$me\mu$: unifying application modeling and cluster exploitation

Albano Alves
ESTiG-IPB
albano@ipb.pt

António Pina
DI-UMinho
pina@di.uminho.pt

José Exposto
ESTiG-IPB
exp@ipb.pt

José Rufino
ESTiG-IPB
rufino@ipb.pt

Abstract

The increasing complexity of high-demand long-running applications has faced programmers with the need to take into account both development hardness and execution time. $me\mu$ provides the flexibility to control the amount of computational and communication power being used in order to maximize resources utilization and to deliver high performance. In this paper we focus on the aspects of the paradigm that go beyond traditional message passing approaches, promoting the idea that by raising the abstraction level of programming models, programmers will make better use of the available resources with clear impact on both productivity and performance.

We introduce the resource as the abstraction used to represent and manage both physical resources – nodes, memory, processors and communication technologies – and logical resources – modules, processes, tasks, threads, groups, etc. We also concentrate on the task of specifying, locating and aggregating resources in order to support the mapping of applications into the target cluster hardware and the explicit management of memory hierarchy.

1. Introduction

The increasing importance of cluster technologies and the need for high-performance put strong demands on the field of parallel computing. In effect, clusters of symmetric multiprocessor (SMP) workstations interconnected by cost-effective, high-performance switching technologies are becoming an attractive alternative to run high-demand applications. This sort of parallel machine is usually assumed to be homogeneous. However, for technological and/or financial reasons, the upgrade of an initial homogeneous cluster may result on a heterogeneous cluster that comprises several technological partitions.

PVM [7] and MPI [12] are both specifications for message passing libraries that have been widely used to create extremely efficient parallel applications that run on many types of parallel computers. Message passing is a powerful and very general method of expressing parallelism that had a major influence on the wide use of parallel computers. Traditionally this paradigm is used mainly to program scientific and engineering algorithms that run as a stand-along application, utilizing the totality of the physical resources of a parallel computer. Nowadays, novel application domains like those that emerge from the increasing importance of the Internet, web computing and distributed computing are putting strong demands on cluster computing.

In another direction, clusters and other modern scalable parallel computer architectures possess multi-level memory hierarchies, comprising processor registers, multiple levels of cache, local memory and remote memory. Although the efficient management of the memory hierarchy is fundamental in high performance computing, most message-passing libraries, including MPI, do not provide locality information about the mapping of processes into cluster hardware and the associated data transfer cost.

In this paper we address the problem of designing and building high-demand complex long-running applications, decomposable into several cooperative components, that may run concurrently in a multi-user, multi-program environment. The shared target parallel machine is the heterogeneous cluster, viewed as a hierarchy system of hardware resources, constituted by several homogeneous technological partitions (sub-clusters), whose interoperability is guaranteed by the existence of multi-interface high-performance communication nodes that bridge sub-clusters.

$me\mu$ is presented as a novel approach to cluster computing shaped by a programming methodology that allows for a reasonable compromise between the designing and building of parallel applications and the mapping of those applications into a hierarchical parallel machine where the applications are required to run. Other design goals include the need to extend the familiar message passing-paradigm with novel abstractions to allow programmers to make better use of the available computing and communication resources. Finally, recognizing the importance of addressing memory hierarchy in programming models, we also propose abstractions to facilitate the explicit management of memory hierarchies by the programmer, since it is critical to the efficient execution of a scalable application.

Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)
1550-6533/04 $ 20.00 IEEE

2. Resource Oriented Computing

As a general purpose approach to the task of cluster computing $me\mu$ is designed as a user-level library to be used by programmers. At the conceptual level it is a restatement of a prototype implementation of a resource oriented computing paradigm aimed to accomplish an efficient and convenient way of modeling long-running complex applications [11].

The model introduces the resource as a generic metaphor that directly incorporates the notion of state, concurrency, locality and distribution. It also supports composition and coordination between applications. Resources are abstractions created to simplify the description of the parallel computer, the modeling and developing of the application and the execution of the application on the target computer.

