The 2010 International Conference on Grid Computing and Applications (GCA'10), July 12-15, 2010, Las Vegas, Nevada, USA.

Skeleton/Pattern Programming with an Adder Operator for Grid and Cloud Platforms

J. Villalobos, B. Wilkinson
Department of Computer Science, University of North Carolina, Charlotte, NC, USA

Abstract— *Pattern operators are extensions to the Pattern/Skeleton parallel programming approach used to apply two types of communication patterns to the same data. The operators are intended to simplify the wide range of possible patterns and skeletons. The abstraction helps manage non-functional concerns on the Grid/Cloud environments. This paper explains how the pattern operators work on synchronous cyclic undirected graph patterns, and it shows examples on how they are used. A prototype was created to test the feasibility of the idea. The example used to show the operator approach is the addition of termination detection to a discrete solution to a PDE. The example can be coded with 27.31% less non-functional code than a similar implementation in MPJ, and its programmability index is 13.5% compared to MPJ's 9.85%. The overhead for an empty pattern with low communication was 15%. The use of pattern operators can reduce the number of skeletons/patterns developed.*

Keywords: Skeletons, patterns, grid, cloud, operators

1. Introduction

The advent of Grid computing during the past decade has created a heterogeneous computing environment where the users of computing resources have been exposed to multiple types of architectures, network topologies, security issues and other non-functional concerns. These challenges need to be addressed. We are using the term heterogeneous environment to refer to the Grid and to cloud computing. Some aspects of cloud computing require the use of abstractions such as skeletons/patterns (SPs), particularly the need to separate the resources of the cloud from the users and developers. The main problem at hand is how to program for this heterogeneous environment given all the possible variations that may need to be accounted for in order to run a single program in different configurations efficiently. The approach we favor is the use of SPs.

Skeletons are directed, acyclic graphs, and their implementations are data parallel. The algorithms create a flow of data that streams from one stage to the other until the necessary algorithms have been performed to it. Patterns are cyclic, undirected graphs. They are data-parallel and they require synchronization during the execution. The start and end of the patterns are the same as the skeleton, that is, they start with some mapping of the initial data to the nodes, and then end by converging the processed data into a sink node. Figure 1 shows the common life cycle for both skeletons and patterns. The most recurring skeletons are map, reduce, workpool/farm, and pipeline. The most recurring patterns are stencil and all-to-all. More complex skeletons like divide-and-conquer can be constructed from simpler skeletons through nesting [1] [2]. The patterns have a wider set of non-standard pattern types that cannot be made from a basic set of patterns as is the case with skeletons.

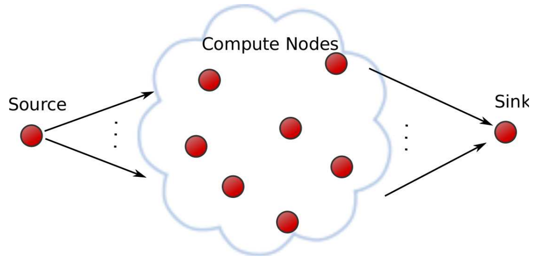

Fig. 1: Basic skeleton pattern organization.

