Journal of Interconnection Networks
Vol. 9, Nos. 1 & 2 (2008) 1–29
© World Scientific Publishing Company

A NOVEL SOFTWARE-BUILT PARALLEL MACHINES AND THEIR INTERCONNECTIONS

MOHAMMAD MURSALIN AKON[*,‡], DHRUBAJYOTI GOSWAMI[†,§]
HON FUNG LI[†,¶], XUEMIN (SHERMAN) SHEN[*,||] and AJIT SINGH[*,**]
[*] *University of Waterloo, Waterloo, Ontario, Canada*
[†] *Concordia University, Montreal, Quebec, Canada*
[‡] *mmakon@ece.uwaterloo.ca*
[§] *goswami@cs.concordia.ca*
[¶] *hfli@cs.concordia.ca*
[||] *xshen@bbcr.uwaterloo.ca*
[**] *asingh@ece.uwaterloo.ca*

Received 26 October 2007
Accepted 23 June 2008

In this paper, we introduce *SPM* (*S*oftware-built *P*arallel *M*achines), a model to create software based virtual parallel machines. With SPM, an application developer simply selects all the required virtual parallel machines from the repository and implements the intended parallel algorithms directly without any need of complex mappings, as if the required processor interconnections are readily available. In addition, we present an implementation of the SPM model, which provides a systematic way to design new virtual machines. Our experiments show that the applications developed using the SPM model and tools give excellent performance, as compared to the applications developed using a generic communication library, such as MPI.

Keywords: Interconnection networks; Architectures of parallel computers; Cluster computing; High Performance Computing (HPC); Parallel patterns.

1. Introduction

Starting from the early age of modern computers, scientists and engineers researched and implemented a wide variety of parallel computer architectures. Due to the newly invented VLSI technology, commercial parallel computers[a] became available in early 1980s. As individual commodity processors became powerful, the majority of the commercial vendors started to build multi-processor parallel computer systems with general purpose commercial processors. As multi-processor systems gained their popularity, researchers developed a wide ranges of processor interconnections. Many of these interconnections have been implemented in real life. For example, ring, 2-dimensional mesh, fat-tree and hypercube interconnections were implemented in CDC Cyberplus, Paragon XP/S, CM-5 and Cosmic Cube parallel machines, respectively [20]. The purpose of these interconnections was to provide efficient execution platforms for different parallel algorithms, i.e., the interconnections replicate the exact communication patterns between different components/modules of the parallel algorithms. However, to be able to execute different parallel algorithms on a given hardware platform, often developers have to find out the mapping of the graph rep-

[||]Corresponding author.
[a]In this paper, we use the terms parallel computers and parallel machines interchangeably.

resenting the communication of different components of the algorithm to that's of the given processor interconnection [26].

With the current powerful general purpose computers and fast networking techniques, computer clusters provide reasonable computing speed for different complex problems with cheaper price. In practice, it can be found that the majority of the top 500 super computing machines are simply large scale computer clusters built from commercial off-the-shelf (or COTS) hardware sets [34]. Therefore, the mapping of interconnections of multi-processors architectures or communication of modules from parallel algorithms becomes difficult. The application developers have to logically create the interconnections required by the underlying parallel algorithms and, complex parallel applications mandate for complex combinations of such interconnections. As a result, application development becomes tedious and error prone.

The concept of design patterns has been used in diverse domains of engineering, ranging from architectural designs in civil engineering [4] to the design of object oriented softwares [18, 28]. Irrespective of its domain, the term *design pattern* always means the solution or a range of solutions to a frequently occurring problem. Mostly, the solutions are provided at the design level and are in the written form [18]. However, the design-level solutions may also be pre-implemented as reusable frameworks [21, 28]. In the area of parallel computing, (parallel) design patterns specify recurring parallel computational problems with similar structural/architectural or behavioral components, and their solution strategies. Examples of the structural parallel patterns are linear array, mesh, hypercube, systolic and wavefront computations, singleton pattern for single-process computation, pipeline and other processor interconnections. Some of the behavioral patters are divide and conquer, compositional framework for control- and data-parallel computation, etc.

In this paper, we propose *SPM* (*S*oftware-built *P*arallel *M*achines), a pattern-based model and tools, to construct and use parallel machines entirely in software. Unlike the research on behavioral patterns [12, 14], which deals with behavioral aspects of parallel computing, SPM focuses on the architectural or structural aspects of parallel processor interconnections or message passing of parallel modules [3, 19]. Each virtual parallel machine in SPM encapsulates the various structural attributes of a parallel pattern in a generic (i.e., application-independent) fashion. A virtual parallel machine can be considered as a specific parallel machine with its own communication, synchronization and structural primitives. Some of the structural attributes and the communication/synchronization primitives are parameterized. An application developer, depending upon the specific needs of an application, chooses the appropriate machines, supplies values for the the required parameters, and finally fills in the application specific code. The virtual machines supply most of the code that are necessary for the low-level parallelism-related issues, without burdening the application developer. Consequently, there exists a clear separation between application dependent and application independent issues (i.e., *separation of specifications or concerns*). The model of SPM is bundled with necessary tools to facilitate the design process of virtual parallel machines. A parallel machine description language (PMDL) is designed and implemented to accelerate the parallel machine design process. A translator reads the declaration of parallel machines written in PMDL and generates related C++ classes. Then an application developer provides necessary application specific code. Combined with SPM library, the final application can be deployed on a computer cluster. Thus, along with all the tools, SPM is a complete parallel programming environment (PE) [13].

The remainder of the paper is organized as follows. In section 2, we present the related preliminaries. A detailed description of the SPM model is given in section 3. We implement the SPM model in section 4. Then, we present two case studies on the usage of the SPM model and tools in section 5. Performance studies of deployed applications are summarized in section 6. Related works are discussed in section 7. Finally, we conclude our paper in section 8.

2. Preliminaries

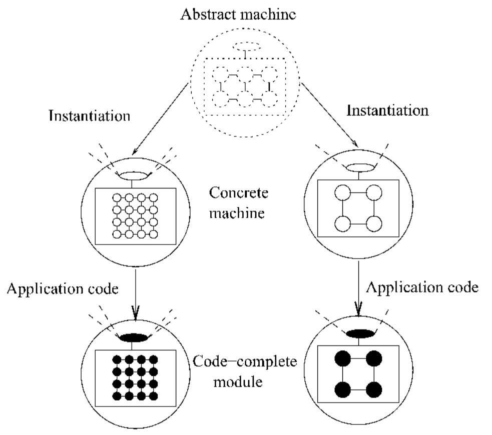
In this section, we present the main terms and concepts used in the paper. At first we give an introduction to virtual parallel machines and their components. Then, we illustrate the concept of virtual parallel machines with a real life application.

2.1. *Overview of A Parallel Machine*

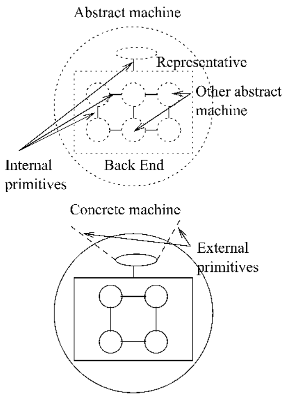
A parallel machine of SPM generically encapsulates the structural/architectural attributes of a multi-processor parallel computer or a message-passing parallel computing pattern. Various phases of an application development using SPM can be roughly illustrated from Fig. 1(a). Each parallel machine is parameterized where each parameter is associated with some architectural attribute of the associated pattern. The value of a parameter is determined during the application development phase. A machine with unbound parameters is called an *abstract (parallel) machine* or an *abstract module*[b]. An abstract machine becomes a *concrete (parallel) machine* or a *concrete module*, when the parameters of the machine are bounded to actual values. A concrete machine is yet to be filled in with application specific code. Filling in a concrete machine with application specific code results in a *code-complete parallel module* or simply a *module*. In Fig. 1(a), different parameter bindings to the same abstract machine can result in different concrete machines.

Each abstract machine (or abstract module) consists of the following set of attributes: (i) *Representative* of a machine represents the machine in its action and interactions with other machines. The initial representative is empty and is subsequently filled with application specific code during application development. (ii) The *back-end* of an abstract machine A_m can be formally represented as $\{A_{m1}, A_{m2}, \ldots,$

[b]The term *module* is used to emphasize that the parallel machines are implemented as software modules.



(a) Abstract and concrete machine, and code complete module



(b) Different components of a machine

Fig. 1. SPM machines and their components

$A_{mn}\}$, where each A_{mi} is itself an abstract machine. The type of each A_{mi} is determined after the abstract module A_m is instantiated. Note that collection of concrete modules inside another concrete module results in a (tree-structured or recursive) hierarchy. Consequently, each A_{mi} is called a *child* of A_m, and A_m is called the *parent.* The children of a machine are *peers* of one another. In this paper, the children of a machine are also referred as *computational nodes* of the associated machine. (iii) *Topology* is the logical connectivity between the children inside the back-end as well as the connectivity between the children and the representative. (iv) *Internal primitives* are the pattern or machine specific communication, synchronization or structural primitives. Interactions among the various modules are performed using these primitives. The internal primitives, the inherent properties of a machine, capture the parallel computing model of the associated parallel machine as well as its connectivity among different processors. Figure 1(b) diagrammatically shows the attributes of an abstract and a concrete 2-D Mesh machine (or parallel modules).