2.1. The methodology

The approach followed in $me\mu$ includes the following three phases: *(a)* the definition and organization of the entities that represent the parallel computer – physical resources –, *(b)* the definition and organization of the entities that represent applications – logical resources – and *(c)* the mapping of the logical entities into the physical resources of the target computer.

$me\mu$ interface is organized around four basic abstractions intended for modeling logical and physical resources: *(1)* domains – to group or confine a hierarchy of related entities; *(2)* operons - to delimit the running contexts of tasks; *(3)* tasks – to support fine-grain concurrency and communication; *(4)* mailboxes - to queue messages sent/retrieved by tasks. Recently, to explicitly manage the intrinsic hierarchical memory of clusters we introduced two other abstractions : *(5)* memory blocks – segments of contiguous memory which may be asynchronously accessed by local or remote tasks; *(6)* memory gathers – to create the notion of a global contiguous memory whose capacity is the sum of several individual distributed memory blocks. Figure 1 shows the graphical representations of the presented abstractions.

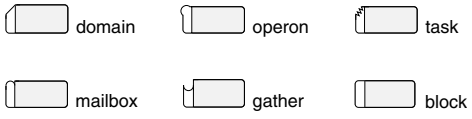
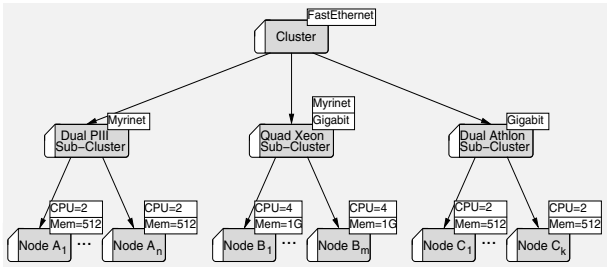
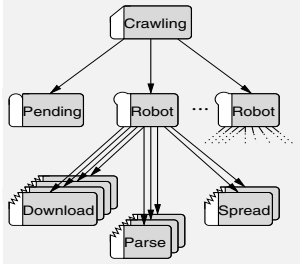


Figure 1. Modeling abstractions.

Modeling: As an instantiation of the first phase of the purposed methodology, the figure 2(a) shows how the physical resources of a parallel machine made of distinct technological partitions may be modeled by a hierarchy of domains. The domain *Cluster*, at the top-level, has as its direct descendants the three sub-clusters *Quad Xeon*, *Dual PIII* and *Dual Athlon*. At each sub-cluster, nodes are modeled by *Node* A_x, *Node* B_x and *Node* C_x domains.



(a)



(b)

Figure 2. Resource modeling examples.

The second phase of the methodology is illustrated in figure 2(b) by a high-level specification of a basic web application. The process of crawling web pages is modeled by a top domain that represents the application whose descendants are a certain number of *Robots* modeled by operon resources. Further refinement introduced three different sorts of tasks (*Download*, *Parse* and *Spread*) to model the three well known crawling stages: 1) downloading of pages from the web, 2) parsing the pages to obtain new URLs and 3) distribution of URLs for future crawling, according to a certain partitioning scheme. Finally *Pending* is modeled as a general global accessible queue used by *Download* tasks to store/retrieve pending URLs.

2.2. Resource properties

Resources are named and globally identified entities to which we may attach lists of relevant properties, which are valid pairs ⟨name, value⟩.

In figure 2(a), *FastEthernet*, *Myrinet* and *Gigabit* are properties that qualify *Cluster* and *Sub-Cluster* domains, while *CPU=2* and *Mem=512* are another sort of properties used to quantify characteristics of nodes A_x and C_x.

Properties inheritance: $me\mu$ defines a basic mechanism that lets resources to accumulate properties inherited from their chain of ancestors. This simple *inheritance* scheme is extended to include *synthesis* and *sharing*. With inheritance resource properties are propagated top-down from ancestors to descendants, while synthesis is a reverse mechanism that allows properties to propagate bottom-up from descendants to ancestors. Sharing is a special form of multiple-inheritance that allows resources to spread all its accumulated properties to other resources, called aliases, not directly related by the ancestor-descendant relationships.