There are three advantages to using SPs over the industry standards today (MPI [3], OpenMP [4], and explicit thread libraries). The first is that SPs hide deadlock and race conditions from the user. They provide implicit parallelization to the user programmer. This is done by giving the user programmer an interface. When the user programmer gives the implementation to a framework, the framework will run the interface while taking care of the race conditions and deadlock. The user is never aware of the problem. The second advantage is a reduction in code and development time. Macdonald et al. showed that coding with SPs requires less coding, and is simpler in comparison to MPI [5]. Aldinucci et al. also have shown frameworks such as Lithium and Muskel that use object-oriented Java to provide the benefits of skeletons to the user programmer [1]. Object-oriented languages have created the abstraction that is necessary for the concepts behind SPs to be provided in a form that is simpler to understand. Previously, projects such as eSkel [6], Sketo [7], and DPnDP [8] provided skeletons using procedural C/C++ (which is not typed) but they create an environment where mistakes are hard to find [8]. The third advantage for SPs came in with the increased need to abstract the parallelization away from the computational resources as is needed in cloud and Grid computing. The abstraction is needed mainly in Grid computing because the environment is made up of multiple architectures and network topologies. Ideally, service providers want an application to run on this environment while minimizing the amount of knowledge the user programmer needs to code it. In the case of cloud computing, the environment tends to be more homogeneous and controlled by a single entity, but the service provider also wants to provide the computation resources in a way that they can optimize the use of the hardware. The hardware optimization leads to servicing more customers. SPs are a pertinent option to abstract the use of the resources because they allow the user programmer to code the problem using the provided API, and they give the Grid/cloud maintainers space to manage the non-functional requirements. The API and interfaces, in effect, create an extra layer between the hardware and the user programmer. One could argue that skeletons have started to be introduced into the cloud; MapReduce can be cited as a successful example of skeleton use in the cloud environment [9]. More skeletons are needed in the future since MapReduce is not optimal for all types of parallel programming algorithms.

Despite the benefits, SPs have some persistent drawbacks. In his manifesto, Cole explains that SPs must "show the pay-back", and he stresses simplicity [6]. However, many still believe that the number of SPs needed to address parallel computing is infinite [10]. This has some implications; the user programmer may be faced with a library of SPs so large that he is discouraged from using the approach. On the other extreme, the user programmer, despite having multiple SPs to choose from, does not find the pattern that he needs for the problem, and therefore is tempted to just use lower level tools. One can intuitively surmise that there can be a basic set of SPs from which all the other SPs can be created. This could be possible by features such as nesting. In nesting, the user programmer is able deploy new SPs from inside the SPs, which allows for an exponential increase of SPs without increasing the size of the basic set. Nesting also allows for the use of libraries that contain SPs themselves. With some operators and a basic set, one could see a Turing-complete (so to speak) set of patterns from which all possible parallel programs can be created.

Section 2 presents a high level explanation about pattern operators. Subsection 2.1 presents an example using Java. It explains the use of interfaces to create computation modules and the interfaces used to create data containers. Subsection 2.2 presents some important details on implementing SPs into a framework we call the Seeds framework. Subsection 2.3 presents the implementation of the adder operator in Seeds using the tools explained in Section 2.2. Section 3 presents the results from measuring the adder operator on the dimensions of performance and programmability. Finally, Section 4 mentions the related work.

2. Pattern Operators

Pattern operators are elements that work on the same piece of data. We present here the addition operator for synchronous patterns. The creation of this operator comes about to address stateful algorithms such as discrete solutions to PDE's and practical solutions to particle dynamics algorithms. In these types of algorithms, there are different communication patterns at different stages of programming. In the example of a discrete simulation of heat distribution, multiple cells on a stencil pattern work in a loop parallel fashion, computing and synchronizing on each iteration. However, every x iterations, they must implement an all-to-all communication pattern to run an algorithm to detect termination. That is used to check if all cells have converged on a value and all the cells should at that point stop computing. Figure 2 shows the example of this approach.

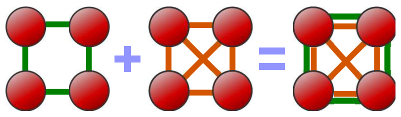

Fig. 2: Adding a Stencil plus an All-to-All synchronous pattern.

Similarly, the practical approach to solving particle dynamics combines multiple communication algorithms. Let us first review a simple version of particle dynamics. In this version, an all-to-all communication pattern is run on every iteration and the information for every particle is used to calculate the future momentum of each particle. This has O(N^2) complexity. Another algorithm has a lower complexity, but its implementation is to use a stencil where the particles will calculate its momentum based on n of its closes neighbors. This stencil pattern is performed for x iterations. Every x iterations, the algorithm switches into an all-to-all pattern of communication to update all particles and reduce the error that the algorithm inevitably accumulates. With just these two examples, it is easy to see that many more algorithms fall into this category where one has multiple layers of communication patterns that work on the same data. In the example of heat distribution, the data is a set of pixels that represent the heat energy present at that point. In the case of particle dynamics, the data represents momentum for each particle at that instant in time.