As is already mentioned, there are pattern specific parameters associated with some of the previous attributes. For instance, if the pattern is a Mesh, then the number of dimensions of the mesh is one parameter, and the nature of the connectivity among the nodes at the edges (i.e., toroidal or non-toroidal) is another parameter. Binding these parameters to actual values, based on the needs of an application, results in a concrete machine/module. A concrete module C_m becomes a code-complete module when: (i) the representative of C_m is filled in with application specific code, and (ii) each child of C_m is code-complete.

All of the attributes of an abstract machine/module are inherited by the corresponding concrete module as well as the code-complete module. In addition, we define the term *external primitives* of a concrete or a code complete module as the set of primitives using which the module (i.e., its representative) can interact with its parent (i.e., representative of the parent) and peers (i.e., representatives of the peers). Unlike internal primitives, which are inherent properties of a module, external primitives are adaptable, i.e., a module adapts to the context of its parent by using the internal primitives of its parent as its external primitives. While filling in the representative of a concrete module with application specific code, the application developer uses the internal and external primitives to interact with other modules in the hierarchy.

A parallel application developed using SPM is a hierarchical collection of (code-complete) modules. Each concrete machine can be considered as an instance of pattern specific virtual machine with its own communication, synchronization and structural primitives. A user fills in these virtual machines with the required code, starting bottoms-up in the hierarchy, to create the complete parallel application. The root of the hierarchy, i.e. a code-complete module with no parent, represents a complete parallel application. Each non-root node of the hierarchy represents a partial parallel application. Each leaf of the hierarchy is called a *singleton module* (and correspondingly, a *singleton abstract machine or module* for the abstract counterpart). Evidently, a singleton module contains only the representative and an empty back-end.

2.2. *Parallel Virtual Machines in Action*

We at first define a face recognition/verification problem and device an efficient solution for parallel computers. Then, evolution of the the solution is described with parallel virtual machines for a better under standing of the concepts described in previous subsection.

2.2.1. *Problem Description and Solution Approach*

Consider a parallel face recognition/verification application. Given a succession of still video images of scenes, the application needs to recognize or verify one or more persons in the scenes using a stored database of faces. Available additional information such as race, age, gender and speech may be used for narrowing the search. A solution to the problem involves the following standard steps: (1) segmentation of faces (also known as face detection) which involves detecting all the faces from an image; (2) feature extraction which involves extracting pre-defined features (e.g., eye, nose, color, etc.) from detected faces, and (3) finally recognizing or verifying a face from the stored database based on the extracted features and the additional information. There are several well known algorithms for each step. A comprehensive survey of many of them can be found in [36].

Let us consider one specific parallel solution to the problem. The initial (i.e., level 0) decomposition of the problem is rather straightforward: the problem can be easily decomposed into three concurrent modules, each involving a step mentioned previously. This initial decomposition creates a pipeline, with each pipeline stage performing one of the steps. Since the problem involves a succession of images, enhancement of performance is anticipated when the pipeline is full, i.e., all the pipeline stages are busy, and the improvement in performance over the sequential counterpart is governed by the slowest of the three pipeline stages. Assume that the rate at which images are generated is much faster than the first stage of the pipeline, that performs the face detection phase. Consequently, in this specific parallel solution, the first stage of the pipeline is actually composed of replicated copies of identical worker modules, each of which executes the specific (sequential) face detection algorithm. The replicated workers work on different independent image frames. Since the image frames are independent, the workers need not exchange any information among themselves. The faces detected by the replicated workers are streamlined (by a master/manager) and buffered for use by the next stage of the pipeline, that performs feature extraction.

The particular feature extraction algorithm used in the second pipeline stage for this specific solution uses the data-parallel paradigm. The algorithm divides each face into a 2-D grid, and concurrent workers process individual elements of the grid. Since the workers process different parts of the same face, they often need to exchange information (unlike the first stage of the pipeline). The second stage of the pipeline is actually a data-parallel grid/mesh with 2-D topology.

The third stage of the pipeline performs face detection or verification, based on the features extracted in the previous stage. There are several well known classification algorithms that can be employed at this stage [36]. However, none of these algorithms guarantees the outcome. In this specific solution, several classification algorithms are employed simultaneously on the same data, and a voting is done on the results. The majority vote wins. As a result, the third stage of the pipeline is composed of a manager and non-identical workers, where each worker performs one classification algorithm. This is unlike the first stage, which is composed of a master/manager and replicated identical workers.

So far, we have abstractly discussed about the type of each stage of the pipeline, without going into its details. Next we refine each stage of the pipeline (i.e., at level 1) by filling in its details. We refine stage 1 of the pipeline by specifying the types of the manager and workers, the maximum number of workers, and their interconnection topology. Similarly, we refine stage 2 of the pipeline by fixing the dimensions of the grid, details like toroidal or non-toroidal, and the type of each grid-element. Similarly, we can refine stage 3.

The previous discussion illustrates the hierarchy involved in application design and development in traditional parallel programming. Application design and development using SPM also follows similar hierarchy and is discussed next.

2.2.2. *An SPM Solution*

Referring to the previous example, since the initial decomposition of the problem leads to a pipeline, the pipeline machine is selected from the repository of available machines. Initially it is an abstract machine. Subsequently the abstract machine is instantiated with the following parameter: number of stages to be 3. Instantiation also involves labeling of each stage of the pipeline with a module type. In this particular case, stage 1 is labeled as the replicated master worker, stage 2 as the data-parallel mesh, and stage 3 as the non-replicated master-worker machine (Fig. 2). The concrete pipeline module is also given an identity named *Root* for the simple reason that it constitutes the root of the hierarchy.

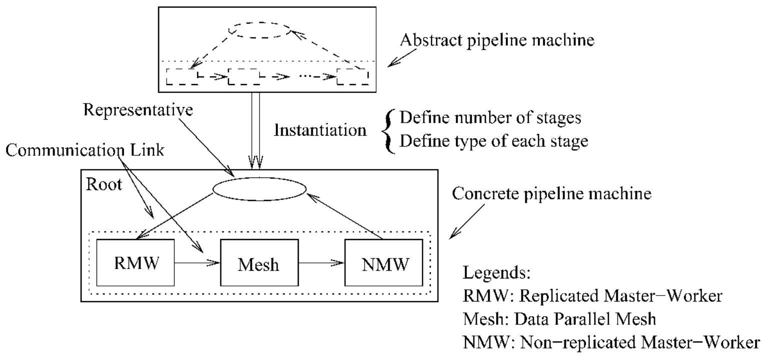

Abstract pipeline machine
Representative
Instantiation
Define number of stages
Define type of each stage
Communication Link
Root
Concrete pipeline machine
RMW
Mesh
NMW
Legends:
RMW: Replicated Master−Worker
Mesh: Data Parallel Mesh
NMW: Non−replicated Master−Worker

Fig. 2. Instantiation of a pipeline machine

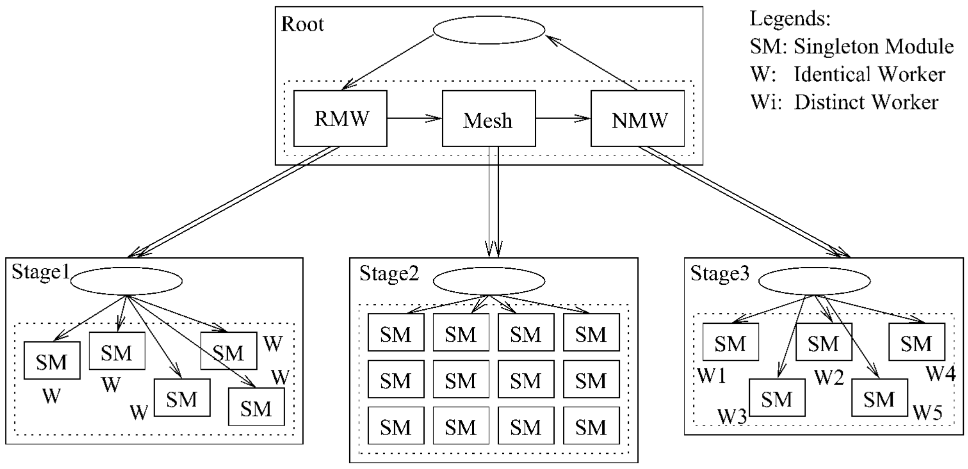


Fig. 3. Instantiation of a pipeline stages

The label of each pipeline stage is merely an abstract entity, and it needs to be instantiated recursively. Fig. 3 shows the subsequent instantiation of the pipeline stages. Stage 1, which is labeled as the (abstract) replicated master worker machine, is instantiated by binding its parameters and labeling accordingly, e.g., the number of replicated workers, type of each worker, etc. Similarly, stage 2 which is an abstract mesh machine is concretized by specifying the dimensions of the mesh (2-D), size of each dimension (a 4×3 mesh), and the type of each child. Same is the case for stage 3 of the pipeline.

The previous methodology of top-down hierarchical instantiation continues until the leaves of the hierarchy are reached. Each leaf is always an instance of *singleton machine/module*. From the definition of code-complete modules, code-completion is a bottoms-up procedure starting at the leaves of the hierarchy and proceeding towards the root (see subsection 2.1). In this example, the developers start with writing application code for the representative of the singleton modules at the leaves of the hierarchy. The next step is to write application code for the representatives of the stage 1 (i.e., the replicated master-worker machine), stage 2 and stage 3, respectively. Finally, the representative of *Root* (i.e., the pipeline machine) is filled up with proper code. The code-complete *Root* module represents the complete parallel application.