An *alias* is a virtual resource node, used as a proxy to one or more existent nodes. In addition to the ancestor and descendants, an alias also has one or more *originals*, which share with it their own full set of accumulated properties. Aliases are represented as dashed shapes and the dashed arrow coming from each original in the direction of an alias represents the mechanism that allows for the sharing of the original accumulated properties with the target resource.

2.3. Specification language

To assist the administrator in specifying the cluster hardware, we provide a very simple language whose syntax is presented in figure 3. It allows for the textual representation of a tree where all nodes are resource domains. Each domain specification is described by an optional tag, a name, a reference to the ancestor and the full set of properties. In the case of an alias domain, the specification also includes a list of references to its originals. The specification of a domain uses tags to reference other domains in the tree.

Specification	::	(comment | Domain)+
Domain	::	[tag · ':'] · name · [Ancestor] · [Originals] · [Properties] · ';'
Ancestor	::	'⟨' · tag · '⟩'
Originals	::	'(' · tag · (',' · tag)* · ')'
Properties	::	'{' · Property · (',' · Property)* · '}'
Property	::	name · ['=' · value]

Figure 3. Specification language syntax.

By using this language, the cluster configuration depicted in figure 2(a) may be specified as shown in figure 4. A specific tool – *meμ_phyparse* – is used to parse the textual specification and make relevant information available to applications through a directory service (see section 3).

2.4. Application launching

To map the abstract logical representation, derived from the second phase, into physical resources, programmers may use a special set of primitives provided by $me\mu$. Next, we illustrated the mapping process used to produce the hierarchy presented in figure 5, which corresponds to the fusion of two hierarchies – the hierarchy that represents the cluster (figure 2(a)) and the hierarchy that represents the application (figure 2(b)).

```
/* Nodes description */
Cl: Cluster {FastEthernet};
Sub1: "Sub-Cluster Dual PIII" <Cl>
      {Myrinet};
Sub2: "Sub-Cluster Quad Xeon" <Cl>
      {Myrinet, Gigabit};
Sub3: "Sub-Cluster Dual Athlon" <Cl>
      {Gigabit};
"Sub-Cluster Myrinet" <Cl> (Sub1, Sub2);

/* Computation Nodes */
"Node A1" <Sub1> {CPU=2, Mem=512};
"Node B1" <Sub2> {CPU=4, Mem=1024};
"Node C1" <Sub3> {CPU=2, Mem=512};
....
```

Figure 4. Physical resource specification.

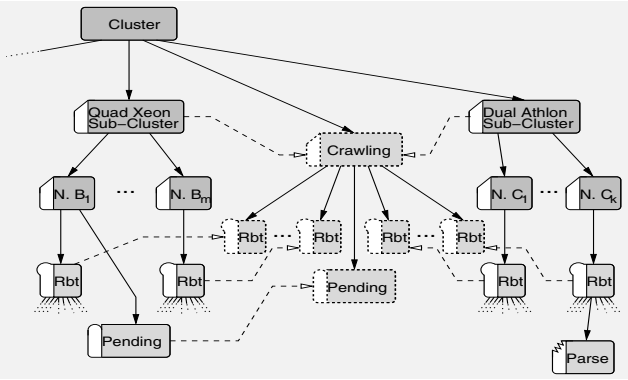


Figure 5. Logical-physical mapping.

The code fragment in figure 6 uses $me\mu$ primitives to launch the *Crawling* application into the target cluster. The first two *meu_lookup* primitives are used to retrieve from the directory the global identifiers of the previously registered *Quad Xeon* and *Dual Athlon* sub-clusters. These identifiers are used to specify the originals of the *Crawling* domain, which is defined as an alias by using the *meu_newalias* primitive.

