High-Performance Grid and Cloud Computing Workshop, *25th IEEE International Parallel & Distributed Processing Symposium*, Anchorage (Alaska) USA, May 16-20, 2011.

Using Hierarchical Dependency Data Flows to Enable Dynamic Scalability on Parallel Patterns

Jeremy Villalobos
Computer Science Department
University of North Carolina
Charlotte, North Carolina 28223
Email: jeremyvillalobos@gmail.com

Barry Wilkinson
Computer Science Department
University of North Carolina
Charlotte, North Carolina 28223
Email: abw@uncc.edu

***Abstract*—Hierarchical dependencies are presented as an extension to data flow programming that allows parallel programs dynamically scale on a heterogeneous environment. The concept can help Grid parallel programs to cope with changes in processors, or Cloud and multi-core frameworks to manage energy use. A data stream with dependencies can be split, which in turn allows for a greater use of processors. The concept shows a 6% overhead when running with split dependencies on shared memory. The overhead on a cluster environment is masked by the network delay. Hierarchical dependencies show a 18.23% increase in non-functional code when the feature was added to a 5-point stencil implementation.**

***Index Terms*—Skeletons, patterns, grid, cloud, multi-core.**

I. Introduction

The increased use of heterogeneous environments and multi-core processors will require all future programmers to incorporate parallel programming into their implementations. However, parallel programs have proven to add a degree of challenge because of the amount of complexity added to even simple algorithms. Previous work shows parallel pattern programming can provide useful parallelization to programmers who are not parallel programming experts[1]. With the initial requirement of maintaining a parallel pattern programming interface for the programmer, we move onto other problems associated with heterogeneous environments such as the Grid, the Cloud, and multi-core environments.

The problem we are tackling involves the ability of a parallel program to automatically adapt to a dynamic system. The Grid, by definition, is a dynamic environment where computational resources can change during the parallel program's run time. For the cloud, the argument is made that automatically scalable programs can be schedule more easily. For the Cloud and multi-core environments, auto-scalability can allow the system to reduce computation units to the most optimum in terms of performance per unit of energy used.

Given the work load at any time, automatically scalable programs can be coalesced to use less processors and shut down idle cores to save energy. The challenge is not so much in providing scalability to a parallel program because that can be added with custom code to a desired application. The issue instead is how can we make parallel applications automatically scale while at the same time preserve the accessible programmability provided by parallel pattern programming.

We tackled the problem first by providing a data flow technique to create parallel programming patterns and second, by enabling the data flow technique to have hierarchical dependencies. The hierarchical dependencies interact with the programmer to allow for a stream of data to be split among processes in the event of an automatic scalability action. The data flow with a hierarchical dependency technique, in turn, is used to provide a programmer with a parallel environment that is still as easy to use as pattern parallel programs, but also has automatic scalability provided with no extra effort, or at least delays the extra effort to the optimization phase. The data flow implementation was added to the Seeds framework, a distributed computing framework geared toward Grid computing [2].

Seeds is a self-deploying framework that uses peer-to-peer JXTA[3] standards to organize a network of computational processes. Seeds follows a development style composed of three layers: expert, advanced, and basic[1]. The expert layer handles the processes and micromanages communications over different media. The expert layer provides abstractions for the advanced layer. The advanced layer is used to create parallel patterns. The basic layer is for the domain-specific programmer. The basic user is interested in choosing a pattern from a menu that will best map a problem onto a parallel platform.

Section II presents the data flow concept and how it is implemented on the Seeds framework. Section III presents the hierarchical dependency concept, that allows the dependencies to split, thereby splitting the synchronization data packets and allowing the parallel program to scale. Section IV discusses how data flows with hierarchical dependencies are used in conjunction with pattern parallel programming to provide automatic scalability to the user programmer. A pipeline skeleton and a 5-point stencil were implemented to test the idea in terms of programmability and performance overhead.

II. Data flow model

A data flow is a directed acyclic graph where the vertices represent processes, and the edges represent communication lines between the processes. Each of the processes therefore can be represented with a set of inputs, a set of outputs, a state, and a computation that is done on its inputs to create

the outputs. This section presents the implementation of such a parallel programming structure onto the Seeds framework and how the data flow implementation is attached to the existing framework's lower layers. The implementation can be divided into: data flow unit implementation, interfaces used by the advanced user, and interfaces used by the basic user. In Seeds, the data flows are Java objects that can be sent over the network to the remote processes. This is done to allow the framework to move the processes around the network without technical difficulty, and also to do this without synchronizing with the programmer through the use of extra interfaces. Each of the data flow objects has a module created by a basic user and a state object. The module contains an executable method that will be called for n iterations until the job is done. The state object stores the dependencies needed for computation. A design decision was made to separate the compute module from the data flow object to allow a data flow to handle multiple modules. This allows the data flow implementation to support pattern operators[4]. To put it all together in one computation cycle, the data flow object collects input packets from its connections. It then calls the compute method from the user's module, which will output a packet. The user's module may or may not store persistent data into the stateful object. At the end of the loop, the data flow object sends the output packets through its respective output connections. The term perceptron will be used to refer to an individual data flow node within the data flow network. The term is borrowed from neural network nomenclature since a data flow resembles that data structure. The advanced user can implement a pattern using the data flow technique by using interfaces. At the advanced user level, the programmer starts up with implementing a data flow template. The implementation of a pattern from this perspective consist of developing a pattern loader, and a data flow. The pattern loader has two signature methods: onLoadPerceptron is called to get a data flow perceptron from the advanced user. And onUnloadPerceptron is called to return the data flow to the advanced user. Once the pattern is implemented using data flows is deployed, the master node uses the signature methods to interact with the advanced user to build the initial data flow network.