3. The SPM Model

SPM is targeted for two categories of users: parallel machine designers and application developers. Often a user falls into both the categories. A parallel machine designer construct abstract parallel machines and store them in the machine repository. It is the responsibility of the designer to ensure the correctness of all the designed machines. The application developers simply choose the proper abstract machines for the target application and go on with the development.

3.1. *Basic Elements of SPM*

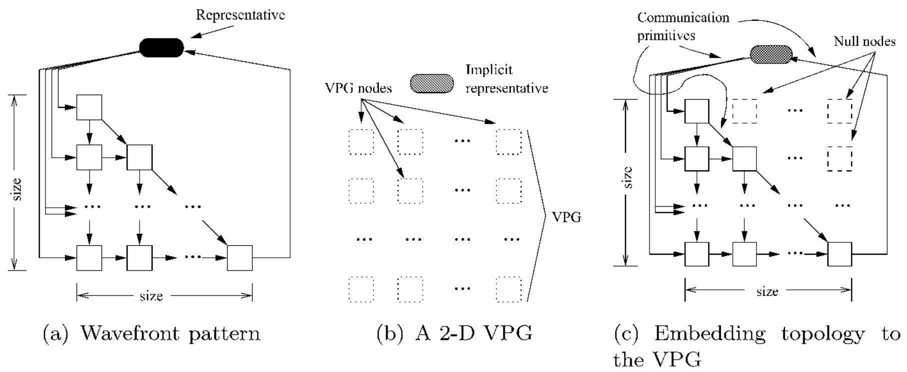
SPM provides a set of virtual k-dimensional processor grids, where $k \in \mathbb{N}$, to embed the topologies of parallel patterns. Each node of the grid is a virtual processor. Each multidimensional *virtual processor grid* (*VPG*) is equipped with its own communication and synchronization primitives. These primitives include operations for synchronous and asynchronous peer-to-peer communication, collective communication, and synchronization specific primitives. We chose to make the VPG primitives a super-set of the basic communication-synchronization primitives supported in some of the prominent parallel programming environments. Our choice is influenced by the MPI standard [24], PVM documentations [25], our experiences different pattern-based systems, and various research articles (e.g., [10]).

In the process of designing an abstract machine, the designer first chooses an architectural parallel pattern. The chosen pattern is implemented as an abstract parallel machine in SPM. For that, the designer decides about the constituents of the new machine (i.e., the back-end components, interconnection topology, primitives, etc) from the description of the pattern. Then, the designer needs to embed the topology of the chosen parallel pattern or the intended parallel machine to the existing grid topology provided by SPM, and maps each of its children (i.e., abstract machines at the back-end) into a VPG node. The mapping of the representative is implicit. Finally, the designer needs to define the communication-synchronization primitives of the new machine on top of the basic primitives provided by SPM. The Parallel Machine Description Language (PMDL), at a level of abstraction much higher than C or C++, aids in the design and implementation phases.

There are several reasons for choosing a k-dimensional grid for embedding the topology of a parallel pattern: (i) any regular or irregular topology can be embedded to a regular k-dimensional grid topology. Sufficient amount of literature already exists in this direction; (ii) From the designer's perspective, it is much easier to mentally visualize a grid, with each node of the grid clearly identified by a k-tuple. This can facilitate in formulating the mapping function(s) from the topology of the pattern to the VPG.

3.2. *Mapping a Pattern*

Fig. 4 shows a mapping where the designer wants to design an abstract *Wavefront* parallel machine, an implementation of the *Wavefront* pattern. Fig. 4(a) is the visualization of the Wavefront patten where the topology of its constituents is shown. The visualization helps the designer to make several design decisions. At first, the designer decides about the *parameters* of the machine. The structure of a Wavefront machine becomes generic if the the number of rows (or the number of columns) is considered to be a parameter rather than a constant. We name this parameter as *size*.



(a) Wavefront pattern (b) A 2-D VPG (c) Embedding topology to the VPG

Fig. 4. Mapping Wavefront pattern into a 2-D VPG

In the case of a Wavefront, the choice of a two dimensional VPG (Fig. 4(b)) is obvious because it facilitates a one-to-one mapping of the children into the nodes of the VPG. Fig. 4(c) shows such a mapping. Note that each VPG has an implicit representative node, to which the representative of the machine is mapped onto. From the figure, it can be seen that even after limiting the height and width of the VPG (to the parameter *size*), there are virtual processors to which no child of the machine are mapped onto. Those virtual processors are called *null virtual processors* or *null nodes*.

The embedding of a pattern into a VPG is complete when the associated communication, synchronization and structural primitives of the machine are defined. In Fig. 4(c), some of the channels for communication primitives are marked. Examples of some communication primitives for a Wavefront machine are: (1) receiving a message from the representative, (2) sending a message to the left neighbor, etc. Examples of some structural primitives are the operations: (1) to check whether a child is located at the last column, (2) to check whether a child is located on the topmost diagonal, etc.

In the SPM model, the topology of a pattern (as in Fig. 4(a)) is called the *topological space* ($\mathcal{TS}$). The topological space is constituted of zero or more computing nodes along with their connectivity. Connectivity among the nodes of $\mathcal{TS}$ is represented by a connectivity function $\mathcal{T}$. The mapping function, $\mathcal{M}$, maps nodes of $\mathcal{TS}$ to the virtual processors of the *VPG space* (designated as $\mathcal{VPG}$). The embedding of a $\mathcal{TS}$ into a $\mathcal{VPG}$ results in an *abstract mapped space*, $\mathcal{P}$, which is a subgraph of the $\mathcal{VPG}$ and is constituted of only the non-null nodes of VPG. Provided that $\mathcal{M}$ and $\mathcal{T}$ are already defined, it is easy to express the connectivity among the non-null virtual processors of the VPG as the composite function: $\mathcal{M}.\mathcal{T}.\mathcal{M}^{-1}$. Once $\mathcal{M}$, $\mathcal{T}$, and $\mathcal{M}.\mathcal{T}.\mathcal{M}^{-1}$ are known, it is a rather straightforward procedure to define any machine specific communication-synchronization and structural primitive in terms of a sequence of SPM provided VPG primitives.

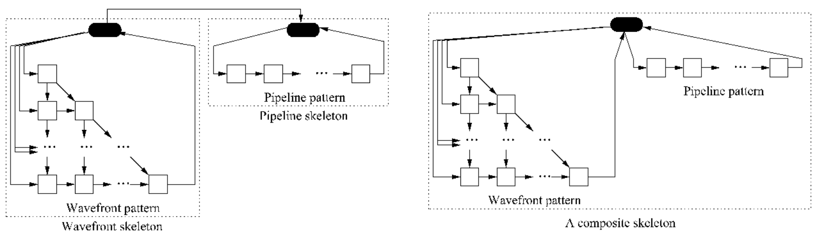
3.3. *Composition*

SPM model supports the idea of composition of abstract machine. Composition is the way to combine simple abstract machines into a complex one. In the following, we describe the motivation of incorporating the idea of composition into SPM, then present the model.

3.3.1. *Motivation Behind Composition*

A large-scale parallel application often requires a composition of multiple machines. It is more desirable to have a single composite machine rather than a collection of smaller machines. Another reason for having a composite machine is performance. Consider the example in Fig. 5(a), where a Wavefront and a Pipeline machine are shown. The output of the rightmost child of the Wavefront is sent back to the representative, the representative routes it to the representative of the Pipeline, which in-turn again routes to the first stage of the Pipeline. Composition of these two machines is shown in Fig. 5(b). It is evident from the figure that composition, in this case, reduces the number of routing requirement by 1 as compared to the case in Fig. 5(a).

Notice that the composition is different from the construction of hierarchy during instantiation. Composition is performed on abstract machines to create a composite



(a) Two separate machines (b) A composite machine

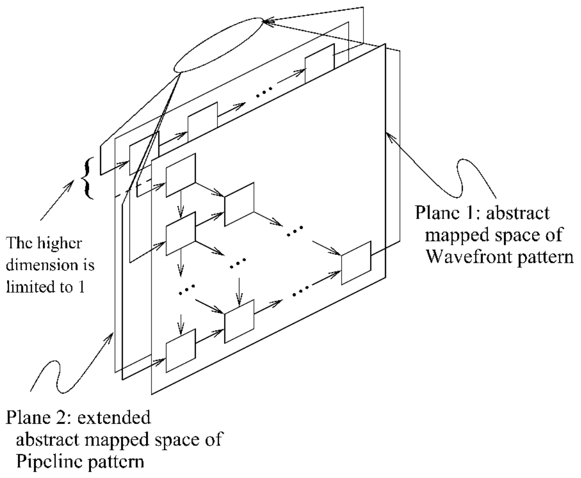
Fig. 5. Composing machines towards performance

structure. Composition is performed by the designers, whereas the machine hierarchy is constructed by the application developers. Composition may require an in-depth knowledge of the *abstract mapped spaces*, whereas the creation of the machine hierarchy does not require that. Composition of two or more machines during instantiation increases the height of the machine hierarchy during application development. On the contrary, use of composite abstract machine during application development reduces the height of the hierarchy.