Then, some operons are created through the *meu_newoperon* primitive. Each operon will be named *Robot* and will execute the *robot.bin* program under spe-

```
subc1 = meu_lookup(ROOT,
                   "Quad Xeon Sub-Cluster",
                   NULL, NULL);
subc2 = meu_lookup(ROOT,
                  "Dual Athlon Sub-Cluster",
                   NULL, NULL);
gidl1 = meu_newgidlist(subc1, subc2);
subcs1_2 = meu_newalias(ROOT, gidl1,
                        "Crawling", NULL);
for(i=1; i<m+k; i++){
    arglist[i] = meu_newarglist(i);
    meu_newoperon(subcs1_2, "Robot", NULL,
                  "robot.bin", arglist[i]);
}
node = meu_lookup(subc1, NULL, DOMAIN,
                  NULL);
meu_newmbox(node, "Pending", NULL);
```

Figure 6. *Crawling* application launching.

cific command line arguments. The target cluster node for the instantiation of each operon will be selected by $me\mu$, among the nodes comprised by the *Crawling* domain, that is, among nodes belonging to *Quad Xeon* and *Dual Athlon* sub-clusters. To preserve the application hierarchy, $me\mu$ automatically creates alias operons below the *Crawling* domain.

Finally a mailbox is created by specifying explicitly the target cluster node.

3. Communicating resources

Message passing is the mechanism devised to support the data transfer between $me\mu$ resources. In order to provide efficiency, on machines that have adequate low-level support, communication abstractions need to be close to the hardware. Those abstractions are layered on top of RoCL, a resource oriented communication library [1].

RoCL: RoCL was designed to exploit Myrinet and Gigabit through low-level communication libraries, thus achieving OS-bypassing and avoiding memory copying. At run-time it creates a fully dynamic distributed system where communication entities may be created and destroyed – a basic single system image.

RoCL is organized around three basic abstractions: resources, communication contexts and buffers that are used to build a variety of high-level message passing facilities. Distinguished features are the support to:(1) fine-grain concurrency and communication, (2) one-sided communication and (3) interoperability among resources residing in different technological communication partitions. Another relevant feature is the inclusion of a global distributed mechanism for resource registering and retrieval that provides for a basic directory service.

Threads and communication: A number of factors are motivating the increasing interest on multithreading in parallel computing environments, notably more irregular, heterogeneous applications, in which the patterns of communication can not be determined prior to execution. However with the traditional message passing approach individual threads are not visible outside a process.

To overcome the limitation of the process oriented message passing model when dealing to threads $me\mu$ introduces a novel *resource oriented* communication model that integrates threads and communication. It offers programmers fine-grain concurrency and communication among resources, as an alternative to traditional thread-safe process oriented communication that does not include the ability to address individual threads. The support to these facilities is delegated to RoCL, which uses the multi-threading capabilities of Linux Kernel through NPTL [13].

The directory: The existence of a rich variety of communicant resources, for which different behaviors may be specified, combined with the facilities of the distributed directory service, also powered by RoCL, is the key to go beyond the traditional message passing model. As so, resources are registered in the directory, at time of instantiation, along with its global identification and other system properties.

Regardless of its basic specification RoCL directory greatly surpass common approaches to general name server discovery mechanisms that use simple keys to register and lookup for global information. Queries submitted to directory are multi-key request packets formed by a certain number of completely specified properties – pairs ⟨name, value⟩ – and other partially specified properties – pairs ⟨name, null⟩. As an alternative, a query may include the resource identifier as a simple key.

When destination global identifiers are unknown, the directory eases the way resources belonging to dynamic long-running applications may concurrently communicate among them. The desired identifier to use as the message destination may always be retrieved by first issuing an appropriate query based on some well-known (public) properties of a target resource, previously registered in the directory.

4. Global memory

A fundamental issue in cluster computing is memory hierarchy and as a consequence the exploitation of data locality is one of the keys to achieve high-performance.

In this section we focus on the presentation of the abstractions and tools we developed to build a global memory address space. The abstractions comprise memory blocks and memory gathers and the tools are the primitives used to instantiate resources based on those abstractions and to support the data transfer between local and global memory.

