

From Design Patterns to Parallel Architectural Skeletons

Dhrubajyoti Goswami, Ajit Singh, and Bruno R. Preiss

*Department of Electrical and Computer Engineering,
University of Waterloo, Waterloo, Ontario
Canada, N2L 3G1*

E-mail: {dgoswami,asingh,brpreiss}@uwaterloo.ca

The concept of design patterns has been extensively studied and applied in the context of object-oriented software design. Similar ideas are being explored in other areas of computing as well. Over the past several years, researchers have been experimenting with the feasibility of employing design-patterns related concepts in the parallel computing domain. In the past, several pattern-based systems have been developed with the intention to facilitate faster parallel application development through the use of pre-implemented and reusable components that are based on frequently used parallel computing design patterns. However, most of these systems face several serious limitations such as limited flexibility, zero extensibility, and ad hoc nature of their components. Lack of flexibility in a parallel programming system limits a programmer to using only the high-level components provided by the system. Lack of extensibility here refers to the fact that most of the existing pattern-based parallel programming systems come with a set of pre-built patterns integrated into the system. However, the system provides no obvious way of increasing the repertoire of patterns when need arises. Also, most of these systems do not offer any generic view of a parallel computing pattern, a fact which may be at the root of several of their shortcomings. This research proposes a generic (i.e., pattern- and application-independent) model for realizing and using parallel design patterns. The term “Parallel Architectural Skeleton” is used to represent the set of generic attributes associated with a pattern. The Parallel Architectural Skeleton Model (PASM) is based on the message-passing paradigm, which makes it suitable for a LAN of workstations and PCs. The model is flexible as it allows the intermixing of high-level patterns with low-level message-passing primitives. An object-oriented and library-based implementation of the model has been completed using C++ and MPI, without necessitating any language extension. The generic model and the library-based implementation allow new patterns to be defined and included into the system. The skeleton-library serves as a “framework” for the systematic, hierarchical development of network-oriented parallel applications.

Key Words: design patterns in parallel computing; parallel programming environments; skeleton-based parallel programming; high-performance computing models; software tools for parallel programming

CONTENTS

1. *Introduction.*
2. *The Architectural Skeleton Model.*
3. *An Object-Oriented Implementation.*
4. *Experiments and Performance Results.*
5. *Comparison with Related Work.*
6. *Conclusion.*

1. INTRODUCTION

The term *design pattern* has been extensively used in the context of object-oriented software design. Patterns in this context describe strategies for solving recurring design problems in systematic and general ways [11]. Similar ideas are being explored in other disciplines of computing as well. For instance, ACE (the Adaptive Communication Environment) [27] is an object-oriented toolkit that implements various network-level patterns to simplify the development of concurrent, event driven communication software. The “Pattern languages of Program Design” series of books [17] are good references for anyone interested in pattern-related topics covering a diverse range of disciplines.

In the parallel computing domain, (parallel) design patterns specify recurring parallel computational problems with similar structural and behavioral components, and their solution strategies. Examples of such recurring patterns are: static and dynamic replication, divide and conquer, data parallel computation with various topologies, compositional framework for irregularly-structured control-parallel computation, pipeline, singleton pattern for single-process (single- or multi-threaded) computation. This paper presents a generic, design-pattern based model and system for building parallel applications.

1.1. Pattern-based Approaches in the Past

As an important difference from the usages of the design-level patterns in the object-oriented domain [11], patterns in parallel computing are often employed not only at the design-level but also at the implementation-level. That is, the design-level patterns are often pre-implemented as reusable components. This is similar in concept to a *framework* [18] in the conventional Software Engineering terminology.

Starting with the late 80s, several pattern-based systems have been built with the intention to facilitate the rapid development of parallel applications through the use of pre-implemented, reusable components. Some of the earlier systems include *Code* [4] and *Frameworks* [28]. Some of the recent systems based on similar ideas are: *Enterprise* [26], *Code2* [5], *HeNCE* [5], *Tracs* [2], and *DPnDP* [30].

Concurrent to the above thread of development, another group of researchers started exploring parallel design patterns from a different perspective. This direction of research concentrates on the idea of replacing explicit parallel programming by the selection and instantiation of a variety of pre-packaged parallel algorithmic forms popularly known as *algorithmic skeletons*. An algorithmic skeleton captures an algorithmic form common to a range of parallel programming applications. In practice, algorithmic skeletons are described as higher order polymorphic functions and are best realized using various functional and

logic programming languages [7, 9, 10]. A survey of some of the research in this direction can be found in [6].

1.2. Limitations of the Previous Approaches

Though the idea of design- and implementation-level patterns holds significant promise, in practice however, most pattern-based systems in parallel computing suffer from one or more of the following serious limitations:

- Most of the pattern-based systems support a limited and fixed set of hard-coded patterns. Often there is no provision for adding new patterns to the system without necessitating major reworking of the system (i.e., lack of extensibility).
- If a certain desired pattern is not supported by such a system, often there is no alternative to the user but to abandon the idea of using the particular approach altogether (lack of flexibility).
- Most of the systems mentioned previously support only a limited set of patterns in ad hoc ways. There is no generic or canonical model describing the structure and behavior of a pattern. Lack of genericness is often at the root of the lack of flexibility and extensibility.
- The research on algorithmic skeletons, on the other hand, concentrates on abstract models for representing parallel algorithmic patterns. These models are best realized inside the realms of pure functional and logic programming languages [9, 10]. The approach is yet to be adopted by the main-stream of parallel computing, where the conventional languages like Fortran, C, C++ and Java are the preferred languages of choice for the majority of developers. Furthermore, most of the works in this direction face some or all of the previously mentioned limitations (for instance, ad hoc set of skeletons, difficulty in skeleton composition, lack of flexibility).

1.3. The Present Approach

In contrast to the previous approaches, the present research proposes a generic model for realizing and using parallel design patterns. The model addresses most of the shortcomings of the existing approaches that were outlined in the previous subsection. Here is a summary of the important characteristics of the Parallel Architectural Skeletons Model (abbreviated as PASM):

- PASM is a generic model for describing parallel design patterns. The model provides genericness in two ways. First, the model provides a standard or canonical method for describing and implementing a parallel pattern. The PASM model itself can be described in a manner independent of any particular pattern or application. Second, each pattern in the model is fully parameterized to accommodate the different structures and behaviors required by different parallel applications.
- The mode of interaction between various components of a pattern is based on pattern-specific message-passing protocols, which makes the model suitable for a cluster of workstations or PCs. The high-level abstractions provided by a pattern hide most of the low-level details which are commonly encountered in any parallel application development (for instance: problem decomposition and distribution, process/thread creation and management, process-processor mapping, communication and synchronization, data marshaling and unmarshaling, load balancing, architecture- and network-specific low-level details).

- Each parallel pattern is described independent of other patterns. Yet different patterns can interact with each other via standard interfaces and well-defined protocols. Consequently, extending the library of patterns becomes a non-issue.

- We have been able to describe every message-passing parallel computing pattern that we have so far encountered in the literature using the PASM model. The present repertoire of patterns covers a wide range of frequently used parallel patterns such as pipeline, static and dynamic replication, divide and conquer, single- and multi-dimensional data parallel computation, composition for irregularly structured control-parallel computation, and hierarchical composition.

- The term *parallel architectural skeleton* is used to represent the set of generic attributes associated with a pattern. Each architectural skeleton is pre-implemented as a reusable component. A user, depending upon the specific needs of a parallel application, chooses the appropriate skeletons, supplies required parameters and application-specific code. Architectural skeletons supply most of the code that is necessary for the low-level, parallelism-related issues mentioned before. In other words, architectural skeletons take care of application-independent parallel computing aspects, whereas the user largely supplies the necessary application code. Accordingly, there exists a clear separation between application code and application-independent issues. From the Software Engineering perspective, this desirable property is commonly known as *separation of specifications* or *separation of concerns*.

- Hierarchical composition is an inherent characteristic of the model. The model is based on the fact that patterns contain other patterns, which results in a tree-structured hierarchy. Accordingly every parallel application is structured in a top-down hierarchical fashion. The model supports hierarchical refinements of patterns, whereby parts of an application can be hierarchically modified without affecting the rest.

