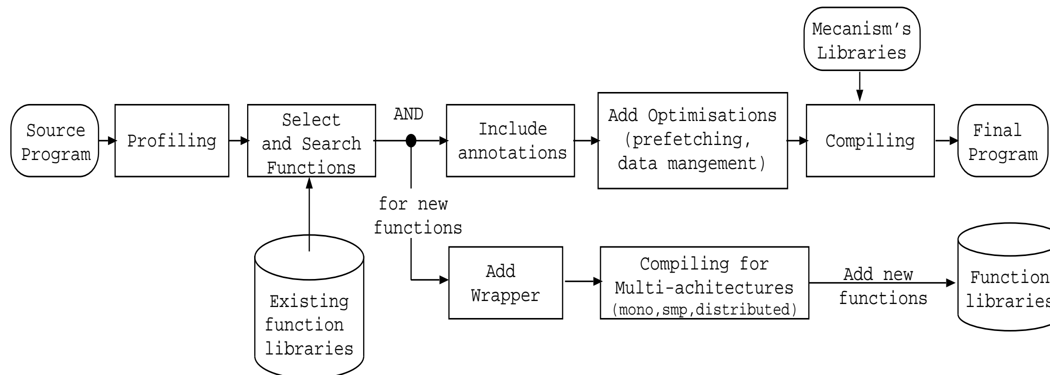

Figure 1: Framework for selecting functions to execute remotely

This process is currently the responsibility of the programmer. However, it should be automated and integrated in the compilation chain for client programs.

2.2 Broker

Basic roles of the broker is to manage the server workloads and to order data allocation, deallocation, reutilization and movements on the server side.

Figure 2 shows the broker high level architecture. The broker manages server workloads using a hierarchical design with two levels. The first selection level is on the critical path of remote execution. So this level is designed for very fast server selections (short term decisions).

Servers are selected based on a list managed by the second level of the broker workload manager. The second level updates the list for the first level from a global view of the servers and takes into account long term server management. The broker imprecise global view is established from its proper selections and from some informations returned by the servers. For example, servers send termination messages to the broker when functions terminate.

2.3 Server side

Servers software architecture uses a multi-threading approach (figure 2). This architecture follows two principles to improve the server reactivity. The first one concerns the request processing at arrival. We use a classical approach where a thread listens to requests while a team of threads are waiting. When a request arrives, the listening thread unlocks one of waiting threads and starts to process the request. The new listening thread waits for a new request. On multiprocessor server, non working threads spin on lock variables. On uniprocessor server, non working thread are waked up by signals. So requests are computed directly by the listening thread. The second principle consists in prefetching the server libraries used by the client programs. Users (or preprocessors) may include annotations (or API function calls) in client program to request the server to prefetch libraries. Prefetchs remove the dynamic library linking of the remote execution critical path.

2.4 Principles for high performance RPC

Performance of OVM relies on three optimization principles: very fast communications, Data reutilization and Asynchronous out-of-order execution model.

Very fast communications The first principle concerns the round-trip latency for RPCs. Figure 2 presents the critical path for remote executions.

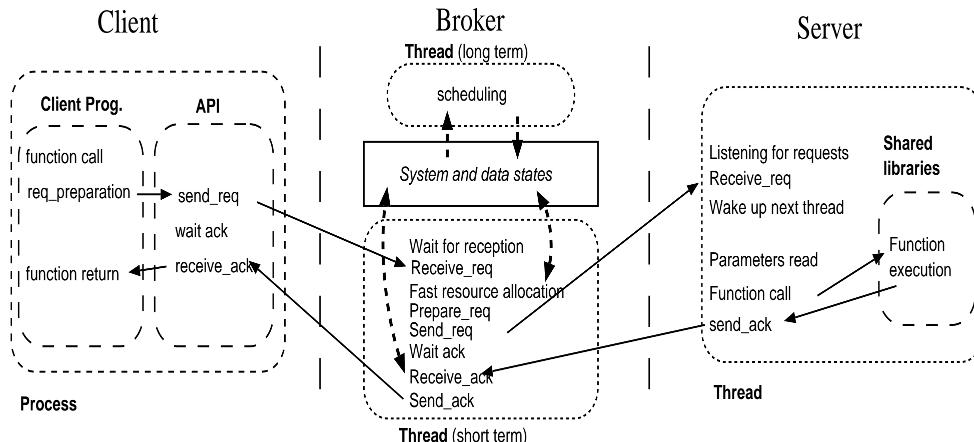

Figure 2: General architecture and critical path for remote executions.

