Mapping application-level components into hierarchical systems resources

Albano Alves[1], António Pina[2], José Exposto[1] and José Rufino[1]

[1] ESTiG, Instituto Politécnico de Bragança, Apartado 134, 5301-857 Bragança, Portugal
{albano, exposto, rufino}@ipb.pt

[2] Dep. de Informática, Universidade do Minho, Campus de Gualtar, 4710-057 Braga, Portugal
pina@di.uminho.pt

Abstract

The appropriate organization of application components and their right mapping into physical resources is fundamental to fully exploit cutting edge technologies especially when hierarchical architectures are used. We present a new approach to combine application modelling with resource management. The proposed programming model allows the mixing of message-passing and global memory facilities and integrates them with the high-level abstractions provided for specifying and organizing application components and resources.

The methodologies and tools we had designed pave the way to build complex systems through the use of cooperative applications, executed simultaneously, involving multiple users, thus extending the MPMD model.

Key words: *Multi-networked cluster, resource management, multi-paradigm programming, logical-physical mapping.*

1 Introduction

Hierarchical architectures, namely clusters of SMP workstations, are popular platforms for high-performance computing. These systems exhibit multiple levels of parallelism that can be exploited to boost applications performance but, unfortunately, appropriate programming tools and methodologies are not widely available. Multi-networked clusters, comprising multiple sub-clusters and multi-technology nodes (to interconnect sub-clusters), introduce another level of parallelism making application development even more difficult.

1.1 Framework

A multi-networked SMP cluster, the platform we focus our work on, requires the ability to combine multiple low-level technologies and the multithreading programming model. In [2] we have presented our first efforts to integrate MVIA and GM low-level protocols in a multithreading environment.

A high-level programming model combining shared memory and message-passing, like the one presented in [8], sounds like the right choice to exploit such a platform. But hierarchical architectures force the programmer to take special care when defining application components. Divide and conquer techniques like those formerly used by Cilk [3] and recently adopted by Ibis [9], to run distributed supercomputing applications, may be useful.

Dynamic hierarchical architectures raise another obstacle: applications must be able to discover suitable physical resources at runtime. Dynamism may result from allowing multiple users to run their parallel applications concurrently. Therefore, applications would take advantage from resource description and allocation facilities, like those provided by CCS [5].

Some computational challenges can further difficult the task of using parallel computing; sometimes multiple applications, eventually from different users, need to cooperate. This means applications must also be able to describe logical components, in order to discover each other. That is, the MPMD model must be extended to enclose the multiple application paradigm.

Here we present a new approach for developing applications that allows to overcome all these adversities.

1.2 Our approach

Our approach, a restatement of pCoR [7], aims to accomplish the efficient and convenient organization of components used in a complex application. We propose a programming library which provides high-level mechanisms for structuring applications and uses RoCL [1], our intermediate-level communication library, to guarantee

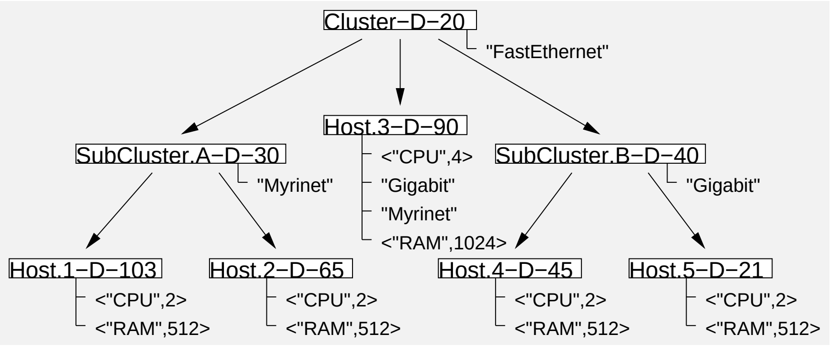


Figure 1. Entity hierarchy example.

component interoperability without compromising performance.

At the moment, we are providing five kinds of entities for application design:

- domains – used to organize (group) physical resources available in the cluster as well as software components represented by other entities;
- operons – used to exploit physical resources at the cluster node level;
- tasks – code fragments (routines), running concurrently, responsible for creating entities and sending messages;
- blocks – contiguous memory segments that may be read/written, total or partially, from any local or remote task;
- gathers – allow to reference a set of blocks using a single identifier therefore creating the notion of global contiguous memory.

Entities manipulated by applications are organized in a tree. Each entity has a global/cluster-wide identifier and some properties. Figure 1 presents a hierarchy describing a cluster with two sub-clusters and a multi-technology node.