III. Hierarchical Dependencies

This section presents the extension to the dependency concept that enables the data flow based framework to provide automatic scalability and grain size. The main extension to the dependency concept concerns splitting a dependency, and does so while providing the benefits of automatic scalability to the advanced user through an interface that is easy to understand. To some extent, the feature can be provided to the basic user either with no extra effort, or through implementation of two to four extra signature methods. A dependency is a stream of data that will flow from one perceptron to the other. Once connected, the dependency can be used for the rest of the computational time. The dependency must be identified uniquely, and its identification should be independent of the hardware where the process is running. this is done to allow the framework to move the process and the dependency for auto-scalability and grain-size.

A. The Hierarchical Dependency Syntax

Each of the dependencies must be assigned a unique ID. However, simply using an integer is not feasible because more dependencies can be created by individual processes, and the hereditary history of a dependency must be maintained in order to allocated a dependency to its corresponding output process. For example, suppose a dependency A is split into B and C. Now, there should be a process by which a perceptron connecting to dependency A would know that by connecting to dependencies B and C, it can produce dependency A. The process should also indicate how many child dependencies are needed to put the parent dependency back together. In order to accomplish this, the dependency ID is an integer followed by a slash (/) and the number of dependencies that make up the whole of the dependency. For example, if we have a dependency 2 that is part of a family of 4 dependencies, its ID would be 2/4. In the implementation, the root dependency is always part of a family of one dependency (n/1), and its children then are attached to the root dependency with a dot. For example, we can create a dependency 2/4 that is child of 1/1. The id is then expressed as 1/1.2/4. This means that, in order to receive dependency 1/1, a perceptron must connect to 1/1.0/4, 1/1.1/4, 1/1.2/4, and 1/1.3/4. Figure 1 shows the basic single level configuration for a binary dependency. In this simple configuration, one can observe three possible arrangements. First we can use the dependencies to establish a point-to-point connection with the dependency source symbolized by a wave symbol, and the dependency consumer or sink symbolized by a ground symbol. The second arrangement can have a dependency splitting itself so that the original output can be split and sent to two remote nodes. The split action is done using an interface implemented by the advanced or basic programmer. The last arrangement is a dependency consumer that requires dependency 1/1, but can only find the partial dependencies in the network (1/1.1/2 and 1/1.2/2). The consumer can connect to both of these dependencies and coalesce the input to create the needed data. The syntax allows the dependencies to split multiple times.

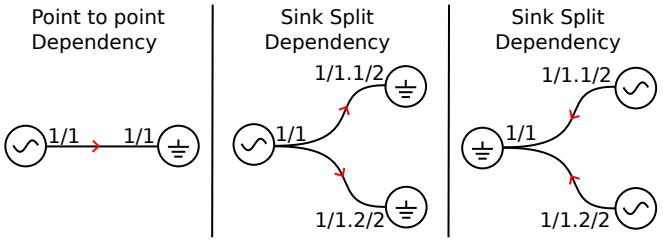

Fig. 1: basic point-to-point, source-split and sink-split dependencies.

Another feature provided by this extension is that only the split perceptron needs to be of concern. The other perceptrons

would see little change at the advanced level and basic level. For example, the perceptrons that were not split will still need the dependencies from the split perceptron; however, the hierarchical dependency will add the output from C.1 and C.2 to make the output for dependency C. This example assumes the dependencies are duplex to keep the example simple.

B. The Hierarchical Segment ID

The local data, or stateful data is also split. To keep track of this, the perceptrons are also assigned an ID that complies with the same syntactical rules as the once explained above. A complete dependency ID is therefore made up of the hierarchical segment ID plus the dependency ID. The two are joined by a colon (:). For example: 2/1:0/1. The hierarchical segments are the root ID in the process arrangement in the minimum CPU configuration. The number of CPU's in the minimum CPU configuration is specified by the advanced or basic programmer.

C. Implementation Details

In this section, we present how to handle split actions when sending and receiving packets, and how to organize a temporary stop of computation, which we call a hibernation action, in order to split the perceptrons into finer grain size perceptrons. The dependency implementation has a tree-like structure that is traversed using recursive methods. A Dependency object therefore has an array of Dependencies called Children. Most of the methods implemented in Dependency can recurse over the dependency tree to return if the dependency has data left, if it is connected, etc. The send and receive methods are where most of the conceptual and technical details concentrate.

