Latency Hiding by Redundant Processing: A Technique for Grid-enabled, Iterative, Synchronous Parallel Programs

Jeremy F. Villalobos
University of North Carolina at Charlote
9201 University City Blvd
Charlotte, NC 28223-0001
(704) 530-7255
jeremyvillalobos@gmail.com

Barry Wilkinson
University of North Carolina at Charlotte
9201 University City Blvd
Charlotte, NC 28223-0001
(704) 687-8381
abw@uncc.edu

ABSTRACT
The increase in interconnected computational resources brought about by the Grid creates the possibility to port multiple parallel programming techniques to it. Porting parallel applications to the Grid could reduce the total computation time, or it could be used to create solutions with higher degrees of resolution. However, the Grid brings with it network state conditions that all too often work to the detriment of expediency in parallel applications. This paper proposes an algorithm designed to significantly reduce the amount of Wide Area Network (WAN) latency experienced when running an interactive synchronous parallel program on the Grid. The algorithm is called Latency Hiding by Redundant Processing (LHRP) and in tests done on two Grid nodes with emulated latency, it complemented Latency Hiding (LH) by performing better than LH on jobs with low internal computation time and performing worst than LH when the computation time was enough to hide the latency.

Keywords
Grid, Synchronous, Parallel, Latency Hiding.

1. INTRODUCTION
Grid Computing uses interconnected computing resources in a geographically distributed area. Multiple projects such as Globus, GrADS, and Cactus [1,2,4] have increased the development of tools that facilitate the creation of grid enabled programs. One aspect that has to be considered is the network state characteristics that are part of any Grid computing organization and the need for new algorithms to deal with this medium. The Grid has inherently high latencies when communicating on a Wide Area Network (WAN). The Grid also provides Grid-enabled programs with a heterogeneous environment for which especial attention has to be paid in order to make efficient use of the resources.

This paper focuses on a technique that can be used to reduce the WAN latency on Iterative Synchronous Parallel Programs (ISPAs) whose internal compute time is too low to allow normal latency hiding to work. It is assumed that WAN latency is always greater than Local Area Network (LAN) latency. This is a realistic assumption since the computers connected through a LAN are closer together than those connected through a WAN. It is also assumed that the ISPA program only requires synchronization with a relatively small amount of processes in its vicinity. Examples of these algorithms can be those found on NetLogo [13], an application to model complex systems. Another application that can serve as an example is LeanMD [11], a molecule simulation program. These simulations and algorithms require only communication among neighboring processes, which is one of the main requirements for LHRP to work when grid enabling these applications.

2. PREVIOUS RELATED WORK
The problem of latency exists in many parallel computing architectures. It is present in supercomputers with little impact to performance. It is also present in cluster computers (networked off-the-shelf computers with special software to perform work in parallel.) Strumpen [12] applied latency hiding to cluster computers. The Latency Hiding (LH) process in parallel programs consists of managing the computation time and the transfer of information over the network such that the idle time on either resource is minimal. The new trend has been porting parallel applications to the Grid, which often works through the Internet. This new platform presents even higher latencies, but with some new characteristics that previous algorithms were not taking into consideration. This section presents similar work that shows that porting parallel application to the Grid is being pursued by other organizations, and that those projects also have devised forms to manage Grid latency.

The idea of porting parallel applications (other than embarrassingly parallel) to the Grid can be considered impractical by some people. But recently, multiple projects have sprung up to provide this capability. MPICH-G2 provides basic tools necessary to run parallel programs on the Globus platform. Other projects include Charm++ [7] and Java-PVM [6]. El Maghraoui [3] also applies multiple tools to run parallel programs on the grid such as check pointing, load balancing, and process migration on a project called Internet Operating System (IOS). The tools, although helpful, are just the beginning steps in order to be able to tackle the task of porting parallel applications. Dealing with latency is still left to the programmer in the case of IOS. Koenig et al. [9] used Charm++ and AMPI in conjunction to provide dynamic, application independent load balancing and latency hiding. The paper proposes the use of multiple "virtual processes" to manage the latency. The CPU's in the grid are loaded with enough virtual processes so that the internal computation exceeds the Grid latency, effectively hiding the latency. The method is innovative in that it adapts MPI applications with little recoding. But it only provides normal latency hiding. It cannot hide latency if there is not enough

work to keep the CPU busy while the information is being transfered. Li et al. [10] propose a peer-to-peer strategy to run asynchronous parallel programs on a Grid made up of clusters and individual workstations. It is a type of SETI@HOME project, but with general purpose asynchronous scientific application. The algorithms covered by this approach include fluid dynamics, aerodynamics, nuclear reactor dynamics and systems of equations. The system deals with the latency by "relaxed synchronization." With relaxed synchronization, the processes can continue to compute even if the information from the neighbors has not being updated. One side effect of this approach is that the simulations take more iterations to converge. But, overall the technique is faster than the synchronized approach. Li et al. [10] however, does not solve the latency problem for other types of n-stencil problems, such as the game of life and other complex adaptive system simulations. LHRP can run both asynchronous and synchronous algorithms. LHRP also hides the latency even if the compute time is less than the transfer time.