2 Basic concepts

The domain is the main structuring element for application design. In fact, tasks and blocks are leaves of the underlying tree used to support the hierarchy that represents the cluster operation, while operons and gathers, which have not to be leaves, impose important limitations for their subtrees.

Basically, the restrictions to the chaining of entities are the following:

- tasks and blocks are always terminal nodes (leaves);
- the ascendancy chain – the node chain from the node to the root – of a task or block must contain an operon;
- the ascendancy chain of an operon must contain a domain;
- the ascendancy chain of an operon can not contain another operon;
- gathers may only have blocks as descendants.

Furthermore, the domain present in the ascendancy chain of an operon must have special properties conferring it the quality of a machine.

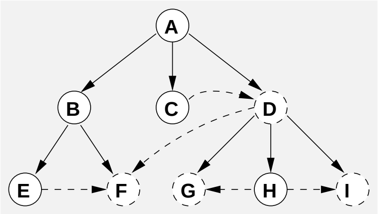
2.1 Aliases

In addition to regular ascendant-descendant relationships present in a tree, it is possible to establish origin-alias relationships. Thus, an entity may have one or more aliases and an alias may result from one or more origin 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 origin) 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 origin entities in the alias. Figure 2 presents some examples of origin-alias relationships, which are represented by dashed arrows.

It is important to note that origin-alias relationships are transitive, that is, $(Y \in aliases(X)) \wedge (Z \in aliases(Y)) \Rightarrow (Z \in aliases(X))$. However, for algorithm simplicity, $X \notin origins(Z)$. The origin of an alias that, by its turn, is an alias is designated by aliased origin in contrast to genuine origins.

Creation of aliases. 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 of a tree, thus, none of the origins of the



$$
\begin{aligned}
&descendants(A) = \{B, C, D\}\\
&ascendant(F) = B\\
&aliases(H) = \{G, I\}\\
&aliases(F) = \{\}\\
&aliases(C) = \{D, F\}\\
&origins(F) = \{D, E\}\\
&origins(D) = \{C\}
\end{aligned}
$$

Figure 2. Ascendants, descendants, origins and aliases.

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.

2.2 Properties

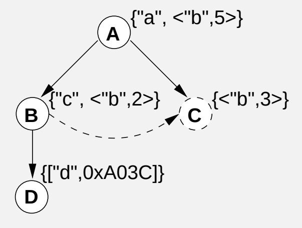
The hierarchical organization of entities is an important tool to structure the components and resources of a parallel/distributed system. However, the entity name and its global identifier may not be enough to support complex discovery operations, required to provide the dynamic (re)organization of entities. It is essential to have a mechanism to thoroughly describe resources, like the specification language used in RSD [4].

We provide a straightforward way of enhancing entity hierarchies by allowing programmers to attach the entities properties of three kinds:

- qualitative – when a single string is used to point out the presence of a particular entity characteristic;
- quantitative – when an extra numeric value is used to quantify the characteristic level (potential);
- descriptive – when an extra byte sequence is used to improve meaningfulness.

Considering the origin-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 for an entity to spread all its properties to its aliases.

Obtaining properties. Figure 3 shows the way properties are determined using a simple example. It is important to note that quantitative properties involve computation aside from union.



$$
\begin{aligned}
p(B) &= \underbrace{\{''c'', \langle ''b'', 2\rangle\}}_{owned} \cup \underbrace{\{''a'', \langle ''b'', 5\rangle\}}_{inherited} \cup \underbrace{\{[''d'', A03C]\}}_{synthesized}\\
p(C) &= \underbrace{\{\langle ''b'', 3\rangle\}}_{owned} \cup \underbrace{\{''a'', \langle ''b'', 5\rangle\}}_{inherited} \cup \underbrace{p(B)}_{shared} =\\
&= \{\underbrace{''a'', ''c''}_{qualitative}, \underbrace{\langle ''b'', 10\rangle}_{quantit.}, \underbrace{[''d'', A03C]}_{descriptive}\}
\end{aligned}
$$

Figure 3. Determining entity properties.

The ability to establish entity relationships and individually attach properties to entities along with the mechanisms of inheritance, synthesis and sharing allows to describe complex systems efficiently. With respect to figure 1, for instance, the inheritance plays an important role in alleviating the need to specify the communication technology at each node.

3 Organization of entities

All entities used to support a particular problem solving environment – application components, users, physical resources, etc – are organized in a single tree.

3.1 Base tree

The creation of entities at runtime expands a base tree, defined by the cluster administrator, which includes some domains to characterize the available hardware resources. The simplest base tree would include the cluster name as the root domain and the cluster nodes (hosts) as leaf domains.