D. The Receive Method

The receive method returns the next packet that has been received from the remote process and has to integrate streams from multiple sources. Figure 2 shows a decision diagram on how a packet is handled by the hierarchical dependency on the top. The bottom of Figure 2 shows a packet traveling up the dependency tree before being handed off to the local process. The dash lines connect the decision from the decision tree that correspond to the path traveled by the packet in the diagram on the bottom.

Following the decision paths from Figure 2, if the stream is null, we assume that it is because the connection is made up of multiple streams, and each of the children contains a stream. Then, we expect that each of the children will return an object, each with the same version, that can then be submitted to an advanced or basic user method that will return the coalesced packet for this dependency. Each Dependency object has a level_id variable. This is a HierarchicalDependencyID object that is used to label the level in the tree. Likewise, the source_id is also a HierarchicalDependencyID object. A comparison is made between the objects. To say x is higher than y is to say that x is a child of y. If x is lower than y is to say x is a parent of y. In the diagram, the comparison is used to determine if the children connections should be used

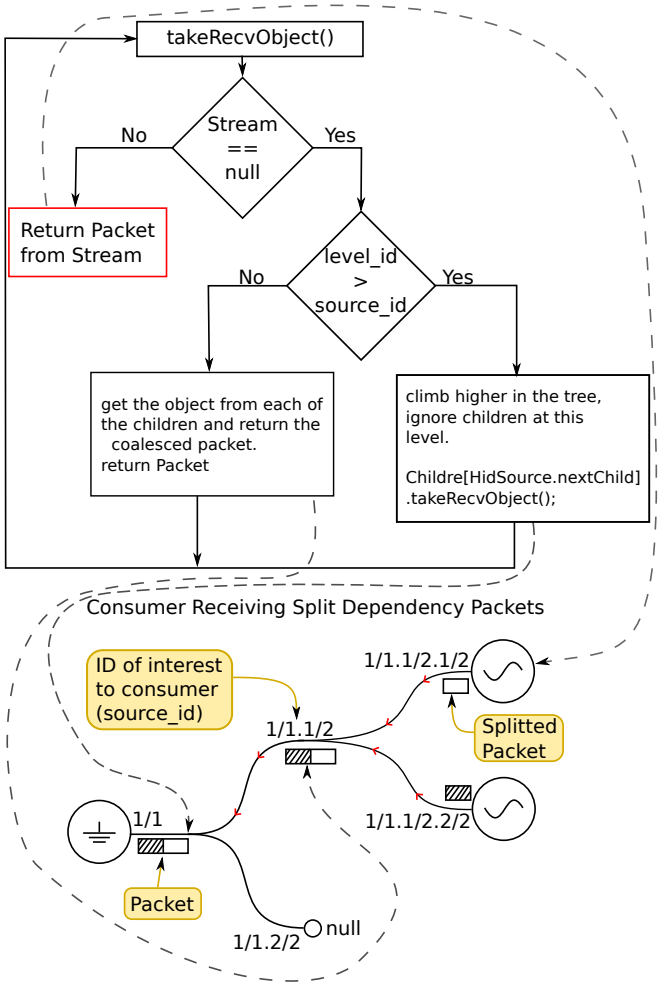

Fig. 2: A decision tree to handle packet on the hierarchical dependency receive method (top), and diagram showing how a packet travels up the dependency tree for a dependency with binary split sub-dependencies (bottom).

to get the next object. Since each of the Dependencies start at a root, in many cases, the dependency that is requested does not fall all the way down to the root, and therefore many of the children connections are not needed. The connection would be null, and an error would happen if level is not accounted for.

E. The Send Method

The send method sends a packet to the remote node. Figure 3 shows a couple of diagrams. On the top, is a decision tree used to handle packets send through the dependency. The bottom of Figure 3 shows the path taken by a packet on its way to being sent through a binary split dependency. Like the receive method, this method assumes the Dependency is a root if the Children array is null, otherwise we look at the level_id of each dependency on the way up the tree to judge when we should start splitting the packet. At some level, the packets starts to be split; however, the remaining part of the packet can keep climbing the tree until it reaches a leaf Dependency that will send the packet over the network.