3. DISCRETE SOLUTION TO THE HEAT DISTRIBUTION PROBLEM

A discrete solution to the heat distribution algorithm was used to test LHRP. The heat distribution algorithm computes iteratively the temperature distribution of a section of a material. The simplified version consists of a bidimensional square made of the same material with some boundary conditions at the edges. The formula to calculate one iteration in this problem is:

$$h_{x,y} = \frac{h_{x+1,y} + h_{x-1,y} + h_{x,y+1} + h_{x,y-1}}{4}$$

The formula is applied to a matrix h whose values represent the temperatures in the material. The location of that cell is denoted by the subscript coordinates x and y. The basic process is to take the average value of the neighbor cells and set this as the new value. The equation is repeated for a number of iterations or until the problem converges. This problem is parallelized by partitioning the data matrix into sections. This is done by a master process. The slave processes receive the data and iterate on it. The implementation of this solution reveals one more obstacle to parallelizing the heat distribution problem. If each of the processes receives exactly the data matrix on which it will be computing, it will have to request its neighbor processors for a single cell of their data matrix every time they are computing on the edge of the data matrix. Communicating often for only small bits of information makes inefficient use of the network and it slows down performance even with LAN latency and speed. The solution is to allocate a data matrix on each processor in such a way that the data matrices overlap by a border of some width of cells. In this way, the computation section can be more easily separated from the synchronization section and the network is better used by transferring large chunks of data at a time. The presence of the cells from a neighbor processor on the local processor are called *ghost points* [14]. For the rest of the paper, a border of *ghost points* will be referred as a border.

Implementing the procedure just mentioned on a LAN cluster with MPI is fairly straightforward. It is also quite efficient. The algorithm's total computation time can be divided into two sections: the internal computation time, and the communication or transfer time. The computation time is when the processor is working to compute a new state in the data based on the previous data. The communication time is when the borders are being exchanged. Since there is no communication while the internal computation is going on, the network is going unused during this time. Then, when the borders are being exchanged, the CPUs are idle. Since the LAN latency is significant in comparison to the speed of the CPUs. The conventional approach leaves plenty of room for improvement. Strumpen [12] solved this by first computing the border, then starting the transferred of the borders in a non-blocking form, and computing the core of the matrices while the borders are being transferred. The last step is to wait for the acknowledgment of the sent and received messages, which is also referred to as the synchronization step.

Latency hiding works fine when the latency is homogeneous throughout the network, or when there is enough internal compute time to hide the highest latency in the network. The issue with the Grid is that the WAN latency tends to be orders of magnitude higher than LAN latencies. Although most of the communication happens withing the LAN, there are points in which the CPUs from different Grid nodes have to communicate and synchronize. These points may not constitute the majority of the communication, but the delay caused by these communications slow down the rest of the process to WAN latency, independent of the system's LAN latencies. Another issue is not having enough internal compute time to hide the latency. Having a reduced amount of internal compute time is expected when porting a program to the Grid. Since a Grid usually has more computer resources that a LAN cluster, it is expected that a program would be able to use more of these resources. If the original data set does not change, this means that a smaller amount of work will be assigned to the Grid's CPUs than it would be to a LAN cluster's CPU. LH's performance goes down with decreasing internal compute time.

4. LATENCY HIDING BY REDUNDANT PROCESSING (LHRP)

Latency Hiding by Redundant Processing(LHRP) addresses the case in which a programmer wants to run an ISPA that does not have enough work to keep the processors busy using LH. It is also helpful to mention that limited bandwidth, 100 Mbps for example, may increase the transfer time by having to wait additional units of time for the transfer to reach the other side.

