PLU FACTORIZATION ON A CLUSTER OF GPUS USING FAST ETHERNET

André Rigland Brodtkorb, Martin Lilleng Sætra and Trygve Fladby

2007

Abstract In this white paper, we present a novel approach to solve linear systems of equations on a cluster using the PLU factorization. We use the graphics processing unit (GPU) as the main computational engine at each node, and a block-cyclic data distribution to solve the system. The local computation is a new way of solving the PLU factorization on the GPU. It utilizes the full four-way vectorized arithmetic found in most GPUs, and a new pivoting strategy. The global algorithm uses the message passing interface (MPI) for communication between nodes. We show that our algorithm is highly efficient on the local nodes, but bounded by the relatively slow network. A faster network will eliminate this bottleneck, and the speed of the local computations show promising results.

1 Introduction

This paper explores the field of general purpose computation on graphics processing units (GPGPU). We specifically target the PLU factorization of a large system of linear equations on a cluster of nodes. Solving large linear systems of equations using dense algorithms is used extensively as a benchmark for clusters and supercomputers. The High Performance LINPACK benchmark (HPL) [1] which computes the PLU factorization, is the standard way of benchmarking and ranking the fastest 500 supercomputers in the world [2]. This benchmark, however, has been criticized for neglecting the importance of faster inter-node communication. This is because the HPL benchmark can run the benchmark with different parameters that compensate for slow network communication by letting each node execute extra computations (e.g., look-ahead).

While the HPL benchmark uses the CPU to compute partial results on each node, we utilize the graphics processing unit (GPU) as the main computational engine to solve the same problem. The GPU is a massively parallel processor with vast amounts of processing power [3]. Current GPUs have a theoretical peak of 400 GFLOPS [4], compared to 90 GFLOPS [4] for current high-end CPUs. When comparing the price[1] per FLOP, the GPU comes out ahead as well with approximately $1.50 per GFLOP, compared to the CPU that costs approximately $18 per GFLOP.

During the last years, we have seen an enormous development in 3D-graphics. The demand for more powerful programmable graphics processing units (GPU) from for example the gaming industry has led to increased flexibility in the processors. The rapid evolution in speed and flexibility has made the GPU interesting for scientific purposes as well. The field of general-purpose computation on GPUs (GPGPU) has emerged as a new and exiting research area [3]. Even though the GPU is a far more powerful and cost-effective processor than the CPU, there is another price. While the CPU has complex logic for branch prediction, cache management, and instruction pipelining, most of the transistors on the GPU are used for pure floating-point operations. There is another architectural difference as well. The CPU is designed to operate on sequential code, such as word processing where each character is entered and processed sequentially. The GPU on the other hand, is designed to simultaneously compute all the pixels that together make up the screen image. In addition, the GPU could traditionally only be accessed via a graphics API, such as OpenGL [5] or DirectX [6]. The architectural differences, and the need to access the GPU through a graphics API require new algorithms and techniques to be employed when the GPU is to be used for general-purpose computing.

2 Background

The Top 500 project [2] was started in 1993 to provide a reliable basis for tracking and detecting trends in the field of high-performance computing. It is a list of the 500 most powerful supercomputers, which is updated twice per year. The ranking of the supercomputer sites is determined by how well they perform on the LINPACK benchmark. A parallel version of LINPACK named HPL [1] was introduced by Dongarra, for this purpose. HPL is short for High-Performance LINPACK Benchmark for Distributed-Memory Computers. HPL utilizes the Message Passing Interface (MPI) and the Basic Linear Algebra Subprograms (BLAS). The algorithm used by HPL implements a two-dimensional block-cyclic data distribution. In addition a look-ahead strategy and bandwidth reducing swap-broadcast algorithm is used to increase performance. The complete operation count sums up to $\mathcal{O}(\frac{2}{3}n^3) + \mathcal{O}(n^2)$.

[1] Prices are from the Norwegian web shop komplett.no 2007-04-23.