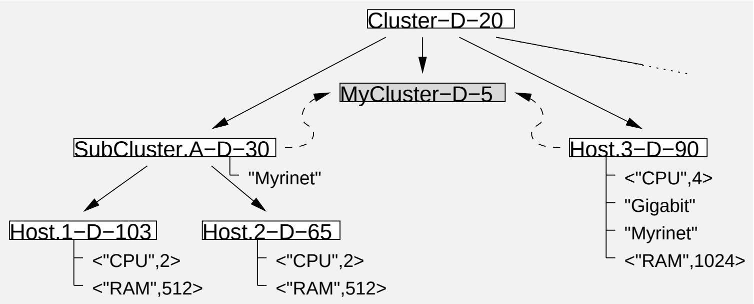


Figure 4. Aggregator domain.

Figure 1 presents an example of a base tree. At each tree node it is pointed out the entity name, the kind of entity (D - domain), the entity identifier (assigned by the intermediate-level communication library) and the list of own properties.

The base entity hierarchy is merely logical. However, the hardware it represents – the hierarchical architecture – is concrete. The hierarchy presented in figure 1, for instance, denotes the multiple levels of parallelism present in the system – multi-processor, multi-machine and multi-cluster. It is also evinced that memory access comprises multiple levels – intra-node, inter-node/intra-cluster and inter-node/inter-cluster.

Since the entities included in the base hierarchy are not created by a specific task, it was decided that their creator will be SYS. The names of the properties attached to the base tree entities can not be used by the programmer when attaching properties to entities created at runtime.

3.2 Creation of entities

The creation of domains and gathers is relatively simple. But to create entities of other kinds it is required to check the aliasing-ascendancy chains to guarantee that the restrictions presented in section 2 are preserved. Operons also obligate to check if at least one of the domains included in the aliasing-ascendancy chains possesses machine specific properties. To create an alias it is not required the existence of an operon in the aliasing-ascendancy chains of a task or block or to exist a machine domain in the aliasing-ascendancy chains of an operon.

It is important to highlight that entities are created by tasks and so it would be useful to obtain the creator of an entity. Thus, the identifier of the task that called the creation primitive is stored as a special property of the entity.

The entity ascendant is also stored as a special property, but the same is not true for descendants. Therefore, to get the descendants of an entity it is necessary to perform a distributed operation.

3.3 Aggregator domains

Considering the process of obtaining entity properties, it would be useful to have an operation that could find or create a domain ensuring the presence of a first group of properties at each node of the sub-tree rooted by that domain and ensuring that a second group of properties is held by the totality of the sub-tree nodes. The creation of such a domain comprises the discovery of domains that partially fulfil the requirements of the two groups of properties followed by the aggregation of those domains by creating an alias that takes them all as origins.

The operation to create an aggregator domain takes as arguments two lists of properties – p_1 and p_2 –, which represent, respectively, the properties that must be guaranteed at each node and the properties that may be disperse among the totality of nodes, and a node – x – from the hierarchy, where the aggregator domain must be inserted. The identifier returned by the operation concerns to an alias domain descendant from x. If the entity x guarantees by itself properties p_1 at all descendants and if p_2 is also present, then the alias is not created and the operation returns the identifier x.

Figure 4 shows an aggregator domain created to guarantee 512MB of RAM at each node and a total of five processors disperse among nodes [1].

4 Laying application components

Application components and physical resources are both represented by entities organized in a tree. Physical resources are described using domains uniquely, while applications may use domains, operons, tasks, blocks and gathers to instantiate components. The appropriate organization of components and the right mapping into physical resources allows to fully exploit cutting edge technologies. Therefore, the programmer must concentrate on two main decisions: which components an application must enclose and how these components are mapped into base tree domains.

[1] The presented tree is a fragment of that one presented in figure 1. Otherwise, the aggregator domain would not make sense.

Determining application components concerns application design/modelling, which is a topic outside the scope of this paper. Rather, we will concentrate on: *(1)* discovery of suitable resources for running an application; *(2)* instantiation of operons and tasks.

Discovering a domain that encloses particular physical resources is not a problem. In fact, application needs can be described using a list of properties which can be used to match a particular base tree domain. In figure 1, for instance, the domain *SubCluster.A* would satisfy a request for three processors interconnected by Myrinet. Of course any ascendant would also satisfy the request, but the library has the ability to return the most restrict domain. When a single domain can not provide the suitable resources for running an application, an aggregator domain can be created.