The main contribution of LHRP is to allow most of the internal processors of each cluster to continue with the computation while the information reaches designated processors. This is done by having redundant copies of some of the data on the clusters on each side of the *Grid gap,* which is the point where two Grid nodes communicate. The *Grid gap* goes over the WAN, therefore the latency is much higher than in inter-cluster communications. LHRP uses some of the processors to redundantly compute some part of the data. The side effect of this behavior is having less CPUs devoted to computing the final answer. However, this may not be a problem since there would be more processors available in a computational Grid in comparison to the amount of processors in a single local cluster

LHRP is design for grids that are made up of cluster computers. The clusters are usually made up of tens or hundreds of workstations or servers. The latency inside the clusters is low. The Grid connects multiple clusters together with a high bandwidth, high latency connection. Figure 1 shows a typical Grid topology for these types of clusters. It is unusual to share a single workstation in the Grid. LHRP is not designed to run a Grid that is made up of single workstations such as it is assumed in [10].

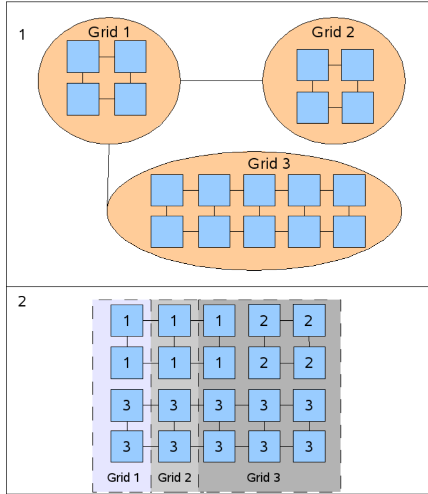

Figure 1: (1) Grid topology for cluster computers. (2) Layout of CPU's for a synchronous parallel application

LHRP's features can be divided into three parts. First, the stages of computation and communication are reviewed. The stages describe when the processes compute, transfer, or do other tasks that are specific to LHRP. The second part explains what is a *far border* and how the nodes are design to behave when they have a far border. The third part explains the versioning system used by the *buffer node.* But first, *s*ome terminology has to be explained before we go into the features of the algorithms

LHRP assigns three different types of procedures depending on the location of the node. The types are: *internal node*, *border node*, and *buffer node*. The *internal node* behaves just like a node on the latency hiding implementation of an n-stencil program with latency hiding. It is isolated from the *Grid gap* by the *border node.* The *border nodes* are between the *internal node*s and the *buffer nodes*. The *buffer* and *border nodes* are the ones that compute over the *Grid gap*. The *border node* behaves almost the same as an *internal node* with all the neighbor processors except for the places where it has to interact with the *buffer node*.

Figure 2 shows an example of how the processors would get different types of assignments depending on their position on the computational Grid. The example in the figure is a 1x6 matrix, CPUs one, two and three belong to grid01 and CPUs 4, 5, and 6 belong to grid02. The rectangles to the sides of the squares represent the border. In a bidimensional CPU matrix, there would also be a border at the top and bottom of the squares. CPU 1 is considered an *internal node* since it only interacts with CPU 2. CPU 2 gets an assignment of *border node* since it has CPU 3 as a neighbor, and CPU 3 is a *buffer node*. Therefore, CPU 2 has to exchange the local borders with CPU 3 as denoted in the figure by the small lines between CPU 2 and CPU 3, but also CPU 2 has to send the same border to CPU 4 which has an identical copy of the data matrix in CPU 3.

The *border node* interacts normally with the *buffer node* that is part of its local Grid node, but also sends the border of information to the *buffer node* from the external Grid. The *border node* has to be able to wait multiple cycles for the acknowledgment of the borders it sends to the far *buffer node*, therefore it is equipped with multiple buffers to store these borders. The amount of extra buffers depends on the ratio of the Grid latency to the internal network latency.

The *buffer node* is the most complicated of the nodes in LHRP. It has two main features. First, it has multiple buffers to receive the borders sent from the *border node* in order to be able to wait for the borders across multiple cycles. Second, it has a different algorithm to compute the core of the matrix in order to be able to keep feeding data to the *border node* while the borders from the external Grid arrive. The *buffer nodes* at each side of the grid latency barrier are assigned the same data set. They handle the redundant processing part of LHRP.

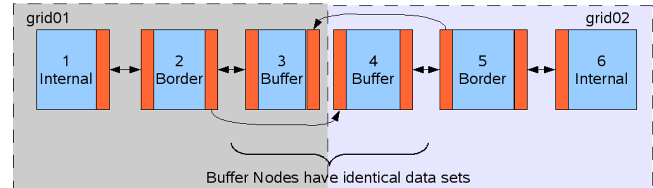

Figure 2: LHRP's CPU assignment example for a 1x6 CPU matrix.