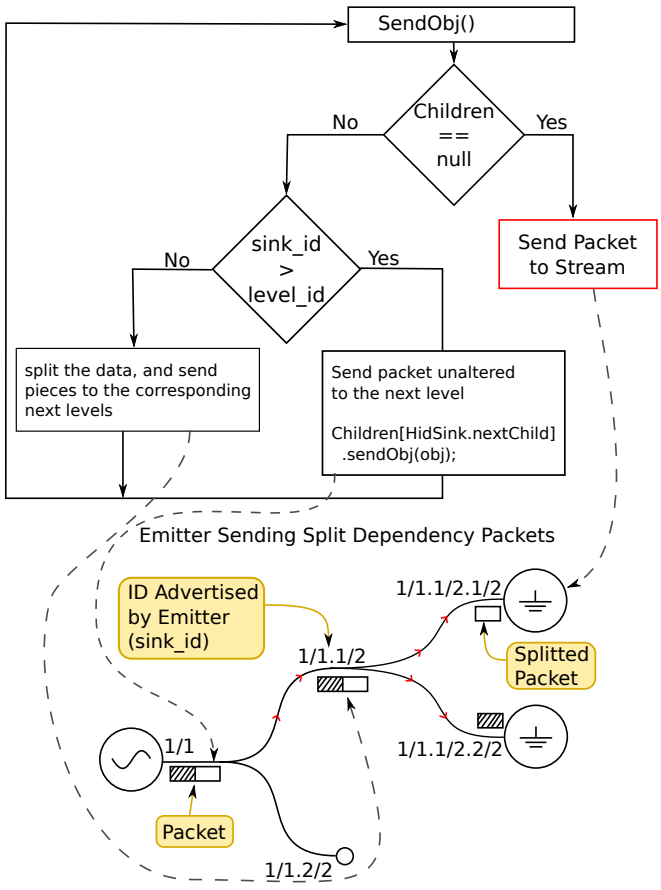

Fig. 3: A decision tree to handle packet on the hierarchical dependency send method (top), and diagram showing how a packet travels down the dependency tree for a dependency with binary split sub-dependencies (bottom).

F. Hibernating Dependencies

When a data flow is selected to return to the master node to be split, we call that state a hibernation state. To accomplish this transition, the framework must be able to freeze the computation node without having knowledge of what the computation node is doing inside. We added three features to enable the framework to hibernate the dependencies. We use a version number on the messages that are transferred among the data flows, and we add an algorithm that will negotiate shutdown at some version in the future. First, we require each of the messages between the computation data flow to be stamped with a version number. This allows each data flow to verify that the message is the next expected message and also allows the data flows to synchronize. This adds about 32 bits to the data flow messages in the Seeds implementation. Alternatively, the implementation can only use the version number on each of the data flows without sending an asserting integer to the remote nodes, this is future work at this point. The second feature to enable the hibernation state is a protocol to negotiate the hibernation at some version in the future. The protocol will compute a version in the future at which point the local data flow and the connection of its neighbors that are involved with the local data flow would disconnect. The framework arranges to hibernate a perceptron at an iteration in a future where the parallel application can finished computing. This will not cause an error because if the computation has ended before reaching a hibernation point, there is no need for the rearrangement of perceptrons. However, if the program is still running at that future iteration, the program can benefit from the increase use of resources. The gray area is a point where the perceptron will hibernate too close to the end of the computation to bring any speedup. The protocol then insures that a hierarchical dependency is hibernated by echoing the message. Figure 4 shows how the protocol behaves on the dependency tree. The message must be echoed in order to spread a hibernation call throughout the dependency tree. If a source emits the call, the sinks will adopt the version stop set by the source, and will echo the call with the number. The echo stops once it comes back to the original emitter. The example where a consumer starts the hibernation (on the right) better shows the need for an echo. In this example, the neighbor sink (on the bottom) will only know of the impending hibernation when the source echoes the hibernation call.

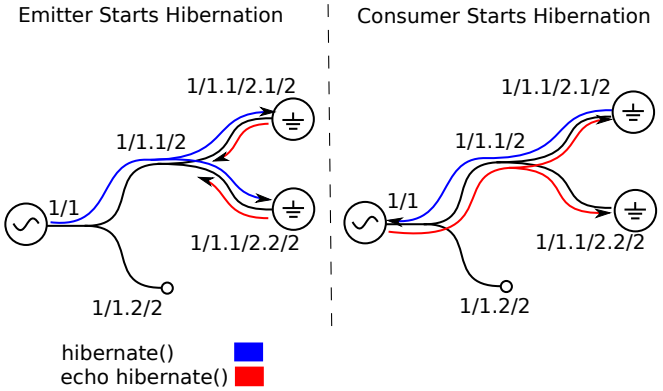

Fig. 4: a hibernation call traversing a hierarchical dependency tree.

