Deploying applications in multi-SAN SMP clusters

Albano Alves*

Escola Superior de Tecnologia e Gestão,
Instituto Politécnico de Bragança,
Campus de Santa Apolónia – Apartado 1038,
5301-854 Bragança, Portugal
E-mail: albano@ipb.pt
*Corresponding author

António Pina

Departamento de Informática,
Universidade do Minho,
Campus de Gualtar 4710-057 Braga, Portugal
E-mail: pina@di.uminho.pt

José Exposto and José Rufino

Escola Superior de Tecnologia e Gestão,
Instituto Politécnico de Bragança,
Campus de Santa Apolónia – Apartado 1038,
5301-854 Bragança, Portugal
E-mail: exp@ipb.pt
E-mail: rufino@ipb.pt

Abstract: The effective exploitation of multi-SAN SMP clusters and the use of generic clusters to support complex information systems require new approaches; multi-SAN SMP clusters introduce new levels of parallelism and traditional environments are mainly used to run scientific computations. In this paper we present a novel approach to the exploitation of clusters that allows integrating in a unique metaphor: the representation of physical resources, the modelling of applications and the mapping of application into physical resources. The proposed abstractions favoured the development of an API that allows combining and benefiting from the shared memory, message passing and global memory paradigms.

Keywords: cluster computing; heterogeneity; resource management; application modelling; logical-physical mapping.

Reference to this paper should be made as follows: Alves, A., Pina, A., Exposto, J. and Rufino, J. (2009) 'Deploying applications in multi-SAN SMP clusters', *Int. J. Computational Science and Engineering*, Vol. 4, No. 3, pp.137–148.

Biographical notes: Albano Alves is a Professor Coordenador at the Instituto Politécnico de Bragança, Portugal. He received his PhD Degree from Universidade do Minho, Portugal, in 2004. Since 2000 he has been researching in cluster computing and distributed systems.

António Pina is a Professor Auxiliar at Departamento de Informática, Universidade do Minho, Portugal. He received his MSc and PhD Degrees in Informatics from Universidade do Minho, Portugal. Throughout his career he has conducted research and published in the areas of scalable parallel computing, parallel languages and environments and information retrieval.

José Exposto is a Professor Adjunto at the Instituto Politécnico de Bragança, Portugal. He is now concluding his PhD Thesis.

José Rufino is a Professor Adjunto at the Instituto Politécnico de Bragança, Portugal. He received his PhD Degree from Universidade do Minho, Portugal, in 2008. Since 2000 he has been researching in cluster computing and distributed systems.

1 Introduction

Common parallel programming tools are intended to develop conventional applications capable of exploiting conventional parallel machines. Non-conventional architectures and emerging application domains will benefit from novel approaches.

1.1 Motivation

Clusters of Symmetric Multi-Processor (SMP) workstations interconnected by a high-performance System Area Network (SAN) technology are becoming an effective alternative for running high-demand applications. The assumed homogeneity of these systems has allowed the development of efficient platforms. However, to expand computing power, new nodes may be added to an initial cluster and novel SAN technologies may be considered to interconnect these nodes, thus creating a heterogeneous system that we name multi-SAN SMP cluster.

Recently, the hierarchical nature of SMP clusters has motivated the investigation of appropriate programming models (Baden and Fink, 2000; Gursoy and Cengiz, 1999). But to effectively exploit multi-SAN SMP clusters and support multiple cooperative applications new approaches are still needed. In fact, PVM and MPI parallel based environments used to exploit and develop parallel programs assume that clusters are simple and straight collections of processors, thus ignoring the various levels of their intrinsic memory hierarchies.

PVM (Geist et al., 1994) and MPI (Snir et al., 1998) 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, making use of 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.

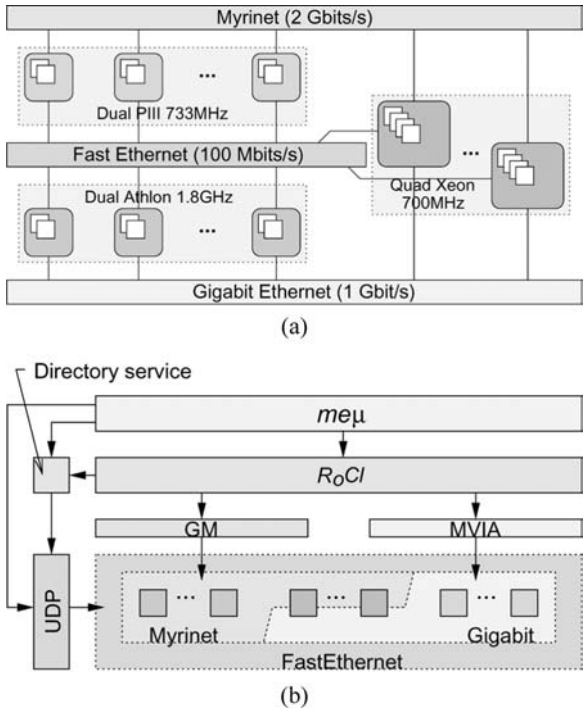
As long as novel programming models and runtime systems are developed, we may consider using clusters to support complex information systems, integrating multiple cooperative applications.

1.2 Our approach

Figure 1(a) presents a practical example of a multi-SAN SMP cluster mixing Myrinet and Gigabit. Multi-interface nodes are used to integrate sub-clusters (technological partitions).

To exploit such a cluster we developed R_oCl (Alves et al., 2003), a communication library that combines GM (Myricom Inc., 2000) – a low-level communication library provided to interface Myrinet – and MVIA – a Modular implementation of the Virtual Interface Architecture (Compaq et al., 1997) that supports Gigabit. R_oCl may be considered a communication-level Single System Image (SSI), since it provides full connectivity among application entities instantiated all over the cluster and also allows the registration and discovery of entities (see Figure 1(b)) through a basic cluster oriented directory service, relying on UDP broadcast.