The *buffer node* uses a versioning system to feed current information to the local *border node* (The node that is part of the LAN and needs to be updated more often.) At the same time, the *buffer node* lets the information coming in from the *far border* node get outdated by the slow latency. Because the data is also on the border node at the other side of the *Grid gap*, there is no node that immediately needs the outdated information.

Stages of Computation and Communication

Figure 3 shows a graphical representation of the compute and transfer modules for LHRP. In step one (1), the borders are computed first as in a standard implementation of a latency hiding n-stencil problem. For step one, all nodes behave the same. In step two (2), The borders are transferred non-blocking as in a standard implementation of LH. For step two, all the nodes behave the same way. Only the *border node* performs step three (3). In step three the node sends its border to a *buffer node*. The node is design so that it can wait multiple cycles for the border to arrive at the *buffer node*. In step four (4) the compute section is done as in LH. At the same time, the *buffer node* executes a versioning algorithm to generate results that allow the rest of the LAN to continue working. Finally, in step five (5), the nodes wait for the acknowledgment of the sent and received messages. the *buffer* and *border nodes* also perform this step with other neighbor CPUs, except if the border CPU is a *far buffer node* or a *far border node.* In this case, the nodes can wait multiple iterations until the acknowledgment arrives.

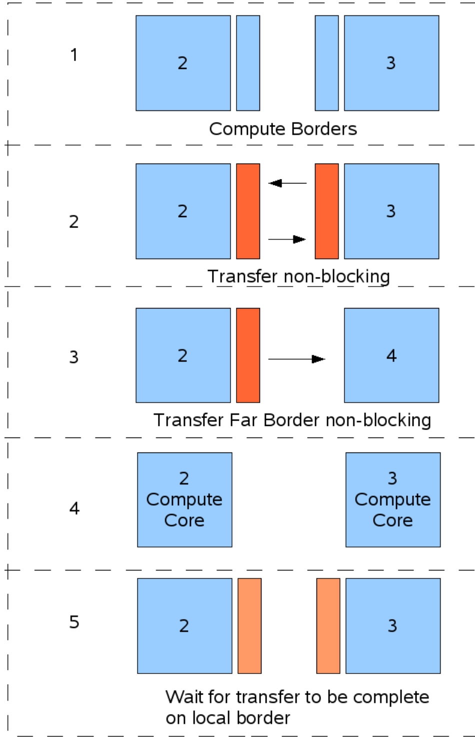

Figure 3: Representation of compute and transfer modules in LHRP

Transferring Data over Grid Gap

The following two concepts are interrelated. As mention previously, both *border node* and *buffer node* are equipped with the ability to send and receive multiple borders respectively. This means that the *border node* can send some quantity n of borders x over the network without having to wait for the acknowledgment of x_{k-1} before sending x_k. For now, let's assume that the *border node* is able to keep its data matrix updated. The result is that the *border node* can maintain a constant output of borders to the *far border node* of $X*\frac{1}{\frac{1}{b}+\lambda}$ borders per unit of time. Where X is the number of borders available to be transferred, λ is the LAN latency, and b is the latency due to the bandwidth in borders per unit of time. Since we have already established that the Grid environment is characterized by high bandwidth and high latency, the model reduces to $X*\frac{1}{\lambda}$. This concept presents an advantage because it sends a stream of borders, which maximizes bandwidth use while reducing the impact of point-to-point latency.

LHRP needs to provide a way that allows the program to send a stream of borders without allowing the data to get outdated. One can reason that if the program waits until it has a batch of borders to send, then the computation will stop after one iteration, since the data falls out of date after just one iteration. So, what the algorithm needs is another process that maintains the processes busy while the batch of borders are sent across the grid gap.

Buffer Node Versioning System

The process needed to maintain the processors busy while the borders are transfered across the *grid gap* is the versioning system. The versioning system uses identical copies of the data matrix at the *buffer nodes* at each side of the *grid gap* to maintain the creation of new borders for the LAN CPUs. Since it uses the same data to maintain the data up to date, this is the part of the algorithm that uses redundant processing.

Grid to Local Latency Ratio = 3													
column coordinates	1	2	3	4	5	6	7	8	9	10	11	12	13
row coordinates	Node 2		Node 3					Node 4					Node 5
1	1		1	1	1	0		0	1	1	1		1
2	2		2	2	1	0		0	1	2	2		2
3	3		3	2	1	1		1	1	2	3		3
4	3		3	2	2	2		2	2	2	3		3
5	3		3	3	3	3		3	3	3	3		3
6	4		4	4	4	3		3	4	4	4		4
7	5		5	5	4	3		3	4	5	5		5
8	6		6	5	4	4		4	4	5	6		6
9	6		6	5	5	5		5	5	5	6		6
10	6		6	6	6	6		6	6	6	6		6
11	7		7	7	7	6		6	7	7	7		7
12	8		8	8	7	6		6	7	8	8		8
13	9		9	8	7	7		7	7	8	9		9
14	9		9	8	8	8		8	8	8	9		9
15	9		9	9	9	9		9	9	9	9		9

Table 1: Model representing the versioning of columns inside a buffer node.