The echo allows the hibernation message to go to all the leafs of a dependency tree. The emitter of the hibernation call waits for the message to make a round-trip. However, given the strict timing demanded by the negotiation algorithm, care must be taken to ensure the processes do not fall out of sync and deadlock. To prevent deadlock, the algorithm in Figure 5 shows how the version stop is negotiated between the local process and the neighbor remote processes. First the main loop enters this negotiation once the advanced layer template or the framework sets the data flow status as hibernation eminent. This starts the negotiation process. The algorithm checks to see if ”wait n version” variable (wnv) is set (1). If so, this means that a neighbor process is close to entering hibernation. Therefore, it would be unsafe for the local process to continue with the hibernation procedure. If wnv is not set, the (2) version_stop is checked to validate if it was set in a previous iteration. If version_stop has not been set, the

procedure will (3) calculate a version_stop based on the speed at which the local process has been working through the previous iteration. The number is then (4) compared with the version_stop of the neighbor processes. If the neighbors is not set, the local process will (5) send the hibernate calls through its communication lines. If the neighbor's version_stop is set, the local process has to (6) judge if the version_stop of its neighbors is far away into the future to join the group and hibernate at that same time. lowerend is calculated for that purpose. If the neighbor's version_stop is larger than the local process lowerend version stop estimate, the local process will (8) join the neighbor's hibernation version. If not, that means there is not enough time (measured in cycle units) to organize a hibernation for the local process. The local process will then (7) set wnv to a save version in the future and continue computing until that time. If version_stop is set, the process will continue to hit the decision tree on the next iterations. This time it will go through the branch where version_stop is true (2). At this point, the local process will still (9) check on the neighbor's version_stop. Although unlikely, the check is useful in case the neighbor's and the local process initiate a hibernation call at the same time, but were not aware of each other due to delays in the network. If this happens, the processes should (10) reset the hibernation call to the highest version among the group of hibernating processes. Notice that this will not lead to an infinite loop where all process continuously update the hibernation version further and further into the future. This will not happen because any new process that enters the hibernate procedure will, at some point, be aware of its neighbor's hibernation state before issuing the hibernation call. Therefore, it will instead enter into the branch that sets wnv variable. The last point in the decision tree is when the current_version reaches the stop_version (11). At that point, the main loop is broken (12), and the data flow can be disconnected and returned to the master node.

The version_stop is calculated by measuring how fast the local process is going through the iterations for the main computation loop. The aim of the algorithm is to set the hibernation action far ahead in the future so that there is enough time for the network to synchronize around the action. Let's first set some variables:

- t = time taken to compute i iterations.
- i = number of iterations done.
- ss = time slot size in millisecond units. The time slot is a constant based on the highest latency expected from the network.
- vts = version per time slot. The number of versions done in one time slot.
- cqs = communication queue size. The undelivered or unsent packets is assumed to be the highest by using the maximum number allowed to be cached in the send and receive queues.
- s = time slots.
- STEP_SIZE = The version_stop will be the ceiling of the next multiple of STEP_SIZE. This improves the change that unsynchronized nodes will hibernation on the same iteration.

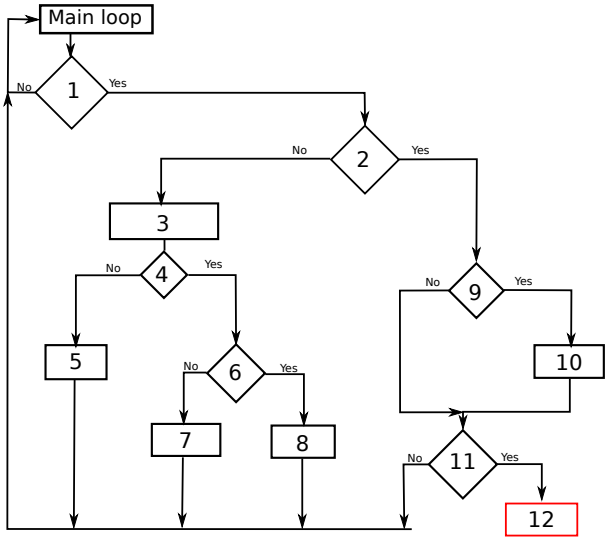

Fig. 5: Decision tree to negotiate a hibernation state for the local process.(1) wnv ¡= zero. (2) is version_stop set ? (3) compute local version_stop get neighbor's version_stop. (4) Does neighbor version_stop exists? (5) set version_stop and send hibernate signal. (6) is neighbor's version_stop more than local lowend. (7) set wnv to next version stop step. (8) Join neighbor's hibernation action. (9) get neighbor's version_stop. (10) version_stop ¡ neighbor's version_stop. (11) Reset communication lines to neighbors' version_stop. (12) current_version = version_stop. (13) break.

Now we can calculate the following variables:

$$s = \frac{t}{ss} \tag{1}$$

$$vts = \frac{i}{s} + cqs \tag{2}$$

Now, vts can be used to calculate version_stop and lowend.

$$version\_stop = current\_version + vts * a \tag{3}$$

The constant a is set to provide tolerance for the processes to synchronized. version_stop sets the iteration at which point the local perceptron hibernates.

$$lowerend = current\_version + vts * b \tag{4}$$

The constant b is set to provide a window where the local process can join a neighbor's hibernation procedure.

$$wnv = current\_version + STEP\_SIZE + vts * c \tag{5}$$

The constant c is set to provide enough time for the neighbor process dependencies to hibernate and split. Once the new perceptrons are on-line and the local perceptron's dependencies connect, a new hibernation call can be attempted. The end result for this procedure is that the perceptron can

be taken off-line safely. The perceptron can then be split into more processing units or it can be coalesced with other perceptrons to adapt to less computation processes. The remote processes that only had their communication lines hibernated but where not hibernated themselves will try to reconnect after the hibernation call has been executed. Once the new data flow are on line, they will emit dependency advertisements that will be caught by neighbor data flows in the network that were already looking for the connection. Once the connections are reestablished, the data flow can resume computing.