A memory block is a segment of contiguous memory which may be asynchronously accessed by local or remote tasks. It may be read/written in portions or all at a time. As depicted in figure 7 memory blocks (B_x) are created as descendants of operons. One ore more memory blocks belonging to the same or different hierarchy of operons, may be joined together to create the illusion of a larger amount of contiguous memory. The key for this facility resides on the definition of a memory gather which is a resource that organizes together in one dimension several variable size distributed memory blocks.

A memory gather has as its descendants a sequence of memory block aliases. Each alias, automatically created by the run-time system, has one only original – a memory block that encloses a specific segment of memory. In figure 7, G_2 is a memory gather that combines several memory blocks by means of the correspondent aliases. The black portion of the small bars under memory block aliases represents the fraction of the contiguous memory segment that is imported to the alias. Note that the same memory block may be aggregated under multiple memory gathers. It is also possible for a memory gather to enclose other memory gathers along with simple blocks, as is the case of G_1 in figure 7.

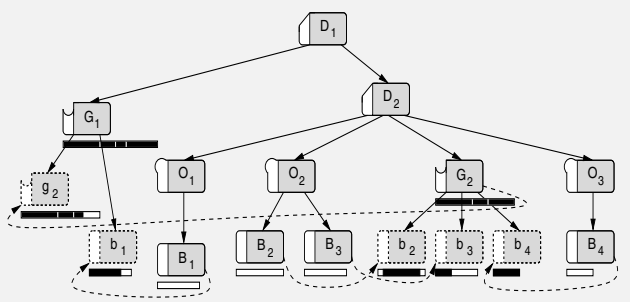


Figure 7. Block gathering example.

Figure 8 shows a code fragment used to create the two global address spaces depicted in figure 7. Memory blocks are created by specifying an ancestor, a name, a memory size and an optional list of properties. Memory gathers are created similarly, but obviously no size is given, since they may grow arbitrarily. When a block is added to a gather, it is possible to specify a fraction of the block (by default the whole block is considered) and an offset, relative to the gather, where the block fraction must be placed (by default blocks are appended to the end of the gather).

Global memory, represented by gathers, is used by explicitly calling $me\mu$ primitives that transfer data from several remote distributed blocks to application local address space. Figure 9 presents a basic example to explain how an

```
meu_newblock(O1, "B1", 8192, NULL);
...
meu_newgather(D1, "G1", NULL);
...
meu_addblock(B1, 32, 8000, G1, -1);
...
meu_addblock(B3, -1, -1, G1, -1);
...
meu_addblock(B4, 0, 6000, G2, 12000);
meu_addblock(G1, 0, 12000, G2, 0);
```

Figure 8. Instantiation of blocks and gathers.

application is able to access data from the global address space. First, it is necessary to obtain a pointer to a local chunk of memory which will be used as a cache to the remote data. Since applications may not need to access the whole global memory and individual nodes can not hold a copy of all remote blocks, it is required to specify a particular global memory fraction. Next, the global memory may be read through a *get* operation. By default, the whole segment specified in the *meu_newref* primitive is read, however programmers may decide to read a sub-fraction at a time. By using the address of the local memory chunk, in practice, global data is handled like regular memory obtained through *malloc*. Finally, to update remote blocks according to local computations, the *put* primitive must be used.

```
p = meu_newref(G2, 500, 13000);
meu_get(p, -1, -1);
/* read/write data through pointer p */
...
meu_put(p, -1, -1);
meu_freeref(p);
```

Figure 9. Global memory usage.

Note that *get* and *put* operations may read/write multiple remote memory blocks, spread among cluster nodes. Low-level RDMA facilities are used to improve performance.

5. Advanced resource managing

$me\mu$ provides a powerful paradigm for resource control and management that allows applications to interact with and manipulate their computing environment. It provides the flexibility to control the amount of computational and communication power being used in order to maximize cluster resources utilization and to deliver high performance to applications.

Resource control and management refers to the ability to define dynamic views, confining particular physical re-

sources required for a specific application or organizing application components in a more appropriate manner, and to adapt an application according to new requirements or hardware constraints.