- Many of the functionalities of a conventional message-passing library, for instance MPI [16] or PVM [13], can be realized within the boundaries of the model. The convergence between the two approaches provides additional flexibility to the user in application development. For example, a user can build a parallel application entirely using high-level architectural skeletons. However, if certain parts of the application are not amenable to employing pattern-based solutions, the user can code such parts using MPI-like low-level message passing primitives, that are supported within the frameworks of the model, and combine all the parts into a single application.

Unlike many other object-oriented and pattern-based approaches found in the literature whose ideas originated purely from the object-oriented design perspective [11, 20, 27], this research originated from a pure parallel programming perspective. In fact, at the beginning, we were not committed to any particular design methodology or implementation style. However, as the work progressed, the model with the desired capabilities turned out to be an ideal candidate for object-oriented style design and implementation. Presently it has been fully implemented as a C++-based template-library, without necessitating any language extension. Implementation of the PASM model is referred to as PASM system.

In summary, PASM is a high-level model and system that provides solutions for frequently occurring parallel computing problems in the form of object-oriented code components. The message-passing aspects of the model allow it to support low-level MPI-like message passing primitives in addition to high-level parallel patterns. A user can mix ready-made high-level patterns with low-level message passing in a single application. The

generic nature of the model and the library-based system allow new patterns to be integrated without requiring any modification to the existing parts of the system.

2. THE ARCHITECTURAL SKELETON MODEL

We start this section with an example. The objective here is to introduce to the reader the various concepts of the model and its implementation, before elaborating them in detail. Variations of the example will be used at different points throughout the rest of the discussion. Words within *italics* are terms with special meanings in the context of the model and are defined subsequently.

2.1. Example 1

For ease of understanding, let us take the familiar example of a Master-Worker application where the *Master module* produces a succession of image frames and sends them to dynamically replicated *Worker modules* for processing. Code segments for this application are shown on the next page. The *Master module* is implemented as an extension of the replication *skeleton* which supports arbitrary degree of replication that is dynamically controlled at run-time, depending on the work-load and the number of available processors. Each *Worker module* extends the singleton *skeleton*, which is used for single-process computation. Each replicated *Worker* is a *child* of the *Master module*.

Each of the *Master* and *Worker modules* has its *representative* code segment which is called *Rep* (lines 18 and 52 respectively). When the *representative* is empty, what we have is an *abstract module*. Filling in of the *representatives* of *Master* and *Worker abstract modules* with application-specific code results in the corresponding *modules*. In other words, an *abstract module* is a *module* without application-specific code.

The *Master module* interacts with each *Worker* using its *internal communication and synchronization protocol*, *PROT_Repl*. The primitive operations *SendWork(...)*, *ReceiveWork(...)*, *ReceiveResult(...)*, etc., (lines 27 and 54) are member functions of *PROT_Repl*. Each *Worker module* interacts with its *parent*, i.e., the *Master module*, using its *external communication and synchronization protocol*, which is *PROT_Repl* in this case (line 54).

Frame (line 21) is a user-defined object of the type *Image* (line 3), whose data attributes can be marshaled, shipped over a communication link, and then un-marshaled, without the usual hassles of data packing and un-packing.

The example illustrates the use of the high-level textual user interface, based on a specification language, for the PASM system. A Perl-script [31] expands the text to full C++. This high-level textual interface helps the user to bypass certain C++-related and other details which are often laborious and monotonous to write, and can easily be automated. As will be evident in section 3, the use of a textual interface is not a language extension. If desired, a user can easily bypass this phase and can directly develop his application in C++ (obviously it will make him write more code).

The code for the skeletons and the associated protocols hide most of the parallelism related details, thus leaving only the application-specific issues to the user. For instance, the single operation *SendWork(...)* (line 27) inside *PROT_Repl* handles issues like: dynamic process creation and mapping; data marshaling, communication and un-marshaling; keeping track of process status. Had all these issues needed to be done from scratch (as with MPI), the user would have required to write hundreds of lines of code in developing the same application. Essentially, the user supplies the application specific code components and the PASM system supplies the necessary code for parallelism.

```

0: // ***** EXAMPLE 1 *****
1: GLOBAL{
2: #include "ImageDef.h"
3:   class Image: public UType
4:   {
5:       // A class definition specifying the attributes of a marshal-able image.
6:       ...
7:   };
8:   // Also, global definitions of the methods ProcessImage(...) and
9:   // WriteImage(...) may go here. These methods are used in the
10:  // following code segments.
11: }
12: // *****
13: // The "Master" module extends the replication skeleton
14: Master EXTENDS ReplicationSkeleton
15: {
16:   // The dynamically replicated children of "Master"
17:   CHILDREN = Worker;
18:   Rep {
19:       int Number_of_Images = 0;
20:       int success;
21:       Image Frame;
22:       while (True){
23:           success = True;
24:           while ((Number_of_Images < MaxImages) && success){
25:               Produce(Frame);
26:               Number_of_Images++;
27:               success = SendWork(Frame); // Keep sending work-loads
28:               // to workers until none is free and can no longer spawn one
29:               // dynamically. SendWork is a member function of the internal
30:               // protocol: PROT_Repl.
31:           }
32:           if (!success) { // Do it myself, if failed to assign work-load
33:               // to a worker.
34:               ProcessImage(Frame);
35:               WriteImage(Frame);
36:           }
37:           if (Number_of_Images == MaxImages) break;
38:       }
39:   }
40:   LOCAL {
41:       // Local functions used by this module go here.
42:       void Produce (Image& Frame)
43:       {
44:           // User code for "Produce" goes here:
45:       }
46:   }
47: }
48: // *****
49: // Each replicated "Worker" module.
50: Worker EXTENDS SingletonSkeleton
51: {
52:   Rep {
53:       Image Frame;
54:       ReceiveWork(Frame); // ReceiveWork is a member function of
55:       // the external protocol: PROT_Repl.
56:       ProcessImage(Frame);
57:       WriteImage(Frame);
58:   }
59: }

```

In the previous example, words in the bold case (e.g., EXTENDS, CHILDREN, LOCAL, GLOBAL) are keywords specific to the textual specification language. As was mentioned

previously, the textual specification language is only an optional thin layer over C++. The Perl-expanded C++-translation of the example is illustrated in section 3, which discusses the implementation issues of the PASM system. Detailed discussions regarding how these keywords are translated into C++ code can be found in [14].

More examples will be illustrated in the following discussion to further elaborate the ideas behind the model and the associated framework.

2.2. The Model

Let us now look at the basic PASM model. A *parallel architectural skeleton* [14, 15] is a set of attributes that encapsulate the structure and the behavior of a parallel pattern in an application independent manner. These attributes are generic for all patterns. Many of these attributes are parameterized where the value of a parameter depends on the needs of an application. Some of these parameters are statically configurable (i.e., at compile time. For instance: the dimensions of a 2-D mesh) while the others are dynamic (for instance: the degree of replication in the case of dynamic replication). User extends a skeleton by specifying the application-dependent static parameters, as needed by the application at hand.

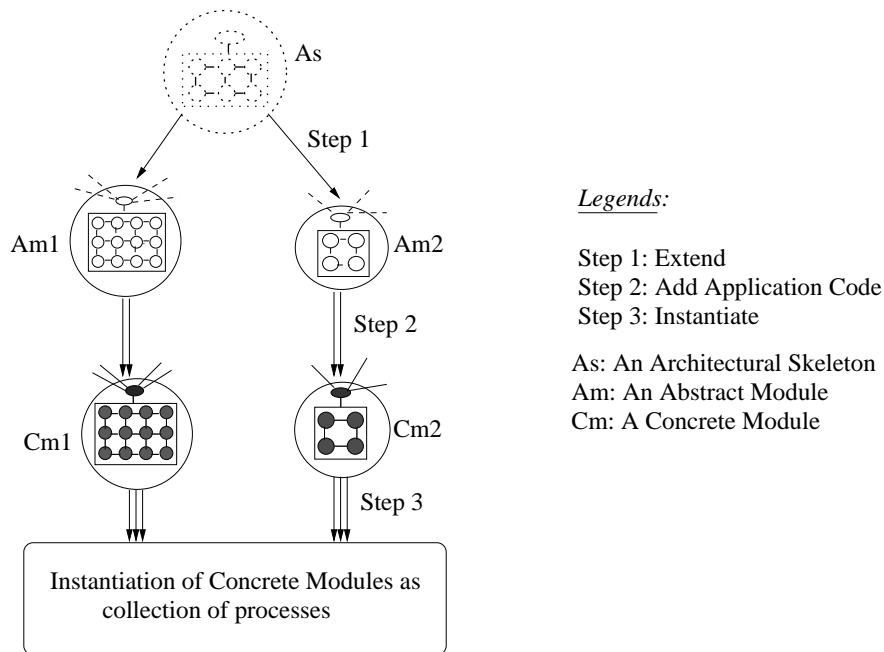


FIG. 1. Application development using architectural skeletons

Figure 1 approximately illustrates the various phases of application development using parallel architectural skeletons. As shown in the figure, different extensions (i.e., different set of parameter values) of the same skeleton can result in somewhat different *abstract parallel computing modules* (abbreviated as an *abstract module*). An abstract module is yet to be filled in with application code. Once an abstract module is supplied with application code, it results in a *concrete parallel computing module* (abbreviated as a *concrete module* or simply a *module*). A parallel application is a systematic collection of mutually interacting, instantiated modules.

