Schooner: An Object-Oriented Run-time Support for Distributed Applications

Nathalie Furmento Françoise Baude

Research rapport I3S RR95-50

Project SLOOP: I3S - CNRS (URA 1376) - UNSA - INRIA
INRIA Centre de Sophia Antipolis
2004, route des Lucioles, BP 93
06902 Valbonne Cedex, France
{Nathalie.Furmento,Francoise.Baude}@unice.fr

Abstract

This article presents Schooner *(which stands for* **S***loop* **C***ommunication Library for* **H***igh-level,* **O***bject-***O***riented, a***N***d* **E***fficient* **R***emote Exchanges) which is a library of classes used for writing parallel and distributed applications in C++. This library consists in an object-oriented run-time support. Its main purposes are:*

- *to be independent of the used communication protocol,*
- *to give the user a structured and abstract view of its application (farm of available computers, set of computation nodes),*
- *to allow a straightforward redefinition of its behaviours. Advantages using of an object-oriented language are mainly modularity and reusability: for example, it is straightforward to add a new message type or to change the application scalability.*

Key-words: *Parallelism, Object paradigm, Run-time environments, Distribution, Portability.*

Contents

1 Introduction

Distributed concurrent computing has established itself as a mainstream technology over the past few years. This mode of high-performance computing is based on heterogeneous networked platforms, frequently comprising powerful workstations, and on software systems that emulate general purpose virtual parallel computers.

The SCHOONER library presented here, which is an object-oriented run-time support, is such a system to help the development of distributed concurrent applications.

In this introduction, we will first review some of the most representative specimen among the plethora of run-time supports whose aims are to manage the underlying machine and network resources and to present a usable programming interface and model to applications. This review and corresponding criticisms will be helpful to explain what are the original purposes of SCHOONER.

1.1 Analysis of the current situation

1.1.1 Existing run-time supports

A de facto standard to support distributed concurrent applications is the PVM library [Sun90]. Its main functions are to help building a parallel virtual machine comprising nodes which are indeed heavyweight (usually Unix) processes, and to manage message-passing among them, usually through interfacing the TCP/IP and UDP/IP protocols. The use of such standard protocols and operating system, available on a wide range of platforms is certainly one of the reasons of the PVM success.

But, the PVM library is not re-entrant. So, its use favours a one-to-one matching between one "physical" node and one communicating computation entity of the application, although a lot of them would require a finer granularity.

So, many libraries enabling to support fine grain, so multithreaded, distributed concurrent applications have been developed. Many of them like for example TPVM [FS95], PT-PVM [KAH95], or PM2 [NM95] can be viewed in fact as multithreaded PVM extensions in the following sense : each PVM node can itself sustain more than one entity of the application thanks to the interfacing of a thread library and the use of a lock/unlock operation each time a thread enters or leaves a PVM library function. Nexus [FKT96], an other run-time support for lightweight distributed applications can be distinguished from the preceding examples in the sense that other libraries than PVM can be used in order to exchange data between heavyweight entities hosting threads[1].

The interfaced thread library is generally in compliance with the POSIX threads standard, and can either be "home made" (like Marcel used in PM2, which provides non-POSIX facilities like thread migration) or more popular (Quickthreads [Kep93], REX [Cra93] for example).

A common characteristic of those "distributed-thread packages"[2] is that they have fixed the scheduling policy. For example, the scheduler implemented in Marcel is priority and time-sharing based (so preemptive). The one implemented in TPVM is non-preemptive, and in the message-passing computation style of TPVM a thread automatically yields control when it has no messages to receive. So the use of one or another "distributed-thread package" can preclude some uses due to the implemented scheduling policy.

On the contrary, Nexus is remarkable in the sense that it does not stand to be a portable "distributed-thread package". In fact, a kind of portability is achieved in the preceding examples because the used thread library is POSIX standard and the used communication protocol could be something different than PVM (MPI [Mes94] for example). Indeed, Nexus is a *toolkit* for developing multithreaded distributed applications. It means that its design is such that different scheduling policies can be incorporated, usually one or another policy being chosen depending on the target operating system and application. For example, if the target application is developed

[1] Note that this distinction does not concern the provided programming model for the developer, which will be discussed later on.