Figure 1 Exploitation of a multi-networked SMP cluster: (a) cluster representation and (b) R_oCl and $m_\varepsilon\mu$ layers



Now we propose a new layer – $m_\varepsilon\mu$ – built on top of R_oCl, intended to assist programmers in setting-up cooperative applications and exploiting cluster resources. Our contribution may be seen as a new methodology comprising three stages:

- the representation of physical resources
- the modelling of application components
- the mapping of application components into physical resources.

This paper is structured as follows: the next section formalises our approach; Sections 3 and 4 detail important aspects of our modelling and programming tools; Sections 5–7 exemplify each one of the stages stated above; finally, in Section 8 we summarise our concluding remarks.

2 Resource oriented computing

The resource oriented computing paradigm is aimed at accomplishing an efficient and convenient way of

modelling long-running complex applications (Pina et al., 2002). In R_oCl the resource abstraction is used to model communication end-points. In $m_\varepsilon\mu$ it is used to simplify the description of the parallel computer and the modelling/development of the application and it plays an important role in supporting application execution.

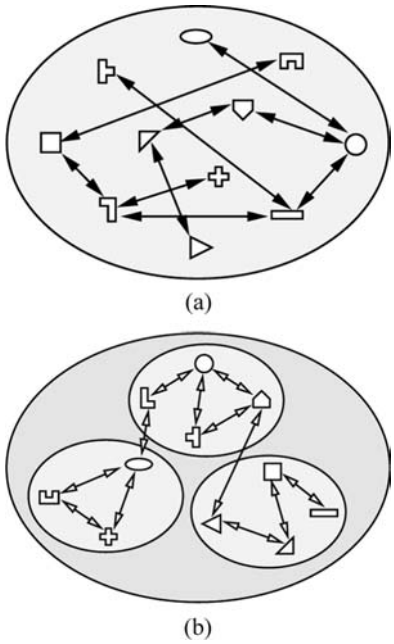
2.1 Resources in R_oCl and $m_\varepsilon\mu$

R_oCl acts as a base layer interfacing low-level communication subsystems generally available in clusters. It constitutes a basic single system image, by providing interconnectivity among communication entities. These entities we named resources are application entities that may exchange messages despite the underlying communication subsystems. The R_oCl dispatching mechanism is able to bridge messages from GM to M-VIA, for instance. A cluster oriented directory service allows programmers to announce and locate application resources thus turning R_oCl into a convenient platform to drive multi-SAN clusters that integrate multiple sub-clusters (Myrinet and Gigabit sub-clusters, for instance) interconnected by multihomed nodes.

$m_\varepsilon\mu$ was implemented over R_oCl and provides higher-level programming abstractions. Basically, it supports the specification of physical resources, the instantiation of logical resources accordingly to the actual organisation of physical resources and high-level communication operations between logical resources.

Figure 2 depicts the main difference between R_oCl and $m_\varepsilon\mu$ abstraction models; while R_oCl (Figure 2(a)) provides unstructured communication between resources, $m_\varepsilon\mu$ (Figure 2(a)) offers mechanisms to organise resources and exploit locality.

Figure 2 Resource interaction in: (a) R_oCl and (b) $m_\varepsilon\mu$



The overall platform is well suited to the development of applications aimed at exploiting multi-SAN clusters that integrate multiple communication technologies such as Myrinet and Gigabit. The implicit programming model has proved to be very useful to manage the distinct locality levels intrinsic to the target system architecture. At the application level, the approach smoothly accommodates the evolution from cluster to multi-cluster grid enabled computing.

2.2 Resource oriented communication

R_oCl is an intermediate-level communication library that allows system programmers to easily develop higher-level programming environments. It uses existing low-level communication libraries to interface networking hardware, like Madeleine (Aumage et al., 2000).

R_oCl defines three major entities: contexts, resources and buffers. Contexts are used to interface the low-level communication subsystems. Resources permit to model both communication and computation entities whose existence is announced by registering them in a global distributed directory service. To minimise memory allocation and pinning operations, R_oCl uses a buffer management system; messages are held on specific registered memory areas to allow zero-copy communication.

R_oCl includes a fully distributed directory service where resources are registered along with their attributes; a directory server is started at each cluster node and all application resources are registered locally. A basic interserver protocol allows applications to locate remote resources; query requests are always addressed to a local server but, at any moment, servers may trigger a global search by broadcasting the request.

The typical sequence of operations in an application includes:

- initialising a R_oCl context
- registering resources
- querying the directory to find remote resources
- requesting message buffers
- sending messages to resources previously found
- retrieving received messages from a local queue.

Resources are animated by application threads which share the communication facilities provided by contexts. On the other hand, R_oCl supports the simultaneous exploitation of multiple communication technologies being responsible for selecting the most appropriate communication medium to deliver messages to a specific destination, for aggregating technologies when the target resource of a message can be reached through distinct technologies and for routing messages at multihomed nodes that interconnect distinct sub-clusters. To provide the above facilities, R_oCl uses a multithreaded dispatching mechanism and includes native support for multithreaded applications.

2.3 Unified modelling and exploitation

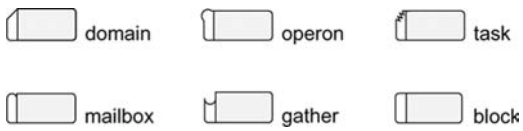
$m_\varepsilon\mu$ programing methodology includes three phases:

- the definition and organisation of the concepts that model the parallel computer – physical resources
- the definition and organisation of the entities that represent applications – logical resources
- the mapping of the logical entities into the physical resources.