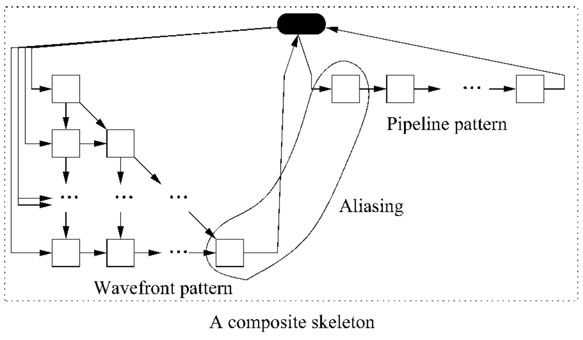
3.3.2. *Model of Composition*

Composition of machines M_i and M_j into a machine M_k results in the union of the parameters, primitives, and the abstract mapped spaces of M_i and M_j. Moreover, newer primitives may be defined for a composite machine, for instance, constituents of machine M_i are mapped onto abstract mapped spaces $\mathcal{P}_{1i}$, $\mathcal{P}_{2i}$, $\ldots$, $\mathcal{P}_{mi}$; and those of M_j are mapped onto $\mathcal{P}_{1j}$, $\mathcal{P}_{2j}$, $\ldots$, $\mathcal{P}_{nj}$; then constituents of M_k will be mapped onto $\mathcal{P}_{1i}$, $\mathcal{P}_{2i}$, $\ldots$, $\mathcal{P}_{mi}$, $\mathcal{P}_{1j}$, $\mathcal{P}_{2j}$, $\ldots$, $\mathcal{P}_{nj}$. We define the *extended mapped space* ($\mathcal{E}$) of a machine as a space which is exactly big enough to hold all the abstract mapped spaces of that machine. Formally, let us assume that a machine M consists of the abstract mapped spaces $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_N$, and the abstract mapped space $\mathcal{P}_i$ is a k_i dimensional space (i.e., result of mapping the topological space of the machine into a k_i dimensional $\mathcal{VPG}$). The extended mapped space, $\mathcal{E}$, would be of dimension $K = \max\{k_i \mid 1 \le i \le N\} + 1$.

To create the extended mapped space $\mathcal{E}$, an abstract mapped space $\mathcal{P}_i$ is extended from k_i dimension to $K-1$ dimension. While extending the dimension, the higher $K-1-k_i$ dimensions are made limited to length 1 to ensure consistency. The length of the K-th dimension of the extended mapped space, $\mathcal{E}$, is N and the extended abstract mapped space of $\mathcal{P}_i$ is placed on the i-th entry of the K-th dimension of $\mathcal{E}$. Fig. 5(b) is redrawn in Fig. 6(a) to reflect the idea of the extended mapped space. The extended mapped space, as is shown in the figure, includes the abstract mapped spaces of the Wavefront and the Pipeline machine. Among the two abstract mapped spaces, the mapped space for the Wavefront is of the highest dimension, which is two. As a result, the extended mapped space is 3-D. The first plane of the extended mapped space includes the mapped space of the Wavefront, and the second plane includes the mapped space of the Pipeline machine.


The higher dimension is limited to 1
Plane 1: abstract mapped space of Wavefront pattern
Plane 2: extended abstract mapped space of Pipeline pattern

(a) The 3-D extended mapped space



(b) Aliasing in a machine

Fig. 6. Extended mapped space and aliasing

In order to achieve more flexibility, SPM provides an *aliasing* facility. Aliasing is the way to combine two nodes from two different abstract mapped spaces of a particular machine. The idea is shown in Fig. 6(b). Aliases in SPM are expressed using an *aliasing function* (designated as $\mathcal{A}$). SPM provides two types of aliasing: (1) *fusion* paradigm, and (2) *linkage* paradigm. In the linkage paradigm, two aliased nodes are connected via a unicast communication channel and both the nodes remain as separate entities. On the other hand, in the fusion paradigm, two aliased nodes are unified into one node. As a result, that unified node becomes members of both of the abstract mapped spaces, where the original nodes belonged to. Assume that $\mathcal{E}$ is the extended mapped space of machine M, and $\mathcal{P}_i$, $\mathcal{P}_j$ and $\mathcal{P}_k$ are three abstract mapped spaces of M. Lets $P_l \subseteq \mathcal{P}_l$ where $l \in \{i, j, k\}$. The aliasing function is defined as, $\mathcal{A} : P_m \rightarrow P_n$ where $m, n \in \{i, j, k\} \wedge m \neq n$. Say, $\mathcal{A}(p_i) = p_j$ and

$\mathcal{A}(p_j) = p_k$, where $p_l \in P_l \wedge l \in \{i, j, k\}$. In the fusion paradigm of the model, those two aliases imply that $\mathcal{A}(p_i) = p_k$. However, this implication is not true for the linkage paradigm.

3.4. *Instantiation*

In SPM, during the instantiation phase, a labeling function $\mathcal{L}$ labels each child of a concrete machine, CM, with an abstract machine type (refer to section 2). In other words, labeling can be considered as specifying the types of the children inside the back-end. These children are subsequently instantiated. Hence, instantiation can be considered as a top-down process in the machine hierarchy. Suppose, in the presence of aliasing, $\mathcal{A}(p_i) = p_j$ and $\mathcal{L}(p_i) = AM_s$ where AM_s is an an abstract machine type. If the aliasing follows the fusion paradigm, then $\mathcal{L}(p_j)$ must also be AM_s. However, if linkage paradigm is used then p_j can be labeled without considering the labeling of p_i, as p_i and p_j are considered to be two separate entities.

4. Implementation

In this section, we discuss our implementation of the SPM model. We also discuss the Parallel Machine Description Language (PMDL), which is mainly targeted towards a machine designer for facilitating design and implementation of new and composite machines. An application developer also may use a small part of the PMDL, during the instantiation phase, as is discussed in the following. A formal description of PMDL is out of the scope of this paper and can be found in [2].

4.1. *Describing A Virtual Parallel Computer*

In this subsection, we illustrate the different features of SPM and the PMDL by describing the design procedure of an abstract *Wavefront* machine discussed previously (section 2), which implements the *Wavefront* pattern.

```
integer size; // The parameter for the machine
// Design of the the Wavefront machine follows:
machine Wavefront(2) { // Embedded into a 2-dimensional VPG
  LOCAL  = {           // local functions
     void init(void) { // The initialization function
        // Set the dimensions of the machine topology
        for (int i = 0; i < GetDimension(); i++)
           SetDimensionLimit(i, size);
     }
     bool non_null(const Location & loc) { // Define non-null nodes
        // loc[0], loc[1], .. indicate position of a VPG node in a
        // specific dimension, i.e. loc[0] is for the lowest
        // dimension, loc[1] is for next dimension, etc.
        if (loc[0] <= loc[1]) // if column number <= row number
           return true;       // then non-null node
        return false;         // otherwise, null node
     }
  };
  INITIALIZE = init; // Set the name of the initialization function
  MAPPING    = non_null; // Set the name of the mapping function
  PRIVATE    = { ... };  // Private primitives
  PUBLIC     = { ...};   // Public primitives
}
```

As shown in the previous PMDL code, the machine description starts with the declaration of the parameters, and subsequently the definitions of the constituents of the machine and their mapping to the 2-D VPG. The initialization function *init* (line 18) limits the length of both of the dimensions of the VPG to the parameter *size*. The function *non_null* (line 19) defines non-null VPG nodes by returning *true* values for all the nodes located on or below the upper diagonal of the VPG. The function defines the embedding of the Wavefront pattern into the VPG. It should be noted that *GetDimension* and *SetDimensionLimit* are two of the built-in structural primitives provided by SPM.

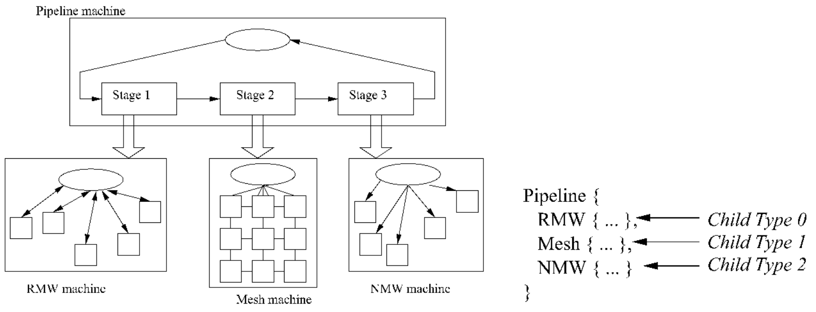
The definition of the machine is not complete unless the topology (i.e., connectivity) of the machine components is defined. In practice, the connectivity is established via defining the internal communication primitives of the machine. SPM divides the internal primitives into two categories: *private* primitives are to be used exclusively by the representative of the machine, whereas *public* primitives are inherited by the children as external primitives. In the case of the Wavefront machine, a *receive message from the child, located at the last column* is a private primitive whereas a *send message to the left peer* is a public primitive. The PMDL code for defining the private and public primitives is shown next:

```
...
machine Wavefront (2) { // Embedded into a 2-dimensional VPG
   LOCAL = { ... };
   INITIALIZE = ...;
   MAPPING = ...;
   // private primitives
   PRIVATE = {
      // Send a message to a child located at <nRow, 0>
      bool SendToNodeAt(int nRow, Msg & m) {
         Location loc;
         loc[0] = 0, loc[1] = nRow;
         return SendChild(loc, m); // VPG primitive provided by SPM
      }
      // Receive a message from the child located at <size - 1, size - 1>
      bool RecvFromLastNode(Msg & m) {
         Location loc;
         loc[0] = loc[1] = size - 1;
         return RecvChild(loc, m); // VPG primitive provided by SPM
      }
      ...
   };
   // Public primitives
   PUBLIC = {
      // COMMUNICATION PRIMITIVES
      // Send message from node <i, j> to <i, j+1>
      void SendRight(Msg &m) {
         Location loc = GetLocation();
         loc[0] = loc[0] + 1;
         SendPeer(loc, m); // VPG primitive provided by SPM
      }
      // Node <i, j> receive message from node <i, j+1>
      void RecvRight(Msg &m) { ... }
      ...
      // Receive message from the representative
      void RecvRepresentative(Msg &m) {
         RecvParent(m); // VPG primitive provided by SPM
      }
      ...
      // STRUCTURAL PRIMITIVES
      // Check if node is located at the first column
      bool IsAtFirstColumn() {
         Location loc = GetLocation();
         return loc[0] == 0; // is column number == 0?
      }
      // Check if node is located at the diagonal
      bool IsAtDiagonal() {
            Location loc = GetLocation();
            return loc[0] == loc[1]; // is column number == row number?
        }
        ...
    };
}
```