IV. Results

The hierarchical dependencies were tested with respect to two measurements. The hierarchical dependencies are intended to provide automatic scalability while adding small or no extra effort to the basic programmer. The effort on the part of the basic programmer is assessed by counting the number of extra lines of code needed in order for the basic user to be able to use automatically scalable patterns. The other question is on performance. How much performance overhead is created by this concept? The questions are addressed using a systolic matrix multiplication algorithm implemented on a pipeline to gauge performance overhead, and a Jacobi solution to the heat distribution problem using a 5-point stencil to measure performance overhead and to measure programmability . The stencil provides a poorer performance measurement method because it introduces unneeded complexity, but it provides a better example to accurately measure programmability on typical parallel programs. The systems used for the results were a dual-core cluster and a 16-core server. The performance test for shared memory was done on a 16-core (Xeon 2.93 GHz) system with 64GB of memory. The OS was Red Hat Linux, Kernel version 2.6.18-164. The Java virtual machine was Java HotSpot(TM) 64-bit Server 10.0-b22 mixed mode. The cluster is made up of four dual-core (4 threads) Xeons (3.4GHz), with 8GB of memory each. They have same version of Java VM and the Linux kernel used was 2.6.9-42 and 2.6.18-92 from Red Had.

A. Performance Overhead

The matrix multiplication algorithm was chosen as an algorithm to showcase the use of automatic scalability because of the simplicity of its implementation, and because the algorithm can be implemented using a pipeline. The matrix multiplication was implemented as mention by Wilkinson et al. [5]. In matrix multiplication we have A x B = C. Equation (6) shows the computation done to row one of matrix A and column one of matrix B to get the value for C1,1

$$a_{1,1}b_{1,1} + a_{1,1}b2,1 + a_{1,3}b_{3,1} + a_{1,4}b_{4,1} + a_{1,5}b_{5,1} = c_{1,1} \quad (6)$$

In the pipeline implementation, A is divided into its columns, and each column is assigned to a stage. We differentiate stages from processes here because there can be more than one stage per processes in our implementation. The matrix B is divided by its rows. The rows from B are then piped through each of the pipeline processes, and each process pipes the row through each of the stages. At the end of the computation, the returned columns correspond to the columns of matrix C.

The matrix multiplication algorithm is implemented in three forms:

Serial: The serial implementation is a pipeline with only one process working on all the stages. The source and sink for matrix B's rows are hosted on a different core or host computer. This version is held as control. Figure 6 shows the process arrangement.

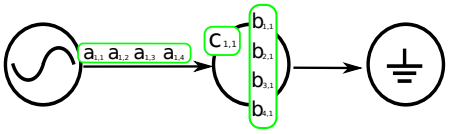

Fig. 6: Serial equivalent of a pipeline.

Sideways Split: The sideways split implementation is used to test the hierarchical dependencies use for this implementation. The perceptron is split in two, which requires the stateful object to be split, and the data packets to be split as well. Figure 7 shows the processes arrangement after the split. The user programmer then is required to implement four signature methods. The signature methods are: SplitStateFul, and CoalesceStateful to split and coalesce the statefull data; and SplitData, and CoalesceData to split and coalesce individual data packets send and received during synchronization. The module was coded using 82 lines before adding the code that enable automatic scalability. The lines of code to enable the automatic scalability were 68. This is an increase of 82.93

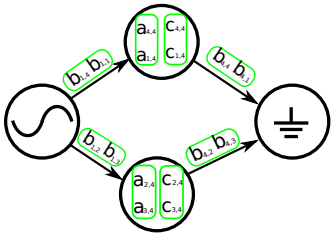

Fig. 7: Sideways split using hierarchical dependencies.

Stage Split: The staged split is a more obvious form to automatically scale this pipeline implementation, since the sideways implementation was done to test the performance and overhead that corresponds to the hierarchical dependencies. In this implementation, the perceptron with ten stages, would be split longways into two perceptrons where each has five stages. This automatic scalability implementation does not add lines of code on the side of the basic programmer. Figure 8 shows the process arrangement for this auto-scalability implementation.

Because both the Sideways split and the Staged split also incurred overhead due to the process of splitting and coalescing the perceptron as well as sending the perceptron back to the master node and getting the new child perceptrons back, the experiment measured only a portion of the total computation.

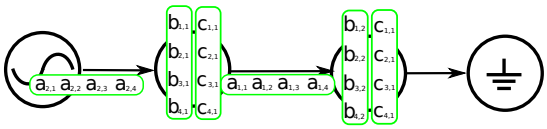

Fig. 8: Staged split using dependencies without hierarchical splitting.