The programing interface is organised around six basic abstractions (see Figure 3) designed for modelling both logical and physical resources:

- domains – to group or confine a hierarchy of related entities
- operons – to delimit the running contexts of tasks
- tasks – to support fine-grain concurrency and communication
- mailboxes – to queue messages sent/retrieved by tasks
- memory blocks – to define segments of contiguous memory
- memory gathers – to create the illusion of global memory.

Figure 3 $m_\varepsilon\mu$ entities



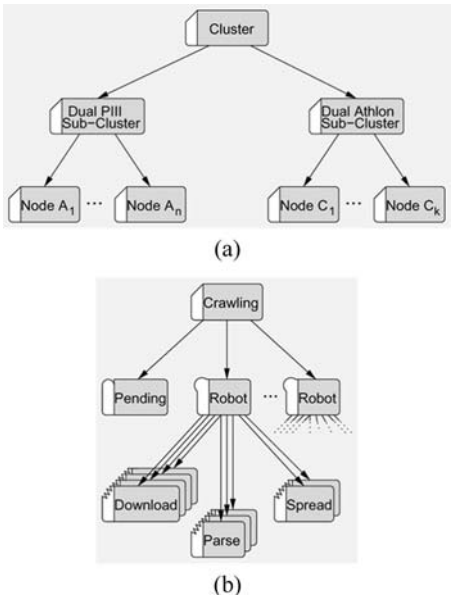
These abstractions are to be disposed of in a tree, reflecting the memory hierarchy of the cluster. Figure 4(a) shows how the physical resources of a parallel machine made of distinct technological partitions may be modelled by a hierarchy of domains. Figure 4(b) presents a high-level specification of a basic web application.

The mapping of logical into physical resources is achieved by merging the base hierarchies – physical resources and logical entities – into a single $m_\varepsilon\mu$ hierarchy.

3 Resource properties

Resources are named and globally identified entities to which we may attach lists of relevant properties. Properties allow to improve the meaningfullness of each entity present in a $m_\varepsilon\mu$ hierarchy by pointing out the presence or defining the value of a particular characteristic. In a $m_\varepsilon\mu$ hierarchy, the properties of an entity are computed taking into account its specific location.

Figure 4 Resource modelling examples: (a) physical hierarchy and (b) logical hierarchy

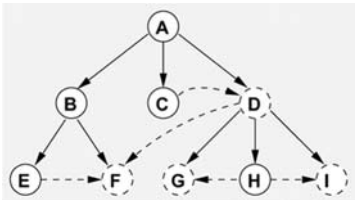


3.1 Aliases

In addition to regular ascendant-descendant relationships present in a tree, it is possible in a $m_\varepsilon\mu$ hierarchy to establish original-alias relationships. Thus, an entity may have one or more aliases and an alias may result from one or more original entities.

The creation of an alias for a given entity corresponds to the creation of another entity, of the same kind, at another point of the tree, and to the storage of the identifier of the first entity (the original) in the second entity (the alias). To create an alias for a given set of entities (all of the same kind) it is required to store the identifiers of all original entities in the alias. Figure 5 presents some examples of original-alias relationships, which are represented by dashed arrows.

Figure 5 Ascendants, descendants, originals and aliases



$descendants(A) = \{B, C, D\}$
$ascendant(F) = B$
$aliases(H) = \{G, I\}$
$aliases(F) = \{\}$
$aliases(C) = \{D\}$
$originals(F) = \{D, E\}$
$originals(D) = \{C\}$

Origin-alias relationships are set up by creating new entities that are inserted in the tree. The insertion of an alias must not misrepresent the regular ascendancy chains[1] of a tree, thus, none of the originals of the alias can belong to the

ascendancy chain of the entity where the alias is attached. Since the ascendancy chain of an entity may contain aliases (including the entity itself), the stated restriction must be applied to the aliasing-ascendancy chains, that is, all node chains that lead to the tree root, including alias entities.

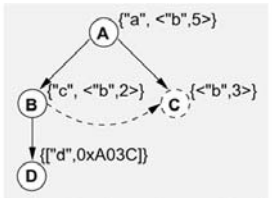
3.2 Obtaining properties

Considering the original-alias and ascendant-descendant relationships, the process to obtain the properties of an entity is by collecting the properties directly attached to that entity (own properties) and the properties obtained through inheritance, synthesis or sharing. The inheritance mechanism ensures that the properties of an entity are passed along to its descendants while the synthesis corresponds to the reverse. Property sharing allows an entity to spread all its properties to its aliases.

Figure 6 shows the way properties are determined using a simple example. It is important to note that some properties may involve computation aside from union.

The ability of establishing entity relationships and individually attach properties to entities along with the mechanisms of inheritance, synthesis and sharing allows the characterisation of complex systems efficiently.

Figure 6 Determining entity properties



$$p(B) = \underbrace{\{\text{"}c\text{"}, \langle \text{"}b\text{"}, 2\rangle\}}_{owned} \cup \underbrace{\{\text{"}a\text{"}, \langle \text{"}b\text{"}, 5\rangle\}}_{inherited} \cup \underbrace{\{[\text{"}d\text{"}, A03C]\}}_{synthesised}$$

$$p(C) = \underbrace{\{\langle \text{"}b\text{"}, 3\rangle\}}_{owned} \cup \underbrace{\{\text{"}a\text{"}, \langle \text{"}b\text{"}, 5\rangle\}}_{inherited} \cup \underbrace{p(B)}_{shared} =$$

$$= \{\text{"}a\text{"}, \text{"}c\text{"}, \langle \text{"}b\text{"}, 10\rangle, [\text{"}d\text{"}, A03C]\}$$

4 Programming paradigms

$m_\varepsilon\mu$ allows programmers to combine three well known parallel programming paradigms: shared memory, message passing and global memory.

