# Using Object-Oriented Techniques for Realizing Parallel Architectural Skeletons

Dhrubajyoti Goswami, Ajit Singh, and Bruno R. Preiss

Department of Electrical and Computer Engineering,
University of Waterloo,
Waterloo, Ontario N2L-3G1, Canada
{goswami,asingh}@etude.uwaterloo.ca
brpreiss@despot.uwaterloo.ca

**Abstract.** The concept of design patterns has recently emerged as a new paradigm in the context of object-oriented design methodology. Similar ideas are being explored in other areas of computing. In the parallel computing domain, design patterns describe recurring parallel computing problems and their solution strategies. Starting with the late 1980's, several pattern-based systems have been built for facilitating parallel application development. However, most of these systems use patterns in ad hoc manners, thus lacking a generic or standard model for using and intermixing different patterns. This substantially hampers the usability of such systems. Lack of flexibility and extensibility are some of the other major concerns associated with most of these systems. In this paper, we propose a generic (i.e., pattern- and application-independent) model for realizing and using parallel design patterns. The term *architectural skeleton* is used to represent the application independent, re-usable set of attributes associated with a pattern. The model can provide most of the functionalities of low level message passing libraries, such as PVM or MPI, plus the benefits of the patterns. This results in tremendous flexibility to the user. It turns out that the model is an ideal 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.

## 1 Introduction

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 (i.e., sequential) computation.

Starting with the late 1980s, several pattern-based systems have been built, e.g. Frameworks and Enterprise [9], CODE and HeNCE [1], DPnDP [10], for facilitating parallel application development. 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 introduced [3] to specify higher-order functions with specific implementations tailored to particular parallel architectures. The algorithmic skeleton research [2] concentrates on various functional and logic programming languages for abstracting and representing recurring parallel patterns.

Some other approaches have taken the path of building a new language to support program composition using skeletons [7], while others are based on the extension of existing languages, e.g. Frameworks and Enterprise [9].

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 fixed set of patterns, and there is no clear way to add 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 [9].

This paper defines an *architectural skeleton* as a generic building-block which captures a parallel design pattern in an application-independent manner. Once the required structural and behavioral parameters are assigned to an architectural skeleton, it results in an *abstract parallel computing module* (abbreviated as an *abstract module*). User subsequently attaches application code to an abstract module, which results in a *concrete parallel computing module* (abbreviated as a *concrete module* or simply a *module*). A parallel application consists of one or more mutually interacting, concrete modules.

The model supports top-down hierarchical refinements such that a module, abstract or concrete, can be a composition of other modules. Consequently, every parallel application can be structured in a top-down hierarchical fashion, starting with the root of the hierarchy.

At the beginning of this research, we were not committed to any particular design methodology or implementation style. However, as the work proceeded, the model with the desired capabilities turned out to be an ideal candidate for object-oriented style of design and implementation. Presently, it has been fully implemented using industry-standard 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.

A pattern can support low- or high-level communication and synchronization protocols. At the same time, multiple patterns can work together based on a generic scheme. Support for hierarchical design and low-level as well as high-level protocols provide the additional flexibility, not found in the existing parallel systems that aim to support design patterns.

## 2 The Architectural Skeleton Based Model

We start this section with an example. The objective is to introduce the various concepts of the model and its implementation, before elaborating them in detail. The same example will be used at various 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 An Example

The following example illustrates a master-worker application, where the `Master` *module* produces a succession of image frames and sends them to dynamically replicated `Worker` *modules* for processing. 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 workload and the number of available processors. Each replicated `Worker` *module* extends a singleton *skeleton*, which is used for single-process computation. Each replicated `Worker` is a *child* of the `Master` *module*.