To achieve very fast RPC, OVM only considers high performance SANs or LANs (Myrinet, SCI, Gigabits Ethernet, Servernet, Giganet) and high performance user level protocols (BIP, PM, Unet, Active messages, Fast Messages, VIA). Because the network hardware of SANs have several key properties (almost error free, packets stay ordered, homogeneous physical support), complex protocol stacks like TCP/IP are not useful in this context and OVM can benefit of the very low latency of these communication support. In this paper Myrinet is used for the communication hardware and OVM RPC are implemented on top of BIP. Fast RPC also relies on the broker and server software architectures. As previously mentioned, they are optimized

for performance and not for security or reliability. In contrast with *Metacomputing* systems where remote executions lead to dynamically create new processes (`fork` system call), OVM follows a multi-thread oriented design like the classical approach for single-application servers.

Fast RPC will be one of the main performance improvements from existing environments such as Netsolve, RCS and Ninf. These environments are worthwhile to use for remote executions with about $1sec$ duration. OVM is designed to allow relevant remote executions with sub $1ms$ duration. Note that OVM does not directly compare with *Metacomputing* environments because it does not currently manage heterogeneity and security. Using high performance protocols precludes to use OVM for MAN and WAN environments.

Data reutilization OVM uses a global naming system for the data sets on clients, broker and servers to limit the data transfers (between these entities). This is a main issue toward high performance for remote executions. Executing an application leads to a sequence of remote function executions. These functions may exhibit data dependencies and locality (read after write, read after read). To avoid multiple exchange transfers between client and server when consecutive functions presents data-flow dependencies, function results and parameters are passed by references and some data sets can stay resident at the server side even after function return. The client may call some OVM control operations to move data to and from the server side. The five main operations are **`create`**, **`kill`**, **`put`**, **`get`** and **`move`**. First operations create (destroy) a variable on the server side and return references that may be used to store and read data sets by client remote function calls. **`put`** operation writes value to existing variables on the server side using references. **`get`** does the reciprocal operation (reads a value from server side and writes it on the client). **`move`** operation allows point to point communication of a variable between two servers.

Figure 3 presents protocols for `Put` and `Get` operations. Numbers describe messages order. Data are effectively transfered on bold lines. **`create`**, **`kill`**, **`put`**, **`get`** and **`move`** are managed by the broker. When the broker selects different servers for data dependent functions, it has the responsibility to move datasets from producing servers to consuming ones. Client is not aware of data transfers issued by the borker.

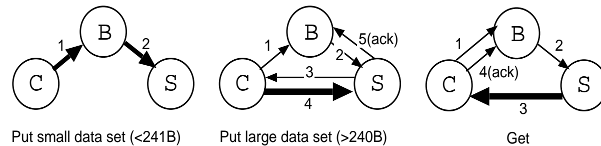

Figure 3: `Put` et `Get` operation protocols.

Asynchronous out-of-order execution model Asynchronous, Out-of-order execution model is the heart of OVM. Like some other client-server systems, OVM allows the client to issue asynchronous RPC. However, a major originality of OVM is that these RPC requests are managed by a out-of-order data-flow mechanism. The broker uses this mechanism for managing the load of the server.

Asynchronous out-of-order mechanism and implementation of OVM are very close, in principle, of the instruction management of an out-of-order superscalar microprocessor.