An abstract module inherits all the properties associated with a skeleton. Besides, it has additional components that depend on the needs of a given application. In object-oriented terminology, an architectural skeleton can be described as the *generalization* of the structural and behavioral properties associated with a particular parallel pattern. An abstract module is an application-specific *specialization* of a skeleton.

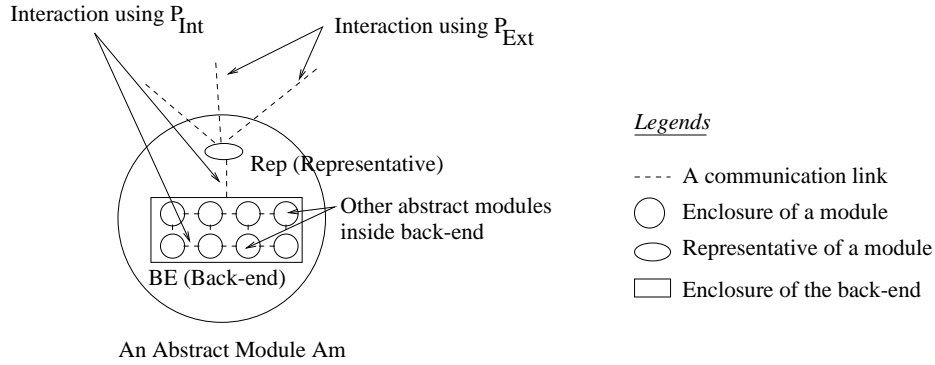


FIG. 2. Structure of an abstract module

Figure 2 diagrammatically illustrates the anatomy of an abstract module (in this case, the module extends the data-parallel architectural skeleton designed for 2-D mesh topology). The various attributes associated with a skeleton (and subsequently inherited by an abstract module and a module) are explained next.

DEFINITION 2.1. An *architectural skeleton*, A_s , is an application-independent abstraction comprising of the following set of generic attributes, $\{Rep, BE, Topology, P_{Int}, P_{Ext}\}$. An *abstract module* is an application-specific extension of a skeleton. Let A_m be such an abstract module that extends the skeleton, A_s . The various attributes inherited by A_m (from A_s) are described in the following:

- Rep is the representative of A_m . When filled in with application code, Rep represents the module in its action and interaction with other modules.
- BE is the back-end of A_m . Formally, $BE = \{Am_1, Am_2, \dots, Am_n\}$, where each Am_i is itself an abstract module. The notion of modules inside another module results in a tree-structured hierarchy. Am , at the root of this tree, is the *parent* and each Am_i is its *child*. Modules Am_i and Am_j belonging to the same back-end are *peers* of one another.
- $Topology$ is the interconnection-topology specification of the modules inside the back-end (BE), and their connectivity specification with Rep .
- P_{Int} is the internal communication-synchronization protocol of A_m and its associated skeleton, A_s . The internal protocol is an inherent property of the skeleton, and it captures both the parallel computing model of the corresponding pattern and the topology. Formally, P_{Int} is defined as a set of primitive commands. Using the primitives inside P_{Int} , the representative of A_m can interact with the modules in its back-end, and a module in the back-end can interact with its peers.
- P_{Ext} is the external communication-synchronization protocol of A_m . Formally, it is defined as a set of primitive commands. Using the primitives inside P_{Ext} , A_m can interact with its parent and the peers. Unlike P_{Int} , which is an inherent property of the skeleton, P_{Ext} is adaptable. I.e., A_m adapts to the context of its parent by using the internal protocol of its parent as its external protocol.

Though an abstract module is an application specific specialization of an architectural skeleton, it is still devoid of any application code. User writes application code for an abstract module using its communication-synchronization protocols, P_{Int} and P_{Ext} . A code-complete abstract module is called a *concrete parallel computing module* (abbreviated as a *concrete module* or a *module*). A concrete module can be formally defined as follows:

DEFINITION 2.2. (a) An abstract module that has no children (i.e., an empty *BE*) becomes *concrete* when its representative, *Rep*, is filled in with application code. (b) An abstract module with children becomes *concrete*, provided each one of its children is a concrete module and its own representative is filled in with application code. A parallel application is a hierarchical collection of mutually interacting concrete modules.

PASM is a hierarchical model. The hierarchy arises from the notion of modules inside modules, which results in the parent-child relationship among modules as mentioned before. This notion of parent-child relationships among modules is equivalent to a tree-structured hierarchy. A parallel application can be viewed as a hierarchical collection of modules, constituting of a root module and its children forming the sub-trees. This tree is called the *HTree* of the application. A singleton module that has no children (refer to the previous example) forms a leaf of the hierarchy.

As can be seen from the preceding discussion, there exists a clear separation between application code and application-independent issues (also known as *separation of specifications*). An architectural skeleton is a pure application-independent abstraction. An abstract module contains some application specific components (e.g., the right parameters for topology, the right protocols that depend on the current context). A concrete module is an application-specific completion. A hierarchy comprising of only abstract modules represents the overall structure of an application, without the application code. From the implementation perspective, such a structure will compile and run, however without doing anything useful.

We will revisit the various concepts described in this section in the following examples. Implementation issues of the model are discussed in section 3.

2.3. Example 2

The following is a more significant example that further elaborates some of the previous concepts. The example illustrates an arbitrary composition of modules using the compositional skeleton. It also illustrates hierarchical refinement, and some of the other useful features of the present object-oriented implementation of the model, for instance: automatic data marshaling and un-marshaling mechanisms; use of operator overloading in C++ to implement certain primitive operations inside protocol classes.

Let us consider the graphics animation program [29] consisting of the three modules: *Generate*, *Geometry* and *Display* (abbreviated as *Gen*, *Geo* and *Dis* in Figure 3) that form a pipeline. The program takes a sequence of graphics image-frames and animates them. *Generate* computes the location and motion of each object for a frame. It then passes the frame to *Geometry* which performs actions such as viewing transformation, projection and clipping. Finally, the frame is passed to *Display* which performs hidden-surface removal and anti-aliasing. Then it stores the frame onto the disk. After this, *Generate* continues with the processing of the next frame and the whole process repeats.

The following code-segments demonstrate one way of implementing the application using the PASM system. The implementation uses the compositional skeleton and the