The `Rep` associated with each *module* is its *representative* (lines 18 and 52). 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 replicated `Worker` using its *internal communication/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` (lines 21 and 3) is a user-defined object 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 a textual specification language. The specification language helps the user bypass certain C++-related and other monotonous details. As it will be evident in the next section, a user can directly write his code in C++, if desired.

```
1:  GLOBAL{
2:  #include "ImageDef.h"
3:     class Image: public UType
4:     {
5:        // A class definition specifying the attributes of an image.
```

```
        ...
    };
    // Similarly, global definitions of ProcessImage(...) and
    // WriteImage(...) go here. These procedures are used in the
    // following code segments.
}

// The "Master" module for dynamic-replication  based computation.
Master EXTENDS ReplicationSkeleton
{
  // The dynamically replicated children of "Master"
  CHILDREN = Worker;
  Rep {
       int Number_of_Images = 0;
       int success;
       Image Frame;
       while (True){
           success = True;
           while ((Number_of_Images < MAXIMAGES) && success){
              Produce(Frame);
              Number_of_Images++;
              success = SendWork(Frame); // Keep sending work-loads
       // to workers until none is free and can no  longer spawn  one
       // dynamically. SendWork is a member function of  the internal
       // protocol: PROT_Repl.
           }
           if (!success) { // Do it  myself, if failed to assign work
                           // to a worker.
                ProcessImage(Frame);
                WriteImage(Frame);
           }
           if (Number_of_Images == MAXIMAGES) break;
        }
  }
  LOCAL {
     // Local functions used by this module go here.
     void Produce (Image& Frame)
     {
       // User code for "Produce"  goes here:
     }
  }
}

// Each replicated "Worker" module.
Worker EXTENDS SingletonSkeleton
{
   Rep {
        Image Frame;
        ReceiveWork(Frame); // ReceiveWork is a member function of
                            // the external protocol: PROT_Repl.
        ProcessImage(Frame);
        WriteImage(Frame);
   }
}
```

### 2.2 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 structural and behavioral parameters as needed by the application at hand. Fig. 1(a) roughly illustrates the various phases of application development using architectural skeletons. As shown in the figure, different extensions 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.

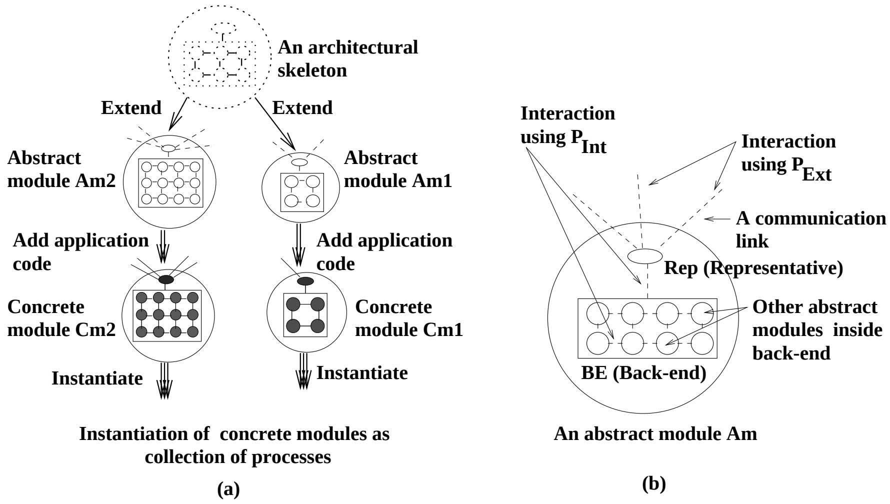


**Fig. 1.** (a) Relationships between an architectural skeleton, an abstract module and a module. (b) Structure of an abstract module

An abstract module inherits all the properties associated with a skeleton. Besides, it has additional parameters 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 design pattern. An abstract module is an application-specific *specialization* of a skeleton.

Irrespective of the pattern type, an abstract module, $A_m$, inherits (from an architectural skeleton) the following generic set of attributes. Fig. 1(b) diagrammatically illustrates the attributes of an abstract module that extends a data-parallel architectural skeleton designed for 2-D mesh topology.