In a typical client sequence, several asynchronous RPC requests are sent to the broker. These requests are examined by the broker to check that arguments of the RPC are ready on the server side. If all arguments related to a RPC are ready, the broker forwards to the server that has been selected for the service execution. If one of the arguments is not ready, the broker stores the request in a table. When servers end a service, not only the result can be transfered to the client, but the broker is also informed of all the arguments that have been modified. The

broker updates the table of delayed requests. Requests for whom this update has completed the arguments list are sent by the broker from waiting table to servers.

3 Performance of basic OVM operations

Global performance of OVM tightly depends on the performance of basic operations like remote procedure call, data movements between client and servers and between servers.

Experimentation platform All performance measurements presented in this paper use the same parallel experimentation platform. The platform connects eight 500 Mhz Pentium III biprocessor nodes by a eight ports Myrinet network. The software environment includes Linux 2.2.14, BIP 0.99d [2] and Linux Pthread library. BIP raw performances on Myrinet is a latency of $5\mu s$ and a bandwidth of 1 Gbit/s. Every tests use one node for the client, one node for the broker and up to six servers.

We should state that although the asynchronous execution model of OVM is critical for high performance parallel applications, the out-of-order execution feature is not relevant in the context of this study: regular applications (or irregular applications without dataflow dependencies) and homogeneous experimentation platform.

Data transfers performance: (`Put`, `Get`, `Move`) We measure the performance for data transfers between a client and a server through the broker and between servers. The protocols used for the `Put`, `Get` and `Move` transfers are described in figure 3.

`Put` operations use two protocols according to the message size. The threshold for protocol change is 240 Bytes. Small messages can be buffered on the broker. Long messages must follow a rendez-vous protocol. The protocol complexity for `Get` operations hides the performance gap between fast and rendez-vous protocols.

Figure 4 presents the communication bandwidths for all transfer types (`Put`, `Get` and `Move`).

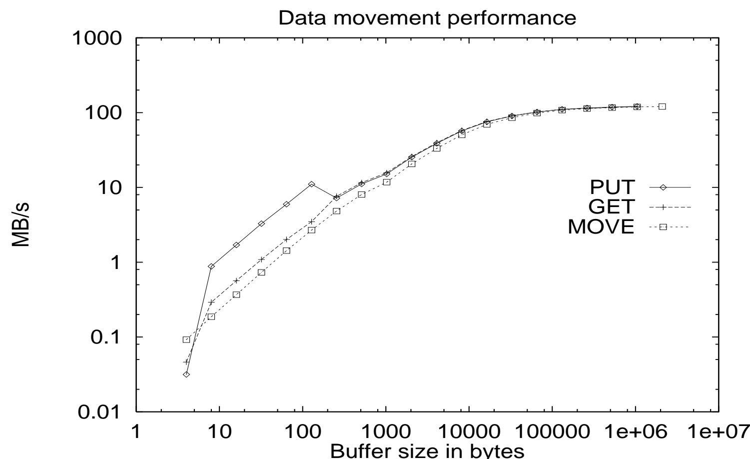

Figure 4: Communication performance (bandwidth) for OVM `Put` `Get` and `Move` operations

The performance plot for `Put` operations corresponds to the reciprocal of inter-sending delay up to 240 bytes. For larger messages, the bandwidth performance corresponds as usual to the reciprocal of half the round-trip time. Client `Get` operations require $32\mu s$ to get small data sets from servers. Client `Put` operations have a $6\mu s$ inter-sending delay.

Remote execution overhead Client request latencies are shown in table 1. Since requests return immediately executing a void function, the remote execution overhead equals the request latency.

asynchronous remote void execution + `get`	50 μs
synchronous remote void execution	32 μs
client part of remote execution	7 μs
client local void function call	0,16 μs

Table 1: Performance for invoking remote operations

Performance results do not consider the servers management algorithm executed by the broker. A test is done for locating the data set on the server side and for selecting the execution server.

Performance of remote global operations Global operations only concerns server side. They corresponds to main MPI global operations (they have the same semantic).

Figure 5 presents the performance for the two versions (OVM and MPI) of data all-to-all exchanging, broadcasting, and reducing on the server side for 6 nodes.