[2] We call them "distributed-thread packages" to avoid confusion with the underlying used thread libraries.

in C++, then if the available C++ library is not reentrant (like for SUN-OS), a non-preemptive scheduling strategy would be recommended. As such, once the choice to use Nexus has been done, the develop-per is insured to be able to use the same interface (Nexus library) for communication and lightweight process management programming, whatever the specificities of the target applications be.

In this sense, SCHOONER is comparable with Nexus because it captures and implements only the common concepts needed to support any distributed concurrent application (see Section 1.1.3) while leaving unspecified what concerns the programming of more specific needs (for example, programming of an "exotic" scheduling policy for distributed discrete-event simulation environments, modification of the order data are marshalled and send if one wants to overlap computation and communication phases, ...).

1.1.2 User provided computation model

Among the various "distributed-thread packages" or toolkits (like Nexus) which integrate multithreading and communications, two main and quite different programming interfaces and models are provided to the user.

The first, examplified by PT-PVM, partly TPVM, is a traditional communicating entities model grounded upon asynchronous message-passing (send/receive) whose semantic is equivalent to the one of PVM. In this sense, these examples are really multithreaded PVM extensions.

The second, examplified by Athapascan-0 [CRG95], PM2, Nexus, is a general asynchronous RPC computation and programming model. Thanks to the definition of remote service entry points, a distributed computation consists in launching remote service calls. Each call is taken in charge by a usually newly created thread on a local or remote heavyweight process (which is able to manage many threads).

It is today not clear which of these two computation models is the most general. Thus, SCHOONER does not impose one or another model and on the contrary, both of them can be defined at the highest level of SCHOONER.

1.1.3 Basic mechanisms

Definition of many different computation models is possible because the main purpose of SCHOONER is to gather all common basic concepts needful to support concurrent distributed applications development and execution on either homogeneous or heterogeneous virtual computers.

The definition of this set of concepts arises from the following observations. At run-time, computation entities of the application need at least some CPU and memory resources, which are always provided by a heavyweight process. These processes form the core of the parallel virtual machine, so one has to be able to define their locations. These processes usually do not share memory so whatever the provided computation model be (RPC or traditional message passing), at least a basic asynchronous send/receive mechanism is needed to exchange data. Because remote data exchanges have to pass the process frontiers, data have to be marshalled and later unmarshalled, and an encoding scheme must be applied whenever the implied hosts do not code primitive data types in the same way.

Notice that threads related concepts do not belong to this basic set because they are not ever provided, like for example in PVM.

1.1.4 Object-oriented approach

The SCHOONER library comprises classes whose aim is to provide to higher levels an interface to this minimal set of basic concepts. But thanks to the use of the object-oriented technology, a developer will be able to easily derive from this minimal set a more sophisticated behaviour, even if the used underlying package is not designed to behave like this. One typical example is the extension of the send/receive basic mechanism in such a way that each incoming or outgoing message is counted (this is needful for distributed discrete-event simulation tools for instance).

In fact, SCHOONER is more than a C++ encapsulation of one or another available communication and distributed computation support libraries. Such libraries are usually a set of C functions which so can be directly called by C or C++ programs. Numerous C++ encapsulations have been developed in order to ease and simplify the call of the C functions. For example, taking advantage of the genericity provided by high-level languages such as C++ or Ada simplifies the marshalling/unmarshalling phases. Para++ [CD95], also an ADA to PVM interface [BFdM94], are examples of such encapsulations of the PVM C library. Nevertheless, the advantages obtained using these encapsulations generally stand at the syntactic level only. Indeed, as opposed to SCHOONER, these encapsulations do not necessarily provide a hierarchy of classes, organizing the various basic concepts which gives the user a more structured and abstract view of its application and an easy way of using these basic concepts to define more complex functionalities.

1.2 Purposes of SCHOONER

SCHOONER is an object-oriented run-time support for C++ concurrent distributed applications.