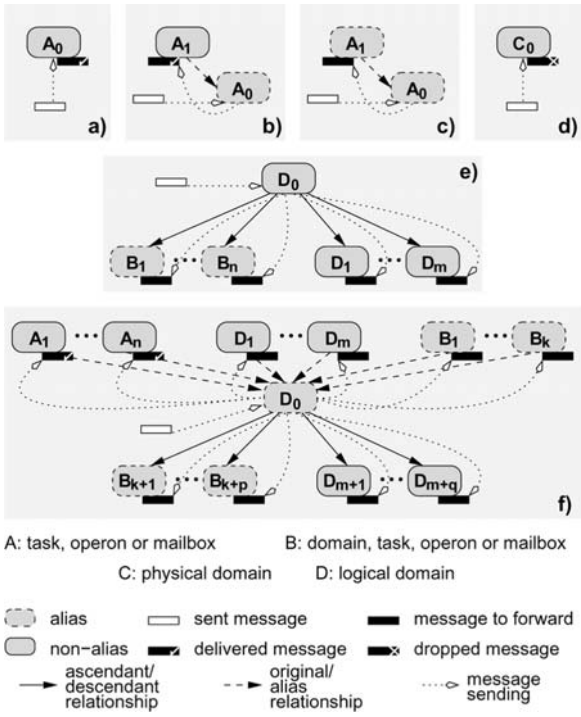
Shared memory programming in $m_\varepsilon\mu$ is equivalent to traditional POSIX threads programming, since $m_\varepsilon\mu$ tasks are directly mapped into POSIX threads.

4.1 Message passing

The approach followed in $m_\varepsilon\mu$ extends the traditional message passing model by integrating distinct types of resources used in the process of modelling an application. In fact, only tasks may generated messages, but the destination of a message may be a task, an operon, a domain or a mailbox.

When a message is addressed to a task, the communication mechanism provided by R_oCl is adequate (see Figure 7(a)). If the message destination is an alias task, then the message must be forwarded to the original task, thus requiring additional functionality (see Figure 7(b)). If the original, by its turn, is an alias, then the forwarding process will continue (see Figure 7(c)).

Figure 7 Message passing scenarios



When a message is sent to an operon, any descendant non-alias task may compete to get these message. The operon may be considered as a repository where the only message copy is stored until a task claims for it. If the operon is an alias, the forwarding mechanism, is activated.

The mailbox acts like the operon, storing the message copy, when it receives a message. But in this case, there will be more tasks that can access the message; any task belonging to the subtree defined by the ascendant of the mailbox may claim the message.

When a message is addressed to a domain, a message copy is sent to each descendant capable of receiving messages (tasks, operons, mailboxes and domains) and, if the domain is an alias, each original will also receive a message copy (see Figure 7(e) and (f)). Any message addressed to a domain that represents a physical resource will be dropped (see Figure 7(d)).

4.2 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.

To build a global memory address space, $m_\varepsilon\mu$ includes two abstractions: memory blocks and gathers. The instantiation of resources based on these abstractions and

the data transfer between local and global memory are accomplished by calling specific $m_\varepsilon\mu$ primitives.

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 8 memory blocks (B_x) are created as descendants of operons. One or 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 organises 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 8, G_2 is a memory gather that combines several memory blocks by means of the corresponding 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 8.

Figure 8 Block gathering example

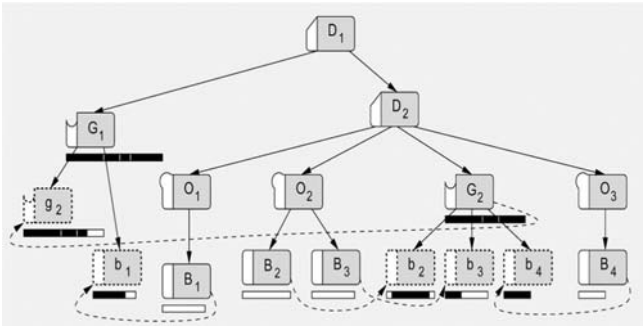


Figure 9 shows a code fragment used to create the two global address spaces depicted in Figure 8. Memory blocks are created by specifying an ascendant, a name, a memory size and an optional list of properties. Memory gathers are created similarly, but obviously no size is given, since they may grow arbitrarily. 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 accessed by explicitly calling $m_\varepsilon\mu$ primitives that transfer data from several remote distributed blocks to application local address space. Figure 10 presents a basic example to explain how an 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.

Figure 9 Instantiation of blocks and gathers

```
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 10 Global memory usage