TABLE I: Speedup using Dynamic Scalability.

	Serial(No Split)	Sideways	Staged
Shared Memory	1	1.65	1.77
Distributed Memory	1	1.46	1.43

The experiment consisted of multiplying two matrices of 2000x2000 doubles. The measurement is done in the middle of the computation to eliminate the influence of the splitting action that happens on iteration 900, and the coalescing action that happens in iteration 2000. the measurement is therefore, the time taken to compute from iteration 1000 to 1500. The measurement was taken on the shared memory server, and on the cluster system. Each test was done 100 times. The relative standard deviation for the Serial, Sideways Split, and Staged Split were 0.05%, 2.03%, and 2.04% respectively. Table I shows the speedup for the experiment on the shared and distributed memory systems.

Sideways scalability does spend more CPU cycles due to the need to constantly split and coalesce the data packets when going from one perceptron that is split to a perceptron that has a trunk dependency. This is a 5% overhead. In the cluster test, the extra computation incurred by the sideways scalability implementation does not have as large of a overhead because it is masked by the network delay. The staged implementation may have experience slightly lower scalability given that it misses half a computation cycle because the second stage has to wait for the data to get to it from the first data computation. In contrast, the sideways split implementation can utilize both child perceptrons as soon as they are available. This shows hierarchical dependencies are not a concern for clusters.

B. Programmability and Performance

The 5-point stencil parallel programming pattern is provided to measure programmability. The algorithm used for the experiment was the Jacobi solution for the heat distribution problem[5]. At the advanced layer, data flows with hierarchical dependencies were used to enable the use of automatic scalability. The auto-scalability feature allows the advanced user programmer to split perceptrons during runtime as requested by a load balancer or operating system.

Figure 9 shows a four perceptron stencil. The advanced user is responsible for allocating an arbitrary numerical identification number to each perceptron and each dependency. In the example on the figure, the dependencies are numbered clockwise starting with the right dependency. Perceptron zero, for example, has output dependencies 0:0 and 0:1. the perceptron has the input dependencies 2:3 and 1:2. The stencil pattern is shown in Figure 12. The signature methods on the

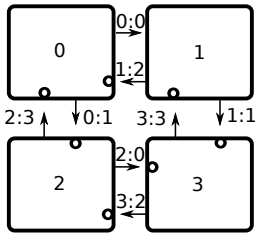

Fig. 9: ID tags for dependencies and perceptrons.

top are required in order to run the application successfully. The bottom signature methods, the ones highlighted, are extra methods that are needed in order to maintain the 5-point 2D stencil general enough for many types of implementations, yet provide automatic scalability. Figure 11 shows graphically how the stencil splits its processing units. In this implementation the perceptron is split into two perceptrons, and the cut is done horizontally. The advanced user is responsible to create new dependencies that will connect the two new perceptrons. The dependencies at the side of the old perceptron would use the hierarchical dependency split feature once deployed.

```
public abstract class Stencil extends BasicLayerInterface{
  boolean OneIterationCompute( StencilData data);
  StencilData DiffuseData(int segment);
  void GatherData(int segment, StencilData dat);
  int getCellCount();
  StencilData[] onSplitState(StencilData data, int level);
  StencilData onCoalesceState(StencilData[] dats,int level);
  Serializable[] splitData( Serializable serial);
  Serializable coalesceData( Serializable[] packets);
}
```

Fig. 10: The stencil pattern required signature methods, and the extra signature methods added to the stencil pattern to provide auto-scalability. Implementing the extra methods is optional.

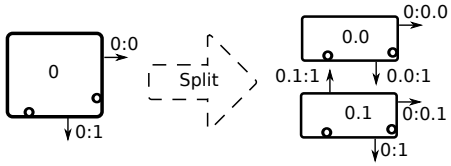

Fig. 11: From left to right, a single perceptron is split into two perceptrons. The new dependencies 0.0:1 and 0.1:1 are created, and the existing dependency 0:0 is split into 0:0.0 and 0:0.1.

The implementation was tested on a shared memory environment, and on a mixed cluster-shared memory environment. The shared memory environment provided useful results that are used to show the technique's performance overhead and its programmability. The mixed cluster-shared environment is mostly used to test the framework's ability to run in that environment using hierarchical dependencies. A performance test was done to measure the overhead incurred by the automatic scalability feature. The total time taken to run the

stencil pattern was measured. Figure 12 shows the results. The static results, as a whole, show the framework's performance when running with the automatic scalability feature turned off. The dynamic test shows the time it takes for the framework to complete the pattern using hierarchical dependencies to automatically scale. The precise procedure is as follows: four processes start running the stencil. At iteration 1,500 the perceptrons hibernate computation and return to the master node to be split into eight perceptrons. Upon allocation of the perceptrons, the computation continues until iteration 3,000 is reached. Finally, the ideal value is calculated using the results from the static measurements of four processes, and the calculated value for eight processes. The graph shows that the framework was able to increase performance during run time at a small performance overhead. The overhead for this test was 0.54% compared to the ideal. In order to enable auto-scalability in the program, the user programmer must add four signature methods in the case of the stencil pattern. For this test, the total increase in lines of code (LOC) was 74. The original pattern implementation was 406. After adding the extra lines, the code increased to 480. Anecdotally speaking, the extra lines of code were not a challenge. How hard this extra step is dependent on the the problem being implemented and the pattern being used. The total increase was 18.23%. The added benefit, is that these lines can be added once the static version of a problem has been debugged during a performance optimization session.