2.1 Example

Figures 3 and 4 show an example where an all-to-all termination detection algorithm is used to determine if there is convergence after performing a stencil algorithm for some number of iterations. A discrete approach to the problem of heat distribution was used to test the code shown in the figures. Figure 3 shows the code used to create the algorithm for heat distribution. Some of the problem-specific code was omitted in the interest of brevity. The class HeatDistribution extends a Stencil abstract class. This requires the user programmer to implement some signature methods. The Javadoc for each signature method is used to instruct the user programmer on the purpose of each method and their interaction within the framework. The DiffuseData() method is used to get the segments of data from the user programmer. GatherData() is used to get the processed segments of data back from the user. OneIterationCompute() is used as the main computation method. Because the algorithms are loop-parallel and the framework needs to gain back control in order to organize multiple patterns, the user is instructed the method should only run one iteration of the main loop in the application. initializeModule() is used to allow the user programmer to pass string arguments to the remote instantiation just after the modules get initialized.

```
public class HeatDistribution extends Stencil {
 private static final long serialVersionUID = 1L;
 int LoopCount ;
 public HeatDistribution(){
  LoopCount = 0;
 }
 @Override
 public StencilData DiffuseData(int segment) {
  int w = 10, h = 10;
  double[][] m = new double[10][10];
  /** init matrix m with file or user input*/
  HeatDistributionData heat = new
             HeatDistributionData(m, w, h);
  return heat;
 }
 @Override
 public void GatherData(int segment, StencilData dat) {
  HeatDistributionData heat = (HeatDistributionData) dat;
    /** print or store results*/
 }
 @Override
 public boolean OneIterationCompute(StencilData data) {
  HeatDistributionData heat = (HeatDistributionData) data;
  double[][] m = new double[heat.Width][heat.Height];
  /** compute core matrix */
  /** compute sides (borders)*/
  /** compute corners */
  /** set if this node is done*/
  heat.matrix = m;
  return false;
 }
 @Override
 public int getCellCount() {
  return 4; //four nodes for this example
 }
 @Override
 public void initializeModule(String[] args) {
  /*not used*/
 }
}
```

Fig. 3: HeatDistribution class extends Stencil and fills in the required interfaces.

Figure 4 shows the TerminationDetection class, which extends CompleteSyncGraph. Similar to the stencil pattern, CompleteSyncGraph also requires some signature methods. The pattern has a DiffuseData() and GatherData() method but they are not used for this example since the second pattern in the operator is used for its computation function only. getCellCount() is the number of processes needed for

```
public class TerminationDetection extends CompleteSyncGraph{
 @Override
 public AllToAllData DiffuseData(int segment) { // not used
  return null;
 }
 @Override
 public void GatherData(int segment, AllToAllData data) {
  // not used
 }
 @Override
 public boolean OneIterationCompute(AllToAllData data) {
  HeatDistributionData d = (HeatDistributionData) data;
  return d.Terminated;
 }
 @Override
 public int getCellCount() { // not used really
  return 4;
 }
 @Override
 public void initializeModule(String[] args) {/*not used*/}
}
```

Fig. 4: Termination detection using all-to-tall pattern.

the computation and must return the same number on both patterns so that communication patterns fit together.

Figure 5 shows the main data object used for both the patterns. The main advantages sought in using the pattern adder is to provide the user programmer with the ability to have two communication patterns work on the same data. Our approach to patterns has the requirement of having all information used for communication travel in the form of serializable objects. Additionally, the stencil pattern adds other signature methods that are needed in order to control the communication on behalf of the user programmer. CompleSyncGraph also adds signature methods. HeatDistributionData implements both StencilData and AllToAllData so that it can be handled by both patterns.