```
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);
```

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

5 Representing physical resources

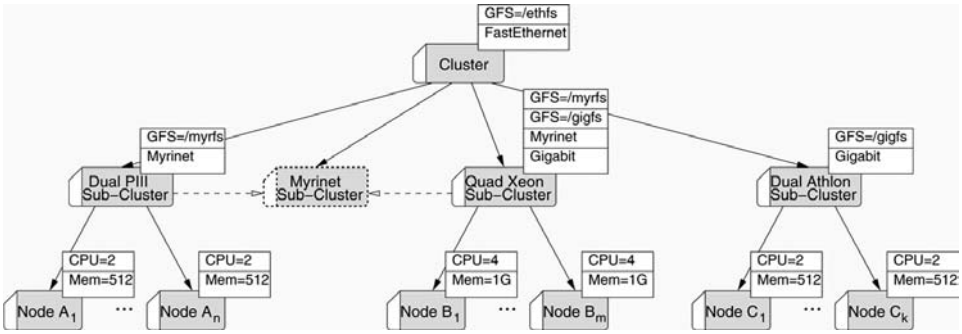
The manipulation of physical resources requires their adequate representation and organisation. Following the intrinsic hierarchical nature of multi-SAN SMP clusters, a tree is used to lay out physical resources. Figure 11 shows a resource hierarchy to represent the cluster of Figure 1(a).

5.1 Basic organisation

Each node of a resource tree – a domain – confines a particular assortment of hardware, characterised by a list of properties. Higher-level domains introduce general resources, such as a common interconnection facility, while leaf domains embody the most specific hardware the runtime system can handle.

Properties are useful to evidence the presence of qualities or to establish values that clarify or quantify facilities. For instance, in Figure 11, the properties `Myrinet` and `Gigabit` divide cluster resources into two classes while the properties `GFS` = ... and `CPU` = ... establish different ways of accessing a global file system and quantify the resource *processor*, respectively.

Figure 11 Cluster resources hierarchy



Because every node inherits properties from its ascendant, it is possible to assign a particular property to all nodes of a subtree by attaching that property to the subtree root node. *Node* A_1 will thus collect the properties $\texttt{GFS} = \texttt{/ethfs}$, $\texttt{FastEthernet}$, $\texttt{GFS} = \texttt{myrfs}$, $\texttt{Myrinet}$, $\texttt{CPU} = 2$ and $\texttt{Mem} = 512$.

By expressing the resources required by an application through a list of properties, the programmer instructs the runtime system to traverse the resource tree and discover a domain whose accumulated properties conform to the requirements. Respecting Figure 11, the domain *Node* A_1 fulfils the requirements $(\texttt{Myrinet}) \wedge (\texttt{CPU} = 2)$, since it inherits the property $\texttt{Myrinet}$ from its ascendant.

If the resources required by an application are spread among the domains of a subtree, the discovery strategy returns the root of that subtree. To combine the properties of all nodes of a subtree at its root, we use a synthesisation mechanism. Hence, *Quad Xeon Sub-Cluster* fulfils the requirements $(\texttt{Myrinet}) \wedge (\texttt{Gigabit}) \wedge (\texttt{CPU} = 4 * \texttt{m})$.

5.2 Virtual views

The inheritance and the synthesisation mechanisms are not adequate when all the required resources cannot be collected by a single domain. Still respecting Figure 11, no domain fulfils the requirements $(\texttt{Myrinet}) \wedge (\texttt{CPU} = 2 * \texttt{n} + 4 * \texttt{m})$.[2] A new domain, symbolising a different view, should therefore be created without compromising current views.

In Figure 11, the domain *Myrinet Sub-cluster* (dashed shape) is an alias whose originals (connected by dashed arrows) are the domains *Dual PIII* and *Quad Xeon*. This alias will therefore inherit the properties of the domain *Cluster* and will also share the properties of its originals, that is, will collect the properties attached to its originals as well as the properties previously inherited or synthesised by those originals.

By combining original/alias and ascendant/descendant relations we are able to represent complex hardware platforms and to provide programmers the mechanisms to dynamically create virtual views according to application requirements. Other well known resource specification approaches, such as the Resource and Service Description (RSD) environment (Brune et al., 1999), do not provide such flexibility.

5.3 Resource specification

To assist the administrator in specifying the cluster hardware, we provide a very simple language whose syntax is presented in Figure 12. 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.

Figure 12 Specification language syntax

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

By using this language, the cluster configuration depicted in Figure 11 may be specified as shown in Figure 13. $m_\varepsilon\mu$ provides a tool to parse the textual specification and make relevant information available to applications through the directory service.

Figure 13 Physical resource specification