OVM implementation of global operations is customized to achieve best performance. MPI implementation corresponds to MPICH version of global operations. Although, global operations are implemented on top of point to point communications for the two versions (OVM and MPI), algorithms used for the two versions may be different.

4 OVM versus MPI on NAS Benchmarks

To measure the performance level of OVM and especially its RPC system, we compare it against MPI for 3 benchmarks (FT, CG, EP) of the NAS 2.3.

MPI codes correspond to the original implementation of the NAS NPB 2.3. To program the OVM version, we start from the sequential version of the NAS 2.3 and add RPC calls in the codes.

Note that we did not attempt to optimize neither the MPI nor the OVM versions according to memory hierarchy of the Pentium III. More generally, the two implementations could be considered as using the same optimization level.

FT, CG and EP exhibit different computation/communication ratios as well as different communication patterns. CG uses a lot of point to point communications with large and small messages. FT uses all-to-all communication patterns. EP is mainly computation bound and requires only one reduction at the end of the program.

Figure 2 compares the execution time (in second) of OVM and MPI implementations of FT, CG and EP for the Class A data set size.

Table 2 demonstrates that OVM versions can approach the performance of MPI version of the NAS benchmarks EP, CG and FT.

However, MPI version outperforms OVM version for all benchmarks. There are two main reasons behind this result. First OVM uses shared libraries on the server side. User functions are linked dynamicaly when the libraries are called. During the liking process, the linker adds some indirections for some references made in the original code. This leads to a lot of extra memory references during the execution. This problem is not related to OVM architecture but to the linux linker used in our experiement. Second, OVM global communications reach the performance of MPI ones only for large messages. Since, NAS benchmarks use relatively short messages, MPI perfoms faster the communications that OVM. This cost should be considered as the overhead of the load balancing features of OVM.