Listing 1: Example on a deadlock in an MPI-2 program

```
MPI_Init(&argc, &argv);

if(processId == 0) {
  MPI_Recv(buf, 10, MPI_INT, 1, 101, MPI_COMM_WORLD, &status);
  MPI_Send(buf, 10, MPI_INT, 0, 100, MPI_COMM_WORLD);
} else(processId == 1) {
  MPI_Recv(buf, 10, MPI_INT, 0, 100, MPI_COMM_WORLD, &status);
  MPI_Send(buf, 10, MPI_INT, 1, 101, MPI_COMM_WORLD);
}

MPI_Finalize();
```

LU factorization on the GPU has previously been implemented by Galoppo et al. [7]. One of their main contributions was index-pair streaming, which uses texture coordinates to make a cache-oblivious algorithm. The index-pair streaming technique sets texture coordinates from the CPU in order for the GPU to pre-fetch data, in contrast to computing them on the fly on the GPU. This data pre-fetch resulted in about 25% speed increase [7]. They also reported their algorithm as faster than ATLAS, but the benchmark was highly synthetic.

To run our application in parallel on multiple nodes, we have utilized the Message Passing Interface 2.0 (MPI-2) [8]. MPI-2 is a C/C++ and Fortran interface for message passing between multiple processes spread over any number of nodes. It can be used in many different setups, e.g., supercomputers, distributed memory clusters, and shared memory clusters. Several implementations of MPI-2 exist, where we have chosen MPICH2 [9] for our application. The most important uses of MPI-2 in our application are the automatic generation of a block-cyclic Cartesian grid of processes and broadcast of data to groups of processes.

There are two concepts related to our use of MPI-2 that require some explanation; communicators, and blocking- and non-blocking calls. A *communicator* in MPI is a collection of processes. Many functions in MPI-2 take a communicator as argument and perform the requested operation on all processes in that communicator. A call to the broadcast function in MPI, for example, can look like this: `MPI_Bcast(buf, 10, MPI_FLOAT, 0, MPI_COMM_WORLD)`. This call will broadcast ten elements of the array `buf` to all processes in the `MPI_COMM_WORLD` communicator. The other processes in the communicator must also call the `MPI_Bcast` function to receive these elements. The `MPI_COMM_WORLD` communicator is a special communicator that contains all processes, and it is initialized automatically by MPI. When an MPI function is called on all processes within a communicator (or group) it is referred to as a collective operation. `MPI_Bcast` is a collective operation.

A *blocking* call will make the application wait for the call to complete before continuing execution. In this way you will know if the call has finished successfully or aborted due to some error. This also means that the application may get deadlocked, where two or more processes have called competing blocking functions that are circularly dependent on each other [10]. For example, if we have two processes that execute the code in Listing 1, it will result in a deadlock. Both processes are waiting for the other to send data, thus blocking program execution. A non-blocking call on the other hand, will not cause the application to wait for the call to return. In this way it is possible to call a function and continue executing the application before the function returns. Collective operations in MPI-2, however, are always blocking.

3 Algorithm

The LU factorization of a matrix A can be written as $LU = A$, where L and U are *lower* and *upper* triangular respectively. Using the Doolittle algorithm, we can construct the upper triangular matrix U using Gaussian elimination. The lower triangular matrix is constructed from the multipliers used to reduce A to an upper triangular form. For our algorithm to be numerically stable, we also permute the rows of A. This is known as partial pivoting, and ensures that the row we are eliminating with creates smaller perturbations of the result than would normally occur. With the permutation of the rows in A, our factorization takes the form $A = P^T LU$, where P is the permutation matrix that permutes rows of A.

Our algorithm has two layers, the global and the local computation. The global algorithm solves the PLU factorization of the matrix spread over all the nodes, shown in Figure 0b, whilst the local algorithm is what each node needs to compute for the global algorithm to be correct.

Each node in the computation receives a block-cyclic part of the matrix, as shown in Figure 0b. Then, all the