```
/* 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};
....
```

5.4 Creating virtual views

The exercise of selecting a particular domain comprising the physical resources required for executing an

application may not be easy. The programmer specifies a list of properties and instructs the system to find the physical domain that matches these properties, but when no single domain assembles all required properties, the system must be able to create an alias.

As an example, consider that the programmer wants to circumscribe some cluster nodes (referring to Figure 11) according to the following specification: each node must have two CPUs, nodes must be interconnected by Gigabit and a total of four CPUs must be available. $m_{\varepsilon}\mu$ should return the alias domain included in the figure, but if the alias is not present the system must create it.

The code fragment presented in Figure 14 uses a few $m_{\varepsilon}\mu$ 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 fulfil the two property lists.

Figure 14 Creation of an aggregator domain

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

6 Application modelling

Following the same approach that we used to represent and organise physical resources, application modelling comprises the definition of a hierarchy of nodes. Like physical domains (used to represent physical resources), application entities (domains, operons, tasks, mailboxes, memory blocks and gathers) are registered in the directory service and programmer may instruct the runtime system to discover a particular entity in the hierarchy of an application component. In fact, application entities may be seen as logical resources that are available to any application component.

6.1 A modelling example

Figure 15 shows a modelling example concerning a simplified version of SIRe, a scalable information retrieval environment. This example is just intended to explain our approach; specific work on web information retrieval may be found e.g., in Brin and Page (1998) and Cho and Garcia-Molina (2002).

Each *Robot* operon represents a robot replica, executing on a single machine, which uses multiple concurrent tasks to perform each of the crawling stages. At each stage, the various tasks compete for work among them. Stages are synchronised through global data structures in the context of an operon. In short, each robot replica exploits an SMP workstation through the shared memory paradigm.

Within the domain *Crawling*, the various robots cooperate by partitioning URLs. After the parse stage, the spread stage will thus deliver to each *Robot* operon its URLs. Therefore *Download* tasks will concurrently fetch messages within each operon. Because no partitioning guarantees, by itself, a perfect balancing of the operons, *Download* tasks may send excedentary URLs to the mailbox *Pending*. This mailbox may be accessed by any idle *Download* task. That way, the cooperation among robots is achieved by message passing.

The indexing system represented by the domain *Indexing* is purposed to maintain a matrix connecting relevant words and URLs. The large amount of memory required to store such a matrix dictate the use of several distributed memory fragments. Therefore, multiple *Indexer* operons are created, each to hold a memory block. Each indexer manages a collection of URLs stored in consecutive matrix rows, in the local memory block, thus avoiding references to remote blocks.

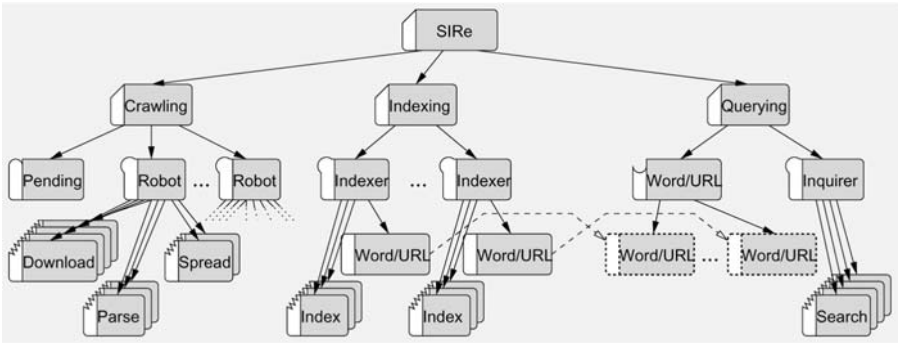
Finally, the querying system uses the disperse memory blocks as a single large global address space to discover the URLs of a given word. Memory blocks are chained through the creation of aliases under a memory block gather which is responsible for redirecting memory references and to provide a basic mutual exclusion access mechanism. Accessing the matrix through the gather *Word/URL* will then result in transparent remote reads throughout a matrix column. The querying system thus exploits multiple nodes through the global memory paradigm.

6.2 Cooperative applications

Clusters may be used to support complex information systems that integrate multiple cooperative applications. In such a scenario, any programmer should be allowed to implement an application (a system module) without knowing all details about the other applications. Distinct applications may also execute under the control of different users. In $m_{\varepsilon}\mu$, all the applications that form the system are represented by a single hierarchy. In fact, all applications running in a cluster at a specific moment will be represented by a single hierarchy of resources, even if they do not belong to the same application system, that is, apart from their cooperation level.

