A Pattern-Based Model for Developing Parallel Applications Using a Network of Processors

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

1 The Problem

Parallel programming is complicated. This complexity arises from the compounding of low-level parallelism related issues with the problems of writing good sequential code. Over the years, various approaches have been proposed to aid parallel program developers. These approaches employ a high-level model of parallel computation, thus hiding the low-level parallelism-related details from the user. Different approaches employ different abstraction techniques, such as communication libraries, macros, new parallel languages and abstract data types. In this paper, we present a pattern-based approach to parallel application development, which uses frequently occurring structures for parallelism. A pattern (more precisely, a parallel computing pattern) is a re-usable, application-independent component providing a commonly used parallel structure. It is implemented as a re-usable code-skeleton for quick and reliable development of parallel applications. We believe that many of the parallel applications in use today are composed of at least one of these commonly-used patterns. One or more of these existing patterns can be composed, through a generic hierarchically-defined model and a textual interface, to create the entire parallel application. The benefit is two-fold: (1) each pattern is well documented through examples and usages, etc. This provides an user the in-depth knowledge about it. As a result, successive usages of the same pattern in different applications should reduce considerable amount of learning time; (2) the re-usable, application-independent components hide most of the low-level parallelism-related details from the user. These low-level details are often quite tedious and error-prone to implement from scratch. Thus, use of these re-usable components should reduce considerable amount of development time.

Unlike most other similar approaches in the past, where the user was constrained to work in a high-level model which made him compromise most of his flexibility, this development model allows the user to intermix patterns with low-level primitives (similar to those in PVM/MPI). This gives the user the comfort of developing his application in a much higher level of abstraction without compromising his flexibility. At the same time, unlike most other previous approaches, this library-based approach is extendible by allowing a user/system designer to add new patterns as per need. Flexibility and extendibility, together with a generic hierarchical view of a parallel computing pattern with associated structural and behavioral components and a resultant well-defined compositional model, makes this approach distinguishable from the similar approaches in the past. To support our model, a development system using the object-oriented design methodologies in C++ and the standard message-passing interface (MPI) has been implemented and tested for a
number of patterns on workstation-clusters. The remaining part of the paper concentrates on the model, illustrates an exemplar user-interface and discusses some results obtained so far.

2 An overview of the model

![Diagram of parallel computing pattern](image)

**Figure 1:** Structure of a parallel computing pattern

A parallel computing pattern $P$ is composed of the following (also refer to Figure 1):

- a logical process entity, called the representative of the pattern, which represents $P$ to the rest of the universe;

- a cluster of zero or more patterns, called the pattern cluster. The patterns inside the pattern cluster are the child patterns of $P$. Any two patterns inside the pattern cluster are called peers. (Note that there are patterns inside a pattern. This gives a hierarchical view to a pattern);

- a topology specification, which determines the interconnection topology of the representatives of the child patterns. The representative of $P$ is implicitly connected to the representatives of the child patterns;

- a parallel computing model, which is an integral part of the pattern. This model determines the way the parent and the child patterns can interact with one another;
• a set of communication protocols: (1) an internal protocol, \( P_{\text{int}} \), for interactions between the representative and the child patterns and for interactions among the peers; (2) an external protocol, \( P_{\text{ext}} \), for its interaction with the rest of the universe. The internal protocol captures both the parallel computing model of \( P \) and the topology of its pattern cluster. The external protocol adapts to the context of the parent;

• a set of statically and/or dynamically configurable parameters. Thus, each pattern is parameterized;

• other optional components, characteristic to the specific pattern.

Figure 1 illustrates the structure of a parallel computing pattern. It should be noted here that the internal protocol \( P_{\text{int}} \) can be further sub-divided as, \( P_{\text{int}} = P_{\text{Rep}} \cup P_{\text{Peer}} \). \( P_{\text{Rep}} \) is for interactions between the representative of \( P \) and the child patterns inside the pattern cluster. Using \( P_{\text{Peer}} \), each child pattern can interact with its peers.

A pattern can be classified into one of three categories: (1) a singleton pattern is one with an empty pattern cluster, i.e. it just contains the representative; (2) a basic pattern is a fundamental pattern in parallel computing, for instance: a static/dynamic replication pattern, a pipeline, a divide-and-conquer tree, a n-D data-parallel mesh, or a hypercube, etc; (3) a compositional pattern is one inside which other patterns can be composed. The internal protocol for this pattern allows the lowest level of network based irregular communication between processes, and will be called \( P_{\text{Net}} \). As an illustration, \( P_{\text{Net}} = \{ \text{Send(...), Receive(...), Broadcast(...),...} \} \).

Patterns can be composed (inside a compositional pattern) as well as can be used as stand-alone patterns. A pattern adapts to the context of its parent \( P \) (i.e. becomes a member of the pattern cluster of \( P \)) by choosing the right external protocol. In other words, its external protocol becomes the internal protocol of \( P \). For a stand-alone pattern, its external protocol is empty.

2.1 An implementation

The modular nature of each pattern makes it an excellent candidate for the object-oriented style of implementation. The present library-based implementation uses C++ on top of MPI. The idea of patterns inside a pattern makes the implementation hierarchical, i.e. there is a hierarchical tree associated with each pattern, where its representative is at the root and the patterns inside its pattern cluster form the sub-trees. A singleton is a leaf. The implementation constructs the hierarchical tree associated with a pattern.

The pattern for data-parallel mesh computation was tested on the following algorithms: (1) 2-D discrete convolution [3], and (2) parallel sorting using regular sampling (PSRS) [4]. Unlike the first algorithm, the second algorithm needs a considerable amount of peer-to-peer communication. Impressive speed-up was obtained on a coarse-grained workload (approximately 6.5 and 5.5 respectively) using a cluster of 10 SparcStations interconnected with a 10 MBit network.