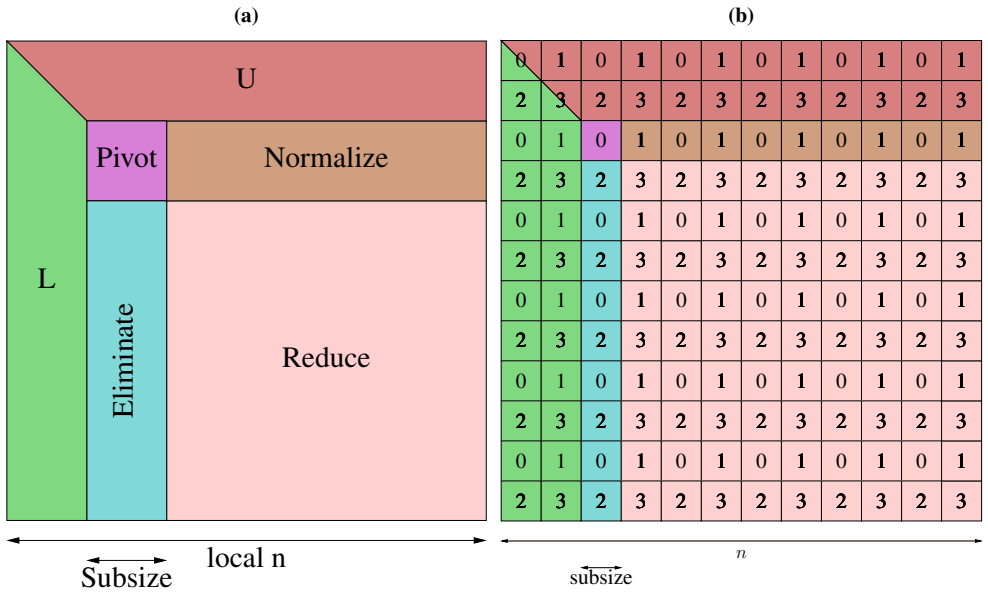

Figure 1: PLU decomposition on a cluster of nodes: (a) the four different parts of the LU factorization. (b) the block-cyclic distribution of data on four nodes, 0, 1, 2 and 3.

processors compute what type of operation they need to compute. Our algorithm splits the computation into four distinct operations: pivot, normalize, eliminate and reduce, as shown in Figure 0a. The operation computed on each node depends on the global position of the pivot operation. All processors that hold elements in the same row as the pivot operation need to compute the normalize operation, and similarly all nodes with elements in the same column as the pivot operation need to compute the eliminate operation. All remaining nodes need to compute the reduction operation. In Figure 0b this means that process 0 is the *pivot*, process 1 executes *normalize*, process 2 *eliminate*, and process 3 *reduce*. The pivot node shifts one down along the diagonal for each global pass.

3.1 Global algorithm

Computing the PLU factorization is an almost embarrassingly parallel operation. However, vanilla implementations demand a lot of data to be transferred between nodes, which is a very costly operation. In addition, many nodes would simply idle as we reach the end of the computation.

To reduce the idling, we distribute the matrix A block cyclically in the same fashion as the HPL algorithm [1]. Figure 0b shows this distribution, where all nodes have a part of the matrix to process throughout the whole factorization, except for the very last block. The last block is computed by the last node in an extra pass. For each pass in the global domain, we compute the result of one row of blocks, and one column of blocks. In the following, we refer to these as *block-row* and *block-column* respectively.

To lessen the amount and number of transfers between nodes, we use partial pivoting within in-core memory, thus eliminating the need to transfer rows between processors. It is trivial to create examples where partial pivoting fails, but sufficient accuracy is attainable in practice. This also holds for our pivoting, which pivots in a subset of the regular pivot candidates.

In order to compute one pass in the global domain, we have to execute the four different operations *pivot, normalize, eliminate* and *reduce*. It should be mentioned that this data distribution, and splitting into different operations per node allows for multiple nodes, not only four as shown in this example. In the third pass of this algorithm, we have the following situation (see also Figure 2):

Pivot: The pivot position (process 0) must compute the PLU factorization of the current active pivot block in its local domain. The block size is subsize $\times$ subsize. In addition, it has to reduce the rest of the local matrix according to the computed L and U. These blocks belong elsewhere in the global domain (see Figure 0b). In each global pass, there is always only one pivot node.