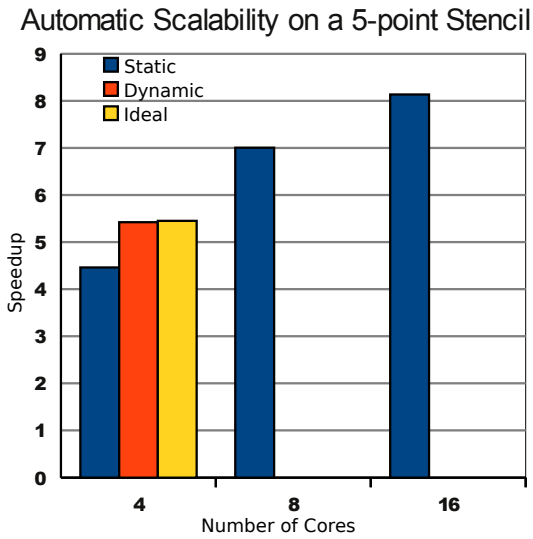

Fig. 12: Performance test to measure auto-scalability's overhead. The y axis shows time.

V. Related Work

Pattern based parallel programming has been done by multiple previous projects [6],[1],[7],[8]. Benefits of the parallel programming approach have being documented by Bromling et al.[1]. The use of multiple development layers was part of the CO2P3S project[1]. Data flows have being used to implement parallel patterns in Muskel[6] and DriadLINQ [9], however only for directed acyclic graphs were used on those projects given that DAGs are preferred for throughput computing. Seeds 5-point stencil allows for undirected acyclic behavior. The use of data flow perceptrons can be compared to synchronous languages such as Luster[10] but their similarity is mostly conceptual since synchronous languages are designed for different applications than those used in parallel computing. Seeds data flows handle dependencies as streams. In that regard, the concept is similar to the StreamIt language[11].

VI. Conclusion

The use of hierarchical dependencies in data flows, in conjunction with the use of parallel pattern programming, can help to provide accessible parallel development to domain specific programmers while at the same time allows parallel computing experts the abstraction space to resolve non-functional concerns without having to know what the parallel program does. In this research, we concentrated our effort on providing automatic scalability and automatic grain size. The patterns allow the programmer enough freedom to implement a solution while at the same time providing scalability to the parallel program. Future work will test how well the Seeds framework scales with our current implementation of hierarchical dependencies as well as research on adding a load balancer and how schedulers are affected by the approach.

References

[1] S. Bromling, S. MacDonald, J. Anvik, J. Schaeffer, D. Szafron, and K. Tan, Pattern-Based Parallel Programming, in Proceedings of the 2002 International Conference on Parallel Processing, p. 257, 2002.

[2] J. Villalobos, Parallel Grid Application Framework. [Online]. Available: http://coit-grid01.uncc.edu/seeds/tutorials.php. [Accessed: 22-Nov-2010].

[3] JXTA, https://jxta.dev.java.net/, Oct-2009. [Online]. Available: https://jxta.dev.java.net/. [Accessed: 14-Oct-2009].

[4] J. Villalobos and B. A. Wilkinson, Skeleton/Pattern Programming with an Adder Operator for Grid and Cloud Platforms, Proceedings of the 2009 International Conference on Grid Computing & Applications, GCA 2010, Jul. 2010.

[5] B. Wilkinson, Parallel programming : techniques and applications using networked workstations and parallel computers, 1st ed. Delhi: Pearson Education Asia, 2002.

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

[7] M. Aldinucci, M. Danelutto, and P. Teti, An advanced environment supporting structured parallel programming in Java, Future Generation Computer Systems, vol. 19, no. 5, pp. 611-626, 2003.

[8] M. Cole, Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming, Parallel Computing, vol. 30, no. 3, pp. 389-406, 2004.

[9] Y. Yu et al., DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language, in Proceedings of the 8th USENIX conference on Operating systems design and implementation, pp. 1-14, 2008.

[10] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, LUSTRE: a declarative language for real-time programming, in Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '87, pp. 178-188, 1987.

[11] William Thies, Michal Karczmarek, and Saman Amarasinghe, StreamIt: A Language for Streaming Applications, Proc. 11th Intl Conf. Compiler Construction, LNCS 2304, Springer-Verlag, pp. 179-196, 2002.