OVM: Out-of-order execution parallel virtual machine

George Bosilca*, Gilles Fedak, Franck Cappello

Laboratoire de Recherche en Informatique (UMR 8623), Universite Paris-Sud, 95405 Orsay Cedex, France

Abstract

High performance computing on parallel architectures currently uses different approaches depending on the hardware memory model of the architecture, the abstraction level of the programming environment and the nature of the application. In this article, we introduce an original client–server execution model based on RPCs called out-of-order parallel virtual machine (OVM). OVM aims to provide three main features: portability through a unique memory model, load-balancing using a plug-in support and high performance provided by several optimizations. The main optimizations are: non-blocking RPCs, data-flow management, persistent and non-persistent data, static data set distribution, dynamic scheduling and asynchronous global operations. We present OVM general architecture and demonstrate high performance for regular parallel applications, a parallel application with load balancing needs and a parallel application with real-time constraints. We firstly compare the performance of OVM and MPI for three kernels of the NAS 2.3. Then we illustrate the performance capability of OVM for a large real-life application that needs a load balancing support called AIRES. Finally, we present the performance of a real-time version of the PovRay ray-tracer demonstrating the reactivity of OVM. © 2002 Published by Elsevier Science B.V.

Keywords: RPC; OVM; Beowulf; Cluster

1. Introduction

Today parallel programmers are facing at least three decision parameters when choosing a parallel programming model. The first parameter concerns the fast evolution of parallel architectures. Ten years ago, dominant high performance computers were vector machines. In the nineties, MPP with message passing or shared memory systems became very popular. Today parallel computers are clusters of multiprocessors gathering SMP nodes connected by a high speed network. The computers of the next generation, with IBM Power4, Alpha 21364 processors or Mips r14000, will be clusters of NUMA nodes with a very deep memory hierarchy (four or five levels). The programmer should choose very carefully a programming model that will be efficient and portable across several generations of parallel machines.

The second parameter concerns the abstraction level of the programming model. The programmer has a choice between three major categories. The hardware inspired approaches allow to program the parallel computers using their native memory model: message passing, shared memory or even a mix of them. Some other models use abstractions such as communicating threads, tuple space or communicating objects to provide a programming model portable across a large range of platforms. Finally, languages like HPF [1] offer a higher level of abstraction, allowing the programmer to manipulate applications features (data parallelism) and not architecture features (memory system). Performance is crucial for a parallel architecture and hardware inspired models are often considered as the most efficient way to program...
parallel computers. However, portability may enforce to choose a high level programming model.

The nature of the application is the third parameter. The application could be a regular one with (e.g. dense matrix computation) or without (e.g. sparse matrix computation) predictable behavior. Some applications need to dynamically expand and reduce the number of tasks during the execution. Typical examples of such applications are branch and bound methods. Some other applications require high responsiveness and regular throughput. Depending on the application nature, the programmer may choose an existing programming model or develop his proper execution model on top of an existing programming model. In addition to the time consumption of this later approach, the programmer will still face the problems of performance and portability.

The out-of-order execution parallel virtual machine (OVM) system is an original computing model designed to reach high performance with a client–server architecture. In this paper, we demonstrate that OVM provides high performance on regular and irregular parallel applications with load balancing needs and on a parallel application with some real-time constraints. Moreover, because OVM provide a higher abstraction level than hardware inspired environments, it is portable across a large range of architectures.

Section 2 presents OVM principles, design and basic performance. Section 3 compares it with a traditional programming approach (MPI) for high performance regular applications (NAS NPB 2.3 benchmark). Section 4 presents the parallelization and the performance of a large real-life and regular application with huge load balancing needs called AIRES using OVM. Section 5 describes the performance of a real-time version of the PovRay ray-tracer parallelized with OVM.

2. OVM

2.1. General principles

OVM is a parallel environment based on client–server architecture. It provides: (a) an API for programming applications starting from scratch or from an existing sequential version and (b) a runtime support managing multiple computing resources. The general architecture of OVM is based on two main features: an RPC style programming interface and a build-in dynamic load balancing mechanism. Fig. 1 presents the general organization of OVM parallel execution model.