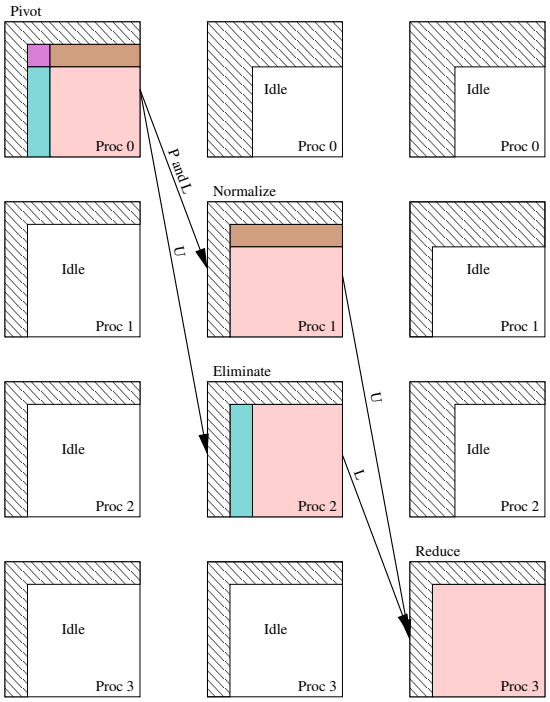

Figure 2: Data send patterns for PLU decomposition using four nodes in the third global pass (corresponds to the situation in Figure 0b). The shaded areas represent the part of the matrix we already have computed.

Normalize: The normalize operation (process 1) needs to compute U according to the P and L computed by the *pivot* operation. It will also have to reduce all remaining elements in the local matrix, which again belong elsewhere in the global domain. There are $s - 1$ nodes that compute the normalize operation in each global pass, where s is the width and height of the processor grid.

Eliminate: Eliminate (process 2) calculates the multipliers needed to forward substitute one block by using the computed U's from *pivot*. In addition, it has to reduce the rest of the local matrix, according to the computed U. In each global pass, the number of eliminate nodes is also $s - 1$.

Reduce: The reduce operation simply reduces the local matrix according to the L and U computed in *eliminate* and *normalize* respectively. All remaining processes compute this operation, $s \times s - 2(s - 1) - 1$ nodes.

As stated in the list of operations, the different processes depend on data from other processes. This dependency is not static, but varies with the operation the current node is set to execute. Figure 2 shows how the data is sent in the already used example. The nodes waiting for data cannot continue before they have received the data. This effectively limits the computational speed to the slowest node. The HPL [1] algorithm uses look-ahead to remedy this somewhat. As this chart shows, there is still quite a lot of idling for the four nodes. The pivot node, for example, computes its result and then waits until all other nodes have completed their computations.

3.2 Local algorithm

The local algorithm includes four stages *pivot, eliminate, normalize* and *reduce*, but first we will introduce the matrix representation. The data is row-wise represented in four-wide vectors [11]. This is to utilize as much computational power and bandwidth as possible, since most GPUs can execute one MAD instruction on four-long vectors per clock cycle. The advantage of this packing scheme is that it does not require restructuring of the data in main memory before it is sent to the GPU[2]. Another reason for this choice is that it fits well with the solution we have for pivoting. In addition to storing the matrix, we add an extra column leftmost in the matrix, as shown in in Figure 2a. This column is used to speed up the calculation of the next pivot element, explained later. Because

[2]Assuming its width is divisible by four.

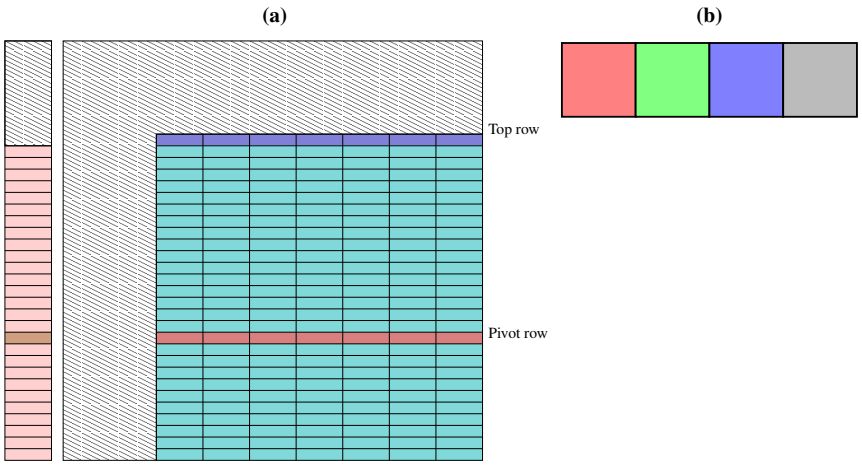

Figure 3: Data representation on the GPU: (a) Row interchange of the multipliers (leftmost column) and the rest of the matrix (cyan part). (b) The leftmost column of the texture, with both the multiplier, and the reduced next column in the PLU factorization. The multiplier is stored in the red color channel, and the reduced next column is stored in the blue color channel.

the result of writing to the same buffer as we read from is explicitly undefined in OpenGL, we have to use an extra texture. The two textures are used as one virtual matrix, but we alternate between reading / writing and writing / reading to the front and back textures, respectively. This technique is referred to as ping-ponging in the field of GPGPU.

