Architectural Skeletons: The Re-Usable Building-Blocks for Parallel Applications

Dhrubajyoti Goswami, Ajit Singh and Bruno R. Preiss
Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, Ontario
N2L-3G1, Canada

Abstract *Design pattern concepts are being used in the various disciplines of computing. In the parallel computing domain, design patterns specify recurring parallel computing problems and their solution strategies. The paper defines a generic (i.e. application and pattern independent) model for realizing and using parallel design patterns. The term architectural skeleton is coined to represent the application independent, generic set of attributes associated with a pattern. The model is aimed at providing many of the functionalities of MPI, plus the benefits of the patterns. The intent is to provide a considerable amount of flexibility to the user in application development. The generic model also enhances usability. As it turns out, the model is a natural candidate for object-oriented style of design and implementation. It is currently implemented as a C++ template-library without requiring any language extension. The generic model, together with the object-oriented and library-based approach, facilitates extensibility (i.e. new patterns can be added to the system library without requiring any major modifications to the existing system).*

Keywords: Design patterns in parallel computing, parallel programming environments, skeleton-based parallel programming, high performance computing models, software tools for parallel programming.

1 Introduction

The idea of design-patterns is well recognized in the various disciplines of computing. In the context of object-oriented design methodologies, *design patterns* [4] are used to specify solution strategies for solving recurring design problems in systematic and general ways. Similarly, in the parallel computing domain, design patterns specify recurring parallel computational patterns and their solution strategies. Examples of such recurring patterns are: static and dynamic replication, divide and conquer, data parallel pattern with various topologies, compositional framework for irregularly-structured control-parallel computation, systolic array, singleton pattern for single-process (single- or multi-threaded) computation.

Starting with the late 1980s, several pattern-based systems have been built, e.g. Frameworks and Enterprise [6], CODE and HeNCE [1], DPnDP [7], with the good intention of facilitating application development using some of these patterns. In a separate but similar thread of work, a group of researchers started exploring parallel patterns as high level functional constructs. The term *algorithmic skeleton* was coined [3] to specify higher-order functions with specific implementations tailored to particular parallel architectures. Most of the algorithmic skeleton research [2] concentrates on various functional and logic programming languages for abstracting and representing recurring parallel patterns.

However, most of the pattern-based systems mentioned previously support only a limited set of patterns in ad hoc ways. There is no generic or canonical model of a pattern. This substantially hampers the usability of such systems. Besides usability, there are two other very important aspects: flexibility and extensibility. Most of the systems are hard-coded with a limited set of patterns, and there is no clear way for adding new patterns to the system when required (i.e. lack of extensibility). Besides, if a certain desired parallel structure is not supported by a design-pattern-based system, there is often no alternative but to entirely abandon the idea of using the particular system (i.e. lack of flexibility). A detailed discussion regarding the desirable characteristics and the shortcomings of some of these systems can be found in [6].

Algorithmic skeletons, on the other hand, rely on abstract mathematical models exclusively based on various functional and logic programming languages. Though such abstract models make good contributions to academics, they face serious problems with respect to the mainstream adoption of pattern-based parallel computing, where the conventional languages like C, C++, Fortran and Java are the languages of developer's choice.

This paper defines an *architectural skeleton* as a generic building-block in terms of pattern- and application-specific virtual machines. A virtual machine specifies the application-related structure and the communication-synchronization behavior of particular parallel design pattern. Unlike an algorithmic skeleton, an architectural skeleton is not a high level functional construct. Rather, it contains the ingredients (i.e. application independent attributes) for constructing a collection of pattern- and application-specific virtual machines. Each such virtual machine is equipped with pattern related communication-synchronization protocols, using which the virtual machine can interact with other such machines. The model supports top-down hierarchical refinements such that a virtual machine can contain other virtual machines. Consequently, every parallel application can be structured in a top-down hierarchical fashion, starting with the root of the hierarchy. The next section elaborates the model.

The hierarchy and the protocols, together with the support for using many of the functionalities similar to those in MPI and PVM, are aimed at providing the necessary flexibility to the user in application development. The model naturally supports object-oriented style of design and implementation. As a result, it can be elegantly implemented using C++ without requiring any language extensions, or graphical user interfaces. At the same time, it does not preclude the use of GUIs for building high level programming environments. Presently, it has been fully implemented using C++. The generic model, together with the object-oriented and library based approach, facilitates extensibility. I.e. new patterns can be added to the library without requiring any overall change to the existing system.