Distinct applications may cooperate if they are able to identify entry points to access software components running in the cluster. These entry points, in $m_{\varepsilon}\mu$, are those entities that allow message passing and remote memory access. Considering that all logical entities used to model each application are registered in the directory service and that it is easy to discover entities according to their known (public) properties, it is only necessary to announce, using mechanisms external to $m_{\varepsilon}\mu$, the way these entry points may be exploited (what kind of messages

Figure 15 Modelling example of the SIRe system



the application component is prepared to receive, which information is stored in a specific memory region, etc.).

The programming paradigms to accomplish the cooperation between applications are therefore message passing and global memory. To establish a cooperation relationship with other application, based on message passing, an application should demand the discovery of a specific domain, operon, task or mailbox, owned by that application. In the same manner, based on external block memory and memory gather identifiers, an application will be able to read and write data from other applications.

It is possible to realise more sophisticated forms of cooperation, by creating aliases whose originals belong to external application components. The creation of entities under domains, operons and memory gathers of other applications allows an application to mix its hierarchy with others. For instance, if an application creates an alias task below an external domain (of other application), it will receive a copy of any message addressed to that domain and cooperation will be feasible.

7 Mapping resources

The last step of our methodology consists of merging the two separate hierarchies produced on the previous stages to yield a single hierarchy.

7.1 Laying out logical resources

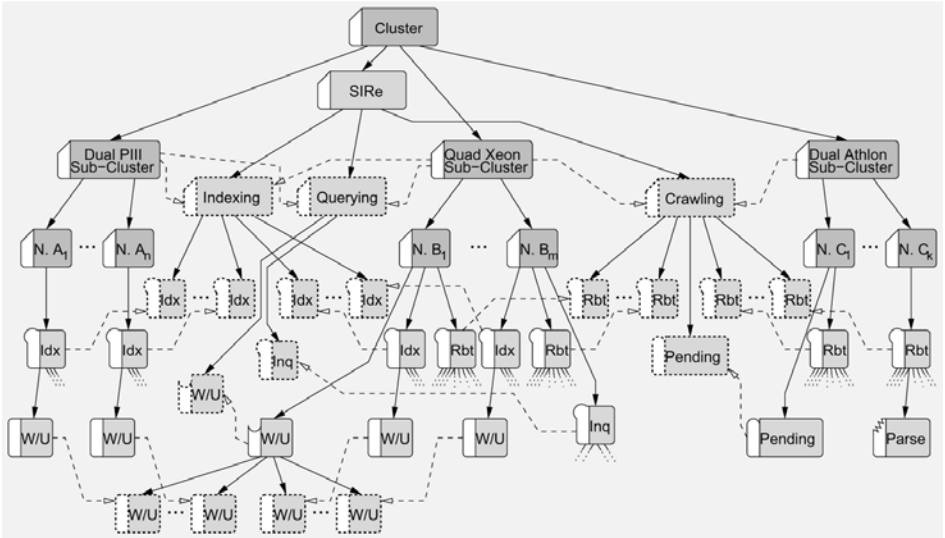
Figure 16 presents a possibility of integrating the application depicted in Figure 15 into the physical resources depicted in Figure 11.

Operons, mailboxes and memory block gathers must be instantiated under original domains of the physical resources hierarchy. Tasks and memory blocks are created inside operons and so have no relevant role on hardware appropriation. In Figure 16, the application domain *Crawling* is fundamental to establish the physical resources used by the crawling sub-system, since the operons *Robot* are automatically spread among cluster nodes placed under the originals of that alias domain.

To preserve the application hierarchy conceived by the programmer, the runtime system may create aliases for those entities instantiated under original physical resource domains. Therefore, two distinct views are always present: the programmer's view and the system view.

The task *Parse* in Figure 16, for instance, can be reached by two distinct paths: $Cluster \rightarrow Dual\ Athlon \rightarrow Node\ C_k \rightarrow Robot \rightarrow Parse$ – the system view – and $Cluster \rightarrow SIRe \rightarrow Crawling \rightarrow Robot^{(Alias)} \rightarrow Parse$ – the programmer's view. No alias is created for the task *Parse* because the two views had already been integrated by the alias domain *Robot*; aliases allow jumping between views.

Figure 16 Mapping logical hierarchy into physical



Programmer's skills are obviously fundamental to obtain an optimal fine-grain mapping. However, if the programmer instantiates application entities below the physical hierarchy root, the runtime system will guarantee that the application executes but efficiency may decay.

7.2 Dynamic creation of resources

Logical resources are created at application start-up, since the runtime system automatically creates an initial operon and a task, and when tasks execute primitives with that specific purpose. To create a logical resource it is necessary to specify the identifier of the desired ascendant and the identifiers of all originals in addition to the resource name and properties. To obtain the identifiers required to specify the ascendant and the originals, applications have to discover the target resources based on their known properties.

When applications request the creation of operons, mailboxes or memory block gathers, the runtime system is responsible for discovering a domain that represents a cluster node. In fact, programmers may specify a higher-level domain confining multiple domains that represent cluster nodes. The runtime system will thus traverse the corresponding sub-tree in order to select an adequate domain.

The code fragment in Figure 17 shows how the $m_\varepsilon\mu$ primitives may be used to launch the *Crawling* application.

Figure 17 *Crawling* application launching