In the previous code, the machine specific private primitives (e.g., *SendToNodeAt*, *RecvFromLastNode*, etc.) and the public primitives (e.g., *SendRight*, *IsAtDiagonal*, etc.) are defined inside the language constructs *PUBLIC* and *PRIVATE* respectively. These primitives are built on top of the basic SPM provided primitives for the VPG, i.e., *SendChild*, *RecvPeer*, *GetLocation*, etc.

4.2. *Describing a Instantiation*

The PMDL also provides supports to an application developer during the instantiation phase. However, it should be noted that the application developer has to do majority of the development work using pure C++[c]. Let us consider the machine hierarchy of Fig. 7(a), decided by an application developer for the face recognition application described earlier. The root of the hierarchy is constituted by a *Pipeline* machine. The different types of machines that are available for the back-end of the Pipeline are abstract RMW (*Replicated Master and Worker*), Mesh (*Data Parallel Mesh*) and NMW (*Non-replicated Master Worker*) machines. The corresponding PMDL code is given in Fig. 7(b). In this code, RMW, Mesh and NMW are made available as first, second and third available machine types (numerically type 0, 1 and 2), respectively. A separate mapping function binds the first available type to the first stage, second type to second stage and so on. Later on, the labeled children are instantiated and thus instantiation proceeds top-down in the hierarchy.



```
Pipeline {
  RMW { ... },  <--- Child Type 0
  Mesh { ... }, <--- Child Type 1
  NMW { ... }   <--- Child Type 2
}
```

(a) Pictorial view of hierarchy (b) Expressing hierarchy in PMDL

Fig. 7. Two levels of machine hierarchy

```
// File name: Root.mc
...
```

[c]At this moment, the SPM implementation only supports C++. Other languages can be used with proper wrapper and tools, such as SWIG [33].

```
machine Pipeline (1) {
   LOCAL = {
      ...
      void label(void) {
         Location loc;
         // for all children ....
         for (loc[0] = 0; loc[0] < GetDimensionLimit(0); loc[0]++) {
            // label i-th child with i-th available type (abstract machine)
            AddLabel(&PipelineRoot, loc, i);
         }
      }
   };
   INITIALIZE = ...;
   MAPPING = ...;
   PRIVATE = ...;
   PUBLIC  = ...;
   RULE = label; // the rule for labeling
}
```

The above PMDL code shows the second step of the labeling process where stages are labeled with the available machines. Breaking the labeling process into two parts make the instantiation process more flexible. For example, the replicated and non-replicated master and worker machines could be designed as a single master and worker (MW) abstract machine in the first place. Then, if all the workers are labeled with a single machine type, the MW machine could be treated as replicated master and worker, otherwise, it would turn to a non-replicated master worker machine. In our current implementation, an instance of a machine is created by simply copying the machine description file to the project directory. If the designer prefers a different name for the instance, the new name is reflected through the name of the file.

5. Case Studies

For the sake of brevity, we choose to describe a simple but elegant parallel application. The detail development process is presented in subsection 5.1. Then in subsection 5.2, composition of X-Tree parallel machine from two simpler parallel machine is addressed.

5.1. *An Image Convolution Application*

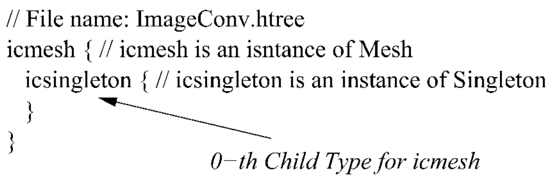
Image convolution is an important application in the domain of image processing. Here we describe a step by step procedure to develop a parallel image convolution application using SPM and related tools.

5.1.1. *Problem Description*

Image convolution is performed by applying a mask to each of the image pixels. The simplest way to make the operations parallel is to divide the whole image into columns and/or rows. Different parts of the image are distributed to different processes and each process computes the convolution of its assigned part. Unfortunately, there are dependencies among these computing processes, i.e., each process needs to exchange data with its logically neighboring processes.

5.1.2. *Instantiating Abstract Machines*

As the problem description suggests, the application demands a 2-D *Mesh* machine with identical children. The representative of the machine is mainly responsible for data partitioning (and all the file system I/O, depending on the underlying hardware constraints). The identical children of the mesh machine perform the actual convolution. In this case, each child of the mesh is a *Singleton* machine. The two-level hierarchy for this application is shown in Fig. 8. In the figure, *icmesh* is an instance of an abstract mesh machine, and *icsingleton* is an instance of an abstract singleton machine. It should be noted that *icsingleton* represents each identical child of the mesh machine.



```
// File name: ImageConv.htree
icmesh { // icmesh is an isntance of Mesh
  icsingleton { // icsingleton is an instance of Singleton
  }
}
```

Fig. 8. The two-level machine hierarchy for an image convolution application

Due to the number of available computing nodes and processors at a given the cluster, 20 sequential processes performing the convolution would yield the best performance. For an input image of size 2048×1536, we chose to divide the image among 5×4 children, each of which is a singleton module and hence performs sequential computation. Hence, we instantiate the *icmesh* machine as follows:

```
// File name: icmesh.mc
// The icmesh machine: an instance of Mesh machine
integer k = 2; // A k-D VPG to which mesh is embedded into
// Bind the different parameters for the mesh as follows:
bool fWrapping = false; // A non-toroidal mesh
machine Mesh (k) {
   // this is another way to set the size of each dimension
   LIMITS = {5, 4}; // Binding: row = 5 and columns = 4
   ...
}
```

We also specify a labeling function that labels each of the identical children of the *icmesh* machine as *icsingleton*, an instance of the abstract singleton machine. The corresponding PMDL code is shown in the following :

```
// File name: icmesh.mc
...
machine Mesh (k) {
   ...
   LOCAL = {
      ...
      void label(void) {
         Location loc;
         // for all children ....
         for (loc[1] = 0; loc[1] < GetDimensionLimit(1); loc[1]++) {
            for (loc[0] = 0; loc[0] < GetDimensionLimit(0); loc[0]++) {
               // Label each child as icsingleton (i.e., 0-th child type
               // as mentioned in the hierarchy file ImageConv.htree
               AddLabel(&Mesh, loc, 0);
            }
         }
      }
   };
   ...
   RULE = label; // the rule for labeling
}
```

At the second level of the hierarchy, the *icsingleton* machine has no parameter to bind. Moreover, a singleton machine has an empty back-end and hence it has no children to be labeled. So, instantiation of the *icsingleton* machine is a void procedure.

5.1.3. *Code-Complete Modules*

Filling in the representatives of each of the *icmesh* and *icsingleton* results in the respective code complete modules and hence the complete parallel application (refer to section 2). Before implementing the code complete modules, first we need to generate the C++ code for the machine hierarchy. The SPM-provided translator takes the hierarchy as input and generates one C++ object per machine. The developer needs to fill in the representative code for each of the objects. The *Rep* method of an object is interpreted as the representative of the corresponding machine. In the case of the image convolution application, the *icsingleton* and *icmesh* objects are generated, and subsequently they are filled in with application specific code as follows:

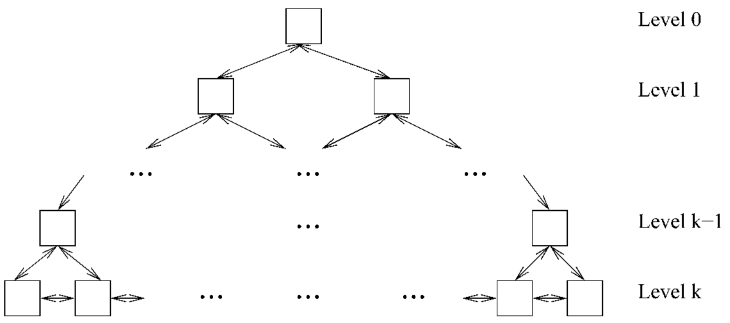
```
class icmesh : ... {
...
public:
   icmesh(...) : ... { ... }
   void Rep(void) { // Representative of mesh module
      // Fill in application specific code as follows:
      MsgImage imgMain, mask;
      // Read the image and the mask from file into imgMain
      // and mask objects
      ...
      // Now partition the image
      int nParts = GetDimensionLimits(0) * GetDimensionLimits(1);
      MsgImage * imgParts = new MsgImage[nParts];
      // Now, Divide imgMain among imgParts
      ...
      // Now send the partitions to the children
      ScatterToChildren(imgParts); // An internal primitive (section 2)
      BroadcastToChildren(mask); // An internal primitive (section 2)

      // Now gather the convoluted image partitions from children
      GatherFromChildren(imgParts);

      // Combine the convoluted image partitions into the imgMain object
      ...
      // Write the result back to a file
      ...
  }
  ...
}
class icsingleton : ... {
...
public:
   icsingleton(...) : ... { ... }
   // NEWLY ADDED METHODS (BY DEVELOPER) BEGINS
   void RecvRight(Msg &m) {
      static int * p = {+1, 0}; // Right node in a 2-d mesh
      External.RecvNeighbor(p, m); // ``External'' stands for an external
                                   // primitive. Refer to section 2.
   }
   void SendLeft(Msg &m) {
      static int * p = {-1, 0}; // Left node in a 2-d mesh
      External.SendNeighbor(p, m);
   }
   ...
   bool IsAtFirstColumn(void) {
      return External.IsAtBeginning(0);
   }
   ...
   // ADDED METHODS ENDS
   void Rep(void) { // Representative of singleton module
      // Fill in with your code
      MsgImage imageIn, imgi, mask; // The image partition and the mask
      ...
      // receive the image partition from parent
      External.RecvRepresentative(imageIn);
      // Receive the mask
      External.RecvRepresentative(mask);
      // Exchange information with neighbors
      ...
      // Now convolute
      ...
      // imageOut contains only the part of the image to be sent back
      MsgImage imageOut(imageIn.dx(), imageIn.dy());
      ...
      // Send the result back to parent
      External.SendRepresentative(imageOut);
  }
};
...
```

The previous code uses the *MsgImage* class. This class is inherited from SPM library provided *Msg* class. The *Msg* class is a generic message container used by all SPM provided built-in communication primitives. It has two abstract methods: *Marshal()* and *Unmarshal()*, which specify how the corresponding message object should be packed and unpacked at the sender and receiver, respectively. These two abstract methods need to be overwritten by an application developer, as is shown below for the *MsgImage* class:

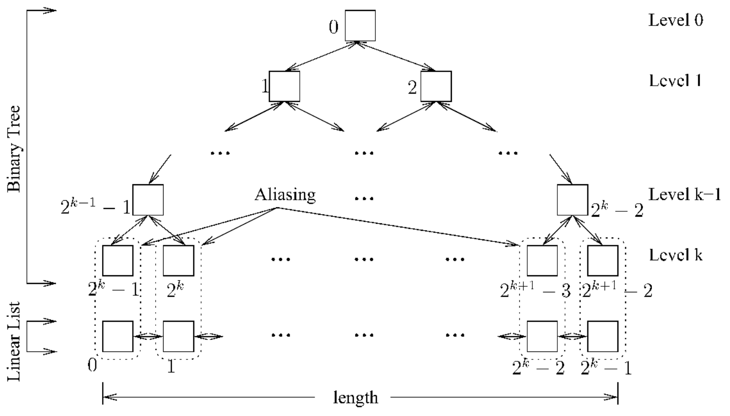
```
class MsgImage : public Msg {
   int width, height;
   int * data;
public:
   MsgImage(void) : Msg(), width(0), height(0), data(NULL) { }
   MsgImage(int _height, int _width) : Msg(), height(_height),
      width(_width) {
      data = ...;
   }
   ...
   void SetImage(const gdImagePtr im) { ... }
   int GetWidth(void) { return width; }
   // other methods
   ...
   // FOLLOWING METHODS MUST BE OVER WRITTEN
   // How to marshal  this object
   void Marshal(void) {
      // marshal width
      MarshalData(width);
      // marshal height
      MarshalData(height);
      // marshal image data
      MarshalData(data, width * height);
   }
   // How to unmarshal this object
   void Unmarshal(void) {
      // unmarshalling must be in same order of marshalling
      // unmarshal width
      UnmarshalData(width);
      // and height
      UnmarshalData(height);
      if (data) delete []data;
      data = new int[width * height];
      // we have proper memory, now unmarshal image data
      UnmarshalData(data, height * width);
   }
};
```

5.2. *A Virtual X-Tree Parallel Machine*

We design an abstract partial X-Tree [7, 16] machine by composing a *Binary Tree* and a *Linear List* machine. The partial X-Tree pattern is shown in Figure 9(a). In the pattern, each leaf of a binary tree is connected to the neighboring leaves. Assume we already have a *Binary Tree* and a *Linear List* machine, as implementations of the Binary Tree and Linear List pattern, respectively. The abstract mapped space of both of the machines are built on-top of two separate one dimensional VPGs. In Fig. 9(b), the number shown beside each of the parallel computational nodes indicates the address of the VPG node where it is mapped onto. As shown in the figure, the height of the tree in the Binary Tree pattern is a design choice, i.e., a parameter for the Binary Tree machine. Similarly, the length of the list is a parameter for the Linear List machine. In PMDL code, those two parameters are represented as *height* and *length*, respectively.



(a) The X-Tree pattern



(b) Composing Binary Tree and Linear List machines

Fig. 9. Designing X-Tree machine through composition

Fig. 9(b) also shows the aliases required to compose those two machines to design the targeted X-Tree machine. If the height of the tree is $k+1$, i.e., the highest level is k, the leaves would have addresses from 2^k-1 to $2^{k+1}-2$. To compose a Linear List with a Binary Tree, a Linear List must be of length $2^k (= (2^{k+1} - 2) - (2^k - 1) + 1)$. Node i of the abstract mapped space of Binary Tree should be aliased with node j of the abstract mapped space of Linear List, where $i = 2^k - 1 + j$ and $0 \leq j < 2^k$. Following PMDL code reflects this idea.

```
integer height; // from Binary Tree machine
integer length = integer(pow(2, height - 1)); // if height=k+1, length=2^k
machine BinaryTree(1) { // Binary Tree
   ...
}
machine LinearList(1) { // Linear List
   ...
}
alias {
   LOCAL =
      void doAlias(void) {
         for (int j = 0; j < length; j++) { // 0 <= j < 2^k
            Location lBT, lLL;

            // for node from binary tree: 2^k - 1 + j = length - 1 + j
            lBT[0] = length - 1 + j;
            // for node from linear list: j
            lLL[0] = j;
            AddAlias(&BinaryTree, lBT, &LinearList, lLL);
         }
      }
   }
   RULE = doAlias; // rule for aliasing
}
```

6. Discussion on Deployment Efficiencies

In this section, we describe different performance related studies of applications deployed using SPM. We also discuss our findings from our small scale usability studies.

6.1. *Performance Studies*

We developed several applications using SPM. Here, we present one of the representative case. We divide the discussion into two subsubsections. At first, we describe the problem in hand and the solution approach. Then, we describe our findings.

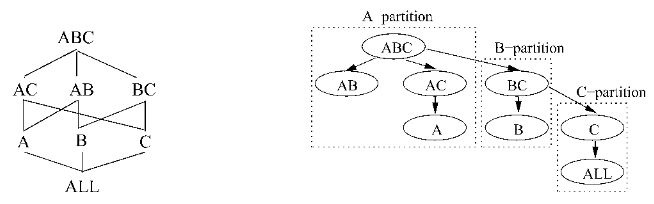
6.1.1. *Problem Description*

The decision support system (DSS) needs analytical data to have a comprehensive view about the performance of the enterprise. Often queries on such analytical data are complex and require multi-dimensional view of the enterprise data. Codd et al. coined the term On-Line Analytical Processing (OLAP) which creates, manipulates, animates and synthesizes information from Enterprise Data Models (EDM) [11]. An OLAP application usually analyzes a huge amount of data. On the contrary, a user would expect to have a real-time performance from the system. As a result, speed is a primary goal in this class of applications [11]. To make interactive analysis, OLAP databases usually pre-compute various aggregates on various combinations of attributes, often in the form of data cubes. However, speed is still a critical factor for this pre-computation as it affects how often the aggregates are revised.

Several techniques have been proposed to speed up the data cube computational procedure [1]. Recent research efforts demonstrate that parallel computation of the data cube is the most effective solution [15].