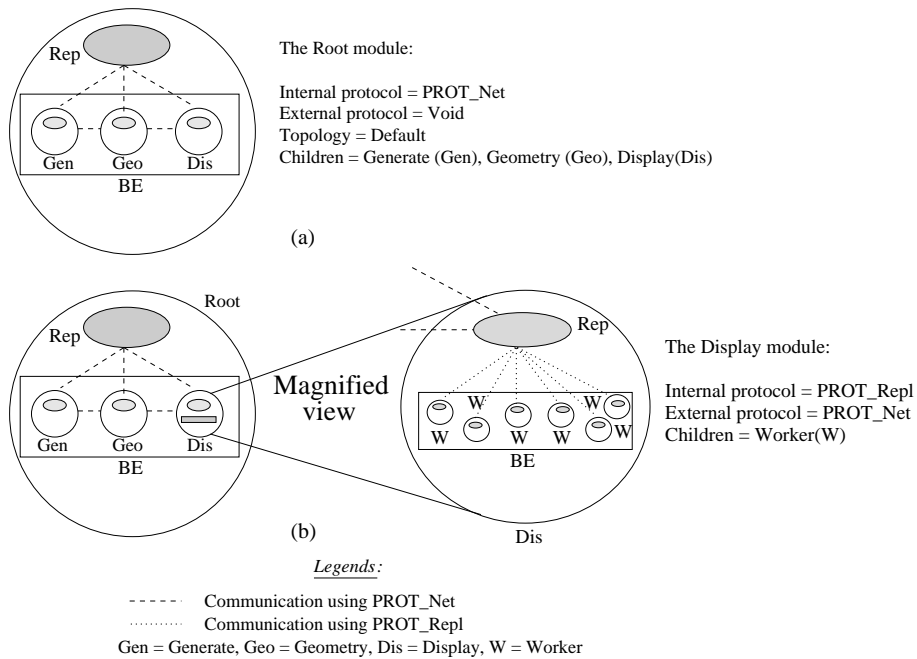


FIG. 3. Structure of the animation application before and after refinement

singleton skeleton. The Root compositional module (i.e., Root extends the compositional skeleton) forms the root of the hierarchy, and its three children, i.e., Generate, Geometry and Display, form the subtrees. Each of the three children is a singleton module (i.e., it extends the singleton skeleton) and hence is a leaf of the hierarchy. The internal protocol, P_{Int} , for the compositional skeleton is $PROT_Net = \{Send(...), Receive(...), Broadcast(...), Spawn(...), \dots\}$. Consequently, PROT_Net becomes the external protocol, P_{Ext} , for each of the three children (refer to section 2.2). By default, the modules composed inside a compositional module form all-to-all interconnection topology, which is the case here. Figure 3(a) illustrates the general structure of the application in the context of the model.

GenerateGeometry and GeometryDisplay are user defined classes, whose data attributes can be marshaled, shipped over a communication link and then un-marshaled, without the usual hassles of data packing and un-packing (as in MPI or PVM). Their constituent data members are either system defined wrappers of standard data-types or other user defined types. The example also illustrates the use of C++ operator overloading as an alternative way for implementing and using certain primitive operations inside PROT_Net (e.g., Send(...), Receive(...)).

```
// ***** EXAMPLE 2 *****
GLOBAL {
#include "geom.h"
#define MaxImages 120

// The following defines a marshal-able class.
class GenerateGeometry : public UType {
    Int imageNumber; // "Int" is a System defined marshal-able wrapper for "int"
    ObjTable table; // "ObjTable" is a marshal-able class defined in "geom.h"
public:
    // Marshal() an object of "this" type
    virtual void Marshal() {imageNumber.Marshal(); table.Marshal();};
};
```

```

    // Un-marshal an object of "this" type
    virtual void UnMarshal() {imageNumber.UnMarshal(); table.UnMarshal();};
    // Constructor(s) etc...
    ...
};

// Another marshal-able class definition.
class GeometryDisplay : public UType {
    Int imageNumber;
    Int nPoly;
    PolyTable table; // "PolyTable" is another marshal-able class defined in "geom.h"
public:
    virtual void Marshal() {imageNumber.Marshal(); nPoly.Marshal(); table.Marshal();};
    virtual void UnMarshal() {imageNumber.UnMarshal(); nPoly.UnMarshal();
        table.UnMarshal();};

    // Constructor(s) etc...
    ...
}
}

// *****
// The "Root" module, which is at the root of the hierarchy. It has three child
// modules: Generate, Geometry and Display.
Root EXTENDS CompositionalSkeleton
{
    CHILDREN = Generate, Geometry, Display;
    Rep {
        // The representative code goes here. In this particular application, the
        // representative has no functionality and thus it just contains an empty
        // loop.
    }
}
// *****
// The "Generate" module, which extends the singleton skeleton.
Generate EXTENDS SingletonSkeleton
{
    // A singleton module can have no children.
    Rep {
        // The representative code goes here.
        int image;
        GenerateGeometry Work;
        for (image = 0; image < MaxImages ; image++){
            ComputeObjects (Work);
            Geometry << Work; // A primitive of the external protocol, PROT_Net.
            // An alternative option is to use: Send(Geometry, Work, context).
        }
    }
    // All local definitions go below:
    LOCAL {
        void ComputeObjects(GenerateGeometry& Work)
        {
            //User code for "ComputeObjects" goes here.
        }
    }
}
// *****
// The "Geometry" module.
Geometry EXTENDS SingletonSkeleton
{
    Rep {
        int image = 0;
        GenerateGeometry Work;
        GeometryDisplay Frame;
        for (image = 0; image < MaxImages ; image++){

```

```

        Generate >> Work; // A primitive of the external protocol, PROT_Net.
        // An alternative option is to use: Receive(Generate, Work, context).
        DoConversion(Work, Frame);
        Display << Frame;
    }
}
LOCAL {
// local definition of DoConversion(...) goes here.
}
}
// *****
// The "Display" module.
Display EXTENDS SingletonSkeleton
{
    Rep {
        int image;
        GeometryDisplay Frame;
        for (image = 0; image < MaxImages ; image++) {
            Geometry >> Frame;
            DoHidden(Frame);
            WriteImage(Frame);
        }
    }
    LOCAL {
// Local definitions of DoHidden(...) and WriteImage(...) go here.
    }
}
// *****

```

The previous user code is expanded by the PASM system to generate the front-end C++ file, Pmain.cc, which is subsequently compiled and linked with the skeleton-library to generate the executable. PASM system is discussed further in section 3.

2.3.1. Refinement

It is generally the case that the Display module that performs hidden surface removal and anti-aliasing is the most time intensive of the three modules. This will slow down the entire application. The best possible way to resolve this is to distribute the work-load of Display among dynamically replicated (i.e., replicated dynamically based on work-load) workers. Accordingly, the singleton Display module is replaced with another module, of identical name, that extends the replication skeleton. The work-load of the new Display module is dynamically distributed among replicated copies of its children, Worker, modules. Each Worker extends the singleton skeleton. Figure 3(b) illustrates the structure of the modified application and the new Display module.

Regarding topology, the modules composed inside a replication module are not interconnected with one another, however each one of them is connected with its parent. Their physical distribution is dynamic. The internal protocol of the replication skeleton is PROT_Repl, which becomes the external protocol of each replicated Worker. The external protocol of Display remains the same as before, i.e., PROT_Net.

The modified part of the application is illustrated next. The reader will notice quite a bit of commonality between the modified part of the application and the example illustrated in section 2.1. Note that none of the other modules, i.e., Generate and Geometry, is affected by this change. This type of localized replacement that works towards the overall betterment of an application is called a *refinement* (e.g., part of the application is *refined*).

```

// *****
// The refined "Display" module.

```

```

Display EXTENDS ReplicationSkeleton
{
    //The dynamically replicated children of "Display"
    CHILDREN = Worker;
    Rep {
        int image = 0;
        int success;
        GeometryDisplay Frame;
        while (True){
            success = True;
            while ((image < MaxImages) && success){
                Geometry >> Frame; // A member of external protocol, PROT_Net.
                image++;
                success = SendWork(Frame); //A member of internal protocol, PROT_Repl.
            }
            if (!success) { // Do it myself, if not successful in assigning
                // it to a worker.
                DoHidden(Frame);
                WriteImage(Frame);
            }
            if (image == MaxImages) break;
        }
    }
    LOCAL {
        // Local definitions of DoHidden(...) and WriteImage(...) go here.
        // Another possibility is to define them globally, since these procedures
        // are used inside more than one module (refer below).
    }
}
// *****
// Each replicated "Worker" module.
Worker EXTENDS SingletonSkeleton
{
    Rep {
        GeometryDisplay Frame;
        ReceiveWork(Frame); // A member of the external protocol, PROT_Repl.
        DoHidden(Frame);
        WriteImage(Frame);
    }
    LOCAL {
        // Local definitions of DoHidden(...) and WriteImage(...) go here.
    }
}
// *****

```

The next example further elaborates the PASM model and the system.

2.4. Example 3

This last example illustrates code fragments for a parallel implementation of the Jacobi iterative scheme. Jacobi and its variants are frequently used for solving sparse linear systems. Sparse systems are often encountered in various scientific and engineering applications, for instance: computational fluid dynamics, thermodynamics, electro-magnetics. In this particular application, the Jacobi scheme is applied on a square grid with given boundary conditions. The algorithm iterates over all inner grid points (of the square grid), and at each point it calculates a certain value of the point (for instance: temperature) based on the values of the neighboring grid points. The algorithm repeats until all the values converge, which correspond to the solution of the sparse system of linear equations. For simplicity, the algorithm shown here repeats for a fixed number of iterations (i.e., MaxIterations) which is reasonable enough to give a converged solution, if one exists.