5.1. Dynamic views

The paradigm proposed by meμ provides for a unified management of resources using the same abstractions to both logical and physical resources. As so, the exercise of selecting a particular operon as the place to instantiate a specific task is similar to the one of selecting a domain comprising the physical resources required for executing an application; the programmer specifies a list of properties and instructs the system to find a physical or logical resource that matches these properties. However, when no single resource assembles all required properties, a mechanism to aggregate multiple resources is still required.

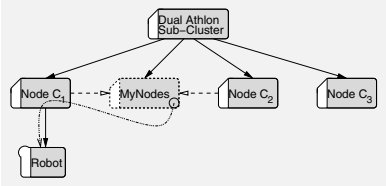


Figure 10. Example of an aggregator domain.

$me\mu$ allows to dynamically create an aggregator domain as a way to establish a different view to the existent resources. As an example, consider that the programmer wants to circumscribe some cluster nodes (referring to figure 2(a)) according to the following specification: each node must have 2 CPUs, nodes must be interconnected by Gigabit and a total of 4 CPUs must be available. Figure 10 shows an alias domain which accomplishes these requirements.

```
propl1 = meu_newproplist("CPU", 2);
meu_addprop(propl1, "Gigabit");
propl2 = meu_newproplist("CPU", 4);
domain = meu_newaggregate(propl1, propl2,
                    ROOT, "MyNodes", NULL);
```

Figure 11. Creation of an aggregator domain.

The code fragment presented in figure 11 uses a few $me\mu$ primitives to create the desired aggregator domain. First, two lists of properties are created: the first one to specify the properties each entity must own and the second one to specify the properties owned by the collection of entities represented by the aggregator. The *meu_newaggreator* primitive uses these two lists to retrieve from the directory the resources that all together match both conditions, starting at the root domain. On success the primitive creates an alias domain, whose accumulated properties imported from its originals fulfill the two property lists.

Aggregator domains may also be created around the properties of logical resources or the mixing of properties of logical and physical resources.

5.2. Programmer's views

In what follows, by taking as an example the evolution of a dynamic Crawling application, we demonstrate the $me\mu$ features that support application scaling. The general idea is to show how an initial view of the physical system resources may later be derived into distinct programmer's views shaped by different application requirements.

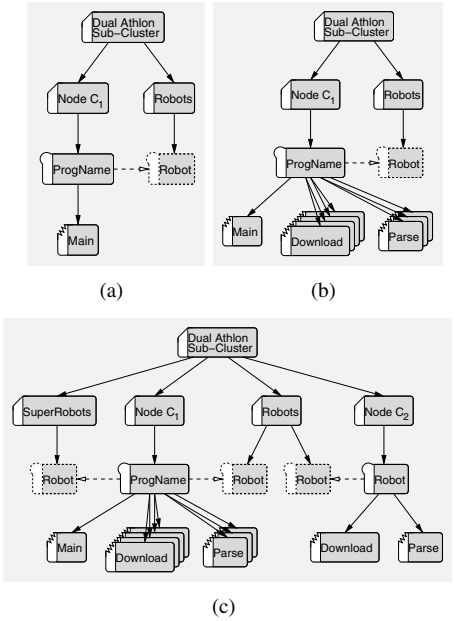


Figure 12. Application scaling.

Assuming the crawling application is launched via command line at *Node* C_1, an operon *ProgrName* and a task *Main* (see figure 12(a)) are automatically created. As the programmer prefers its own application view, a new domain *Robots* is created to hold an alias operon. This alias (*Robot*) will be used by the application instead of the original operon (*ProgrName*).

The next step, illustrated in figure 12(b), corresponds to the creation of several parse and download tasks, instantiated by the task *main*, as descendants of *Robot*. Since *Robot* is an alias, the run-time will follow the link to reach operon *ProgName*, where tasks will effectively be instantiated.

Finally, the configuration depicted in figure 12(c) shows how the application may be scaled up, in order to respond to higher demands. A new cluster node is exploited by creating another operon (via the *meu_newoperon* primitive) and an alias is created under the domain *Robots*, in order to maintain the application view. That way, the new application entities along with the initial entities may be accessed through a single resource. Simplifying, we might say that the domain *Robots* increased its power from one to two operons.