- *Rep* is the representative of the abstract module. When filled in with application code, *Rep* represents the module in its action and interaction with other modules.
- *BE* is the back-end of the abstract module. Formally, $BE = \{A_{m1}, A_{m2},\ldots, A_{mn}\}$, where each $A_{mi}$ is itself an abstract module. Note that the notion of a collection of abstract modules inside another abstract module results in a tree-structured hierarchy. Consequently, each $A_{mi}$ is called a *child* of $A_m$, and $A_m$ is called the *parent*. Abstract modules belonging to the same back-end are *peers* of one another.
- *Topology* is the interconnection topology specification of the abstract modules inside the back-end, and their connectivity specification with *Rep*.
- $P_{Int}$ is the internal communication/synchronization protocol of the abstract module. The internal protocol is an inherent property of the associated skeleton and it captures the parallel computing model of the corresponding pattern as well as the topology. Using the primitives inside $P_{Int}$, the representative of $A_m$ can interact with the abstract modules in its back-end, and an abstract module in the back-end can interact with its peers.
- $P_{Ext}$ is the external communication/synchronization protocol of the abstract module. Using the primitives inside $P_{Ext}$, an abstract module (i.e., its representative) 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 module $A_m$ adapts to the context of its parent by using the internal protocol of its parent as its external protocol. For example, in our previous master-worker example, `PROT_Repl` is used as $P_{Ext}$ for the `Worker` module and as $P_{Int}$ for the `Master` module.

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*, or simply a *module*. A parallel application is a hierarchical collection of instances of the modules.

As mentioned before, the notion of parent-child relationships among modules result 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 the previous example, the `Master` module forms the root of the hierarchy, and the dynamically replicated children `Worker` modules form the sub-trees. (2) In an application consisting 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 concept 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.

## 3 An Object-Oriented Implementation

The model is presently implemented in industry standard C++ (SunCC compiler, V 4.1), 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, whose parser is 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 monotonous C++-based details that can easily be generated using PERL. An expert in C++, for example, may want to directly develop his application in C++.

Other important features of the implementation include: (1) use of C++ operator-overloading to implement certain primitives inside protocol classes (e.g. `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 Specification Language Parser

The example in the previous section illustrates the current user's interface which uses a specification language. The specification language parser, implemented in PERL, translates it to produce the back-end C++ file: *Pmain.cc*. It is subsequently compiled and linked with the skeleton library to produce the executable file, designated to run on a workstation cluster.

The following is the skeleton of the automatically generated file *Pmain.cc*. As mentioned before, a user can directly write his code in C++. The specification language and its parser merely reduce some of the extra work by automating certain monotonous steps.

```
#include "BasicDef.h"
#include "SingletonSkeleton.h"
// Similarly other "include" files go here.
//Any global definitions will go below:
//-----------------------------------
#include "ImageDef.h"
  class Image: public UType
  {
    // A class definition specifying  the attributes of an 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 go 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 go below:
};
void Pmain()
{
    Master TopLevel_366;
    TopLevel_366.Run();
}
```

As it is evident from the previous code, C++ templates are used extensively in the implementation. `PROT_Repl` is the internal protocol for the replication skeleton. Consequently, `PROT_Repl` becomes the external protocol for a `Worker` module. Since the `Master` is at the root of the hierarchy, its external protocol is undefined (as specified by `Void`).

### 3.2 Implementing Architectural Skeletons

Fig. 2 illustrates the high-level class diagram behind the design of the skeleton library, using the standard UML notation. For simplicity, the figure does not illustrate the relationships between the skeleton classes and the various protocols. Besides, the various attributes and the methods associated with each class, and the formal parameters associated with each inherited skeleton class are not shown for a cleaner representation.

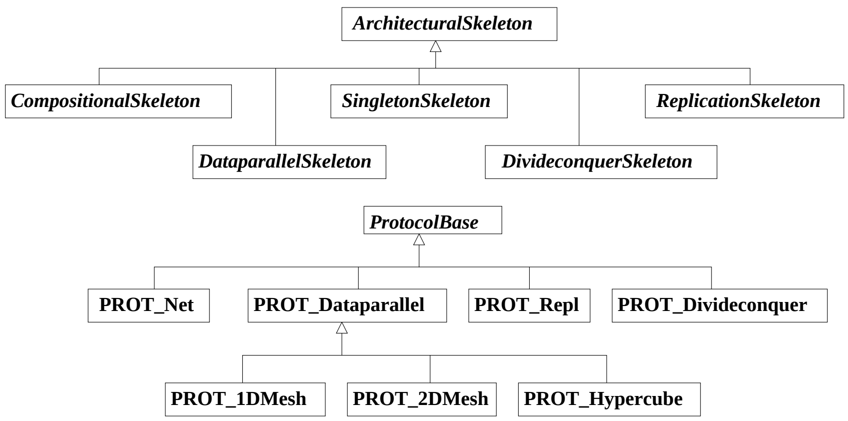


**Fig. 2.** High level class diagram for the skeleton-library

Fig. 3 shows more details of the high level design pertaining to the replication and the singleton skeletons, using UML notation. These skeletons are used in our master-worker example in Sect. 2.1. The figure also illustrates the high level class diagram of these skeletons as used in the example. As it is evident from the figure, each skeleton class is implemented as a template, with its list of formal parameters. A template class is subsequently parameterized with actual application-specific value-list to create a bound-class, which is further extended by the user.