The data-parallel skeleton with mesh topology (1- or 2-D mesh) is the most appropriate for this application. The module `FrontEnd` extends the data-parallel skeleton, and corresponds to the front-end of a data-parallel mesh. The back-end of the mesh comprises of the identical singleton modules, `MeshElement`. The data-parallel skeleton supports different topologies and associated protocols (as explained in Figure 4 in the next section). The 1-D mesh topology and the corresponding protocol, `PROT_1DMesh`, are explicitly selected for this application. Accordingly, `PROT_1DMesh` becomes the external protocol of each child, `MeshElement`, module.

The given square grid is equally partitioned among the mesh-elements (i.e., the children of `FrontEnd`). Granularity of the parallel application is related to the number of grid-points mapped per mesh-element. Nearest neighbor communication is needed at the inner mesh boundaries. Relevant pieces of the code are illustrated in the following:

```
// ***** EXAMPLE 3 *****
#include "Grid.h"
...
// Front-end of the 1-D mesh.
FrontEnd EXTENDS DataParallelSkeleton
{
    CHILDREN = MeshElement;
    PROTOCOL = PROT_1DMesh; // In this case, we need to explicitly specify the
        // internal protocol, since more than one choices are possible.
    Rep {
        ...
        int N = SetMeshWidth(4) // Set mesh-width to 4. It is a member of
            // PROT_1DMesh. Mesh-width is one parameter that can be
            // configured either statically or dynamically.
        Grid A(1000,1000); // A 1000 X 1000 marshal-able grid.
        ReadIn (A);
        PartitionGrid (A,N); // Partition the grid row-wise among the N = 4 children
        CollectResults (A,N); // Collect the results from the children.
        ...
    }
    LOCAL {
        // Definitions of ReadIn (...), PartitionGrid (...), CollectResults (...) and
        // other methods and variables may go here (or may be defined globally).
    }
}
// *****
// Each element of the 1-D mesh.
MeshElement EXTENDS SingletonSkeleton
{
    Rep {
        ...
        int context = ...;
        Grid A;
        ReceiveFromRep(A,context); // It is a member of external protocol, PROT_1DMesh.
        // In this particular case, A is a 252 X 1000 grid. There are two extra rows
        // (i.e., rows 0 and 251) for holding boundary rows from neighboring elements.
        Grid B = A;
        ...
        int lb, ub;
        int nRows = A.Rows();
        int nColumns = A.Columns();
        int myPosition = getMyPosition(); // Get my position in the 1-D mesh.
            // It is a member primitive of PROT_1DMesh.
        int meshWidth = getMeshWidth(); // Get the width of the 1-D mesh. It is a
            // member primitive of PROT_1DMesh.
        if (myPosition == 0) lb = 2; else lb = 1;
        if (myPosition == (meshWidth - 1)) ub = nRows - 3; else ub = nRows - 2;
    }
}
```

```

// MaxIterations is chosen as some reasonable value.

for (int k = 0; k < MaxIterations; k++){
    if (myPosition > 0) Peer[Left] << A[1]; // Each row of A is a
                                           // marshal-able object.
    if (myPosition < (meshWidth - 1)) Peer[Right] >> A[nRows - 1];
    if (myPosition < (meshWidth - 1)) Peer[Right] << A[nRows - 2];
    if (myPosition > 0) Peer[Left] >> A[0];

    // The above four statements illustrate communication with peers
    // (in this case, nearest neighbor communication). Different types of
    // communication, including broadcasting to peers, are possible,
    // which are member primitives of PROT_1DMesh. The above statements
    // also illustrate the use of C++ operator-overloading in implementing
    // certain primitive operations. An alternative option is to use
    // functions calls, e.g., SendToLeft(...), ReceiveFromRight(...),
    // SendToOffset(...), etc.

    for (int i = lb; i <= ub; i++){
        for (int j = 1; j <= nColumns - 2; j++){
            B[i][j] = (A[i][j-1] + A[i-1][j] + A[i+1][j] + A[i][j+1])/4;
        }
        A = B;
    }
    SendToRep(A, context); // A member of external protocol, PROT_1DMesh.
}
}
// *****

```

Peer-to-peer interaction (in this case, nearest neighbor) is an inherent characteristic of the Jacobi iterative scheme and is supported by the various primitive commands inside PROT_1DMesh. Peer-to-peer types of interactions could not be modeled by most of the earlier pattern-based systems. Performance results for Jacobi are discussed in Section 4.

3. AN OBJECT-ORIENTED IMPLEMENTATION

The following discussion addresses some of the key issues related to the present object-oriented implementation of the model. The interested reader can refer to [14] for more details on implementation-related issues.

The model is presently implemented using industry standard C++ (SunCC compiler, V 4.1), without necessitating any language extension. The implementation uses the MPI library provided by LAM 6.1 [1]. A textual user interface helps the user during the implementation phase of an application. Application code written using the textual interface is expanded by a Perl-based [31] script to produce the front-end C++ code, which is subsequently compiled and linked with the skeleton-library to produce the executable. The use of the textual interface is not a language extension, but merely an optional feature that helps the user to skip certain laborious and often monotonous steps in the development process. If desired, the user can bypass the textual-interface phase and directly work in C++.

Other important features of the implementation, as illustrated in the previous examples, include: (1) use of C++ operator-overloading to implement certain primitive operations inside protocol classes (for instance: Send(...) and Receive(...) inside PROT_Net). (2) Use of marshaling and un-marshaling mechanisms whereby the data attributes of an entire object can be marshaled, shipped over a communication link and then un-marshaled, without the usual hassles of data packing and unpacking.

3.1. The User Interface

The examples in the previous section illustrated the use of the current textual user interface that is based on a specification language. A Perl-script parses and expands the user code to produce the front-end C++ file, *Pmain.cc*. It is subsequently compiled and linked with the skeleton library to produce the executable, designated to run on a workstation cluster.

The following is the skeleton of the automatically generated file *Pmain.cc* for the example in section 2.1. As is evident from the following code-skeleton, a user can directly develop his application code in C++, thus bypassing the textual interface.

```
#include "BasicDef.h"
#include "SingletonSkeleton.h"
#include "ReplicationSkeleton.h"
#include "PROT_Repl.h"
#include "VoidClass.h"

// The items defined inside GLOBAL are copied in the following as it is.
// *****
#include "ImageDef.h"
class Image: public UType
{
    // A class definition specifying the attributes of a marshal-able image.
};
// Similarly, the other global definitions inside "GLOBAL" follow:
// *****
// Generated code for module: "Worker"
class Worker : public SingletonSkeleton <PROT_Repl>
{
public:
    Worker(Void& _v) {};
    virtual void Rep() { // The representative code goes here. }
    // Miscellaneous local definitions are copied below:
};
// *****
// Generated code for module: "Master"
class Master : public ReplicationSkeleton <Worker, PROT_Repl, Void>
{
    Master() {};
    virtual void Rep() { // The representative code goes here. }
    // Miscellaneous local definitions are copied below:
};
// *****
void Pmain()
{
    Master TopLevel_366;
    TopLevel_366.Run();
}
// *****
```

As the previous code-segments suggest, C++ templates are used to realize certain statically-configurable parameters associated with the attributes (e.g., choosing the right protocols, specification of the child module(s)). *PROT_Repl* is the internal protocol for the replication skeleton. Consequently, *PROT_Repl* becomes the external protocol for each replicated *Worker* module. Since the *Master* is at the root of the hierarchy, its external protocol is void (as specified by *Void* as one of the template value-parameters).

3.2. Implementing Architectural Skeletons: Reusability and Extensibility

Figure 4 illustrates the high-level class diagram behind the design of the skeleton library. The figure uses the standard UML [3] notation. For simplicity, the figure does not illustrate

the relationships between the skeleton- and the protocol-classes. Moreover, the various attributes and the methods associated with each class, and the formal parameters, in the form of templates, associated with each inherited skeleton-class are not shown for a cleaner representation.