In Table 1, the rows represent different cycles for each of the nodes. The red separations (columns 2 and 12) represent the network latency and the black separation (column 7) represents the Grid latency. Therefore, we can imagined the version number going over the network, either red or black columns, into the other node. The purpose of the example's algorithm is for all the cells in a one dimensional matrix to move on to the next version, that is from version k to version $k + 1$, only if its neighbors are also in version k. For the table, the Grid to internal latency ratio is three, which means that on average a border sent from node two will take about three cycles to reach node four. Once the borders start to be transferred, the subsequent data is transferred at the speed the bandwidth allows. The numbers do not represent data, but the version of the data. For example, in order for one cell to move to version two, its neighbors must be in version one. This strategy create the need for a temporary storage matrix, since some of the data may move on to higher versions, and the old version may be needed by a neighbor that has not been able to advance in version. As an example, look at column-row coordinate (4,2). On this cell, the column was able to compute version two; however, the cell to its right is stuck at the time with version one. Eventually (5,2) will need to compute version two, but (4,2) will have a version higher than the one needed. In order to provide the correct old version, a matrix of temporary old values is necessary. Only one matrix is needed to store temporary values since the cells will go at most one version ahead of its neighbors.

The gray highlight indicates when the first message starts to be transferred from node two and five. Since it takes three cycles for the border to arrive at the *buffer node*, it is until row three that it is received by node three. This delay creates a type of cascading of versions which is also highlighted in gray. Once this cascade reaches the cells that are adjacent to the border of the *border nodes*, the *border nodes* are left waiting for the version to get up to date. Meanwhile, the *buffer node* will keep working on getting the latest *far border* from the far *border node* and updating the version of the cells until they are ready to

transfer borders with the *border nodes*. The whole process takes WAN latency units of time to restart.

At this point, it may be clear how the interaction between the border transfer over the *grid gap* and the *buffer node's* versioning algorithms work together to provide latency hiding with the use of redundant processing. To be thorough, lets analyze the interaction now that both functions are well explained. The *border node* sends its borders over the *grid gap* as soon as they are computed. While the border is on its way to a remote Grid node, the *buffer node's* versioning system creates new up to date borders to be used by the LAN cluster. Note that because there are identical sets of data at the *buffer nodes*, there is no need to wait for information to proceed. The versioning system is able to allow the *border node* to send its borders over the *grid gap* in batches. But, the process does not last forever. Eventually, the *buffer node* also runs out of up to date information and it has to wait for borders to arrive from a remote Grid node. The time for the whole cycle to start again is the *grid latency*. In the end, the two processes working together are able to send the borders in batches over the *grid gap* while at the same time maintaining the local CPUs busy.

Modeling LHRP's Performance

This section creates a model that helps predict LHRP's performance. LHRP does not completely hide the Grid latency, but it does reduce it to less than what it would be with LH as long as the compute time does not hide enough of the Grid latency. The following describes the algorithm's theoretical performance:

Let G be the Grid latency, I is the internal latency and B is the amount of the data tuples being used by the *buffer node*. For the example being used in this paper, B is the width of the data matrix assigned to the *buffer nodes*. The average latency created by LHRP can be computed to be:

$$(1)\quad \frac{G+B*I}{B}$$

Equation 1 considers the Grid latency for the first transfer. The subsequent transfers are started every I units of time. Once the buffer runs out, the Grid latency is incurred again. The algorithm assumes that the number of iterations required by the problem is much higher than B. An example of the behavior explained in this formula can show how dramatic the improvement is. Suppose a Grid presents a LAN latency of 10 ms which is typical for an internal network. The WAN latency is 200 ms, and the programmer decides to use a buffer of 50. The average latency for the program is 14 ms.