Both of these modules are inserted into the framework using a bootstrapping executable class. Figure 6 shows the executable the user programmer implements in order to add the stencil pattern plus the CompleteSyncGraph pattern. The two are added using an Operand class, which is used to hold together three characteristics each pattern needs, which are: the initialization arguments if any, the host anchors if any, and an instance of the pattern's module. Anchors are used to tie a special node the host that has to run it. The main use for the anchor is to specify where the source and sink nodes are to be run, since they usually have to be where the data is. The executable class also has some code to start the framework and shutdown the framework, which will self-deploy. After creating the operands, the pattern-adder operator is deployed by starting a new pattern called an AdderOperator. The framework, by default, will spawn and monitor the new pattern on a separate thread. The user programmer can just wait for the pattern to complete using waitOnPattern() method.

```
ublic class HeatDistributionData
        implements StencilData , AllToAllData {
 boolean Terminated;
 public double [][] matrix;
 public int Width,Height;
 SyncData[] Sides;
 public HeatDistributionData(double [][]m
                         , int width , int height){
 }
 /** Stencil data signature methods */
 @Override
 public Data getBottom() {}
 @Override
 public Data getLeft() {}
 @Override
 public Data getRight() {}
 @Override
 public Data getTop() {}
 @Override
 public void setBottom(Data data) {}
 @Override
 public void setLeft(Data data) {}
 @Override
 public void setRight(Data data) {}
 @Override
 public void setTop(Data data) {     }
 /** The All-to-All Data signature methods */
 @Override
 public Data getSyncData() {/**return data for all*/}
 @Override
 public void setSyncDataList(List<Data> dat) {
    /** get data from all */
 }
}
```

Fig. 5: The main data extends both the StencilData and AllToAllData. The object is used to hold the state-full data for the main processing loops.

```
public class RunHeatDistribution {
 public static void main(String[] args) {
  Deployer deploy;
  try {
   Seeds.start("/path/of/shuttle/folder/pgaf", false);
    /**first pattern */
    Operand f = new Operand(
        (String) null
        , new Anchor("Kronos", DataFlowRoll.SINK_SOURCE)
        , new HeatDistribution() );
    /**second pattern*/
    Operand s = new Operand(
        (String) null
        , new Anchor("Kronos", DataFlowRoll.SINK_SOURCE)
        , new TerminationDetection() );
    /**create the operator*/
    AdderOperator add = new AdderOperator(
         new ModuleAdder( 100, f, 1, s )   );
    /**start pattern and get tracking id*/
    PipeID p_id = Seeds.startPattern( add );
    /**wait for pattern to finish*/
    Seeds.waitOnPattern(p_id);
   Seeds.stop();
  } catch (Exception e) {
    /**catch exceptions*/
  }
 }
}
```

Fig. 6: RunHeatDistribution is used to create the operator and start the pattern.

2.2 Implementation Details

The SPs are divided into unstructured and structured. The unstructured patterns are the templates such as the map, reduce, and workpool/farm patterns. They are referred to as unstructured because the number of processes can change dynamically. A workpool/farm can work with 10 nodes the same as it works with 1000 nodes without the need to modify its implementation or having to add special code to the framework to handle the change. These are features that are inherent in the definition of these three skeletons. The other algorithms that require a specific number of processes are the pipeline skeleton, the synchronous loop-parallel patterns such as the stencil and its modifications, and the complete graph or all-to-all pattern.

Our Seeds framework implements the differences by using UnorderedTemplate for the unstructured skeletons, and OrderedTemplate for the structured SP's. Furthermore, each template is inherited to implement the specific pattern. PipeLineTemplate inherits OrderedTemplate to implement the skeletons. Figures 7 and 8 show the UML for the different classes that inherit OrderedTemplate and UnorderedTemplate respectively.

Map and reduce skeletons are not implemented. There is a template created for convenience called LoaderTemplate. LoaderTemplate inherits UnorderedTemplate and its goal is to load some initial data into the computation units for OrderedTemplate algorithms. Once the computation units are loaded, they can start working on one of the OrderedTemplate implementations. At the end of the OrderedTemplate SP, control is returned to LoaderTemplate, which sends back the data to the sink node. The data may have been modified during the computation as it would happen when implementing a stencil, or the data may not have any significance once the job is complete.