It is important to note that, besides the system view, the programmer may define multiple views to the application entities. The *SuperRobots* domain and its descendant alias, for instance, are used to confine a particular operon of the whole crawling application.

6. Advanced groups

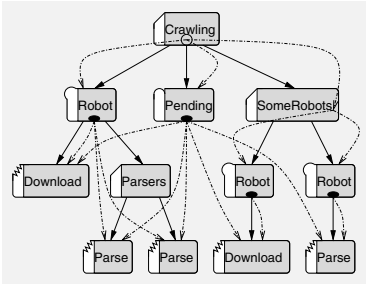
When writing parallel programs it is useful to use the group abstractions to ease the naming problem of interrelated entities. $me\mu$ domains are the equivalent to groups, however domains may include members of many types, thus creating the notion of hybrid groups. A message addressed to a domain is, automatically, forwarded to all its members – the originals and the descendants – thus performing selective broadcast.

Advanced group features have been included in $me\mu$ to provide for more elaborate forms of organization and to support other styles of communication among resources that easily coexist with traditional message passing. Figure 13 is used to briefly summarize the following features: group chaining and group rebuilding.

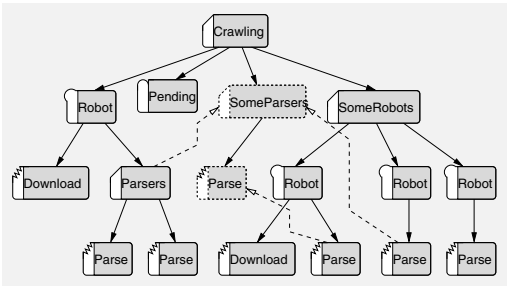
Selective broadcast and chaining: The situation depicted in figure 13(a) represents two active groups – *Crawling* and *SomeRobots*; solid line arrows mean a membership relation, dotted line arrows correspond to a point-to-point communication, a small black-ball stands for a stored message and the small circle at *Crawling* indicates the message initial destination. The first group is a hybrid group whose members are an operon, a mailbox and another domain. The second group, which is also member of the first domain (group chaining), includes two operons.

In what follows we trace the communication path of a message sent by a task (internal or external) to *Crawling*.

We start by applying the rule that states that (a) whenever a message arrives at a domain a copy of the message is forwarded to all its members. In this situation, the reception of a message, by the operon and the mailbox, is expected to result on its immediate storage (see the small black-ball).



(a)



(b)

Figure 13. Chaining of hybrid groups.

When the domain *SomeRobots* receives the forwarded message, to preserve group behavior, that message is forwarded again and stored in its two descendant operons.

Later on, the four stored messages may be effectively consumed by obeying to the next following rules: (b) tasks belonging to the sub-tree whose root is an operon may consume messages stored on it and (c) messages stored in a mailbox may be extracted by any task belonging to the tree defined by the ancestor of that mailbox. Obviously, a task may benefit from both stated rules.

Group rebuilding: In figure 13(b), *SomeParser* is a new alias domain which has as its originals the domain *Parsers* and one of the tasks *Parse*. As its direct descendant an alias task is used to establish a membership relation with other of the tasks *Parse*.

As a side effect of this group rebuilding, all tasks, excluding the bottom right one, would receive a copy of any message sent to *Crawling*. The reasons for this result comes from the application of a new rule, in addition to the rules presented above: (d) any message sent to an alias is forwarded to its originals. The operons and the mailbox will continue to receive message copies as stated before.

7. Discussion

The integration of various communication technologies has motivated the development of some intermediate-level communication libraries. A refer-

ence work is Madeleine [3], which also permits to exploit Myrinet and Gigabit. However, the most popular intermediate-level libraries do not introduce relevant abstractions when compared to low-level communication libraries. Basically, they offer node-to-node communication or, in some cases, port-to-port communication, and programmers have the responsibility to map application entities into communication end points. RoCL introduces a new communication paradigm, supported by a low-level directory service, which allows to add communication facilities to any computational entity.