2 The Model

An *architectural skeleton* is a collection of attributes, which encapsulate the structure and the behavior of a parallel design pattern in an application independent manner. User extends a skeleton by filling in the various application specific parameters associated with these attributes. User's extension of a skeleton results in one or more *virtual machine*s. A virtual machine is yet to be filled in with application specific code. Once a virtual machine is complete with application code, it results in a *parallel computing module*. A parallel application is a systematic collection of instantiations of these modules. Figure 1 approximately illustrates the various phases of application development using architectural skeletons. The figure here shows the 2-D Mesh pattern and associated structural parameters consisting of the dimensions of the mesh.

A virtual machine inherits all the properties associated with a skeleton. Besides, it has additional properties pertaining to the 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 design pattern. A virtual machine is an application-specific *specialization* of a skeleton.

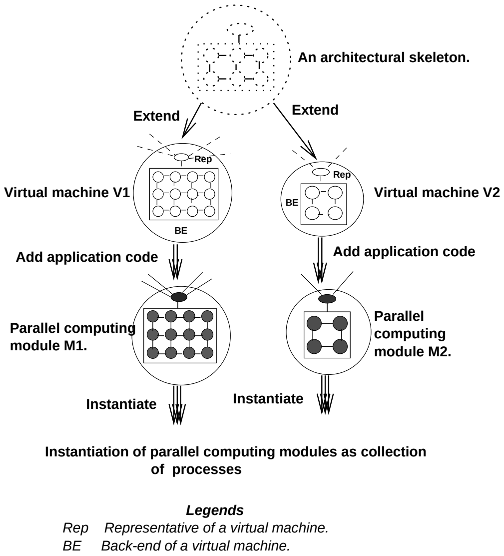

Figure 1: Relationships between an architectural skeleton, a virtual machine and a module

Since all the attributes associated with an architectural skeleton are inherited by a virtual machine, and the attributes convey more useful meaning in the context of a virtual machine, the following description precisely does so. Irrespective of the pattern type, a virtual machine, V_m, has the following generic set of attributes:

- *Rep* is the representative of the virtual machine. When filled in with application specific code, *Rep* acts as the entry and the exit point for V_m in terms of its interaction with other virtual machines.

- *BE* is the back-end of the virtual machine. Formally, $BE = \{V_{m1}, V_{m2}, ..., V_{mn}\}$, where each V_{mi} is itself a virtual machine. Note that the notion of a collection of virtual machines inside another virtual machine results in a tree-structured hierarchy. Consequently, each V_{mi} is called a *child* of V_m, and V_m is called the *parent*. Virtual machines belonging to the same back-end are *peers* of one another.

- *Topology* is the interconnection tolpology specification of the virtual machines inside back-end, and their connectivity specification with *Rep*.

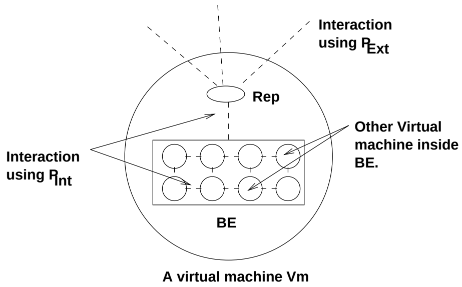

Figure 2: A virtual machine

- P_{Int} is the internal communication-synchronization protocol of the virtual machine. The internal protocol is an inherent property of the skeleton, and it captures the parallel computing model of the pattern and the topology that the skeleton is designed for. Using P_{Int}, V_m can interact with the virtual machines inside its back-end, and a virtual machine inside the back-end can interact with its peers.

- P_{Ext} is the external communication-synchronization protocol of the virtual machine. Using the primitives inside P_{Ext}, a virtual machine 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. the virtual machine V_m adopts to the context of its parent by adapting the internal protocol of its parent as its external protocol.

Figure 2 diagrammatically illustrates the above mentioned attributes for a specific virtual machine, which extends a data-parallel architectural skeleton designed for 2-D mesh topology.

Though a virtual machine is an application specific specialization of an architectural skeleton, it is still devoid of any application code. User writes application code on the virtual machines using the communication-synchronization protocols, P_{Int} and P_{Ext}. When a virtual machine and all its children are complete with application code, the resultant virtual machine is alternately called a *parallel computing module*. A parallel application is a hierarchical collection of the instantiated copies of the modules.

As mentioned before, the notion of parent-child relationships among virtual machines (and parallel computing modules) results in 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 *hierarchical tree* associated with the application. For instance: (1) in a dynamically replicated master-worker application, the *Master* module forms the root of the hierarchy, and the dynamically replicated child *Worker* modules form the sub-trees. (2) In an application consiting of the three modules: *Producer*, *Worker* and *Consumer*, a *compositional module* forms the root of the hierarchy, and its three children (i.e. *Producer*, *Worker* and *Consumer*) form the sub-trees.