Its low level defines the minimal set of concepts that are useful for either coarse or fine grained concurrent applications development. Thanks to inheritance mechanisms, this set can straightforwardly be redefined and gives the user a structured and abstract view of its application support (farm of available computers, set of computation entities supports, ...).

Moreover, this set of concepts can be implemented using the appropriate subset of any of the C libraries for communication and distributed computation supports, because in each of them, these concepts are defined (like in Nexus) or even implemented (like for example in TPVM, PM2). Using PVM for example, we would use the `pvm_addhost()`, `pvm_spawn()`, `pvm_send()`, `pvm_receive()` functions. Using PM2 for example, we would use the `pm2_start_modules()` function, and define an efficient remote service (using a `QUICK_ASYNC_LRPC`) in order to send/receive any data on a module.

Depending on the user's needs, SCHOONER can also provides various multithreaded high-level computation models whose semantic could follow either a traditional message-passing model or an RPC computation model. In this way, SCHOONER can for example be implemented using a subset of the functions of a specific "distributed-thread package" and at the same time provides to application developers a true object-oriented encapsulation of the whole specific used package!

1.3 Plan

The rest of this paper is organized as follows. Section 2 describes the library model and its design choices. Section 3 gives an overview of implementation problems, and Section 4 presents two examples of SCHOONER use.

2 Presentation of SCHOONER

2.1 The programming model

The SCHOONER library can be split into three layers (see Figure 1) :

1. the first layer is an interface to any communication protocol;
2. the second layer allows the user to create computation nodes also called clusters; these clusters can interact thanks to active messages (see Section 2.2);
3. a third layer supports communicating objects which can also interact via messages (see Section 2.3). This layer could be replaced by a RPC-like model, if needed.

Figure 2 gives a run-time example of an application built upon SCHOONER.