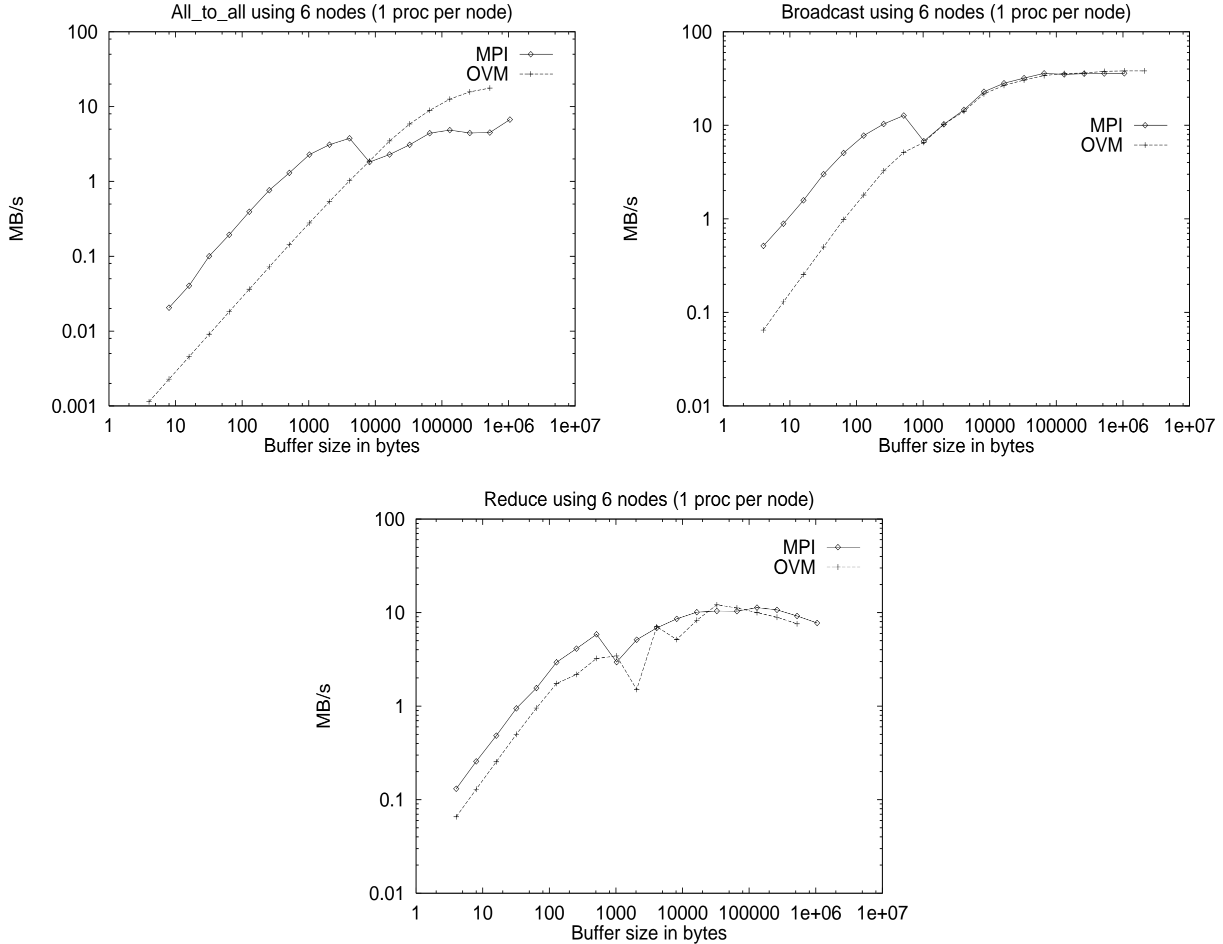

Figure 5: Performance of global operations (all-to-all, broadcast and reduce) for OVM (server side) and MPI using 6 nodes

5 OVM for a real life irregular application

The Pierre Auger Observatory project is an international effort to study high energy cosmic rays. Two giant detector arrays, each covering 3000 square kilometers, will be constructed in the Northern and Southern Hemispheres. Each will consist of 1600 particle detectors and an atmospheric fluorescence detector. The objective of the arrays is to measure the arrival direction, energy, and mass composition of cosmic ray air showers above 10^{19} eV.

Associated to this observatory is the Air Shower Extended Simulations (AIRES) application. AIRES simulates particle showers produced after the incidence of high energy cosmic rays on the earth's atmosphere. The complete AIRES 2.2.1 source code consists of more than 590 routines, adding up to more than 80,000 source lines.

AIRES simulates the particle shower from the high energy cosmic ray entry in the atmosphere up to earth surface. Basically, the application manages a stack of particles for which it computes the incidence on the atmospheric particles that are closer to earth surface. The simulation proceeds step by step following an iteration process. When the high energy cosmic ray enter the atmosphere it collides with a first set of particles that are stored in the stack. This is the first iteration of the simulation. Each of the collided particle will collide with other atmospheric particles. This collision generation is processed iteratively until the end of the simulation. There

	OVM 4 CPUs	MPI 4 CPUs	OVM 8 CPUs	MPI 8 CPUs
EP	105.09	95.94 (-9.5%)	54.42	48.38 (-12%)
CG	12.35	11.84 (-4%)	8.50	7.72 (-9%)
FT	50.09	42.42 (-15%)	30.93	25.75 (-15%)

Table 2: Comparison of the execution time (in second) of OVM and MPI implementations of the NAS benchmarks EP, CG and FT for 4 and 8 processors. Figures in brakets give the difference between MPI and OVM performance in percentage using OVM values as references

is no collision between particles belonging to deferent showers. So, the global shower can be viewed as a hierarchy of independent showers. When the energy of a particle decreases down to a threshold, it may be discarded for the next simulation iteration. The particle collision process follows a Monte-Carlo approach. Collision between particles depends on the probability law and keeping a particle that has reached the threshold also depends on a probability law.

As a consequence of the Monte-Carlo simulation approach, the number of particles inside the stack evolves during the execution, decreasing or increasing in each iteration.

The figure 6 presents the evolution of the stack during a typical simulation.