```
sire = meu_newdomain(ROOT, "SIRe", NULL);
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(sire, 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]);
}
```

After discovering the location for a specific logical resource, the runtime system instantiates that resource and registers it in the local directory server. The creation and registration of logical resources is completely distributed and asynchronous.

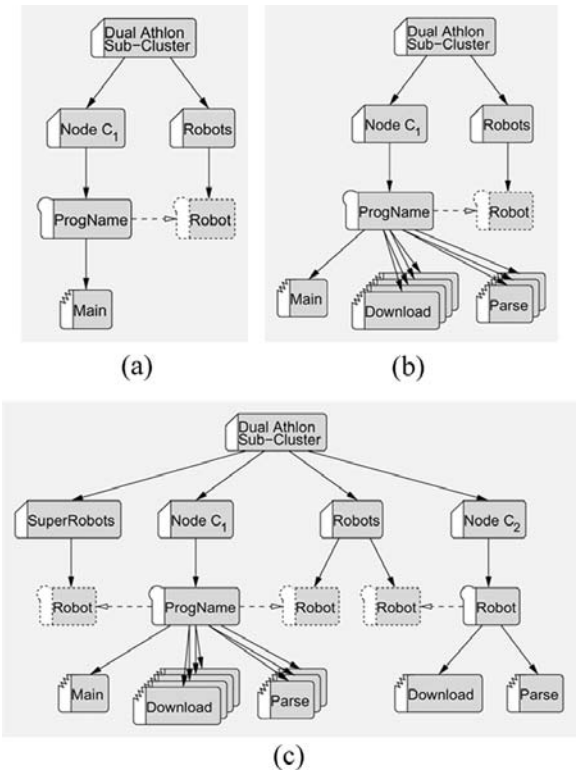
7.3 Programmer's views

In what follows, by taking as an example the evolution of a dynamic *Crawling* application, we demonstrate the $m_\varepsilon\mu$ 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.

Assuming the crawling application is launched via command line at *Node* C_1, an operon *ProgName* and a task *Main* (see Figure 18(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 (*ProgName*).

The next step, illustrated in Figure 18(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.

Figure 18 Application scaling



Finally, the configuration depicted in Figure 18(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.

8 Discussion

The integration of various communication technologies has motivated the development of some intermediate-level communication libraries. Generally, these 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. R_oCl introduces a new communication paradigm, supported by a low-level directory service, which allows to easily add communication facilities to any computational entity.

Traditionally, the execution of high performance applications is supported by powerful SSIs that transparently manage cluster resources to guarantee high availability and to hide the low-level architecture, e.g., Gallard et al. (2002). Our approach is to rely on a basic communication-level SSI (R_oCl) used to implement simple high-level abstractions that allow programmers to directly manage physical resources. $m_\varepsilon\mu$ provides high-level mechanisms for structuring applications and for mapping their components into physical resources.

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 (combining MPI and OpenMP, for instance). Some innovative approaches had also been presented, as is the case of Kelp (Fink and Baden, 1997). $m_\varepsilon\mu$, because of its mechanisms to map logical into physical resources, is another approach to the same problem. However, $m_\varepsilon\mu$ considers another level of parallelism which is not addressed by common approaches: sub-clusters.

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. $m_\varepsilon\mu$ 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 (Nieplocha et al., 1996). $m_\varepsilon\mu$ uses a similar approach, but global memory abstractions are entirely integrated with the remaining abstractions.

When compared to a multi-SAN SMP cluster, a metacomputing system is necessarily a much more complex system. Investigation of resource management architectures has already been done in the context of metacomputing, e.g., Czajkowski et al. (1998). However, by extending the resource concept to include both physical and logical resources and by integrating on a single abstraction layer

- the representation of physical resources
- the modelling of applications
- the mapping of application components into physical resources, our approach is innovative.

The message passing paradigm still remains the main choice for grid programming as attested by current ports like MPICH-G2 (Karonis et al., 2003) and other platforms to run unmodified PVM/MPI applications (Royo et al., 2000). The underlying resource oriented computation model of $m_\varepsilon\mu$ provides the necessary abstractions to accommodate the migration from cluster to multi-cluster grid enabled computing.

More recently, R_oCl has been configured for execution in grid alike multi-cluster environments, that we name virtual clusters, conforming to the deployment model used by NPACI Rocks (Sacerdoti et al., 2004), a robust and scalable toolkit used to create large scale clusters. The approach allows individual users of different administrative domains to select a certain number of nodes and a front-end, among all machines in the multi-cluster, to build and setup a customised virtual cluster. To achieve this functionality we use OpenVPN and PVFS (Figueiredo et al., 2001). OpenVPN allows communication between nodes of different private networks. PVFS introduces the concepts of shadow accounts and NFS proxies to support global file access across virtual cluster nodes.

Programmers need the power of grids to build and deploy applications that break free of single resource limitations to solve large-scale problems. Although much work has been done in the context of Globus (Czajkowski et al., 2001) to exploit grid resources, our approach uses a straight model but provides much of the operations identified in Allen et al. (2003) as key functionality for developing grid applications.

Acknowledgement

We would like to thank FCT/MCT, Portugal, for its support under contract POSI/CHS/41739/2001.

References

Allen, G., Goodale, T., Russell, M., Seidel, E. and Shalf, J. (2003) *Grid Computing: Making the Global Infrastructure a Reality*, Classifying and Enabling Grid Applications, John Wiley & Sons, Ltd., West Sussex, England.

Alves, A., Pina, A., Exposto, J. and Rufino, J. (2003) 'RoCL: a resource oriented communication library', *Euro-Par 2003*, pp.969–979.

Aumage, O., Bougé, L., Denis, A., Méhaut, J-F., Mercier, G., Namyst, R. and Prylli, L. (2000) 'Madeleine II: a portable and efficient communication library for high-performance cluster computing', *CLUSTER'00*, pp.78–87.

Baden, S.B. and Fink, S.J. (2000) 'A programming methodology for dual-tier multicomputers', *IEEE Transactions on Software Engineering*, Vol. 26, No. 3, pp.212–226.

Brin, S. and Page, L. (1998) 'The anatomy of a large-scale hypertextual web search engine', *Computer Networks and ISDN Systems*, Vol. 30, Nos. 1–7, pp.107–117.

Brune, M., Reinefeld, A. and Varnholt, J. (1999) 'A resource description environment for distributed computing systems', *Proc. 8th International Symposium on High Performance Distributed Computing*, Redondo Beach, CA, pp.279–286.

Cho, J. and Garcia-Molina, H. (2002) 'Parallel crawlers', *Proc. 11th International World-Wide Web Conference*, Hawaii, pp.124–135.

Compaq, Intel and Microsoft Corporations (1997) *Virtual Interface Architecture Specification*, Version 1.0, Technical Report, available at http://download.intel.com/design/servers/vi/VI_Arch_Specification10.pdf

Czajkowski, K., Fitzgerald, S., Foster, I. and Kesselman, C. (2001) 'Grid information services for distributed resource sharing', *HPDC'01*, pp.181–194.

Czajkowski, K., Foster, I., Karonis, N., Kesselman, C., Martin, S., Smith, W. and Tuecke, S. (1998) 'A resource management architecture for metacomputing systems', *IPPS/SPDP'98*, pp.62–82.

Figueiredo, R., Kapadia, N. and Fortes, J. (2001) 'The punch virtual file system: seamless access to decentralised storage services in a computational grid', *HPDC'01*, San Francisco, CA, pp.334–344.

Fink, S.J. and Baden, S.B. (1997) 'Runtime support for multi-tier programming of block-structured applications on SMP clusters', *International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE'97)*, Marina del Rey, pp.1–8.

Gallard, P., Morin, C. and Lottiaux, R. (2002) 'Dynamic resource management in a cluster for high-availability', *Euro-Par 2002*, pp.589–592.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V. (1994) *PVM: Parallel Virtual Machine. A Users Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, MA, USA.

Gursoy, A. and Cengiz, I. (1999) 'Mechanism for programming SMP clusters', *PDPTA'99*, Vol. IV, pp.1723–1729.

Karonis, N., Toonen, B. and Foster, I. (2003) 'MPICH-G2: a grid-enabled implementation of the message passing interface', *Parallel and Distributed Computing*, Vol. 63, No. 5, pp.551–563.

Myricom Inc. (2000) *The GM Message Passing System*, Reference Manual, available at http://www.myri.com/scs/GM/doc/gm.pdf

Nieplocha, J., Harrison, R. and Foster, I. (1996) *Advances in High Performance Computing*, Chapter Explicit Management of Memory Hierarchy, Kluwer Academic Publishers, Norwell, MA, USA.

Pina, A., Oliveira, V., Moreira, C. and Alves, A. (2002) 'pCoR: a prototype for resource oriented computing', *HPC 2002*, pp.251–262.

Royo, D., Kapadia, N. and Fortes, J. (2000) 'Running PVM applications in the PUNCH wide area network-computing environment', *Proc. 4th International Meeting VECPAR 2000*, Oporto, pp.90–95.

Sacerdoti, F.D., Chandra, S. and Bhatia, K. (2004) 'Grid systems deployment and management using rocks', *Proc. International Conference on Custer Computing, CLUSTER'04*, San Diego, CA, pp.337–345.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D. and Dongarra, J. (1998) *MPI – The Complete Reference*, MIT Press, Cambridge, MA, USA.

Notes

[1]The term ascendancy chain is used to represent the node chain from a node to the root.

[2]n and m stand for the number of nodes of sub-clusters *Dual PIII* and *Quad Xeon*.