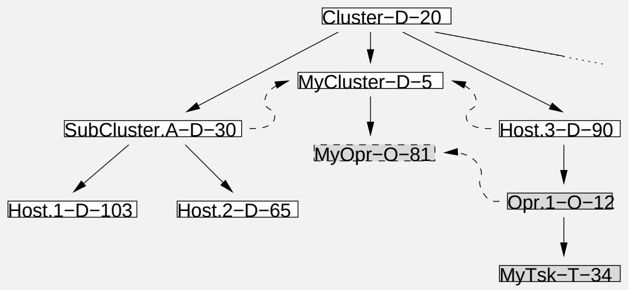


Figure 5. Operon and task creation.

The creation of operons and tasks is determinant for the exploitation of resources. By specifying a domain, the programmer is defining the host where a particular module must run. Figure 5 shows the instantiation of an operon inside an aggregator domain. The system is responsible for selecting an appropriate machine, starting up the operon and creating an alias above the specified domain. The programmer does not need to concern about machines; the aggregator domain – a virtual domain that represents the resources available for the application – will be the entry point for the application components hierarchy.

Tasks can be created in a similar way; by specifying an operon the programmer defines explicitly the place to execute a routine but by defining the domain the programmer moves the responsibility of finding an operon to the system. Figure 5 shows the instantiation of a task when operon *81* is specified.

5 Message-passing

Messages are always generated by tasks. Thus, the origin field will always concern to an identifier of a non-alias task, since aliases are not active entities.

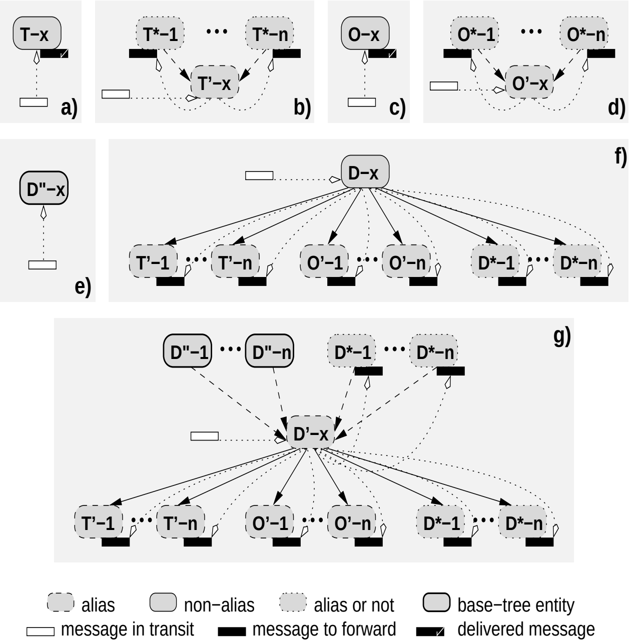


Figure 6. Possible scenarios for message delivery.

The destination of a message may be: a task, an alias of a task, an operon, an alias of an operon, a domain or an alias of a domain. In is important to note that base tree domains are not valid destinations for application-level messages.

To send a message to a task, the communication mechanism provided by an intermediate-level communication library is adequate (fig. 6-*a)*). But, if the destination is an alias of a task, the message must be forwarded to the origin, what requires extra functionality. If the alias has multiple origins, then a message copy is sent to each of them (fig. 6-*b)*). Note that origins can be aliased (and not genuine) what requires successive forwarding.

If the message destination is an operon, then any non-alias descendant task may compete for accessing the message; the operon stores the single message copy and one of the descendant tasks may access it (fig. 6-*c)*). If the operon is an alias, the proceeding is similar to the one presented for tasks (fig. 6-*d)*).

When a message is addressed to a domain, a message copy is forwarded to each of the descendants (tasks, operons or domains) and, if the domain is an alias, to all origin domains, excluding base tree domains (fig. 6-*f),g)*). Any message addressed to a base tree domain is discarded (fig. 6-*e)*).

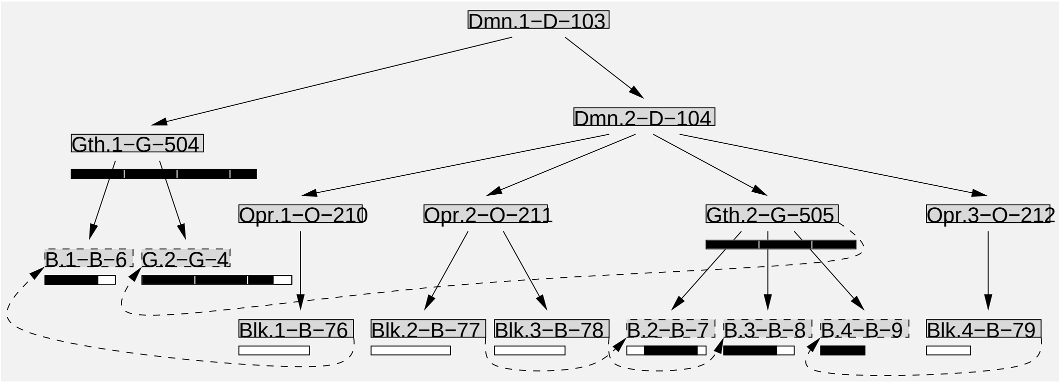


Figure 7. Block gathering example.