For negligible compute time, LH latency formula would be just the Grid latency. For the last example the result would be 200 ms. However, the average latency is not the only factor to consider. Another two factors that should be considered when comparing LH to LHRP are the loss in computational resources by LHRP due to redundant computation and the computation time. Computation time could be a variable to observe at which point LHRP would start to be a better approach than LH. Let W denote some total amount of work that has to be done in units of time. The work can be divided by the number of CPUs performing non-redundant work, with C being used by each algorithm. The revised formula to predict performance for the *Grid gap* is:

$$max\left(\frac{G+B*I}{B},\frac{W}{C}\right)\ .$$

Figure 4 shows the estimated total time taken to complete an average cycle by LH vs LHRP. The x axis represents the computation time per subcell. The Grid latency for the model is 200 ms and the internal network latency is 10 ms. From the chart, it can be inferred that for this example, it makes sense to use LHRP when the computation takes 11,000 ns per subcell or less. Once the computation time goes higher than that, this creates enough computation time to hide the transfers with LH. At that point, the redundant node in LHRP starts to reduce its performance.

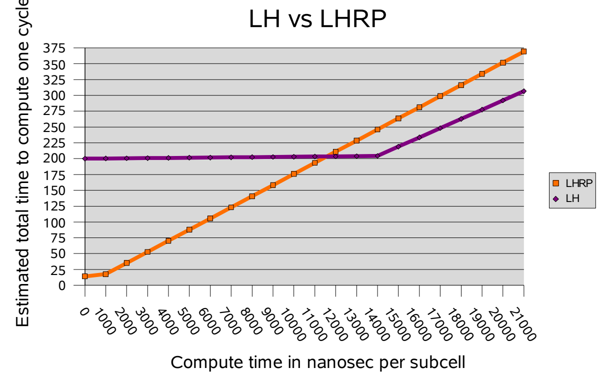

Figure 4: Estimated total time to complete one cycle as the compute time changes.

Figure 5 shows the plot with a variable Grid latency from 10 ms to 170 ms and leaves the computation time factor constant at 2000 ns per subcell. The internal latency is left constant at 10 ms. It can be seen that LHRP starts out being slower than LH mainly because of the node lost to redundant computation. Once the Grid latency goes above 40 ms, LHRP starts gaining speed over LH. This prediction will be revisited in the experimental results section.

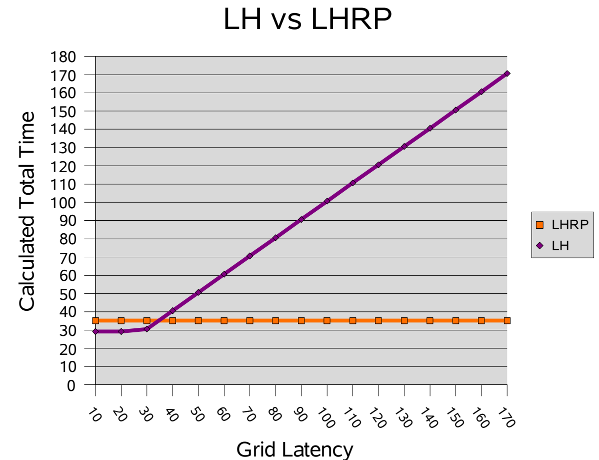

Figure 5: LH vs LHRP with variable Grid or WAN latency.

LHRP can have multiple Grids with multiple grid latencies in between. Only one grid division is considered in this section to simplify the explanation of the algorithm. Only one requirement is needed for any number of grid divisions. The CPU matrix has to have a straight line across the CPU matrix, horizontal or vertical, that denotes the Grid division. In Grid division, having

a corner in the middle of the CPU matrix, for example, increases the size *of far borders* in at least one processor which requires the *buffer node* to be more complicated. LHRP does not have a feature to deal with this processor arrangement.

LHRP has one more issue. The algorithm as described up to now may need more accommodations when the *buffer node* needs to exchange borders. The main issue here is that the versions may not be the same. One *buffer node* may receive a far border before its neighboring *buffer nodes* that are above or below it in the CPU matrix. One way to deal with this is to have the *buffer nodes* share the version number of the columns they have. In a way this would tie up the *buffer nodes* to be synchronized to the same versions. The program created to produce the experimental results does not include this feature; therefore, it can only work with one row in the CPU matrix. It can be speculated that this additional level of complexness would slow LHRP somewhat. But considering that the *buffer nodes* would be communicating through their local networks, the cost would probably still justify the use of this procedure. Implementing the version sharing feature will be done for future work.

5. EXPERIMENTAL RESULTS

The test of LHRP consisted of a CPU matrix of one by six. A single row as the CPU matrix's height is used since the exchange of versions between *buffer nodes* was not implemented at the time of testing. The algorithm was compared with LH. The performance was observed while two factors were varied: the Grid latency and the computation time. The environment used for the test was made up of two blade Xeon servers with two multi-threaded CPU, which is effectively four CPUs each. Each of the servers had a complete installation of the Globus Toolkit [4,5] as well as MPICH-G2 [8]. The latencies were emulated. The network consisted of a 100 Mbps LAN.