3.2.1 Pivot

The pivot procedure computes the PLU factorization of A, but stops when one block-row and one block-column has been computed (see Figure 0b). It can roughly be split into two tasks: multiplier calculation, and reduction, each explained below. To compute a single row and column, we start by permuting the first column simultaneously as we compute the multipliers. Then, we reduce the rest of the matrix, whilst permuting the rows here as well.

To compute one column of multipliers, we read from the correct location in the source texture, and write to the leftmost column in the destination, as shown in Figure 2a. The top element is rendered at the position of the pivot element. Because the multiplier for the top row always is one, we do not need to compute it. In addition to computing the multipliers, we also compute the values of the column to the right of the pivot position and store in one of the other color channels (see Figure 2b).

When the computation is complete, we transfer the multipliers and the reduced next column to the CPU using a pixel buffer object (PBO). The PBO uses asynchronous read-back to the CPU, allowing both the CPU and the GPU to continue execution. When the whole leftmost column has been transferred to the CPU, the next pivot element is found by the CPU. Simultaneously as the data is copied, and the CPU searches for the pivot element, the GPU subtracts the multiplier times the top row throughout the rest of the matrix. The top and pivot row are also interchanged simultaneously in the same manner as in the first column. In addition, we employ the index pair streaming technique to increase performance [7]. When the computation is complete, the top row is copied to the CPU, again using a PBO. The algorithm continues until we have computed the whole block-row of U, and block-column of L.

3.2.2 Normalize

The normalize step computed on the local domain executes as follows: The L matrix from this global time-step's pivot node is uploaded to the GPU as a texture. Then, we execute a for-loop that sequentially computes one row of U at a time. First, the current top row and pivot row are swapped, simultaneously as we eliminate using the multipliers in L. Because we are using two buffers, we read back the pivot row simultaneously using PBO's, and store them in main memory. When all rows in the block-row have been computed, U is sent to all nodes in the same column for the reduction operation.

Listing 2: Setting up row- and column-communicators

```
/* Set up row communicators */
MPI_Cart_sub(origcom, {0, 1}, &rowcom);

/* Set up column communicators */
MPI_Cart_sub(origcom, {1, 0}, &colcom);
```

3.2.3 Eliminate

The elimination procedure calculates multipliers. Normalized rows (U) are sent from the current time-step's pivot node, and the multipliers are calculated using these. The elimination step follows much of the same procedure as the pivot step, but it is a simpler case since there is no complications with row interchanges. This is again because the pivot node only pivots within in-core memory.

3.2.4 Reduce

The reduction step is trivial on the local node. Using a for-loop, we sequentially reduce the whole remaining sub-matrix by looking up one row from U and one column from L, and calculating the reduced A as $A_{i,j} := A_{i,j} - L_{i,k} \cdot U_{k,j}$.

3.2.5 Sending of data

This section describes how data is sent between different nodes. The use of MPI-2 for this inter-node communication will also be explained in detail.

Based on the algorithm discussed in Section 3.1 we have the following communication scenarios:

1. Sending data to all processes in the same row as active process (to *normalize* and *reduce*).
2. Sending data to all processes in the same column as active process (to *eliminate* and *reduce*).

For broadcasting data to all processes in the same row as the active process, the broadcast function in MPI, `MPI_Bcast`, is used. This function takes a communicator, a pointer to the data, and a count of data elements as arguments. When called, it broadcasts the data to all processes within that communicator. Broadcasting data to the same row as yourself is done by calling `MPI_Bcast` with the row communicator.

To broadcast to columns we use the column communicator instead of the row communicator.

Since the `MPI_Bcast` function is collective, it needs to be called in every process within the current communicator. This implies that each process needs to know a priori from which node it will receive the next broadcast. In our application we have a function dedicated to calculate this. This function bases the calculation on which global pass the process is currently in, and which type it currently is (*pivot, normalize, eliminate* or *reduce*). This method is fairly complicated, but can be briefly explained as follows: The *normalize* nodes will always receive a broadcast from the *pivot* node, which is the diagonal element in its row communicator. *Eliminate* is similar, but will receive from the diagonal element in its column communicator. Finally, *reduce* will receive data from *normalize*, which is the node with the same column index as the current node, and the same row index as the current *pivot* node. *Reduce* also receives data from *eliminate*, which is computed in a similar fashion.