The development of parallel applications requires high-level abstractions, which may be built on top of intermediate-level libraries. This is the case of PM2 [9] over Madeleine, Panda [2], MPICH over Madeleine, etc. These abstractions are usually intended to efficiently exploit the available resources and not to ease application modeling, namely when the programmer wants to model the application as a collection of threads that share cluster resources. In some cases the multithreading support is specific, thus difficulting the use of modules that use POSIX threads. $me\mu$ includes abstractions to ease the modeling of applications and, by relying on RoCL, it ensures POSIX multithreading functionality.

To exploit the various locality levels that it is possible to identify in a cluster, some authors proposed programming models that combine message passing and shared memory, e.g. the use of both MPI and OpenMP. Some innovative approaches had also been presented, as is the case of Kelp [6]. $me\mu$, because of its mechanisms to map logical into physical resources, is another approach to the same problem. However, $me\mu$ considers another level of parallelism which is not addressed by common approaches: sub-clusters. It is also possible, in $me\mu$, to extend the operation to remote clusters. IMPI [8] addresses this last topic but it does not include relevant abstractions to model applications.

Resource allocation and reservation experienced important advances through the development of resource description languages, like the specification language used in the RSD environment [4], for example. Recently, with the increasing interest on Grid, this topic is considered extremely important because of the concept of meta-computing [5]. $me\mu$ provides very simple mechanisms, but it presents a unique characteristic: a unique mechanism is provided to handle both physical and logical resources.

Message passing is the most used parallel programming paradigm and both MPI and PVM are key references. When programmers experiment new platforms, they expect to be able to use the most important concepts and mechanisms present on these two systems. $me\mu$ includes important features available on traditional platforms but it extends these features to offer more flexibility on the design and execution of applications.

The distributed shared memory paradigm is very convenient for application programming but leads to poor performance because of the overheads of the mechanisms used to guarantee consistency. However, new approaches that eliminate these overheads and expose the programmer to the memory hierarchy details are becoming an effective alternative [10]. $me\mu$ uses a similar approach, but global memory abstractions are entirely integrated with the remaining abstractions.

References

[1] A. Alves, A. Pina, J. Exposto, and J. Rufino. RoCL: A Resource oriented Communication Library. In *Euro-Par 2003*, LNCS 2790, pages 969–979. Springer, 2003.

[2] R. Bhoedjang, T. Rühl, R. Hofman, K. Langendoen, and H. Bal. Panda: A Portable Platform to Support Parallel Programming Languages. In *USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, 1993.

[3] L. Bougé, J.-F. Méhaut, and R. Namyst. Madeleine: An Efficient and Portable Communication Interface for RPC-Based Multithreaded Environments. In *PACT '98*, 1998.

[4] M. Brune, A. Reinefeld, and J. Varnholt. A Resource Description Environment for Distributed Computing Systems. In *International Symposium on High Performance Distributed Computing*, pages 279–286, 1999.

[5] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. In *IPPS/SPDP'98*, pages 62–82, 1998.

[6] S. J. Fink and S. B. Baden. Runtime Support for Multi-Tier Programming of Block-Structured Applications on SMP Clusters. In *International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE '97)*, pages 1–8, 1997.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation. MIT Pres, 1994.

[8] W. L. George, J. G. Hagedorn, and J. E. Devaney. IMPI: Making MPI interoperable. *Journal of Research of the National Institute of Standands and Technology*, 105(3):343–428, 2000.

[9] R. Namyst and J. Méhaut. PM^2: Parallel Multithreaded Machine. A computing environment for distributed architectures. In *ParCo'95*, 1995.

[10] J. Nieplocha, R. Harrison, and I. Foster. *Advances in High Perf. Computing*, chapter Explicit Management of Memory Hierarchy, pages 185–198. Kluwer, 1996.

[11] A. Pina, V. Oliveira, C. Moreira, and A. Alves. pCoR: a prototype for resource oriented computing. In *HPC 2002*, pages 251–262. WITpress, 2002.

[12] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - The Complete Reference*. Scientific and Engineering Computation. MIT Pres, 1998.

[13] Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for Linux, 2003.