Grid Latency

Grid latency is varied from 0 to 700 ms latency with an emulated transfer of 100 Mbps. LH is supposed to be faster than LHRP when the Grid Latency is small, because LH is using 6 processors to do the work and LHRP is effectively using only 5. Figure 6 shows the results of this test.

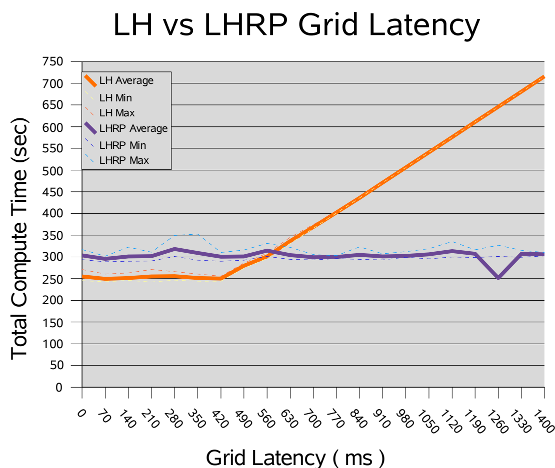

Figure 6: Total time computed in seconds and Grid latency in milliseconds.

The internal latency for the Grid latency test was constant at 10 ms and compute time was held constant to 30 ns per subcell. The data matrix was 300x300, each processor got 50x300=15000 cells for LH and 60x300=18000 cells for LHRP which determines the work the CPU would need to do. Table 2 shows calculations on how long the processors should take to compute the sub-matrices with different numbers of cells.

Table 2: Difference in load on a processor due to the redundant use of one processor

Algorithms	*Number of Cells*	*Compute Time per Subcell*	*Total Time to Compute on Cycle*
LH	15,000	30	0.45 ms
LHRP	18,000	30	0.54 ms

Note that the performance of LHRP over LH is very similar to the predicted results shown above. It is important to note that the emulation algorithms only emulate the delay part of the network, but other factors such as congestion could change these results in a real-world test.

Other measurements to consider include the use of system memory because LHRP requires buffers to receive the borders. LHRP also needs to keep an extra matrix with the same dimensions of the data matrix to store old versions of the columns. Table 3 shows the difference in memory footprint in bytes when solving a 500x500 cell matrix. The average increase in memory from using LH to using LHRP is 21%. The increase is not significant and CPU performance as well as memory capacity are projected to go up in the foreseeable future. Latencies of any kind, on the other hand, are not expected to be reduced because of physical limits such as the speed of light over fiber optics and the speed of electrons through copper cable.

Table 3: LH vs LHRP. Memory foot print of each of the six processes while computing on the same data. The amounts are in bytes.

Process	*LH*	*LHRP*
0	35,724	41,856
1	7,408	8,228
2	7,408	10,932
3	7,408	10,928
4	7,412	8,528
5	7,408	8,232
6	7,544	8,532

Compute Time

In the next test, the computation time is changed. LH is better than LHRP when the compute time is high because this provides enough time to transfer the borders, and LH is using six processors to do the work while LHRP is using five. Figure 7 shows the results gathered from the experiments. The total time computed is expressed in seconds; the compute delay is in nanoseconds per subcell. There is some difference in the units,

mainly because the theoretical model only runs one cycle averaging values where more than one cycle is needed. In contrast, the experiment included a few hundred cycles to allow for the differences to build up and measure the average performance. Nevertheless, for future work the theoretical model can be modified to predict the programs behavior with greater accuracy.

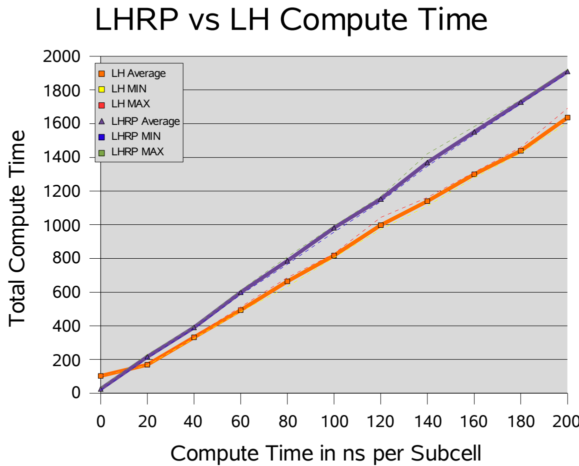

Figure 7: LH vs LHRP. Compute time test.