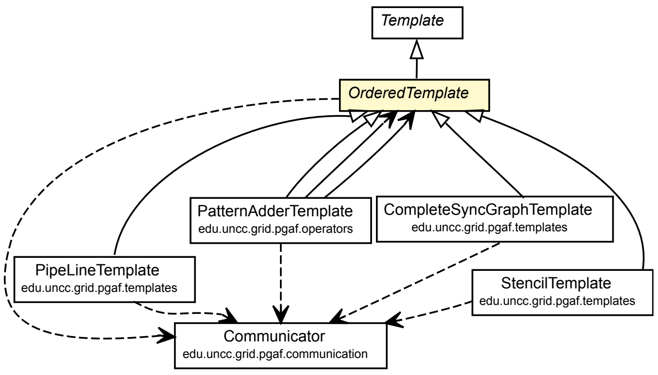
Template
OrderedTemplate
PatternAdderTemplate
edu.uncc.grid.pgaf.operators
CompleteSyncGraphTemplate
edu.uncc.grid.pgaf.templates
PipeLineTemplate
edu.uncc.grid.pgaf.templates
StencilTemplate
edu.uncc.grid.pgaf.templates
Communicator
edu.uncc.grid.pgaf.communication

Fig. 7: The OrderedTemplate needs a specific number of processes. The Stencil and CompleteSyncGraph inherit this class.

In order to provide structure to the user programmer, an interface is created using abstract classes that inherit the BasicLayerInterface abstract class. This is used to provide the user programmer with the signature functions that must be implemented in order to successfully interact with the framework. The interface is like a form presented to the user with instructions on how it should be filled in. Figure 9 shows the UML diagram for the BasicLayerInterface class and some interfaces that inherit it.

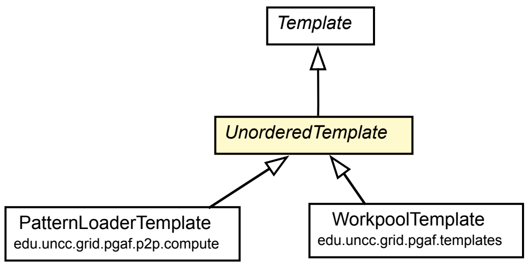

Fig. 8: Template class. OrderedTemplate and UnorderedTemplate extend it.

The basic layer: If the user wants to implement some skeleton or pattern, he needs to create at least three classes: the module that implements an SP interface, at least one class that implements a Data interface, and an executable class that connects the module to the framework, and deploys the framework if it is not running already. An example of this procedure can be seen in Section 2.1.

The advanced layer: This advanced layer user is interested in implementing a pattern that cannot be implemented with the existing skeletons/patterns, or he wants to test a new optimization method, or he wants to provide a pattern that is more convenient to use than the options already offered by the framework. In this case, the user needs to implement at least two classes, and one interface. The advanced user must decide if the SP needs a structured or unstructured implementation, a class that inherits one or the other must be created. If an unordered template is created, only one class is needed. If the SP is structured, then the advanced user must also implement a LoaderTemplate. The advance user may see the need to create Data interfaces to be able to steer the user programmer into the communication patterns that are required by the advanced user's SP; however, this is optional.

The expert layer: The expert layer is the machine room - it has many primitive data structures and behavioral patterns that look like an MPI implementation. The objects that the expert layer provide to the layer above tend to be complicated because the heterogeneity of the environment. It has long if statements that account for each type of network the nodes might be in, and each type of memory management system potentially available. The expert layer is only for Grid computing experts and parallel programming researchers.

2.3 The Operators

The operators at present are only implemented for the OrderedTemplate SPs because only the addition operator seems to be beneficial in reducing SPs complexity for the user programmer. Future endeavors may include adding operators to unstructured skeletons, to get similar benefits as we show can be had from the addition operator.