An OVM program uses three program entities: the client program, a function library loaded by all servers and a scheduling plug-in that will control the load balancing strategy of the broker. The client executes a sequential control program. Currently supported languages are Fortran and C. The programmer inserts OVM RPC calls in the client program. During the execution, when the client reaches an OVM RPC statement, the OVM library requests the RPC execution to the broker. The broker receives the RPC request, selects a server and schedules the RPC request to the server. Because OVM RPCs are non-blocking, the client can launch several outstanding RPCs subsequently. The broker manages these RPCs, launching them on different servers according to the scheduling policy. As a result, several outstanding RPCs are running concurrently on different servers leading to a parallel execution.

2.2. OVM programming approach

OVM programming model for regular applications is close to the SPMD programming style. The programmer has to distribute the data sets among the computational nodes. He also has the responsibility to decompose the global workload in subtasks. The programmer may use global data movements for the communications between the client and the servers and between the servers. There are several significant differences between the OVM programming approach and the traditional SPMD message passing approach. In OVM, the programmer does not manage: (a) the workload distribution among the computational nodes,
(b) the point to point communication between nodes and, (c) the synchronization of computational nodes.

Load balancing is managed by the broker. Point to point communications are initiated by the broker according to the workload management and correspond to subtask migrations from one computational node to another. The synchronization is governed only by the dependencies among operations (computations and communications).

Programmer makes requests to OVM annotating a sequential program with directives. Directive annotations concern data management (creation, movement) between client and server, global operations and identification of the functions to execute remotely. A pre-processor translates the programmer directives into low level API function calls. In the translation process, the pre-processor prepares the requests for the broker. Programmer may also directly use some API functions although this programming level is not recommended because real applications usually require a very large number of API function calls.

As an example, Fig. 2 presents some parts of the NAS2.3 CG benchmark. This code corresponds to the client part of the application. All OVM RPC calls are asynchronous.

This parallel version directly derives from the sequential version of CG. The OVM version of CG starts on the client with some sequential and initialization sections (this part of the code is not shown on the figure). As the first program interaction with OVM, the client requests to create arrays colidx, rowstr, and a on the server side. For this implementation, these arrays are duplicated on all servers. Note that the original MPI implementation of CG also duplicates these arrays on all MPI nodes. The first part of the code also creates params on the server side. This variable will serve for distributing the work among the servers. It contains a parameter set (mainly loop boundaries) which is different on all servers. The client transfers the actual content of colidx, rowstr, and a using the OVM broadcast global operation. For params, the client makes several individual transfers using the put operation. This part of the code is not shown on the figure. CG continues on the client with a sequential section. Then the code reaches the main iteration block. This block calls the conjugate gradient subroutine and contains some normalization operations. This code section is also executed on the client. The next part of the code shows the conjugate gradient subroutine. Some parts of this subroutine stay executed on the client such as the loop with label 101. The next part is the main computational loop nest. This is actually the only part of the whole CG program that uses OVM RPCs. The loop nest itself is not shown on the figure. It is a part of the server function library. The P array is the only variable which is modified by the loop nest. This is also the only variable actually distributed among the servers. So it is broadcasted and gathered, respectively, before and after each iteration of the loop with label 111. The OVM preprocessor provides OVM loops.

```
CC ovm req{ CREATE, colidx, rowstr, a, params, ...}
CC ovm req{ BROADCAST, sizedata(colidx_id), colidx, (XX=1, NB_SERV; COLIDX XX)}
CC ovm req{ BROADCAST, sizedata(rowstr_id), rowstr, (XX=1, NB_SERV; ROWSTR XX)}
CC ovm req{ BROADCAST, sizedata(a_id), a, (XX=1, NB_SERV; A XX)}

............
  do 40 it = 1, niter
    call conj_grad( colidx, ..., rnorm )
    do 41 j=1, lastcol - firstcol+1
      norm_temp1(1) = norm_temp1(1) + x[j]*z(j)
  
  subroutine conj_grad ( ... )
    do 111 cgit = 1, cgitmax
      ovm req{ BROADCAST, size(p_id), p, (XX=1, NB_SERV; P XX)}
      ovm req{ SERV, SERVICES.CG, PARAMS_XX, COLIDX XX, ROWSTR XX, A_XX, P_XX }
      ovm req{ END_FOR }
      ovm req{ GATHER, size(q_it), q, (XX=1, NB_SERV; P_XX) }
    do 141 j=1, lastcol - firstcol+1
      sum = sum + p[j]*q[j]

............
```