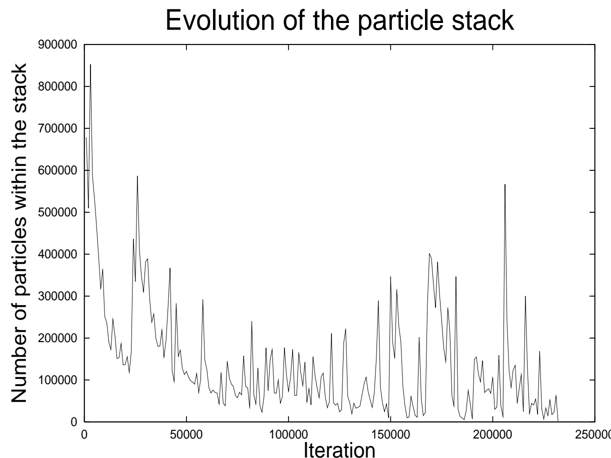

Figure 6: Evolution of the particle stack during a typical simulation

OVM implementation of AIRES decomposes the application in two codes: client program and server library. AIRES parallel execution follows tree steps: initialization, computation, result gathering.

During the initialization, the client start the simulation from a parameter file. When the iteration count reaches a threshold, the client stops the simulation and sends its simulation context to the servers. The context mainly consists in the client particle stack when it has stopped the simulation.

During the simulation, the client invokes remote sub-shower simulation using a specific set of parameters for each sub-shower. These parameters describe the stack part that will be used for the sub-shower simulation.

Because the client invokes nonblocking asynchronous remote executions, several sub-shower simulations are executed concurrently on the server side. Since the simulation duration depends on the probability laws, it differs for all sub-showers. When a server ends a simulation, it send a signal to the broker that send another parameter set corresponding to another sub-shower simulation.

As previously mentioned, we do not investigate load balancing issues in this paper. So sub-shower placement follows a very simple round-robin scheme, with a first end first serve load managing policy.

The experimental platform uses one node for the client, one node for the broker and 6 nodes for servers.

Table 3 presents the speedup for the simulation of medium size shower. The speedup is computed from the global execution times of the sequential version and the OVM implementation of AIRES. All the computation time is taken into account including the sequential initialization part. To hide the large execution time variance related to Monte-Carlo approach, the execution times for the sequential and OVM versions are computed from 20 executions. In order to feed each server, several non-blocking outstanding requests are sent to each server.

The load balancing strategy uses the following features:

- a) iso energy partitioning of the stack i.e. partitions of the stack have the same energy,
- b) over partitioning of the stack i.e. outstanding requests per server is set to 5,
- c) dynamic load balancing with a first come first serve strategy.

	6 CPUs	12 CPUs
Speedup	5.66	11.31

Table 3: Speedup of OVM version of *Aires* over the sequential version for 6 and 12 servers.

Table 3 shows that OVM version provides sub-linear speed-up due to the sequential initialization part of the program. However, the speed-up figures show that OVM can provide high speed-up for a very irregular real world parallel application which is the main application domain target of OVM. It also shows that OVM and RPC programming style can be used efficiently to program parallel implementation of irregular applications.

Note that performance of OVM implementation of AIRES strongly relies on the OVM RPC performance

6 Related Work

Several project have proposed an optimized version of traditional sunRPC like [3]. Optimizations include reducing the context switching cost, reducing the number of arguments copies and specializing code sections that are most often executed.

High performance communication libraries like Active Messages [4] and Fast Messages [5] use a protocol that work functionally like RPCs. When a message is received, a handler is called to manage the message. The message header encompasses the name of the handler to be invoked upon message reception. When the handler ends the message management, it sends back to the message sender an answer (return of a function call).

Multithread environments like Nexus[6], PM2[7], Athapascan[8] use RPC style communication for communication between threads. RPC is used in these environments for remote write, remote read and for invoking remote executions. These environments are mainly dedicated to irregular parallel applications. They use a thread migration mechanism to implements the load balancing.