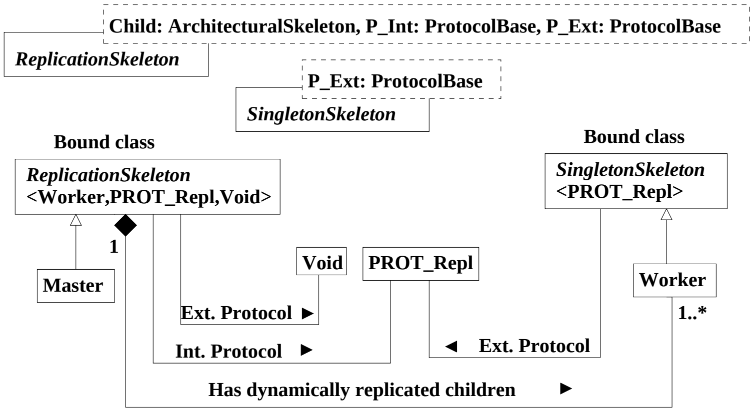


**Fig. 3.** High level class diagram for the example

### 3.3 The Dynamic Execution Model

After viewing the static class diagram of the skeleton library and the application, we are in a position to briefly discuss the dynamic execution model. The execution model is SPMD, 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. Consequently, each process falls through the same hierarchical tree associated with the application, starting at the root of the tree. Fig. 4(a) illustrates the hierarchical tree associated with the master-worker example discussed in Sect. 2.1. Fig. 4(b) illustrates the hierarchical tree associated with a hypothetical `Producer-Worker-Consumer` application. Here, the `Worker` module further sub-divides its work-loads among dynamically replicated sub-workers.

A node of a hierarchical tree 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. Each process starts at the root of the hierarchy and then traverses down the tree to its designated node.

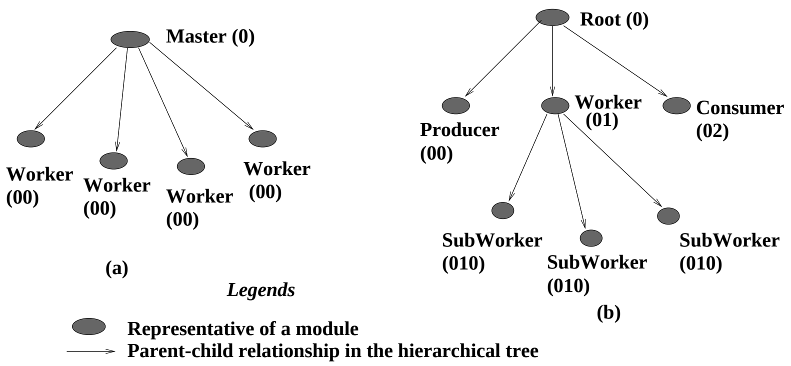


**Fig. 4.** Hierarchical tree and its traversal scheme

How does a process know, 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 Fig. 4, the printed string inside parentheses pairs beside each node is precisely the identification string of the process which executes it. The dynamically replicated `Workers` in the first application are all identical, therefore they execute the same code and have the same identification string. The same situation applies to the dynamically replicated `SubWorkers` in the second application.

All of the above issues are completely hidden from the user. In fact, for instance, a user follows the general structure as illustrated by the master-worker application in Sect. 2.1, and writes his application with the perspective that he is dealing with individual parallel computing 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 parallel computing module.

## 4 Performance Results

Experiments were conducted to assess the performance of the system. The results were compared with direct MPI-based implementations. The performance difference is within $\pm 5\%$, which can be attributed to the fact that the skeleton-library is implemented as an extremely thin layer on top of MPI. The results obtained for the PQSRS algorithm [8] and the 2-D discrete convolution algorithm [6] are described in this section. Detailed results with different applications can be found in [5].