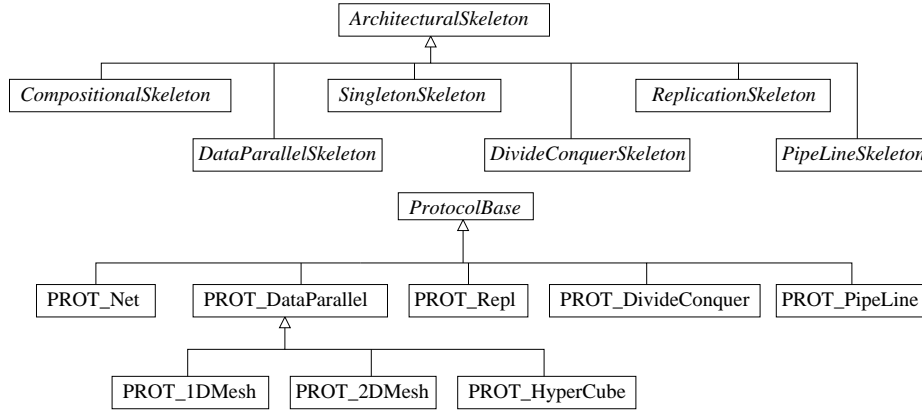


FIG. 4. High level class diagram behind the design of the skeleton library

From the implementor's or an experienced user's perspective, certain features of the object-oriented design, in conjunction with the generic nature of the model, favor reuse and extension of the skeleton library. The generic model helps, because it provides a clear picture regarding what the different components of a skeleton are and what their functionalities are going to be (compare it with a totally ad hoc approach). Furthermore, from the model's perspective, each module is an independent entity whose only interface with the outside world is through its representative and the adaptable external protocol. Accordingly, what the outside world sees of the module are only through its actions (i.e., input/output and any observable side effects), without knowing exactly how these actions are carried out internally. In other words, the module acts as a black-box to the outside world.

Many of the object-oriented features that are supported in C++, for instance: polymorphism (through the use of C++ templates and virtual methods) and inheritance, favor the reuse and extension of the existing skeleton library. New classes can be defined by extending the existing ones, thus enabling the design and addition of new skeletons and protocols with added functionalities. Completely new skeletons and protocols can be designed by extending the base classes (refer to Figure 4). In each case, a collection of pre-existing virtual methods need to be over-written and some new additional methods might need to be defined in order to reflect the characteristics of the newly designed skeleton.

3.3. The Dynamic Execution Model

The PASM system is based on the SPMD execution model, i.e., each processor in the processor-cluster loads and executes the same file, which results in major savings in terms of management of source, object and executable files. As discussed in section 2.2, every PASM application is associated with a tree-structured hierarchy, comprising of a root module and its children forming the sub-trees. This tree is called the *HTree* for the application. At run time, each process traverses the same *HTree* associated with the application, starting at the root of the tree.

Figure 5(a) illustrates the *HTree* associated with the Master-Worker example discussed in section 2.1. Figure 5(b) illustrates the *HTree* associated with the graphics-animation application before refinement, as discussed in section 2.3. Figure 5(c) illustrates the same application after refinement (refer to section 2.3.1), where the Display module further sub-divides its work-loads among dynamically replicated Workers.

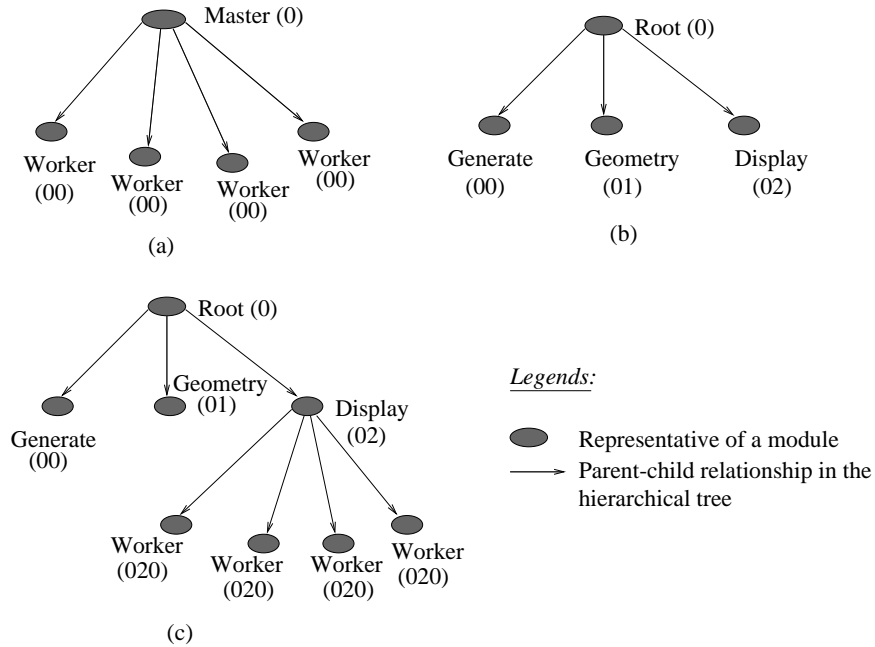


FIG. 5. HTree and its traversal scheme

A node of a *HTree* is essentially the representative of a module. Each process is responsible for executing exactly one node of the tree, i.e., there is a one-to-one correspondence between a process and a representative node. A process starts at the root of the hierarchy and then traverses down the tree to its designated node.

How does a process figure out regarding which path in the tree to traverse? This is achieved as follows: each process is dynamically assigned an identification string, by following a unique labeling scheme. As a process traverses down the tree, it dynamically calculates its path, by following the same scheme. The process traverses down a specific path of the tree, if and only if the already calculated path is a substring of its assigned identification string. When the calculated path matches the identification string, the process is at its designated node.

In figure 5, the printed string inside parentheses pairs beside each node is precisely the identification string associated with the process that executes it. The dynamically replicated Workers, in the first two examples in section 2, are all identical. Therefore they execute the same code and have the same identification string.

All of the previous issues are completely hidden from the user. In fact, a user follows the general structure as illustrated by the examples in section 2, and writes his application with the perspective that he is dealing with individual modules, rather than with individual processes. Without any further aid from the user, the dynamic execution model makes it possible for a process to execute the code segment pertaining to a given module.

To summarize, the discussion in this section demonstrates the feasibility of the approach through a particular implementation of the model. The present MPI-based implementation makes the skeleton-library suitable for relatively coarser-grain parallel applications. However, since the present implementation uses pure C++, nothing prevents a user from writing multi-threaded application code for individual representatives, by using an existing standard thread library (in that case, the user might benefit by using the various critical-section management patterns discussed in [22]) and thus exploit finer-grains of concurrencies in their applications if desired.

4. EXPERIMENTS AND PERFORMANCE RESULTS

Experiments were conducted to assess the performance of the system. The results were compared with direct MPI-based implementations. The performance difference with MPI lies within 5%, which can be attributed to the fact that the skeleton-library is implemented as an extremely thin layer on top of MPI. The thin implementation layer generally implies that any application that demonstrates good performance with direct MPI-based implementation should provide similar performances with a skeleton-based implementation, under all identical conditions.

The following discussion presents the performance results obtained for some well-known parallel applications. Some of these results demonstrate the effect of granularity on performance. Detailed results with different applications and comparisons with MPI can be found in [14].

4.1. PQSRS

Parallel Quick Sort using Regular Sampling, abbreviated PQSRS [25], is a parallel version of quick sort, shown to be effective for a wide variety of MIMD architectures. It uses a combination of master-slave and 1-D mesh patterns, which is easily realized using the data-parallel skeleton for mesh topology and the singleton skeleton (similar in structure to the example in section 2.4). The algorithm works in the following steps: (1) the master module partitions the data items to be sorted to the N children (i.e., slaves). Each child then performs sequential quick sort on its own data items, selects N data items as regular samples, and sends them back to the parent (i.e., master). (2) the master gathers the regular samples from all its children, sorts them, gathers $N - 1$ pivot values and broadcasts them to the children. Each child partitions its portion of sorted items into N disjoint partitions, based on the $N - 1$ pivot values. (3) Child i keeps the i^{th} partition and sends the j^{th} partition to its j^{th} peer. Thus, at this phase, each child has to communicate with all its $N - 1$ peers. (4) Each child receives $N - 1$ partitions from its peers, merges them with its own partition to form a single sorted list, and sends the sorted list back to the master. Finally, the master concatenates the sorted sub-lists from all its children to form the final sorted list.