A data cube of raw data set R with d attributes (denoted as $D_1, D_2, \ldots D_d$) is composed of 2^d different views. Fig. 10(a) shows a *lattice* of a data cube with attribute A, B, C and D. Both control and data parallel [15] paradigm can be used to compute a data cube. In this paper, we consider a data parallel approach where R is partitioned among p parallel computing entities. Each entity computes all 2^d views considering only locally available data. A merge (reduction or gather) on the locally computed data cubes results in the final data cube, DC on entire R [15]. Here, we assume that $|D_1| \geq |D_2| \geq \ldots \geq |D_d|$, where $|D_i|$ is the cardinality of D_i. Denote *D_i-partition* as the set of views starting with D_i (refer to Fig. 10(b)). The computation of data cube can be expressed as,

```
for $i$ = 1 to $d$ do
    1. Partition the data on attribute $D_i$ using Sample Sort
    2. Compute local $D_i$-partition
    3. Marge local $D_i$-partitions to compute global $D_i$-partition
```



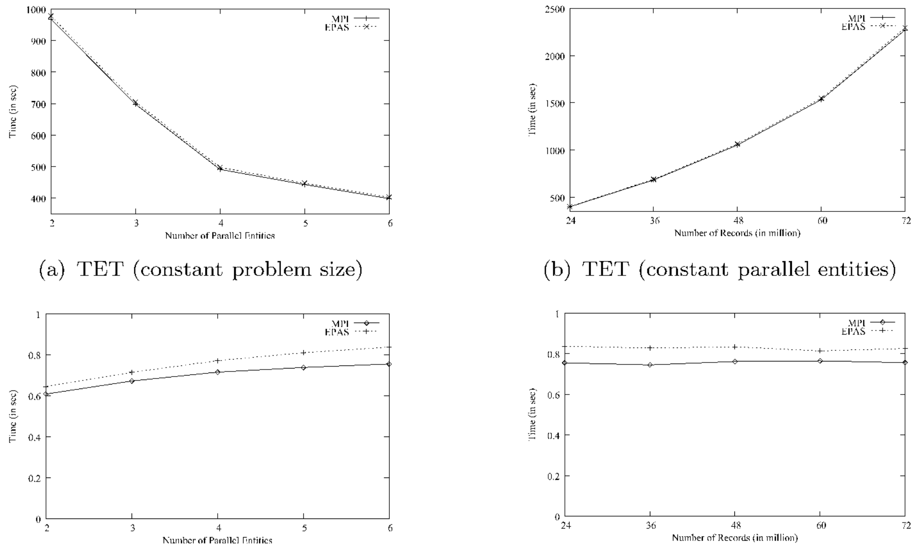
(a) Data cube lattice (b) Data cube partitions

Fig. 10. Computing data cube

Chen, et al., has proposed an Adaptive Sample Sort algorithm that partitions a data set, keeping load balancing in mind [15]. The algorithm sorts the initial data set X_1, X_2, $\ldots X_p$, distributed over processes P_1, P_2, $\ldots P_p$ to X_1', X_2', $\ldots X_p'$ which is globally sorted over the dimensions D_i, D_{i+1}, $\ldots D_d$ for D_i-partition. The algorithm chooses pivot points (for partitioning) through a collaborative regular (over) sampling procedure [22]. Computation of local D_i-partition by a process is a sequential operation without any collaboration. Finally, the merge operation is assigned to a dedicated process (P_0) so that other processes can go on with remaining computations.

6.1.2. *Performance Related Issues*

We have developed the data cube computing applications with the assumption that a parallel process can hold all its local data (consisting of 4 attributes) within the volatile memory. Chen, et al., addressed the issue on performance where this assumption is not valid [15]. To run the application, we use a dedicated cluster with seven nodes. Each node is equipped with dual Pentium III 1 GHz processors, 512 MB memory, local SCSI hard drive and a connection to other nodes through a Gigabit switch. We develop two sets of test cases: (1) variable number of parallel entities, keeping the input size fixed and (2) variable input size, keeping the number of parallel entities fixed. We develop the application using both MPI and SPM to have a better understanding. For all the readings, we consider the average of 25 runs. It should be noted that our SPM library is built as a thin layer on top of MPI-2 [24].



(a) TET (constant problem size) (b) TET (constant parallel entities)

(c) EST (constant problem size) (d) EST (constant parallel entities)

Fig. 11. Effects on performance

Fig. 11(a) and Fig. 11(b) show the total execution time (TET) of the applications developed for both the test cases. It can be seen that SPM applications are doing slightly poorer than the MPI applications. As the SPM run-time system is developed on top of MPI-2, this slowdown is very much expected. To have a better understanding, we divide TET into two segments: (1) environment setup time (EST) is the time to setup the parallel environment (for example, creating the processes, etc) and (2) actual computing time (ACT) is the time to compute the data cube. From Fig. 11(c), it can be seen that EST increments with the increasing number of parallel modules (as well as with more complex architecture of the application). This increment is faster in case of SPM applications. Fig. 11(d) shows that with constant architecture of the parallel application, EST remains fairly constant (except, the effect of the non-determinism). Note that EST takes place only once during the life-time of the application and hence, for an application with long life-time, EST has negligible effect.

The ACT of an SPM application also faces a slowdown. This is mainly due

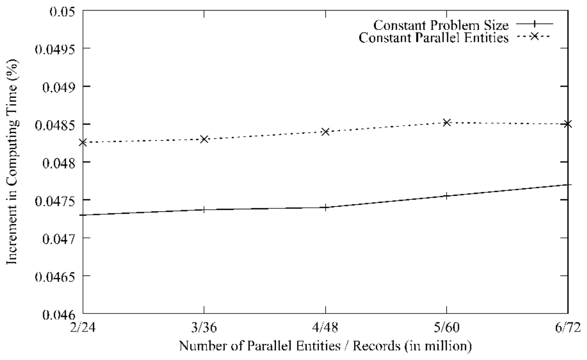


Fig. 12. Increment in computing time

to the generalized communication functions. Though SPM tries to optimize the performance on each *Msg* object by using some rules, it may not produce result that is as optimal as an application, developed using MPI directly. Fig. 12 shows the increment of ACT of SPM applications with respect to MPI applications for both the test cases. For this problem, both curves have very small slope and first test case produces steeper curve than the second (still all readings are less than 0.05%). Note that, increment of ACT depends on the behavior of the application as well as the way *Msg* objects are created. As parallel systems usually are built with an implementation of MPI, our SPM library is readily usable. An implementation directly on top of Sockets [32] is expected to give better performance, but may mandate for a few changes in (firewall) configurations.

6.2. *Usability Studies*

We acknowledge that due to constraints on available resources, our usability test is very limited. However, the finding may put some light on the greater picture. To conduct our usability tests, we chose a group of twelve students at Concordia University, enrolled in an introductory graduate-level course on parallel and distributed computing (COMP6281). Students were asked to compare their experiences with MPI and SPM. The study pointed out five important points:

(1) The learning curve for the SPM model is higher than the MPI model. On the average, the time to learn the SPM model and the PMDL is approximately 30% more than that of the MPI model and its APIs.
(2) Developing parallel applications is significantly easier and less time consuming, if the required abstract machines already exist in the repository. In the case of SPM, it took approximately 50% less time and coding effort as compared to MPI.
(3) Approximately 70% of the saving in development effort was due to lower debugging effort which follows from the correctness of the machines designed with SPM.

(4) The SPM system becomes more beneficial with increased complexity of the given problem, i.e., if the problem structure is simple and the required abstract machines are not available in the repository, it is better to use MPI. Otherwise, using SPM gives better development throughput.
(5) The object-oriented interface and machine specific primitives for communication and synchronization with eloquent names are easier to use as compared to the generic primitives provided by MPI.

7. Related Works

In absence of more competitive works, we present a comparison of SPM with other pattern-based systems. However, all the previous works were developed based on the behavioral aspect of parallel computation. On the other hand, concentration of our work is on the architectural aspect of parallel computation. Some of the earlier systems include *Frameworks* [29], *Enterprise* [27], *Code2* [8], and *HeNCE* [8]. Some of the recent systems are *Tracs* [5], *DPnDP* [31], *COPS* [23], *ASSIST* [35] and eSkel [6]. Another stream of research works explored the use of parallel design patterns for substituting explicit parallel programming by a variety of pre-packaged parallel algorithmic forms, popularly known as *algorithmic skeletons*. Algorithmic skeletons are described as higher order polymorphic functions and are, in practice, realized using various functional and logic programming languages [9,12,14,17]. A comprehensive survey of some these approaches can be found in [9,29,30].

In this paper, we point out several important differences between the MPI model for application development and SPM model for designing machines. By now, it may be clear that the VPG (with the support of *null nodes* and composition) of SPM is more flexible than the linear *rank* or solid recti-linear cartesian topology of MPI. Besides, while designing the primitives for the VPG, we kept the multi-processor architecture and commonality in parallel algorithms in our mind. As a result, from a machine designer's perspective, the basic SPM primitives (for example, *SendChild*, *SendRepresentative*, etc.) are more meaning full than the primitives of MPI (for example, generic *send*s of MPI).

The significance of extending a given library of parallel patterns has been recognized for about two decades now. Early systems such as *Frameworks* [29], *Enterprise* [27], and *HeNCE* [8] do not have any support for extensibility. The next line of parallel design pattern-based systems like *Tracs* [5], *DPnDP* [31], *COPS* [23] and *ASSIST* [35] have some support for extensibility. However, the support is often quite limited in several ways. For example, in *Tracs* [5], an application is developed in two phases. In the first phase, application specific messages are modeled and then tasks are designed using ports and modeled messages. In the second phase, the logical structure among tasks is mentioned through a graphical interface. Besides, *Tracs* supports design of parallel logical structures, i.e. patterns, where message models just have names without any specification of the message structure. However, those logical structures are static in nature with fixed number of nodes, topology and message identifiers. On contrary, SPM allows design of parametric machines and use of machine specific communication primitives with any kind of messages.

Support for extensibility in *DPnDP* [31] is also limited. It provides a platform independent class library, called *NodeLayer*. This library essentially hides the peculiarities of different message passing libraries. It provides the programmer a common interface for message passing between *ports* of different parallel computing *nodes*. Except the abstraction of nodes and ports, *DPnDP* does not facilitate the pattern design procedure in any other way. On the other hand, the SPM model does not only provide a higher level abstraction of parallel entities of a pattern but also provides primitives to create the interconnections. Moreover, while using a *DPnDP* pattern, an application developer needs to be aware of the ports and nodes associated with that pattern. On contrary, in case of SPM, an application developer can only perceive the machine specific components (representative, back-end, primitives, etc.) designed into the machine and every other details are kept hidden.