6. CONCLUSION AND FUTURE WORK

LHRP brings an effective solution to deal with the increase in latency brought by the use of multiple clusters across the Internet through the Grid. It complements LH when the amount of internal computation is too low for it to be used to hide the latency. LHRP trades in CPU power and memory capacity in exchange for lower perceived latency at a time when there is growing performance improvement of the former and flattening performance of the latter. In the experiments conducted, LHRP proved to be a better algorithm over LH when the internal computation was low.

For future work, we will add the *buffer node version sharing* feature. Currently the algorithm can only be used in a one dimensional CPU matrix. The addition of this feature should increase scalability at a small performance cost. LHRP could also be tested with specialized network emulations tools. For the experiment, the latency was emulated withing the program. Also, the model used to predict the programs behavior can be modified to increase accuracy. LHRP could also be extended to accommodate future cluster topologies where LHRP can be used on clusters of multi-core CPU's. In the near future, there will be a 64 core CPU. This type of topology could use LHRP to bring the LAN latency somewhat closer to the inter-CPU latency. Adding LHRP to multi-core CPU's would also create the need to implement it in multidimensional layers. In other words, the block of work assigned to a CPU by the Grid would have to be repartitioned with the LHRP algorithm to distribute the work among the cores. Having LHRP on multiple layer would increase the efficiency of both the Grid latency and the LAN latency in relation to the inter-CPU latency.

7. REFERENCES

[1] Allen, Gabrielle; Benger, Werner; Dramlitsh, Thomas; Goodale, Tom; Hege, Hans-Christian; Lanfermann, Gerd; Merzky, Andre; Radke, Thomas; Seidel, Edward; Shalf, John. Cactus Tools for Grid Applications. *Cluster Computing*, pages 179 – 188, November 2004.

[2] Berman, Francine, Chien, Andrew, Cooper, Keith, Dongarra, Jack, Foster, Ian, Gannon, Dennis, Johnsson, Lennart, Kennedy, Ken, Kesselman, Carl, Mellor-Crumme, John, Reed, Dan, Torczon, Linda, Wolski, Rich. The GrADS Project: Software Support for High-Level Grid Application Development, *International Journal of High Performance Computing Applications*, pages 327-344, 2001.

[3] El Maghraoui, K. A Framework for the Dynamic Reconfiguration of Scientific Applications in Grid Environments. Doctor of Philosophy dissertation, Rensselaer Polytechnic Institute at Troy, New York, 2007

[4] Foster, Ian, Kesselman, Carl. Globus: a Metacomputing Infrastructure Toolkit, *International Journal of High Performance Computing Applications*, pages 115-128, 1997.

[5] Globus. The Globus Toolkit, July 2007. URL: http://www.globus.org/tookit/

[6] Java PVM (JPVM). The JPVM Home Page, September 2007. URL: http://www.cs.virginia.edu/~ajf2j/jpvm.html

[7] Kale, Laxmikant. Krishnan, Sanjeev. Charm++:Aportable concurrent object oriented system based on C++, *Proceedings, Conference on Object Oriented Programming Systems, Languages and Applications*, September 1993.

[8] Karonis, Nicholas T., Brian Toonen and Ian Foster, MPICH-G2: A Grid-enabled implementation of the Message Passing Interface, *Journal of Parallel and Distributed Computing*, Volume 63, Issue 5, Special Issue on Computational Grids, Pages 551-563, May 2003.

[9] Koenig, Gregory. Kale, Laxmikant. Using message-driven objects to mask latency in grid computing applications. *Proceedings of IEEE International Parallel & Distributed Processing Symposium* (IPDPS), 2005.

[10] Li, Z. Parashar, M. A Decentralized Computational Infrastructure for Grid-Based Parallel Asynchronous Iterative Applications. *Journal of Grid Computing*, 2006.

[11] Mehta, V. LeanMD: A Charm++ framework for high performance molecular dynamics simulation on large parallel machines. Master's thesis, Univeristy of Illinois at Urbana-Champaign, 2004.

[12] Strumpen, V.; Casavant, T.L. Exploiting communication latency hiding for parallel network computing: model and analysis. *International Conference on Parallel and Distributed Systems*, pages 622-627, December 1994.

[13] Tisue, S. and U. Wilensky. NetLogo: A Simple Environment for Modeling Complexity. *Proc. International Conference on Complex Systems* (2004).

[14] Wilkinson, Barry; Allen, Michael. *Parallel Programming, Techniques and Applications Using Networked Workstations and Parallel Computers 2nd Ed*. Prentice Hall, New Jersey, 2005.