PQSRS is a non-trivial algorithm which requires a considerable amount of peer-to-peer interaction among the slaves, which is supported by the internal protocol(s) of the data-parallel skeleton. It cannot be implemented using most other pattern based systems discussed previously. Figure 6(a) illustrates the results obtained in sorting 10000 and 24000 randomly generated objects using PQSRS. The underlying hardware is a cluster of Sun Sparc workstations (each is an Ultra 10 Elite with 256 MB of RAM) connected by a 10-megabit Ethernet network. The speed-up ratio is measured with respect to the same sequential quick-sort routine used inside PQSRS. Comparison with direct MPI-based

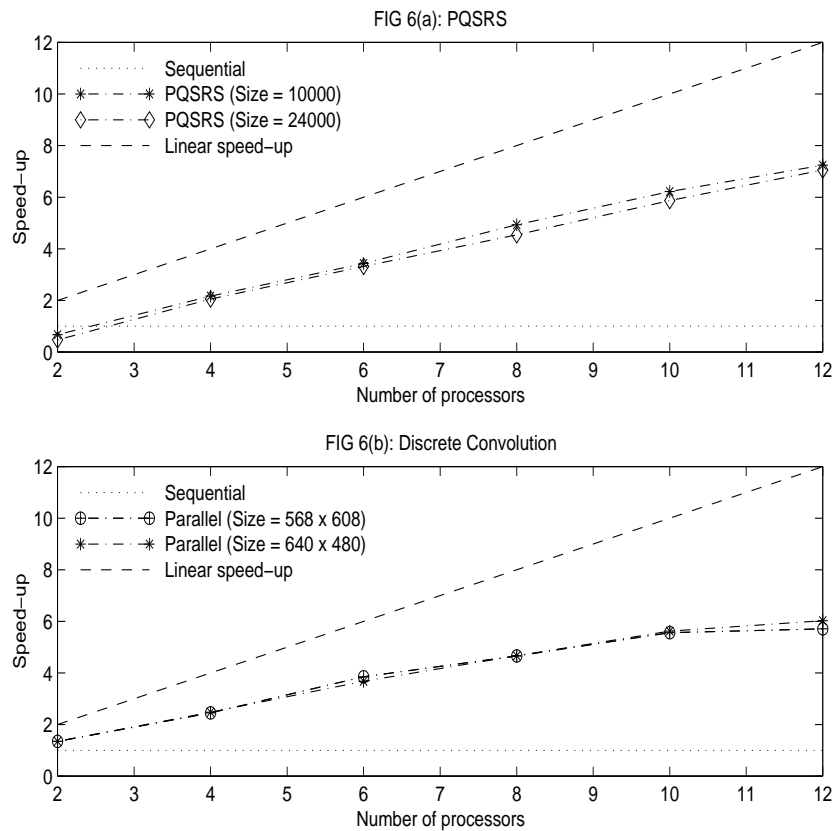


FIG. 6. Speed-up ratio versus number of processors

implementation shows no observable performance difference and hence is not illustrated separately.

4.2. 2-D Discrete Convolution

This is an image processing algorithm used for convoluting a given image through the application of a mask. The mask is applied to each image pixel to produce the convoluted image [23]. As compared to the previous application, this one is relatively simple. Like most other image processing algorithms, it follows the master-slave pattern where the slaves need not interact with one another.

Figure 6(b) illustrates the results obtained for the parallel discrete convolution of two images of sizes 640×480 and 568×608 pixels respectively. The mask used in each case is of size 10×10 . The underlying hardware is identical to that used in the previous application. The speed-up ratio is measured with respect to the best sequential algorithm.

4.3. Jacobi

The Jacobi method and its implementation have already been discussed in section 2.4. As compared to the previous two applications, it is a relatively finer-grain application. As the code fragment for Jacobi (see section 2.4, example 3) illustrates, the parallel implementation requires nearest-neighbor communication.

The granularity (i.e., the ratio of computational time to communication overhead, between two successive communication points) can be increased by mapping more nodes per processor, provided that the corresponding rate of increase in communication overhead is less than the rate of increase in computation per processor. If granularity is more than a certain optimal value, which can vary from situation to situation, theoretically there should be speed-up. Otherwise, the parallel application will show performance degradation.

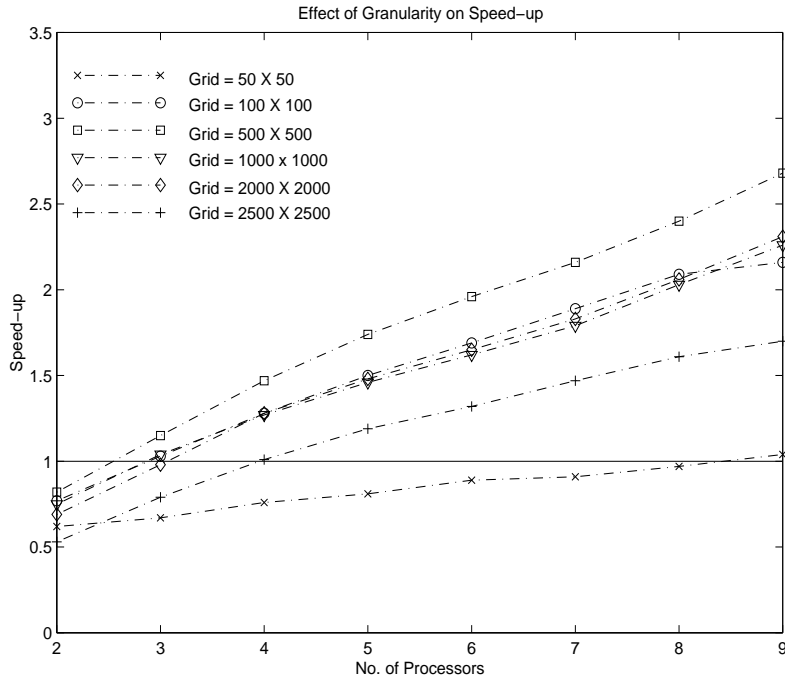


FIG. 7. Effect of granularity on speed-up

Figure 7 illustrates the effect of granularity on speed-up in the case of Jacobi, for multiple sized square grids. As is observed, if the granularity is too small (e.g., in the case of a 50×50 grid), there is slow-down. The optimal speed-up is observed for a grid size close to 500×500 . There are almost identical speed-ups for grids of dimensions 100, 1000 and 2000. When the grid becomes too large (e.g., size 2500×2500), there are other overheads due to message fragmentation and swapping of memory space, etc., which might have contributed to the reduction in speed-up.

5. COMPARISON WITH RELATED WORK

The generic nature of the architectural-skeleton model is one of the features that distinguishes this research from other pattern-based approaches in parallel computing. The generic model contributes towards both flexibility and extensibility, which in turn enhance usability. These are some of the other essential features that are lacking in most of the existing pattern-based approaches to parallel computing.

In the past, several parallel programming systems have supported frequently used parallel interactions [4, 5, 26, 28]. However, in all these cases a fixed number of high-level parallel interactions have been hard-coded into the system. As a consequence, if a user's desired high-level interaction is not supported by a particular system, then the user may have to

give up the idea of using the approach altogether. To achieve higher flexibility, traditionally parallel programmers have relied on low-level communication libraries such as PVM and MPI. So there is clearly a trade off between the ease of development provided by the higher level systems and the flexibility offered by low-level primitive libraries. As discussed, in PASM, a user can mix high-level architectural skeletons with low-level MPI-like message passing. For instance, the internal protocol `PROT_Net` inside the compositional skeleton provides an MPI-like low-level message passing environment. At the same time, the compositional skeleton is just like any other skeleton from the perspective of the generic model and it can easily be intermixed with other skeletons.

Moreover, in all of the previous systems, the supported patterns are tightly integrated into the implementation of the system. So there is no easy way of adding newer patterns without major modifications to the entire system. In PASM, on the other hand, every architectural skeleton is independent of other skeletons and thus adding a new skeleton is a simple matter of extending the skeleton-library.

Tracs [2] is one of the earlier systems known to us that addresses the issue of extensibility. It is a graphical development system, where application development consists of two distinct phases: the definition phase and the configuration phase. During the definition phase, the user graphically defines the three basic components of an application: the message model, the task model and the architecture model. The architecture model defines the software architecture of the parallel application in terms of message and task models. An architecture model defined during this phase can be saved in a user-defined library for later use. During the configuration phase, the programmer constructs the complete application from the basic components, either defined during the definition phase or selected from the system libraries or both. Evidently, *Tracs* supports the idea of extensibility by providing support for an extensible library of user-defined architecture models. However, the type of extensibility realized inside *Tracs* is restrictive. For instance: in *Tracs*, a user can graphically create a 5-slave master-slave pattern and save it inside the library for future use. However, a generic master-slave pattern could have been more useful for this purpose.