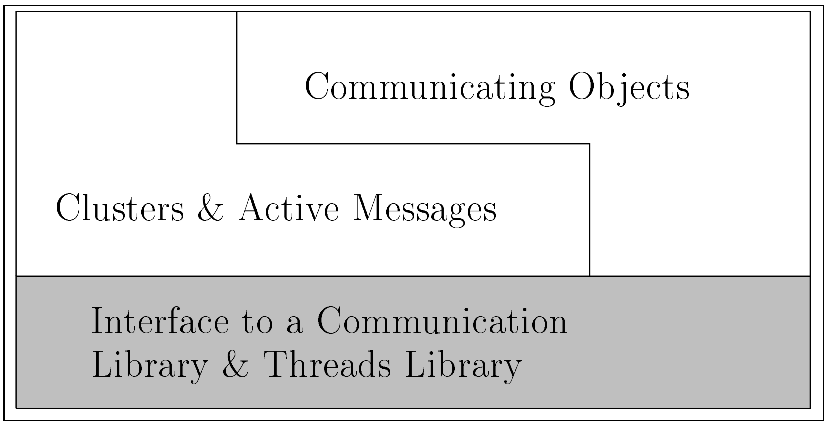

Figure 1: The differents layers of SCHOONER

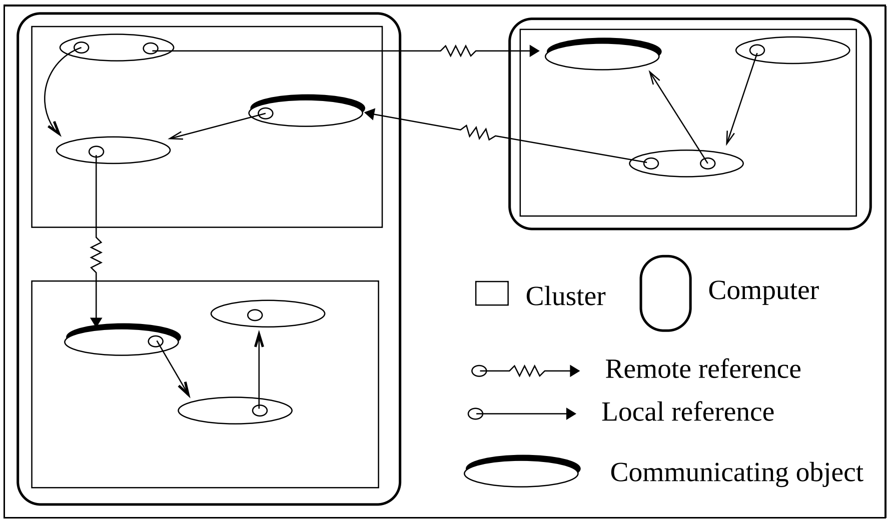

Figure 2: Example of run-time environment

2.2 Clusters and Active Messages

A cluster can be thought of as an entity which designs a context (memory, CPU, ...) on a computer. We can also find this notion in other libraries: environment process of PT-PVM, context in Nexus, modules in PM2.

In most cases, a cluster will be mapped on one Unix process, but we can also choose to group several or all clusters of the application on a same Unix process. In this way, it is very straightforward to change the scalability of the application.

2.2.1 Definition of clusters

A cluster is defined by the `Cluster` class (see Figure 3). An identifier of cluster (`Cluster_Id` class) has a global meaning in the entire application.

The member function `destroy()` is called to destroy the effective cluster entity (which can be remote), whereas the function ~`Cluster()` destroys the C++ object (whose aim was to "trigger" the cluster creation, see Figure 5).

```
class Cluster {
   protected:
      Cluster_Id *id;                              // cluster identifier
   public:
      static Cluster *any();                       // any cluster
      static Cluster *best();                      // best cluster
      static Cluster *current();                   // current cluster

      bool destroy();                              // destroy cluster
      void *operator new(size_t, Computer *);      // allocator
      Cluster();                                   // constructor
      ~Cluster();                                  // destructor
}
```

Figure 3: Partial definition of the `Cluster` class

2.2.2 Creation of clusters

The location for a new `Cluster` object is given thanks to an instance of the `Computer` class (see Figure 4). This class supports various selection functions such as: a specific computer, the "best" computer, ... An instance of this class is defined by three informations: a symbolic name, the real name of the computer (network name, IP address) and its architecture.

Many computers with different symbolic names can be associated to the same physical machine. The use of symbolic name allows to define a virtual architecture which can be mapped on different configurations. We can also have a configuration file which will automatically be used at run-time to instantiate computers.

```
class Computer {
   protected:
      String symbolic_name;                    // symbolic name
      String real_name;                        // real name
      Computer_Arch arch;                      // computer architecture
   public:
      Computer(String&, String&);              // constructor

      static Computer *any();                  // any computer
      static Computer *best();                 // the "best" computer
      static Computer *current();              // current computer
      static Computer *specific(String &);     // the computer with the
                                               // given symbolic name
}
```

Figure 4: Partial definition of the `Computer` class

Figure 5 shows examples of creation of `Cluster` objects. There are two ways to give location of a new cluster:

1. The programmer can specify to the allocator the target computer either by giving an instance of the `Computer` class, or by giving the symbolic name of an existing `Computer`.

2. By default, the cluster will be mapped on the "best" computer which is selected at run-time according to a load balancing scheme.

```
Computer *cp = new Computer("mach", "lorien.m_earth.fr");
Cluster *cl1, *cl2, *cl3;
cl1 = new (Computer::current()) Cluster();            // on current computer
cl3 = new ("mach") Cluster();                         // on specific computer
cl2 = new Cluster();                                  // on default computer
```

Figure 5: Creation of clusters

2.2.3 The "best" clusters and computers

The policies used to choose the "best" computer or the "best" cluster are normally the ones provided by the underlying used communication package (PVM, PM2, ...). They could be redefined as follows : the "best" computer could be the less-loaded one (memory, CPU, ...), and the "best" cluster could be the one which contains the fewest communicating objects.

We can enhance this policy by adding a parameter to the two functions `Cluster::best()` and `Computer::best()`. In this way, SCHOONER enables to easily adapt the selection criteria (memory load, computation power, ...) depending on the target application.

In our future works, we are also interested to study techniques and heuristics to have an automatic and implicit mapping for different SCHOONER objects.