6 Global memory

To accomplish a compromise between the efficient utilization of cluster resources and the convenient programming of applications, some models for the distribution of data across cluster nodes and for the access to that data using one-sided communication have been developed, like global arrays [6].

Block and gather entities are used to create a virtual global storage.

6.1 Block gathering

A set of blocks can be unified through a gather by creating an alias, for each of them, above that gather (fig. 7). The primitive provided for appending a block to a gather, besides the creation of an alias, checks if the block can be appended to the target gather and updates the data used to sequence the various blocks.

A gather may collect blocks that contain the gather ascendant in their aliasing-ascendancy chains. When a block is appended to a gather, by default, a new entry $\langle start, length, id\rangle$ is created, where $start$ stands for the current size of the global memory (which is set to zero when the gather is created), $length$ stands for the block size and id stands for the block identifier. That way, the order by which blocks are appended to the gather is decisive for their sequencing.

Optionally, the place the block must occupy in the virtual global storage may be specified. It is also possible to define the fraction of the block that must be incorporated into the gather. In this case, the programmer will be responsible for filling the whole ”global address space”.

A group of blocks represented by a gather can also be integrated, at once, into another gather. In this case, it is mandatory to specify the fraction of the sequence of blocks that must be integrated, to avoid that the expansion of the first gather interferes with the positioning of other blocks in the second one.

6.2 Global memory access

To access the global memory, first of all, a program must obtain a pointer to a local chunk by providing the gather identifier and the lower and upper bound that define the desired fragment. Next it is possible to update, total or partially, the local chunk through *get* operations, which will read the required remote blocks. The reverse update is accomplished through *put* operations.

Note that the provided pointer allows to read and write local memory which is synchronized with the remote blocks through explicit *get* and *put* primitives that take advantage of low-level RDMA operations. When the program stops accessing the global memory fragment, the local pointer must be freed.

6.2.1 Optimizing access time.

We provide a unique primitive suite to access global memory, disregarding the real location of blocks; application components are expected to use global memory fragments according to localities expressed in the entity hierarchy. Anyway, some optimizations are achieved at library-level:

- if the fragment comprises a single block from the local operon or from the local host, *get* and *put* are innocuous because data can be accessed directly or through a shared memory pointer (returned by *shmget*), respectively;

- if multiple blocks from the local operon or from the local host are comprised, *get* and *put* operations will read and write data through *memcpy*, using a buffer (local chunk), rather than using RDMA operations.

7 Conclusions

We had presented a programming model that provides high-level mechanisms for structuring applications and for mapping its components into physical resources. Resources are represented by a tree which is extended at runtime to include logical application entities. By discovering tree domains that enclose suitable resources and by instantiating operons and tasks using those entry points, multiple users can simultaneously map their application components properly.

The integration of message-passing and global memory with the high-level mechanisms provided for organizing application components allows for an application to exchange data with other applications. This is particularly useful when multiple applications from different users must cooperate.

Currently, these methodologies and tools are being used to design and put into operation a scalable information retrieval environment, exploiting multi-networked clusters with nodes interconnected by Gigabit or Myrinet.

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] A. Alves, A. Pina, J. Exposto, and J. Rufino. ToCL: a thread oriented communication library to interface VIA and GM low-level protocols. In *Conputational Science - ICCS 2003*, LNCS 2658, pages 1022–1031. Springer, 2003.

[3] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, 1995.

[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] A. Keller and A. Reinefeld. CCS Resource Mangement in Networked HPC Systems. In *Heterogeneous Computing Workshop*, pages 44–56. IEEE C. Society Press, 1998.

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

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

[8] E. Speight, H. Abdel-Shafi, and J. Bennett. An Integrated Shared-Memory / Message Passing API for Cluster-Based Multicomputing. In *International Conference on Parallel and Distributed Computing and Networks*, pages 146–153, 1998.

[9] R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Ibis: an Efficient Java-based Grid Programming Environment. In *Joint ACM Java Grande - ISCOPE 2002*, pages 18–27, 2002.