As far as we know, *DPnDP* [30] is the first system that addresses both the issues of flexibility and extensibility. It was a nice attempt, but unfortunately it concentrates only on the structural aspects of a pattern and ignores the behavioral aspects (for instance: parallel computing model, communication-synchronization behavior inside a pattern) altogether. In spite of its limitations, *DPnDP* was a good learning experience and it set up the initial stage for this research.

There are various other systems and research projects in the object-oriented domain that are intended for facilitating parallel application development. A majority of them are based on C++ or its extensions. Some of these systems are pattern-related. A comprehensive discussion on many of these systems can be found in [32]. Though none of them bear similarity to this generic architectural-skeleton model, some of them are worth mentioning here.

HPC++ [12] focuses on a common foundation for portable parallel applications. Parts of its implementation are through libraries and parts through C++ language extensions. One of its key features is the exploration of loop parallelism, as in HPF [21]. Another feature is the parallel extension of the C++ standard template library (STL) [24]. The parallel standard template library (PSTL) provides distributed versions of the STL container classes along with versions of the STL algorithms that have been modified to run in parallel. As a major distinction with most pattern-based systems including this architectural-skeleton approach,

PSTL is a class-library and not a framework (i.e., the user selects the library routines for his application and it is the user's application that dominates). Other major differences lie in the HPF-like features such as loop-parallelism, and the extended C++ language syntax.

POOMA (Parallel Object-Oriented Methods and Applications) [8] is a collection of C++ template-classes for writing high performance scientific applications. It provides high-level data-parallel types (for instance: high-level abstractions for multi-dimensional arrays, computational mesh, etc) that make it easy to write parallel PDE (partial differential equation) solvers without worrying about the low-level details of layout, data transfer, and synchronization. In its restricted problem domain, POOMA is able to provide good amount of optimizations for achieving high performance. In comparison, the architectural-skeleton approach is not restricted to any specific problem domain inside parallel computing, and hence it may not be able to offer that amount of domain-specific optimizations as in POOMA.

Similarly, DAPPLE [19] is another C++ class library that provides the illusion of a data-parallel programming language on conventional hardware and with conventional compilers. DAPPLE defines Vectors and Matrices as basic classes, with all the C++ operators overloaded to provide for element-wise arithmetic. In addition, DAPPLE provides typical data-parallel operations that are most commonly applied. In comparison to DAPPLE's exclusive data-parallel domain, the architectural skeleton library is applicable to a much wider range of parallel programming paradigms.

6. CONCLUSION

The paper presents a generic model for designing and developing parallel applications, and is based on the idea of design patterns. The model is based on the message-passing paradigm which makes it well suited for a cluster of workstations or PCs. An architectural skeleton is a physical abstraction of a pattern in parallel computing. The skeleton-based model is an ideal candidate for implementation using object-oriented techniques. The object-oriented approach can be used to build application-independent library of skeletons, while keeping in mind extensibility as one of the major issues. Other issues of equal importance which form integral parts of the model and the system are: flexibility, re-usability (of code for patterns and of application code), separation of specifications, inherent support for hierarchical pattern composition, and hierarchical refinement.

The present collection of architectural skeletons supports those patterns for coarse-grain message-passing computation which can provide good performance in a networked MIMD environment. Research is in progress to incorporate new skeletons for such an environment. Another immediate research interest is to investigate the usability aspects of the approach. It is planned that experiments involving a controlled group of users be carried out in the near future to assess the usability of the PASM system.

REFERENCES

1. LAM/MPI Parallel Computing. <http://www.lam-mpi.org/>.
2. A. Bartoli, P. Corsini, G. Dini, and C. A. Prete. Graphical Design of Distributed Applications Through Reusable Components. *IEEE Parallel and Distributed Technology*, 3(1):37–50, Spring 1995.
3. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Publishing Company, 1999.
4. J. C. Browne, M. Azam, and S. Sobek. CODE: A Unified Approach to Parallel Programming. *IEEE Software*, pages 10–18, July 1989.

5. 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):75–83, Spring 1995.
6. D. K. G. Campbell. Towards the Classification of Algorithmic Skeletons. Technical Report YCS 276, Department of Computer Science, University of York, 1996.
7. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. The MIT Press, Cambridge, Massachusetts, 1989.
8. J. C. Cummings, J. A. Crotinger, S. W. Haney, W. F. Humphrey, S. R. Karmesin, J. V. W. Reynders, S. A. Smith, and T. J. Williams. Rapid Application Development and Enhanced Code Interoperability using the POOMA Framework. In *SIAM Workshop on Object-Oriented Methods and Code Inter-operability in Scientific and Engineering Computing: OO98*.
9. J. Darlington, A. J. Field, and P. G. Harrison. Parallel Programming Using Skeleton Functions. In *PARLE'93*, Munich, Germany, June 1993. Appeared in *Lecture Notes in Computer Science*, Vol. 694, pages 146-160.
10. I. Foster and R. Stevens. Parallel Programming with Skeletons. In *ICPP'90*.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1994.
12. D. Gannon, P. Beckman, E. Johnson, T. Green, and M. Levine. *HPC++ and the HPC++Lib Toolkit*. Department of Computer Science, Indiana University, and Los Alamos National Laboratory. White paper available at <http://www.extreme.indiana.edu/hpc++/>.
13. A. Geist and V. Sunderam. Network-based Concurrent Computing on the PVM System. *Concurrency: Practice and Experience*, 4(4):293–311, 1992.
14. D. Goswami. *Parallel Architectural Skeletons: Re-Usable Building Blocks in Parallel Applications*. PhD thesis, Department of Electrical and Computer Engineering, University of Waterloo, 2001.
15. D. Goswami, A. Singh, and B. R. Preiss. Using Object-Oriented Techniques for Realizing Parallel Architectural Skeletons. In *the third International Symposium on Computing in Object-oriented Parallel Environments (ISCOPE'99)*, San Francisco, USA, December 1999. Appeared in *Lecture Notes in Computer Science*, Vol. 1732, pages 130-141.
16. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, Cambridge, Massachusetts, 1994.
17. N. Harrison, B. Foote, and H. Rohnert, editors. *Pattern Languages of Program Design 4*. Addison-Wesley Publishing Company, December 1999. Software Patterns Series.
18. R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, June 1988.
19. D. Kotz. A data-parallel programming library for education(DAPPLE). In *Twenty-sixth SIGCSE Technical Symposium on Computer Science Education*, pages 76–81, ACM Press, March 1995.
20. R. G. Lavender and D. C. Schmidt. Active object: An object behavioral pattern for concurrent programming. In J. M. Vlissides, J. O. Coplien, and N. L. Kerth, editors, *Pattern Languages of Program Design-2*, Software Patterns Series. Addison-Wesley Publishing Company, 1996.
21. D. B. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, pages 25–42, February 1993.
22. P. E. McKenney. Selecting locking designs for parallel programs. In J. M. Vlissides, J. O. Coplien, and N. L. Kerth, editors, *Pattern Languages of Program Design-2*, Software Patterns Series. Addison-Wesley Publishing Company, 1996.
23. H. R. Myler and A. R. Weeks. *The Pocket handbook of Image Processing Algorithms in C*. Prentice Hall, 1993.
24. M. Nelson. *C++ Programmer's Guide to the Standard Template Library*. IDG Books Worldwide, 1995.
25. M. J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, Inc., 1994.
26. J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The Enterprise Model for Developing Distributed Applications. *IEEE Parallel and Distributed Technology*, 1(3):85–96, August 1993.
27. D. C. Schmidt. ACE: an Object-Oriented Framework for Developing Distributed Applications. In *6th USENIX C++ Technical Conference*, Cambridge, Massachusetts, April 1994.
28. A. Singh, J. Schaeffer, and M. Green. A Template-Based Tool for Building Applications in a Multicomputer Network Environment. *Parallel Computing* 89, pages 461–466, 1989.
29. A. Singh, J. Schaeffer, and D. Szafron. Experience with parallel programming using code templates. *Concurrency: Practice and Experience*, 10(2):91–120, 1998.

30. S. Siu and A. Singh. Design Patterns for Parallel Computing Using a Network of Processors. In *Sixth IEEE International Symposium on High Performance Distributed Computing*, pages 293–304, Oregon, USA., August 1997.
31. L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 1996.
32. G. V. Wilson and P. Lu, editors. *Parallel Programming using C++*. The MIT Press, 1996.