The notion of the hierarchical tree is important, because the OO implementation dynamically constructs the hierarchy associated with an application, while completely hiding it from the user. A *singleton module*, which has no children, forms a leaf in the hierarchy.

For obtaining better performance, it is often desirable to replace one or more of the parallel computing modules inside an application with other module(s). Such a replacement of a module, where only the sub-tree with the module as its root is affected and the rest of the application remains intact, is called a *refinement.* The ability for refining a parallel application is a desirable characteristic.

The next section includes an example which illustrates most of the important features of the model including refinement.

3 An Implementation

This section presents a brief outline of the approach used for implementing architectural skeletons. A detailed discussion regarding the various implementation-issues is beyond the scope of this paper.

The model is presently implemented in industry standard C++ (SunCC compiler, V 4.1 and upwards), without requiring any language extensions. MPI is used as the underlying communication library. Work is continuing for porting it to GNU C++, and the C++ compiler for AIX (i.e. the xlC compiler) on RS6000.

A specification-language based textual interface, implemented in PERL, helps the user in the various stages of application development. It is parsed to produce the back-end C++ code. However it should be emphasized that use of a specification language is not a language extension. It merely helps the user to bypass certain C++-based details. If desired, a user can directly work in C++.

Other important features of the implementation include:(1) use of C++ operator-overloading to implement certain primitives (e.g. send, receive, etc.) inside protocol classes; (2) use of marshaling and unmarshaling mechanisms, whereby data attributes belonging to an entire object can be marshaled, shipped and then un-marshaled over a communication link without the usual hassles of data packing and unpacking. The following two examples illustrate the several features of the model and the current user's interface.

3.1 Example 1

The following example illustrates a *singleton module*, which does nothing more than printing the string *Hello World.* Being a single process entity, a singleton has no children. *Rep* is the representative. When the representative code is not filled, what we have is a singleton machine. The example uses the current specification language.

```
// My simple sequential program.
MyModule EXTENDS SingletonSkeleton
{
    Rep {
        printf ("Hello World\n");
    }
}
```

The parser, implemented in PERL, translates the above to produce the following C++ code.

```
// Automatically generated file: Pmain.cc.
#include "BasicDef.h"
#include "VoidClass.h"
#include "SingletonSkeleton.h"
// Global definitions will go below:
//-------------------------------------
//-------------------------------------
// Generated code for module: "MyModule"
class MyModule : public SingletonSkeleton <Void>
{
public:
   MyModule() {};
   virtual void Rep() {
      printf ("Hello World\n");
   }
  // Miscellaneous local definitions go below:
  //--------------------------------
  //--------------------------------
};
void Pmain()
{
  MyModule TopLevel_524;
  TopLevel_524.Run();
}
```

It should be evident that a user can directly write his program in C++. The parser merely reduces some of his extra work. The generated file *Pmain.cc* is compiled and linked with the skeleton library to produce the executatble file. The executable can be run on a cluster of workstations (a single workstation in this case), which have MPI installed on them.

In this specific example, the singleton module is stand-alone, i.e. it has no parent and peers, and hence its adaptable external protocol is undefined. The adaptable external protocol is realized as a C++ template, which is specified as *Void* in this case. The module forms both the root and the leaf of the hierarchy. Since the back-end of a singleton machine is empty, its internal protocol is also undefined (not shown here).

3.2 Example 2

Let us consider the graphics animation program [6] consisting of three modules: *Generate*, *Geometry* and *Display*. The program takes a sequence of graphics images, called 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 illustration demonstrates one way of implementing it using the specification language. The implementation uses a compositional skeleton and a singleton skeleton. Here, the *Root* compositional module forms the root of the hierarchy, and its three children, *Generate*, *Geometry* and *Display* form the subtrees. Each of the three children is a singleton module, and hence is a leaf of the hierarchy. The internal communication-synchronization protocol for a compositional machine is PROT_Net = {Send(...), Receive(...), Broadcast(...), Spawn(...),...}. Consequently, PROT_Net becomes the external communication-synchronization protocol for each of the three children.

GenerateGeometry and *GeometryDisplay* are user defined objects, whose data attributes can be marshaled, shipped and then unmarshaled over a communication link, without the usual hassles of data packing and unpacking. 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 primitives (e.g. *Send(...)*, *Receive(...)*).

```
GLOBAL {
#include "geom.h"
#define MAXIMAGES 120

class GenerateGeometry : public UType {
  Int imageNumber;//System defined wrapper for "int"
  ObjTable table;//"objTable" is defined in "geom.h"
public:
  virtual void Marshal() {imageNumber.Marshal();
                          table.Marshal();};
  virtual void UnMarshal() {imageNumber.UnMarshal();
                            table.UnMarshal();};
  // Constructor(s) etc...
  ...
};
class GeometryDisplay : public UType {
  Int imageNumber;
  Int nPoly;
  PolyTable table;
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 case, the representative is idle.
   }
}
// The "Generate" module, which extends a singleton
// skeleton.
Generate EXTENDS SingletonSkeleton
{
  // A singleton extension 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).
    }
  }
  // All local definitions go below:
  LOCAL {
     void ComputeObjects(GenerateGeometry& Work)
     {
     //User code for "ComputeObjects" will go 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).
        DoConversion(Work, Frame);
        Display << Frame;
     }
  }
  LOCAL {
  // local definition for DoConversion(...)
  }
}
// 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 for DoHidden(...) and
  // WriteImage(...).
  }
}
```