2.2.4 Active messages

The model for inter-cluster communications is based on an active message mechanism. Active messages [vECGS92] allows to extend the *message-passing model* to a *single-sided operation*. Once a message is received by a cluster, a processing function dependent of the received message type is automatically triggered. Furthermore, the call to the reception function is implicit: the destination object does not need to call the reception function in order to get the newly available messages.

This active message mechanism implements the basic asynchronous send/receive mechanism described in Section 1.1.3.

This mechanism will also be especially useful to support the implementation of the communication protocols of the C++ // language (see Section 4.2).

An active message object is an instance of a class which publicly inherits (directly or indirectly) from the `Message` class (see Figure 6). This class defines the following member functions:

1. `flat(...)` which is used to flatten an object `Message` into an object `Buffer` in order to send it via the communication network,

2. `rebuild(...)` which is used to build an object `Message` from an object `Buffer`,

3. `process(...)` which specifies the processing to perform when an object of class `Message` is received by a cluster.

```
class Message {
   public:
      virtual Buffer *flat(Buffer *buff=0)=0;
      static Message *rebuild(Buffer *buff, Message *msg=0);
      virtual void process()=0;
}
```

Figure 6: Partial definition of the `Message` class

The definition of a new class of active messages is very easy: the user must (1) define a new class which inherits from the `Message` class, with the associated data, (2) write the two marshalling functions (`flat()` and `rebuild()`), and (3) write the `process()` function.

2.3 Communicating Objects

A possible third layer of SCHOONER offers a communicating objects model. An application built upon this layer can be seen as a set of communicating objects based on a traditional message-passing model, and we can have 1 or n communicating objects mapped on a cluster object.

2.3.1 Definition and creation of communicating objects

Creation of a communicating object is done through creation of a proxy object. A proxy is a local representant of the communicating object which is on another (local or remote) cluster. The proxy allows to send data to the communicating object.

A communicating object is defined by two classes: the `Proxy_Comm_Object` class (see Figure 7) and the `Comm_Object` class (see Figure 8). When a cluster wants to create a communicating object, in fact, it has to create an instance of the `Proxy_Comm_Object` class. An instance of the `Comm_Object` class is then automatically created on the cluster object given in parameter to the allocator of the `Proxy_Comm_Object` class.

```
class Proxy_Comm_Object {
  protected:
    Comm_Object_Id *id;                       // object identifier
    Cluster *cl;                              // object location
  public:
    virtual void send(Data_Comm_Object *);    // send function

    void *operator new(size_t, Cluster *);    // allocator
    Proxy_Comm_Object();                      // constructor
    virtual ~Proxy_Comm_Object();             // destructor
}
```

Figure 7: Partial definition of the `Proxy_Comm_Object` class

```
class Comm_Object {
  protected:
    Comm_Object_Id *id;                          // object identifier
  public:
    virtual void receive(Data_Comm_Object *);    // entry point

    Comm_Object();                               // constructor
    virtual ~Comm_Object();                      // destructor
}
```

Figure 8: Partial definition of the `Comm_Object` class

As for the creation of a cluster object, there are two ways to give the location of a new communicating object: either by specifying to the allocator the target cluster, or by default, the new object will be mapped on the "best" cluster (see Figure 9).

2.3.2 Communication and migration

Sending data to a communicating object is done via its proxy (see Figure 9). Nothing special is required concerning these data. They just have to be embedded in a class which inherits from

```
Proxy_Comm_Object *p1, *p2;
p1 = new (cl1) Proxy_Comm_Object();   // on specific cluster
p2 = new Proxy_Comm_Object();         // on default cluster

Data_Comm_Object_Init *d = ...
p1→send(d);                           // Data are transparently embedded in an active
                                      // message and sent to cluster cl1
```

Figure 9: Creation of communicating objects

the `Data_Comm_Object` class (see Figure 10).

```
class Data_Comm_Object {
   public:
      virtual Buffer *flat(Buffer *buff)=0;
      static Message *rebuild(Buffer *buff, Data_Comm_Object *d=0);
}
```

Figure 10: Partial definition of the `Data_Comm_Object` class

These data are automatically sent by an active message, whose `process()` function triggers the call of the recipient `Comm_Object::receive(Data_Comm_Object *)` function. Thus, the user never has to call this last function.