Fig. 2. Sketch of the client part of CG NAS benchmark parallelized with OVM.
translates the OVM FOR loop in several C and OVM API statements in order to construct an RPC launching loop. The name of the service (function) called on the server side is \textit{SERVICE}\_\textit{CG}. In this part of the code, several outstanding non-blocking RPCs will be sent from the client to the broker. The broker will forward the requests to different servers leading to a concurrent execution of the RPCs. Fig. 3 presents the server part code of \textit{CG}.

The code corresponding to the server part of \textit{CG} is nearly the same as the main computational loop nest of the original sequential version. The \textit{params} variable implements the loop boundaries. OVM implementation of \textit{CG} demonstrates that the program is close to the sequential version except for the parallel part. For this part only some simple modifications are applied to the original code. As for SPMD programming style, the programmer should distribute data and works among the processors. However, he does not deal with point to point communications and he only cares about the global operations on distributed data sets.

The programmer has the responsibility to select the parts of his program to execute remotely.

2.3. OVM high performance RPC

Performance of OVM relies on several optimization principles. Some of them are related to software architecture design and communication support. The other are more related to an extension of the RPC programming style.

2.3.1. Reduced RPC critical path

Performance of OVM programs leans on the capability of the system to launch remote execution with a minimum latency because all potential concurrent executions are launched sequentially from the client. As for communication libraries, we define a critical path (Fig. 4) between the client and the server. The client, broker and server software architectures have been designed to minimize the critical path.

As previously mentioned, a pre-processor translates the OVM directives in the client code into C statements and OVM API calls. There is no buffer copy between client code and API functions. The API functions directly call the communication library functions. There is no scheduling management on the client side. The client sends synchronous or asynchronous requests

---

**Fig. 3.** Server part of the CG program.

**Fig. 4.** General architecture and critical path for remote executions.
and waits for the result. The result is provided by the broker. The basic roles of the broker are to schedule the RPC requests on the servers, manage the server workloads and to order data allocation, deallocation, reutilization and movements on the server side. Since scheduling and workload management may require some substantial computations, the broker architecture uses a hierarchical design with two levels. The first selection level is dedicated to fast execution. So this level only uses fast scheduling algorithms (simple) for broker selection (short term decisions). Basically, the broker checks all arguments and launches the execution to a server if all arguments are available on this server. If any argument is missing on the server or is not available due to data dependences, the request is stored in a list of pending requests. The second level deals with the execution order of the pending requests and the server workload. The long term scheduler may decide to migrate partially or totally the workload of a server to one or several other servers.

The servers software architecture uses a multi-threading approach (Fig. 4). This architecture uses a classical approach where a thread listens to requests while a pool 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. The programmer may add some prefetch annotations (or API function calls) in the client program to order libraries prefetch on the server side.

2.3.2. Very fast communications

Of course round-trip latency for RPCs not only depends on the architecture of the broker and the server but also on the communication support. To achieve very fast RPCs, OVM relies on high performance SANs or LANs (Myrinet, SCI, Gigabit Ethernet, Servernet, Gigabit, Quadrix) and user level protocols (GM, PM, Active messages, Fast Messages, VIA, BIP).