The operators are implemented by inheriting OrderedTemplate. Once this OrderedTemplate is loaded, OperatorTemplate runs the first SP to load the initial computational units. Then it enters into the main loop-parallel cycle. It run the first operand for n iterations, and then it runs the next SP for x iterations until either one of the SPs return true. The computation for these SPs return false if the program is not done computing. When the main loop-parallel cycle is done, the operator pattern returns the processed data units to the first operand's GatherData() method. The operator implements LoaderTemplate to load and unload the initial data from the first SP. The second SP only contributes the computation operation, and the Diffuse/Gather operations are ignored for this SP. Figure 9 shows a diagram that describes most of the interaction among the classes that happens when running the operator template. The small tabs inside the square are used to mention the BasicLayerInterface class that is used by the Template class. For example: Stencil class inherits BasiclayerInterface, and it is used by StecilTemplate class. Together, both classes implement a stencil pattern. The two patterns are added into the PatternAdder interface, which is then executed by the AdderTemplate. Because the adder template is an OrderedTemplate, it must specify a PatternLoader interface, which is the PatternAdderLoader class. PatterAdderLoader inherits PatternLoader interface, and it is executed by the PatternLoaderTemplate. Finally, Seeds can execute the LoaderTemplate directly because it is an UnorderedTemplate. All templates implement a function for the client side node and one function for the server side node. The server side corresponds to the source and sink nodes, and the client side corresponds to the compute nodes.

3. Results

Tests were performed to validate the pattern adder operator. The two main concerns for extensions to the SP programming approach are the performance impact created by the extension, and programmability of the extension. In order to measure the performance overhead created by the pattern adder operator, we implemented a simple algorithm that uses both the stencil pattern and the complete pattern. The algorithm is trivial; it consists of sending a long integer type to the neighbor processes in the stencil pattern, and it repeats the process for the complete pattern. We consider this an empty grain size pattern. The time to run through one iteration is measured for the stencil pattern and for the complete pattern. The time taken to run the process is also measured for the pattern adder operator. The overhead is the difference between the pattern operator's time and the stencil plus the complete pattern's time. Figure 10 shows the result of this test. The test was performed on a 16-core

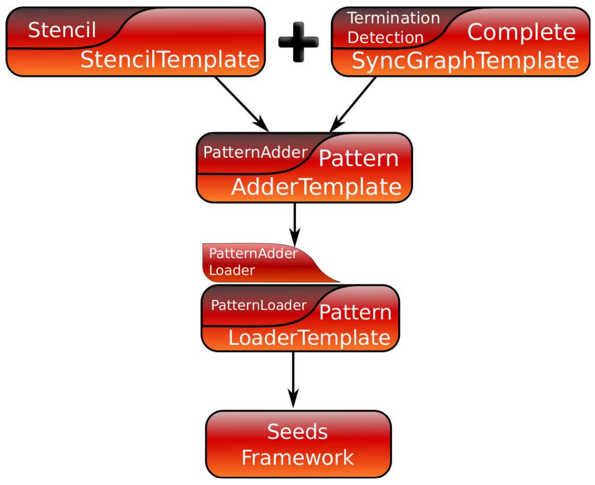

Fig. 9: Interface + Template pairs are drawn on the same square. The diagram shows the hierarchical interaction between the classes in order to execute a PatternAdder operator.

Xeon 2.93GHz server with 64GB memory. The number of processes is varied from 4 to 16. All the communications for this experiment were through shared memory. The results show that the overhead goes down as more processes are used for the computation. This is in part because the increasing communication overhead helps mask the overhead due to the operator. The overhead in comparison to an empty grain size is 15%, so grain size has to be adjusted to justify the use of the operator. The network speed also has an effect on the overhead. As Figure 10 shows, the increase in communication overhead reduces the overhead incurred due to the operator. The same test was done on a cluster of 3 dual-CPU Xeons (3.4GHz) with 8GB memory running four threads per server. The overhead for this test on an empty grain size pattern was 0.03% for nine processes. The network used was a Gigabit Ethernet.

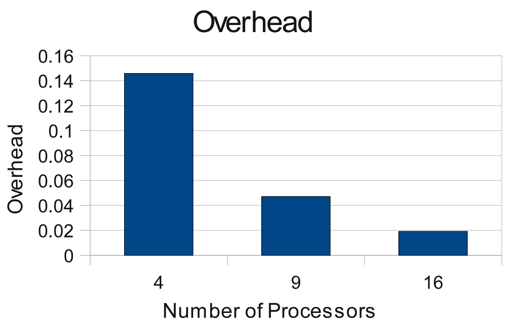

Fig. 10: Operator overhead measured on a shared-memory multi-core server.