We have added a migration capability for the communicating objects. An object can migrate to another cluster by using the primitive `migrate(Cluster *)`. According to load balancing schemes, we could have an implicit and automatic migration of communicating objects. When an object migrates, it is possible to inform or not all the clusters of the application about this new location. In case the clusters get not informed, one of the two following policies could be adopted:

1. A "forward" mechanism: when a communicating object migrates, the cluster can store its new location. If data for this communicating object arrive on this cluster, it has to (1) forward data to the new host cluster, and (2) inform the cluster which has sent the data of the communicating object new location.

2. If data for a communicating object which has just migrated arrive on its last host cluster, they are automatically sent back to the sender cluster. This one then broadcasts a message to all clusters in order to know the new location of the communicating object.

2.3.3 Example of code

Figure 11 presents an example of code using the Schooner library. The code written in italics is supposed to represent an implicit code: it is automatically triggered by Schooner, the user does not have to call it.

3 Implementation

3.1 Actual versions of Schooner

We have already developed two versions of Schooner:

1. The first one only uses PVM, and we can only map one communicating object on a cluster object. In other words, a communicating object is equivalent to a PVM process (Unix process).

`Cluster *cl1 = new ("mach") Cluster();`	⟶	// Initialization of cluster cl1
`Proxy_Comm_Object *p;`		
`p = new (cl1) Proxy_Comm_Object();`		
⇒ *Message_Comm_Object_Creation *m;*		
m = new Message_... (coid);		
send(m, cl1);	⟶	*Comm_Object *c = new Comm_Object();*
		c→set_id(coid); // Identifiant of comm_object
`Data_Comm_Object_Init *d;`		
`d = new Data_Comm_Object_Init(34);`		
`p→send(d);`		
⇒ *Message_Comm_Object *m2;*		
m2 = new Message_Comm_Object(d);		
send(m2, p→get_cluster());	⟶	*c→receive(d);*
`cl1→destroy();`		
⇒ *Message_Destroy_Cluster *m3= ...*		
send(m3, cl1);	⟶	*System::end();*

Figure 11: Example of SCHOONER code

2. The second version uses a "distributed-thread package" based on the RPC programming model (PM2). In fact, we only use a subset of PM2 as explained in Section 1.2.

We have developed these two versions with the following purpose: applications build upon SCHOONER should need only a new compilation to use the PVM or the PM2 SCHOONER version.

3.2 The different managers

Each cluster of the application runs various manager objects:

- An object of the `Configuration_Manager` class which allows to have a global view of the application support. This class defines functions to register a new computer or a new cluster, to obtain the list of available clusters, ...
- Communications between various objects of the application are performed through cluster objects. Thus one object of the `Communication_Manager` class on each cluster allows to exchange messages with the other clusters of the application.

  The main goal of this class is to prepare data according to the underlying used message-passing library (e.g. PVM, PM2).
- An instance of the `Comm_Object_Manager` class allows to manage proxies and communicating objects.

These objects are automatically created during the initialization phase of a cluster object. The `Configuration_Manager`, `Communication_Manager`, and `Comm_Object_Manager` classes can be derived in order to extend their functionalities.

4 Examples of SCHOONER use

4.1 A Distributed Version of PROSIT

PROSIT is a new discrete event simulation system. A sequential version of this library has been developed [MS95], and a distributed version using SCHOONER is currently under development.

A first version was built upon the second layer of SCHOONER, and so only uses clusters and active messages concepts, by deriving new active messages classes (see Section 2.2.4). We also try to develop a version by using the SCHOONER communicating objects.

4.2 The C++ // language

We are also currently developping the run-time of the C++ // language [CBR96]. This parallel extension of C++ is designed and implemented in the framework of a MIMD model, and unifies the concepts of class and unit of parallelism into the *active object concept*.