2.3.3. Non-blocking RPCs

OVM provides blocking and non-blocking RPCs. Non-blocking RPC is the cornerstone to provide concurrency for RPC programming style. Typically, the client loops on non-blocking RPC calls to launch a large set of concurrent remote executions. The client

2.3.4. Decoupled RPCs and persistent data

OVM provides two ways for communicating arguments to RPCs. The first way is similar to traditional RPCs: arguments are passed to the remote function by values and the data are sent from the client to the server (crossing the broker) within the RPC message. This approach of argument communication fits the cases where: (1) the size of the arguments is small (less than a threshold depending on the network performance) or (2) transferred data could not be reused by the server for subsequent RPCs. When the arguments are large and when the arguments can be reused by the server, it is more profitable to store them at the server side. This feature corresponds to the notion of persistent data on the server side. RPCs using persistent data on server side are called decoupled RPCs. In decoupled RPCs, arguments are passed by reference. Values of the argument must already exist at the server side before the RPC call. OVM provides five main operations related to persistent data on server side (move data from client to server and vice versa): create, kill, put, get and move. The two first operations create (respectively kill) a persistent variable on the server side and return references that may be used to store and read data sets by the client remote function calls. put operation writes values to existing persistent 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 persistent variable between two servers.

Fig. 5 presents the protocols for put and get operations. The numbers on the figure describe the
messages order. The values are effectively transferred on bold lines. The broker manages the create, kill, put, get and move operations. When the broker selects different servers for data-dependent functions, it has the responsibility to move persistent data sets from producing servers to consuming ones. Client is not aware of the data transfers issued by the broker.

Decoupled RPC is a main issue toward high performance with RPC programming style because the execution a real application leads to a sequence of remote function calls that may exhibit data dependencies and locality (read after write, read after read). Persistent data allows to avoid multiple exchange transfers between a client and a server when consecutive functions calls have such locality properties.

2.3.5. Data-flow execution

During the execution, a data-flow graph is built from the RPCs requests. In a typical client sequence, the client sends several asynchronous RPC requests to the broker. The broker peruses these requests to check that arguments of the RPC are ready on the server side. If one of the arguments is not ready, the broker stores the request in the pending list. When a server ends a service, the result can be transferred to the client, and it informs the broker of all the arguments that have been modified. The broker updates the pending list of delayed requests. Requests ready to launch are sent by the broker to servers.

In principle, the data-flow mechanism and implementation of OVM are very close to the instruction management of an out-of-order super-scalar microprocessor.

2.3.6. Static data distribution and dynamic scheduling

OVM provides two ways for controlling task distribution among the server according to task complexity. We define task complexity as a two parameters criterion. The first parameter concerns task duration. The second parameter concern the size of the task arguments. The programmer is responsible to select the appropriate control approach according to his program.

For coarse grain tasks and large data sets, it often would be preferable to use OVM features for static distribution: (a) persistent data and (b) task allocation on server that is driven by data distribution. The programmer decomposes a large data set into several subsets and allocates them on server side using OVM persistent data operations (create and put). The broker will distribute them on the servers using a round-robin scheme. When the client launches RPCs, it describes on which subset the function should be applied. The broker automatically selects the server that holds the required subset.

For fine grain tasks and small argument size, it would be preferable to let the broker selects itself the server. Using persistent data would prevent this "on-line" server selection. So dynamic scheduling relies on non-persistent data and dynamic server allocation by the broker. The programmer uses RPCs that encapsulate the argument in the message. When the broker receives such RPCs, it selects a server based on its workload without checking for the data dependencies. The fine grain scheduling is based on a first come first serve approach. The two control approaches can be used individually, subsequently or simultaneously inside the same program.

2.3.7. Asynchronous global operations

OVM provides a set of global operations on persistent data. These operations collect data that are spread among servers (gather), distribute data among servers (scatter), compute global result from distributed data (reduce), copy data from the client to several servers (broadcast) and exchange data among servers (alltoall).

These operations are executed like asynchronous RPCs. They are launched by the client and managed by the broker. If there are several outstanding global operations, they are scheduled accordingly to the data dependencies and server workload. So the termination order of several consecutive global operations may not be related to the launch order. Moreover, the synchronous progress of the sub-operations related to a global operation is not guaranteed.