3.3 Refinement

It is generally the case that *Display* is the most time consuming of the three child modules. Consequently, the singleton *Display* module is *refine*d to a dynamically replicated *Display* module. None of the other modules is affected by this change. The change in the implementation is illustrated next.

```
// 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;
         image++;
         success = SendWork(Frame); // Keep sending
// work-loads to workers until none is free and
// can no longer spawn one dynamically. It is a
// member of the internal protocol: PROT_Repl.
      }
      if (!success) { // Do it myself, if not
              //succesful in assigning it to a worker.
        DoHidden(Frame);
        WriteImage(Frame);
      }
      if (image == MAXIMAGES) break;
    }
  }
  LOCAL {
  // Local definitions for DoHidden(...) and
  // WriteImage(...).
  }
}
// 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 for DoHidden(...) and
     // WriteImage(...).
     }
}
```

Only the subtree, with *Display* at its root, is affected by this change and the rest of the application remains intact. The internal communication-synchronization protocol for a replication machine is PROT_Repl = {SendWork(...), ReceiveResult(...), ReceiveWork(...), SendResult(...),...}. Consequently, PROT_Repl becomes the external communication-synchronization protocol for each replicated child *Worker*.

The other modules can also be refined in a similar fashion.

4 Experimental Results

Experiments were conducted to assess the performance of the system. The same set of experiments was conducted using MPI and the

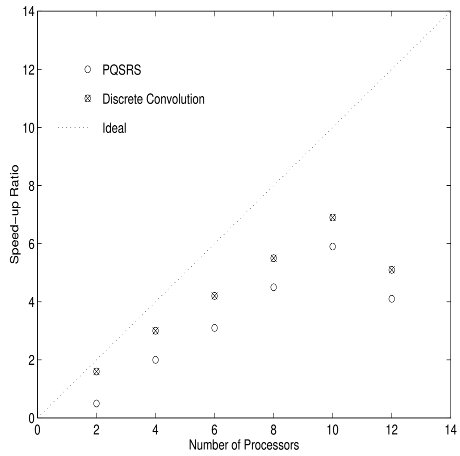

Figure 3: Speed-up Ratio Vs. Number of processors

results were compared. The measured difference in performance is within $\pm5\%$, which can be attributed to the fact that the skeleton-library is implemented as an extremely thin layer on top of MPI. Figure 3 illustrates the performance results obtained for PQSRS (parallel quick sort using regular sampling) and 2-D discrete convolution algorithms. Both implementations use data-parallel skeletons for mesh topology. PROT_Mesh is the internal communication-synchronization protocol. The first application requires a significant amount of peer-to-peer interaction which is supported by the protocol as part of the data-parallel skeleton, and it cannot be implemented using most other pattern-based systems. Detailed descriptions of the experiments and the results can be found in [5].

5 Conclusion

The abstraction of the high-level architectural-skeletons as re-usable components for commonly used parallel computational patterns is a challenging goal. To the best of our knowledge, this skeleton-based approach is the first of its kind that aims at providing a standard model for a parallel computing pattern. The approach can be used to build application-independent library of skeletons, while keeping in mind flexibility and extensibility as two of the major issues. The present set of architectural-skeletons supports coarse-grain message-passing patterns that provide good performance in a networked MIMD environment. Incorporation of new skeletons for such an environment is an ongoing research activity.

References

[1] 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.

[2] D.K.G. Campbell. Towards the Classification of Algorithmic Skeletons. Technical Report YCS 276, Department of Computer Science, University of York, 1996.

[3] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* The MIT Press, Cambridge, Massachusetts, 1989.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, 1994.

[5] D. Goswami. *A Design Pattern Based Approach for Developing Parallel Applications.* PhD thesis, Department of Electrical and Computer Engineering, University of Waterloo, 1999. In Preparation.

[6] A. Singh, J. Schaeffer, and D. Szafron. Experience with parallel programming using code templates. *Concurrency: Practice and Experience*, 10(2):91–120, February 1998.

[7] 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.