Next, we measured the programmability. For this test we implemented the heat distribution algorithm using MPJ-Express. We also implemented the problem using the Seeds framework, and a serial version of the problem was used as control. The lines of code (LOC) were tagged on each implementation with the tags: functional, non-functional, automatic, comment, and log. The main tags are:

- Functional: The code that is dedicated to solving the problem. Most of the LOC in the serial implementation are considered functional.
- Non-functional: Is code primarily written to organize parallel processes and communications. MPJ-Express, and Seeds include code of this type to solve the problem in parallel.
- Automatic: This code is generated code by the IDE. Eclipse was used for the test. The generated code includes the class declaration, import lines, package declaration, and interface signature functions. Setters and getter can also be included as automatic code.

The programmability index is defined as:

$$\frac{\text{functional}}{\text{functional} + \text{non-functional}} \tag{1}$$

A higher ratio means a program with less non-functional code, and therefore we assume more readable and parallelized in less time. Figure 11 shows the result from this assessment. Seeds reduces the number of non-functional code by 27.31% over MPJ implementation. MPJ's programmability index for this implementation is 9.85%, and Seed's programmability index is 13.50%.

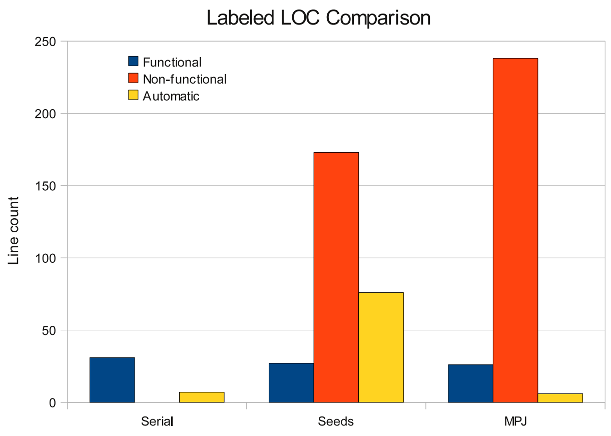

Fig. 11: the y axis shows the number of lines of code for each implementation. The LOC were counted for the serial implementation as well as for the Seeds and MPJ implementation.

4. Related Work

As referenced in Section 1. our work builds upon several frameworks designed to implement skeletons, patterns or both. Cole proposed a series of guidelines that must be met in order to create feasible skeleton interfaces [6]. McDonalds et al. created the three-layer development environment for the CO^{2}P^{3}S project. On that project, the three layers were created as a way to leave some flexibility in the framework; it allows the programmer to create new patterns if needed. We have incorporated that idea into Seeds. Similarly, Aldinucci et al. has provided three layer development environments to explored the use of skeletons as a way to manage non-functional requirements on behalf of the user programmer [11]. Future research will give Seeds non-functional requirement features such as scalability and load-balancing. The same middle layer (the advanced layer) will be used for that purpose. Seeds assumes all the other non-functional requirements, such as security, are done by the Grid middleware. The nested feature is important for any SP framework. Lithium [1], Muskel [2] and other frameworks have implemented the feature [6]. Our work comes closest to the work of Gomez et al. Their pattern operators implement the same concept that set us in the direction to create the pattern operators [12]. Their work is based on workflows and includes other types of operators that are used to manage non-functional concerns. The user programmer, in the Triana framework is given more control over the resources where the program runs. Our work differs from Triana in that we provide the pattern operators as a pure object-oriented framework without the need for XML language constructs, scripts, or a GUI. Also, we have measured the effects of the tool. The use we intend for pattern operators is targeted toward high performance parallel computing in a heterogeneous environment. A similar idea by Gomez et al. that can be useful to port from the problem solving environment (PSE) into SP frameworks is to use pattern operators to add a behavioral pattern (such as check pointing, visualization, and interaction) to a SP parallel application.