Global operations work within a communication group. Before launching a global operation, the client describes the data set to use for a global operation. The broker sets-up a communication group including all servers that hold a part of the data set. The client may request several global communications. Since each data set may be spread among a subset of the servers: (1) not all servers may belong to one
communication group and (2) a server can belong to one or several communication groups at the same time. If several global operations are launched concurrently, the broker manages the communication groups and makes global operations progress concurrently.

3. Performance of basic OVM operations

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

3.1. Experimentation platform

All performance measurements presented in this paper use the same parallel experimentation platform. The platform connects 66 $\times$ 200 MHz Pentium Pro nodes by a Myrinet network. 64 nodes are used as servers (64 processors), the two other nodes are the client and the broker.

The software environment includes Linux 2.2.17, Score 3.2 [2] and the PGI compilers. Score/PM raw performances on Myrinet is a latency of 5 $\mu$s and a bandwidth of 1 Gbit/s.

Table 1 shows the client request latencies. Since the requests return immediately after executing a void function, the remote execution overhead equals the request latency.

3.2. Remote execution overhead

Table 1 shows the client request latencies. Since the requests return immediately after executing a void function, the remote execution overhead equals the request latency.

3.3. Performance of remote global operations

Global operations only concern server sides. They correspond to MPI global operations (they have the same semantics).

Fig. 6 presents the performance of the two versions (OVM and MPI) of the all-to-all data exchange on the server side for 4, 16 and 64 processors. Globally, OVM global operations have a higher latency than the MPI ones but a similar bandwidth for messages longer than 1024 bytes.

OVM implementation of global operations is customized to achieve best performance. MPI implemen-

![Fig. 6. Performance of the all-to-all global operations for OVM (server side) and MPI.](image)
tation corresponds to the optimized MPICH version
developed at RWCP [2]. Although global operations
are implemented on top of point to point communica-
tions for the two versions (OVM and MPI), the algo-
rithms used for the two versions may be different.

4. OVM versus MPI on NAS Benchmarks

To measure the performance level of OVM for reg-
ular applications, we compare it against MPI for three
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 optimize nei-
ther the MPI nor the OVM versions according to mem-
ory hierarchy. These two implementations could be
considered as using the same optimization level.

FT, CG and EP exhibit different computa-
tion/communication ratios as well as different com-
munication 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. Table 2 compares the execu-
tion time according to the number of nodes for OVM
and MPI implementations of FT, CG and EP (Class A).

Table 2 demonstrates that OVM versions can ap-
proach the performance of MPI versions of the NAS
benchmarks EP and FT. A different array distribution
leading to a poor cache utilization is the main rea-
son behind the lower performance of CG with OVM.
These results show that the overhead of the built-in
load balancing mechanism within OVM is negligible
up to 32 nodes.

5. OVM for a real-life irregular application

The Pierre Auger Observatory project is an inter-
national effort to study high energy cosmic rays. Two
giant detector arrays, each covering 3000 km², will be
constructed in the northern and southern hemispheres.

Associated to this observatory is the air shower ex-
tended simulations (AIRES) application. AIRES sim-
ulates 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 Fortran 77 routines, adding up to more
than 80,000 source lines.

Basically, the application manages a stack of parti-
cles for which it computes the incidence on the atmo-
spheric particles that are closer to earth surface. The
simulation proceeds step by step following an itera-
tion process. When the high energy cosmic ray enter
the atmosphere it collides with a first set of particles
that are stored in the stack.

Each of the collided particles will collide with other
atmospheric particles. This collision generation is pro-
cessed iteratively until the end of the simulation. There
is no collision between particles belonging to differ-
ent sub-showers. So, the global shower can be viewed
as a hierarchy of independent showers.

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, the number of particles inside
the stack evolves during the execution, decreasing or
increasing in each iteration. The Fig. 7 presents the
evolution of the stack during a typical simulation.

The OVM implementation of AIRES aims to pro-
vide an interactive use of AIRES. So the OVM im-
plementation must speed-up the computation of an air
shower. Another approach should be to launch several
independent air shower simulations on different nodes.