3 The user’s viewpoint

The discussion in the previous section is for the academic interest. What the user sees and uses is a much simplified view of each pattern and its usages through examples, etc. A simple example will clarify most of the issues. The present C++-based user interface for using a stand-alone dynamic-replication pattern will look as follows:
class UsersReplicationPattern: public ReplicationPattern <Worker, P_Repln, P_Void>
{

  // Application specific procedures and variables go here.
  public:
    void Rep() { // This representative code is written by user.
      SetReplicationWidth(10); // The maximum number of replicated workers.
      // This is a configurable parameter for this pattern.
      while (thereIsSomething()) {
        readData(data);
        SendWork(data); // Send work to a free Worker.
        // If none is free, dynamically spawn one if needed.
        if (ReceiveResult(data)) // Receive result from a Worker.
          writeResult(data);
        else
dosomethingElse();
      }
    }
}

class Worker : public SingletonPattern <P_Repln>
{
  // External protocol = P_Repln. A singleton's internal protocol is empty.
  // Application specific procedures and variables go here.
  public:
    void Rep() { // This representative code is written by user.
      while (ThereIsSomething()) {
        ReceiveWork(data)) // Receive work from parent.
          processData(data);
        SendResult(data); // Send result to parent.
      }
    }
}

Pmain() // The user's main
{
  UsersReplicationPattern Root; // Root of the hierarchical tree.
  Root.Run(); // Implementation of Run() is hidden from user.
}

Here, ReplicationPattern and SingletonPattern are defined in the library. The first one is a basic pattern supporting dynamic-replication and the other is a singleton. What we see here is the user's interface for using the replication pattern. The application specific code is written by the user inside each Rep() (the representative). Other than the filling of the Rep()'s and the application specific procedures and variables, the above structure is completely application-independent and can be mechanically generated. The above user's interface without the application specific components is called the skeleton of the application. The back-bone of the skeleton is available in the system's library.
The dynamically- replicated Workers (which are singletons) constitute the pattern-cluster of UsersReplicationPattern. By default, the internal protocol of UsersReplicationPattern is \( P_{Repl} \), which is dynamic-replication protocol. Its adaptable external protocol is void (since it is stand-alone). The Worker adapts to the context of its parent by choosing the right external protocol, \( P_{Repl} \) (which is the internal protocol of its parent). Being a singleton with an empty pattern-cluster, a Worker has no internal protocol.

The commands in bold-face are the components of the protocol \( P_{Repl} \), i.e. \( P_{Repl} = \{ \text{SendWork}(\ldots), \text{ReceiveResultNb}(\ldots), \text{ReceiveResult}(\ldots), \text{ReceiveWorkNb}(\ldots), \ldots \} \). As a characteristic of this pattern, the peers (i.e workers) do not interact with one another and hence \( P_{Peer} = \phi \). All low level details like dynamic process creation, mapping, communication, synchronization, data-packaging and un-packaging, marshaling and un-marshal- ing, etc. are implemented inside the patterns and the protocols, and system defined wrapper classes, so that the user does not have to worry about these issues. In fact, it will be evident here that the user does not see any explicit parallelism in his code.

If needed, the Worker can later be refined to some other pattern (i.e. the implementation supports hierarchical refinement of patterns, conforming to the hierarchical model), while keeping the parent completely intact.

4 Contributions

The pattern-based approach to parallel programming is not new. In fact, similar ideas originated in the late 1980's with systems like CODE[2] and Frameworks[6]. A detailed discussion of the past approaches and their limitations can be found in [7]. Some recent systems include Enterprise[5] and Tracs[1]. As mentioned before, flexibility and extendibility were two of the major concerns with most of the earlier approaches. In most cases, the user was supplied with a few hard-coded patterns. If his application did not require those patterns, he was out of luck. Besides, due to the lack of a generic model and mutually agreeable interface, intermixing of patterns was confusing and often impossible. To the best of our knowledge, Tracs was the only extendible system. But, its patterns are not parameterizable and as a result they are not generic. Besides, being entirely graphical, its GUI imposes its own limitations.

What these approaches lacked the most is that they exclusively concentrated on the structure of a pattern. Being entirely structural, there is no way of distinguishing a static master-slave from a dynamic-replication pattern, or a 2-D data-parallel mesh from a systolic array. Evidently, besides the structural part, a pattern has its own behavior which distinguishes it from the rest. This behavior comes from the parallel computing model of a pattern and the associated protocol(s). This work associates the notion of behavior to a pattern. This, together with the generic hierarchical definition of a pattern, allows patterns to be included inside another pattern through mutually agreeable interfaces (by choosing the right protocol(s)). This elegantly allows hierarchical composition of patterns and their successive refinement(s) inside an application.

The compositional pattern together with its internal protocol, \( P_{Net} \), provides the maximum flexibility to the user. If nothing works out for him, he can still compose his application using singletons, which can interact using \( P_{Net} \). More on flexibility is raised in the next section. The library-based approach together with a generic view of a pattern supports extendibility (the user/system designer will be able to add new patterns to the library, conforming to the model). Extendibility, in turn, enhances flexibility.
5 Conclusion and Future directions

Some recent quantitative studies, from the software engineering point of view, at the University of Waterloo showed that patterns lower software complexity as compared to code written using MPI. A set of software metrics were generated for pattern- as well as non-pattern-based code for the same set of applications. More independent work is underway in this direction.

Presently the possibility of a simpler PERL-based user interface is being investigated, which will automatically generate the back-end C++-skeleton. One more issue under consideration is to provide a pattern supporting explicit dynamic spawning. This will provide the maximum possible comfort to the user in developing his application.

References