5. Conclusion and Future Work

Much of the literature focuses on skeletons, and somewhat on patterns. Our work shows how the number of synchronous patterns can be reduced by using the addition operator. However, it does not show how the synchronous patterns can be scaled automatically for the number of resources. Specifically, the interfaces shown require the user to specify how many processes need to be used to work on the program. A better approach would allow for an automatic scalability of these patterns. Aldinucci has mentioned these goals in his literature. We also believe this goal would better promote patterns for the grid and cloud environments.

We presented an object oriented implementation to provide an adder operator to skeletons/pattern parallel applications. A sample program was shown and its creation was discussed from the user programmer's perspective. Subsequently, the implementation of the pattern adders was presented. The advance user's perspective was discussed, and some notes about the Seeds framework and the expert programmer's perspective was also discussed. The pattern operator can be used to reduce the number of patterns that are needed by the user programmer. We believe that providing a basic set of skeletons/patterns plus useful operators will increase the popularity of this parallel programming model.

6. Acknowledgment

We thank the UMLGraph.org team for providing the tools create class diagrams from source code [13]. The Seeds framework uses packages for SSH access using the Ganymed libraries [14]. Globus [15] access is done through Java Cog [16]. Network overlay organization is done using JXTATM [17], and use of the UPNP communication protocol for routers is also implemented [18]. Thanks to all these software providers.

References

[1] M. Aldinucci, M. Danelutto, and P. Teti, "An advanced environment supporting structured parallel programming in Java," Future Generation Computer Systems, vol. 19, 2003, pp. 611-626.

[2] Marco Aldinucci, Marco Danelutto, and Patrizio Dazzi, "Muskel: an Expandable Skeleton Environment," Scalable Computing: Practice and Experience, 2008, pp. 325-341.

[3] R. Hempel, "The MPI Standard for Message Passing," Proceedings of the nternational Conference and Exhibition on High-Performance Computing and Networking Volume II: Networking and Tools, Springer-Verlag, 1994, pp. 247-252.

[4] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," Computational Science & Engineering, IEEE, vol. 5, 1998, pp. 46-55.

[5] S. MacDonald, K. Tan, J. Schaeffer, and D. Szafron, "Deferring design pattern decisions and automating structural pattern changes using a design-pattern-based programming system," ACM Trans. Program. Lang. Syst., vol. 31, 2009, pp. 1-49.

[6] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," Parallel Computing, vol. 30, 2004, pp. 389-406.

[7] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu, "A library of constructive skeletons for sequential style of parallel programming," Proceedings of the 1st international conference on Scalable information systems - InfoScale '06, Hong Kong: 2006, pp. 13-es.

[8] S. Siu, M.D. Simone, D. Goswami, and A. Singh, Design patterns for parallel programming, 1996.

[9] J. Dean and S. Ghemawat, "MapReduce," Communications of the ACM, vol. 51, 2008, p. 107.

[10] T. Mattson, Patterns for parallel programming, Boston [u.a.]: Addison-Wesley, 2007.

[11] M. Aldinucci, M. Danelutto, and P. Kilpatrick, "Autonomic management of non-functional concerns in distributed & parallel application programming," Parallel and Distributed Processing Symposium, International, Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 1-12.

[12] M.C. Gomes, O.F. Rana, and J.C. Cunha, "Pattern operators for grid environments," Sci. Program., vol. 11, 2003, pp. 237-261.

[13] "UMLGraph - Declarative Drawing of UML Diagrams," http://www.umlgraph.org/index.html, Mar. 2010.

[14] "Ganymed SSH-2 for Java," http://www.ganymed.ethz.ch/ssh2/, Mar. 2010.

[15] I. Foster, "Globus Toolkit Version 4: Software for Service-Oriented Systems," Network and Parallel Computing, 2005, pp. 2-13.

[16] G. von Laszewski, I. Foster, and J. Gawor, "CoG kits," Proceedings of the ACM 2000 conference on Java Grande - JAVA '00, San Francisco, California, United States: 2000, pp. 97-106.

[17] "jxta: JXTA^{TM} Community Projects," https://jxta.dev.java.net/, Mar. 2010.

[18] "UPNPLib," http://www.sbbi.net/site/upnp/index.html, Mar. 2010.