PQSRS (Parallel Quick Sort using Regular Sampling) is a parallel version of quick sort, shown to be effective for a wide variety of MIMD architectures. It uses a master-slave pattern, which is easily realized using the data-parallel skeleton for mesh topology and the singleton skeleton (analogous to the structure of the example in Sect. 2.1). The algorithm works in the following steps: (1) the master 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. The second set of experiments involves a relatively trivial application, called the 2-D discrete convolution algorithm. This is an image processing algorithm, where a mask is applied to the image pixels to produce a convoluted image. Like most other image processing algorithms, it follows the master-slave pattern and can be elegantly implemented using the same set of skeletons as in the example presented in Sect. 2.1. Unlike PQSRS, no peer-to-peer interaction among slaves is needed for this specific application.

Fig. 5 illustrates the results obtained for sorting 10000 randomly generated objects using PQSRS, and for the discrete convolution of a $400 \times 400$ pixel image using a $5 \times 5$ mask. The underlying hardware is a cluster of Sun Sparc workstations connected by a 10-megabit ethernet network. For the discrete convolution algorithm, the speed-up ratio was measured with respect to the best sequential algorithm. In the case of PQSRS, the speed-up ratio was measured with respect to the same sequential quick-sort routine used inside PQSRS. As it turns out in both the applications, the granularity becomes too small with more than 10 processors and hence the performance gradually degrades.

## 5 Conclusion

The paper presents a generic model for designing and developing parallel applications, which is based on the idea of design patterns. Patterns are abstracted as architectural skeletons. The 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 flexibility and extensibility as two of the major issues. The present set of architectural-skeletons supports patterns for coarse-grain message-passing computation that

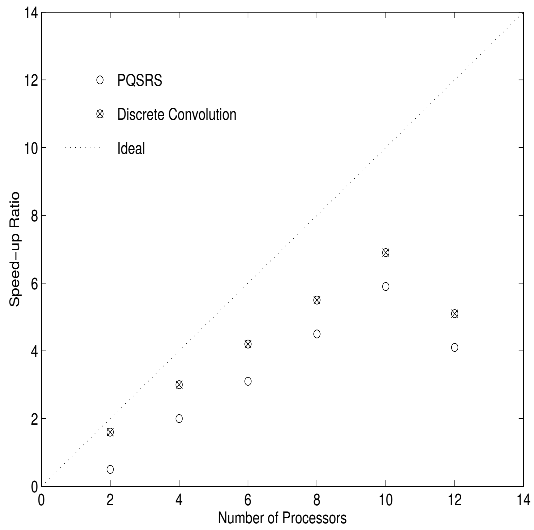


**Fig. 5.** Speed-up ratio versus number of processors

provide good performance in a networked MIMD environment. Incorporation of new skeletons for such an environment is an ongoing research activity.

## References


1. Browne, J.C., Hyder, S.I., Dongarra, J., Moore, K., Newton, P.: Visual Programming and Debugging for Parallel Computing. IEEE Parallel and Distributed Technology. (1995) 3(1):75–83
2. Campbell, D.K.G.: Towards the Classification of Algorithmic Skeletons. Technical Report YCS 276, Department of Computer Science, University of York (1996)
3. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. The MIT Press, Cambridge, Massachusetts (1989)
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company (1994)
5. Goswami, D.: A Design Pattern Based Approach for Developing Parallel Applications. PhD thesis, Department of Electrical and Computer Engineering, University of Waterloo. (In preparation)
6. Myler, H.R., Weeks, A.R.: The Pocket handbook of Image Processing Algorithms in C. Prentice Hall (1993)
7. Pandey, R., Browne, J.C.: A Compositional Approach to Concurrent Programming. In Proc. of 1996 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96), California. (1996) 1489–1500
8. Quinn, M.J.: Parallel Computing: Theory and Practice. McGraw-Hill, Inc. (1994)
9. Singh, A., Schaeffer, J., Szafron, D.: Experience with parallel programming using code templates. Concurrency: Practice and Experience. (1998) 10(2):91–120
10. Siu, S., Singh, A.: Design Patterns for Parallel Computing Using a Network of Processors. In Sixth IEEE International Symposium on High Performance Distributed Computing, Oregon, USA. (1997) 293–304