Data exchanges between active objects can follow three different protocols, all being asynchronous:

1. *communication with rendezvous*: the sender is blocked until: (1) a rendezvous is made with the receiver which is interrupted in order to enter the rendezvous, (2) the message is transmitted into the receiver context.
2. *reactive communication* : the communication is non blocking for the sender, but the receiver is also interrupted when a new message arrives (no rendezvous).
3. *pro-active communication* : the communication is non blocking for the sender, and the receiver is not interrupted; some explicit receiving action has to be undertaken by the receiver in order to get the newly arrived messages(s).

Each active object can follow only one C++ // communication protocol. Furthermore, when an active object `A` sends a message to another active object `B`, it has to use the communication protocol of `B`.

C++ // active objects derive from SCHOONER communicating objects. The class which defines active objects redefines the `Comm_Object::receive()` function in order to implement the three C++ // communication protocols. The reactivity property of the first two communication protocols is easily implemented thanks to the SCHOONER active message mechanism.

5 Conclusion

We have presented SCHOONER which is a library of classes used for writing parallel and distributed applications in C++. This library is effectively used, in particular, for C++ // and PROSIT.

We could in the future develop another SCHOONER third layer to provide a RPC-like programming model instead of a communicating object model. We also plan to use Nexus to support the low levels of the SCHOONER library.

References

[BFdM94] Françoise Baude, Nathalie Furmento, and Daniel Lafaye de Micheaux. Managing true parallelism in ADA through PVM. In *1st European* PVM *Users' Group Meeting*, Rome, Italy, October 1994. `http://www.netlib.org/pvm3/epvmug94/`, Also: Research rapport I3S 94-62.

[CBR96] Denis Caromel, Fabrice Belloncle, and Yves Roudier. *Parallel Programming Using C++*, chapter The C++// System. G. Wilson ed., MIT Press, 1996. To appear.

[CD95] Olivier Couland and Eric Dillon. Para++ : C++ bindings for message passing libraries : User guide. Technical Report n°174, INRIA Lorraine, June 1995.

[Cra93] S. Crane. The REX lightweight process library. Technical report, Imperial College of Science and Technology, London, England, 1993. `ftp:gummo.doc.ic.ac.uk`.

[CRG95] M. Christaller, M. R. Castaneda Retiz, and T. Gautier. Control Parallelism on top of PVM: The ATHAPASCAN Environment. In Dongarra and al., editors, *Proceedings of EuroPVM'95*, volume 5 of *Parallélisme, Réseaux et Répartition*, pages 71–76. HERMES, September 1995.

[FKT96] Ian Foster, Carl Kesselman, and Steven Tuecke. The Nexus Approach to Integrating MultiThreading and Communication. In *Parallel Programming Environments for High Performance Computing*, pages 53–67, Alpe d'Huez, France, April 1996.

[FS95] Adam Ferrari and V. S. Sunderam. TPVM: Distributed Concurrent Computing with Lightweight Processes. Technical report, University of Virginia, 1995.

[KAH95] Oliver Krone, Marc Aguilar, and Béat Hirsbrunner. PT-PVM : Using PVM in a multi-threaded environment. In Dongarra and al., editors, *Proceedings of Euro-PVM'95*, volume 5 of *Parallélisme, Réseaux et Répartition*, pages 83–88. HERMES, September 1995.

[Kep93] D. Keppel. Tools and Techniques for Building Fast Portable Threads Packages. Technical Report UWCSE 93-05-06, University of Washington, May 1993.

[Mes94] Message Passing Interface Forum. *Document for a Standard Message-Passing Interface*, February 1994.

[MS95] Philippe Mussi and Günther Siegel. The Prosit Sequential Simulator: A Test-bed for Object Oriented Discrete Event Simulation. In *European Simulation Symposium*, pages 297–301, Erlangen, Germany, October 1995.

[NM95] Raymond Namyst and Jean-François Méhaut. PM^2: Parallel Multithreaded Machine. A computing environment for distributed architectures. In *ParCo'95*, Gent, Belgium, September 1995.

[Sun90] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency - Practice and Experience*, 2(4):315–339, December 1990.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth C. Goldstein, and Klaus E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architectures*, Gold Coast, Australia, May 1992. ACM Press.