This will improve the throughput but not the response
time of one air shower simulation. We decompose the
application in two codes: the client program and the
server library. The AIRES parallel execution follows
tree steps: initialization, computation, result gathering.

During the initialization, the client starts the simu-
lation 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 con-
text mainly consists of the client particle stack when it has stopped the simulation. We call this part of the execution the sequential part. Note that this part includes the reading of the parameter file and the writing of intermediary computations (check-pointing).

During the simulation, the client invokes a 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 non-blocking asynchronous remote executions, several sub-shower simulations are executed concurrently on the server side. Since the simulation duration depends on probability laws, it differs for all sub-showers. When a server ends a simulation, it sends a signal to the broker that sends another parameter set corresponding to another sub-shower simulation. We call this part of the computation the parallel part. Sub-shower placement follows a very simple round-robin scheme, with a first end first serve load managing policy.

Fig. 8 presents the speed-up according to the number of servers for the simulation of a medium size shower. The speed-up is computed from the global execution times of the sequential version and the OVM implementation of AİRES. The figure presents the speed-up for: (1) the complete application including the sequential part and (2) the application without the sequential part. One should notice that the sequential part is not relevant because it encompasses initialization and check-pointing. 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 them.

The load balancing strategy uses the following features: (a) an iso energy partitioning of the stack (i.e. the partitions of the stack have the same energy), (b) an over partitioning of the stack (i.e. the outstanding requests per server is set to 8) and (c) a dynamic load balancing algorithm with a first come first serve strategy.

Fig. 8 shows that the OVM version provides nearly a linear speed-up on the parallel part for a very irregular real world parallel application. This performance strongly relies on the OVM high performance RPC, dynamic scheduling and global operations.

6. Real-time ray tracing with OVM

The Persistence Of Vision Ray-Tracer (PovRay) [3] is an open-source software which provides high quality 3D image synthesis through ray tracing. Basically, PovRay consists in a scene description language and a ray tracing engine. In the original version of PovRay, a numerical sequential program computes the ray tracing. PVMPov and PovMPI are two parallel versions
of the original PovRay program. They mostly allow
to compute larger images or images with greater com-
plexity. The OVM implementation of PovRay is int-
tended to provide real-time image synthesis. So the
aim is to speed-up the computation compared to a
uniprocessor considering a constant size and a constant
complexity problem (scene). To test the reactiveness
and throughput of the OVM implementation, we use a
simple scheme to generate several and periodic synthe-
sized images of the same scene. The viewpoint simply
rotates around the scene with a predefined shift angle.

In principle, the parallelization of the image synthe-
sis relies on distributing the computation of the image
pixel among the computing nodes. Some mechanisms
should be used to balance the load of the nodes be-
cause: (1) the computation time for each pixel (or set
of) is not necessarily the same, (2) the total computa-
tion time should be reduced as much as possible. Two
approaches could be used to load balance the execu-
tions: static and dynamic tasks allocation.

One of our aims was to adopt a very simple strat-
agy to parallelize PovRay using OVM. So the image
is split into blocks, the mapping of the blocks among
the processor uses a round-robin scheme and the load
balancing mechanisms uses a static approach.

The first implementation issue when parallelizing
an application with OVM is to split the application in
two parts: the client and the services (functions to call
on server side).

In our parallelization, the client starts with the ini-
tialization and then asks to generate several images
(the viewpoint turning around the scene). The servers
compute the image in parallel, each of them having
the responsibility of a part of the image. The client
gathers the image and possibly displays it on a screen.

The performance evaluation concerns two param-
eters: the reactiveness and the throughput. Fig. 9
presents the computation time for an image according
to the number of nodes and the image size. Large im-
ages should be considered as benchmarks. Since they
require some substantial computations, the overhead
of the client and servers coordination could be con-
sidered as negligible. When the image size decreases,
the overhead takes a greater part in the total execution
time.