Metacomputing environments such as RCS[9], Ninf[10] and Netsolve[11] use RPC paradigm as interface between the client (user) and several high performance computers. Heterogeneity, server selection, load balancing and fault tolerance are managed by a third entity sometimes called "Agent". TCP/IP and XDR protocol of traditional RPC help to handle the heterogeneity issue.

The Manta [12] project also focus on Metacomputing. It uses JAVA for programming parallel applications. The RMI protocol of JAVA is used for communication between parallel tasks.

The RMI protocol stack has been optimized and work on top of a user level high performance communication library. Manta RMI on top of Myrinet reaches a latency of $39.9\mu s$.

7 Concluding remarks

One main feature of OVM is its RPC programming style. In this paper we have addressed the issue of high performance parallel computing using this programming paradigm and for regular as well as irregular applications.

Regular applications testbed is considered as very difficult for an environment designed for irregular applications because all overheads related to dynamic load-balancing are taken into account in the performance results. For this testbed, we have demonstrated that RPC programming style can approach the performance of other parallel programming paradigms (designed for regular applications) for the NAS benchmarks, at least for small number of processors.

Irregular parallel applications are the main targets of OVM. We have presented the parallization approach and the performance of a real world application called AIRES. Althrough the code is very large, we have proposed a simple parallelization scheme based on OVM RPC programming style. Performance evaluation shows nearly optimal speed-up compared to the original sequential version.

These two tests demonstrate the programmation and performance capabilities of OVM RPC programming style.

The next issue for OVM is the scalability. It will rely on a hierarchy of brokers. In principle, the client issues requests to a metabroker that distributes them among several second level brokers. Each second level broker manages a set of servers.

References

[1] Richard M. Adler. Distributed coordination models for client/server computing. *Computer*, 28(4):14–22, April 1995.

[2] L. Prylli and B. Tourancheau. Bip: a new protocol designed for high performance networking on myrinet. In *Workshop on Personal Computers based Networks Of Workstations*, 1998.

[3] M. A. Blumrich, K. Liand R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual memory mapped network interface for the shrimp multicomputer. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 142–153, 1994.

[4] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Actives messages : a mechanism for integrated communication and computation. In *Proceedings of ISCA'92*, pages 256–266, 1992.

[5] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast Messages: Efficient, portable communication for workstation clusters and MPPs. *IEEE Concurrency*, 5(2):60–73, April/ June 1997.

[6] Ian Foster, Carl Kesselman, and Steven Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, August 1996.

[7] R. Namyst and J.-F. Méhaut. PM^2: Parallel multithreaded machine. A computing environment for distributed architectures. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and D. Trystram, editors, *Parallel Computing: State-of-the-Art and Perspectives, Proceedings of the Conference ParCo'95, 19-22 September 1995, Ghent, Belgium*, volume 11 of *Advances in Parallel Computing*, pages 279–285, Amsterdam, February 1996. Elsevier, North-Holland.

[8] J. Briat, I. Ginzburg, M. Pasin, and B. Plateau. Athapascan runtime : Efficiency for irregular problems. *In Proceedings of the Europar'97 Conference*, pages 590–599, 1997.

[9] P. Arbenz, W. Gander, and M. Oettli. Remote computation system. *Parallel Computing*, 23(10):1421–1428, October 1997.

[10] A. Takefusa, U. Nagashima, S. Matsuoka, H. Ogawa, H. Nakada, S. Sekiguchi, H. Takagi, and M. Sato. Multi-client LAN/WAN performance analysis of Ninf: A high performance global computing system. In ACM, editor, *SC'97: High Performance Networking and Computing: Proceedings of the 1997 ACM/IEEE SC97 Conference: November 15–21, 1997, San Jose, California, USA.* ACM Press and IEEE Computer Society Press, 1997.

[11] Henri Casanova and Jack Dongarra. NetSolve: A network-enabled server for solving computational science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.

[12] J. Maassen, R. Nieuwpoort, R. Veldema, H. Bal, and A. Plaat. An efficient implementation of java's remote method invocation. *In ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1999.