The COPS [23] system is more flexible in the way that it allows the design of new patterns. A designer can design a pattern template, possibly with several parameters. A parameter in COPS can take one of several values, listed by the pattern designer. The designer needs to write code for all possible combinations of values that the parameters can have. This, in tern, helps the COPS system to generate optimized code for a particular set of parameter values. However, it requires greater effort to write patterns with increasing number of parameters as well as parameter values. Moreover, COPS does not have a formal model, facilitating the pattern design procedure. It should be noted that COPS supports composition neither at pattern design phase nor at application development phase. As a result, if a developer needs a composite pattern, it has be designed from scratch, even though the required smaller patterns are available in the pattern repository.

The ASSIST [35] model allows three kinds of topology among parallel modules (*parmods*) i.e., *multi-dimensional array*, *none* (they work independently of each other) and *one* (sequential component with features like non-determinism, etc.). The *multi-dimensional array* topology can easily express data parallelism whereas the *none* topology can easily express independent task parallelism. But, in real life, a parallel application is a complex composition of both of them. In ASSIST, it is difficult to describe such complex compositional structures. Although ASSIST model allows a composed module to be a part of another bigger module, it does not allow nesting of parmods. Consequently, if a problem requires a parmod to be parallelized further, a re-design of the whole parmod is mandatory. Moreover, this model also lacks the support of parametric patterns. So, the application developer either needs to design her solution space conforming to the restricted structure of the available patterns (in the repository) or needs to design new patterns according to the needs of her application.

The eSkel [6] model uses the parallel programming model of MPI [24], and relies on services from MPI. In the eSkel project, each parallel pattern is realized as a skeleton. A skeleton, along with the attributes and functionalities, is presented through a library of a set of C functions. An MPI programmer may simply learn a new skeleton library and start using it. The dependency of eSkel to MPI makes it limited to the MPI users and systems. Moreover, the skeleton library does not provide encapsulations between different skeletons and other skeleton related functions. Besides, the eSkel model has no notion of extensibility where the last reported eSkel library includes only a few skeletons. In contrary, the SPM model does not rely on any other programming model and can be implemented on different parallel environments, ranging from sockets to MPI. Though the SPM repository includes a large number of parallel machines, addition of new machines is trivial. The modular organization of the machines provides encapsulation which also hides the complex details of the underlying platform.

The regular topologies found in the domain of parallel computing covers so many number of problems that researchers started to think about specific architectures that support those regularities. This resulted in architectures with specific processor topologies, for example processors connected in mesh or hypercube topology. But a computation of one pattern runs inefficiently on an architecture that follows a different pattern. Moreover, most of the parallel applications are compositions of more than one patterns. As a result, researchers found a way around to embed the patterns in software components. The SPM model is partial in favoring the designers to describe such a regular structures, without compromising the generality to express an arbitrary structure / topology. As a pattern / machine design tools none of the previous systems favors the designer to design those regularities, found in the parallel patterns.

8. Conclusion

In this paper, we have introduced SPM model with our implementation (i.e., PMDL). With SPM, a parallel application developer simply chooses proper virtual parallel machines from the repository, and implements the parallel algorithms directly on the the chosen virtual machines. Though our repository of virtual parallel machine is rich, an architecture designer can easily add a new or a composite virtual machine using PMDL by following some systematic steps. Our implementation is platform independent, and provides modular (i.e., object-oriented) interface to the application developer. We have found that applications deployed using SPM give easier development experiences, and excellent execution performances, even though the run-time library is built on top of MPI.

Currently we are investigating the issues of *performance modelling and profiling* for SPM machines. *Synchronous slicing*, a method to extract the communication synchronization behavior of a given application, is of particular interest. We are working on the issues of *static and dynamic optimizations* and *fault tolerance* aspects of applications developed using SPM. We are also in the process of developing a graphical user interface (GUI) to make the parallel application development process easier. Our current research will be reported in our future works.

Acknowledgment

This research has been financially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

1. Sameet Agarwal, Rakesh Agrawal, Prasad M. Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. On the computation of multidimensional aggregates. In *In 22nd Int. Conf. Very Large Databases, VLDB*, pages 506–521, 1996.
2. Mohammad Mursalin Akon. A model for composible and extensible parallel architectural skeletons. Master's thesis, Concordia University, Canada, 2004.
3. Mohammad Mursalin Akon, Dhrubajyoti Goswami, and Hon Fung Li. A model for designing and implementing parallel applications using extensible architectural skeletons. In *The Eighth International Conference on Parallel Computing Technologies*, pages 367–380, Russia, 2005.
4. Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, New York, USA, 1977.
5. Alberto Bartoli, Paolo Corsini, Gianluca Dini, and Cosimo Antonio Prete. Graphical design of distributed applications through reusable components. *IEEE Parallel and Distributed Technology*, 3(1):37–50, 1995.
6. Anne Benoit, Murray Cole, J. Hillston, and S. Gilmore. Flexible skeletal programming with eSkel. In *EuroPar 2005*, Lisbon, Portugal, 2005.
7. Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-Tree: An index structure for high-dimensional data. In *In 22nd International Conference on Very Large Databases*, pages 28–39, San Francisco, U.S.A., 1996. Morgan Kaufmann Publishers.
8. J. C. Browne, S. I. Hyder, J. Dongarra, K. Moore, and P. Newton. Visual programming and debugging for parallel computing. *IEEE Parallel and Distributed Technology*, 3(1), 1995.
9. D. K. G. Campbell. Towards the classification of algorithmic skeletons. Technical Report YCS 276, Department of Computer Science, University of York, 1996.
10. Fan Chan, Jiannong Cao, and Yudong Sun. High-level abstractions for message passing parallel programming. *Parallel Computing*, 29:1589–1621, 2003.
11. E. F. Codd, S. B. Codd, and C. T. Smalley. Providing OLAP to user-analysts: An IT mandate. Technical report, E. F. Codd and Associates, CA, 1993.
12. Mure Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, Massachusetts, 1989.
13. Pasqua D'Ambra, Marco Danelutto, Daniela di Serafino, and Marco Lapegna. Advanced environments for parallel and distributed applications: a view of current status. *Journal of Parallel Computing*, 28:1637–1662, December 2002.
14. J. Darlington, A. J. Field, and P. G. Harrison. Parallel programming using skeleton functions. In *Lecture Notes in Computer Science*, volume 694, pages 146–160, Munich, Germany, June 1993.

15. F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin. Parallelizing the data cube. *Parallel and Distributed Databases*, 15(3):219–236, May 2004.
16. Alvin M. Despain and David A. Patterson. X-Tree: A tree structured multi-processor computer architecture. In *In 5th annual symposium on Computer architecture*, pages 144–151, 1978.
17. I. Foster and R. Stevens. Parallel programming with skeletons. In *ICPP'90*, 1990.
18. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addision-Wesley Publishing Company, New York, USA, 1994.
19. D. Goswami, A. Singh, and B. R. Preiss. From design patterns to parallel architectural skeletons. *Journal of Parallel and Distributed Computing*, 62(4):669–695, 2002.
20. Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing.* Addison Wesley, 2003.
21. Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.
22. Xiaobo Li, Paul Lu, Jonathan Schaeffer, John Shillington, Pok Sze Wong, and Hanmao Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10):1079–1103, 1993.
23. Steve MacDonald, Duane Szafron, Jonathan Schaffer, and Steven Bromling. From patterns to frameworks to parallel programs. *Parallel Computing*, 28(12):1663–1683, 2002.
24. MPI Forum. Message passing interface forum, 2007.
25. PVM. Parallel virtual machine, 2007.
26. Michael J. Quinn. *Parallel computing: Theory and Practice.* McGraw-Hill, Inc, New York, NY, USA, 1993.
27. Jonathan Schaeffer, Duane Szafron, Greg Lobe, and Ian Parsons. The enterprise model for developing distributed applications. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(3):85–96, 1993.
28. D. C. Schmidt. ACE: an object-oriented framework for developing distributed applications. In *In 6th USENIX C++ Technical Conference*, Cambridge, Massachusetts, 1994.
29. A. Singh, J. Schaeffer, and M. Green. A template-based tool for building applications in a multicomputer network environment. In *Parallel Computing 89*, pages 461–466, 1989.
30. A. Singh, J. Schaeffer, and M. Green. A template based approach to generation of distributed application using a network of workstations. *IEEE Transaction of Parallel and Distributed Systems*, 2(1):52–67, 1991.
31. Stephen Siu and Ajit Singh. Design patterns for parallel computing using a network of processors. In *6th International Symposium on High Performance Distributed Computing (HPDC '97)*, pages 293–304, Portland, OR, August 1997.
32. W. Richard Stevens. *Advanced Programming in the UNIX(R) Environment.* Addison-Wesley Professional, Boston, MA, second edition, 2005.
33. SWIG. Simplified wrapper and interface generator, 2007.
34. Top500. Top 500 supercomputer site, 2007.
35. Marco Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, 2002.
36. W. Y. Zhao, R. Chellappa, A. Rosenfeld, and P. J. Phillips. Face recognition: A literature survey. Technical Report CAR-TR-948, University of Maryland, MD, USA, October 2000.