The Fig. 9 shows that the execution time decreases
with the image size at the same rate. This test demon-
strates the ability of this PovRay implementation
to speed-up the computation of small images. The
speed-up for the four images is nearly linear accord-
The actual speed-up is 29 for 32 processors for all scenes. Note that some scenes like galleon may not scale as well due to some features (a lot of textured triangles) that do not match with the static load balancing algorithm used in our implementation. The main features of OVM for real-time ray tracing are persistent data at the server side, global operations and data-flow analysis. Globally, these results demonstrate the scalability of the OVM implementation of PovRay and its ability to sustain periodic rendering.

7. Related works

The RPC programming model has been used, since its introduction in 1984, mainly in the context of client–server systems across LAN, MAN and WAN. Most of the current extensions of the RPC model use object management mechanisms and object-oriented programming style (Remote Method Invocation in Java, CORBA RPC). Few experiments have been done in the context of high performance computing with the original RPC programming style. Several research projects like Active messages [4], SHRIMP [5] and Fast Messages [6] provide either a fast communication system with an RPC like API or a performance optimized implementation of traditional RPC. None of them address the issue of RPC programming style performance in the context of high performance parallel computing.

Problem solving environments such as Netsolve [7] and Ninf [8] provide a support for remote execution of library functions. The user calls the remote execution of a sequential or parallel computation through a programming language or an interactive environment like Matlab or Scilab. These environments are typically used to call large computations. These environments are worthwhile to use for remote executions with a duration of several seconds on a LAN and several minutes.
8. Concluding remarks

In this paper, we have addressed the issue of high performance parallel computing using OVM (an

RPC-based execution environment) for regular, irregular parallel applications and parallel applications

with some real-time constraints.

The regular parallel applications are considered as very difficult tests for an environment addressing irregular applications because the performance results
takes into account all overheads related to the dynamic load-balancing mechanism. In this context, we have demonstrated that OVM can approach the performance
of other parallel programming paradigms (designed for regular applications) for three kernels of the NAS benchmarks.

The irregular parallel applications are other targets of OVM. We have presented the parallelization approach and the performance of a real world application called ARIES. Although the code is very large, we have proposed a simple parallelization scheme based on the OVM RPC programming style. The performance
evaluation shows nearly an optimal speed-up compared to the original sequential version.

The last performance evaluation has concerned a real-time version of the PovRay ray-tracer. A parallel real-time version of PovRay need to face two problems: the real-time demand (frames per second) and a linear speed-up according to the number of processors. Our implementation of PovRay with OVM match these two severe constraints.

The next issues for OVM are: (1) the performance evaluation of a large range of platforms from SMP machines to clusters of SMPs, (2) the scalability evaluation and improvement of the OVM runtime for large parallel machines and (3) study of scheduling strategies used by the broker to improve data locality.

Acknowledgements

Our thanks to Olivier Richard for his participation to OVM. Most of the results presented in this paper have been obtained at the Distributed System Software Laboratory of the RWCP. We wish to deeply thank Prof. Ishikawa, A. Hori, H. Tezuka for their help and support.

References

George Bosilca is a PhD student in computer science from Paris-Sud University, Orsay, France since 1999. His PhD work focuses on quality of services and parallel programming models on distributed architectures. His research interests also include security on cluster computing and remote idle computers over Internet.

Gilles Fedak is a PhD student in computer science from Paris-Sud University, Orsay, France since 1999. His PhD work focuses on high performances computing using remote idle computers over Internet. He is the main contributor to the XtreMWeb project which is a peer-to-peer global computing system. www.xtremsweb.net.

Franck Cappello is a CNRS researcher leading the Cluster and GRID Group at LRI (Computer Research Laboratory) in Paris-Sud University, Orsay, France. His research interests include parallel programming models, parallel runtime environments and performance evaluation of clusters, clusters of multiprocessors and large scale distributed systems. He is working on hybrid programming models (MPI + OpenMP), multi-purpose runtime environments for parallel applications (OVM) and peer-to-peer global computing using federations of idle computers over Internet (XtreMWeb).