To facilitate the communication needed by our algorithm, row- and column-wise communicators are set up. Listing 2 shows the code used to create these communicators. In this listing, the array sent as the second parameter sets which dimension we wish to keep in the new communicators. When we create the row communicators we keep the y-dimension intact, and when creating the column communicators we do the opposite and keep the x-dimension. When the code is executed, each process will set up a row communicator called `rowcom` and a column communicator called `colcom` relative to the process' location in the grid.

4 Results

The cluster which we benchmarked our application on consists of four one-CPU, one-GPU nodes as shown in Figure 4. The nodes were all equipped were Intel Pentium 4 processors with Hyper-Threading Technology (HTT) and 2 GB of RAM. All nodes had an NVIDIA GeForce 7800 GT graphics adapter on a PCI-Express $16\times$ slot.

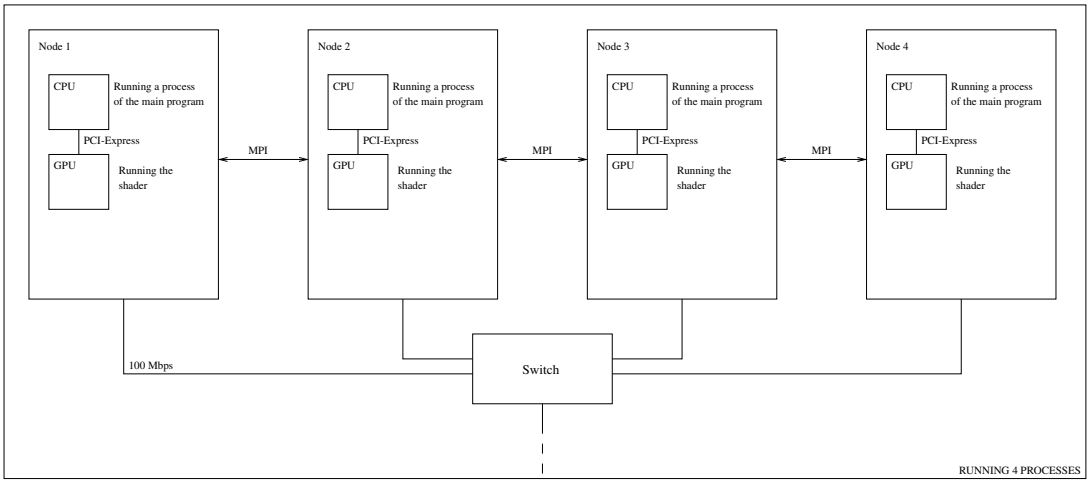

Figure 4: Overview of physical setup of nodes.

4.1 Benchmark

Benchmarking of our algorithms showed that it gives sufficiently accurate results considering that all computation is executed on single precision hardware.

When benchmarking the algorithm, we have varied several variables to identify possible bottlenecks. The variables we have varied are:

1. Number of nodes.
2. Number of processes.
3. The size of the block to factorize in each global pass (subsize).
4. The total size of the problem matrix (n).

In addition, we have benchmarked the pivot operation on a single node executed on the full matrix, as well as only network communication. This gives us performance results for our network setup, the local algorithm, as well as the global algorithm, enabling analysis of the limiting factor.

Table 1 shows the time used to compute the PLU factorization while varying the number of nodes, size of the matrix, and the block size. The maximum achieved performance is 3.5 GFLOPS (for $n = 4096$ on two nodes), and the general trend seems to suggest that using only two nodes is faster than using four. This can somewhat be explained by interprocess communication being faster with two processes per node, than one process per node, as this eliminates a lot of network communication.

Table 2 shows the time used to compute the PLU factorization while varying the number of processes on four nodes. As the table shows, the speed of the algorithm can be greatly influenced by tuning this parameter. However, the optimal number of processes seems to vary with the size of the matrix. The maximum achieved performance achieved was now increased to 4.2 GFLOPS (16 processes on four nodes). We also timed the network-communication, and measured the percentage of the total time used for network communication. The percentages show that there is a substantial time used to send and receive data alone.

To analyze the impact of the network, we ran the network communication while varying the number of nodes. Table 3 shows the time of the network communication, and the impact of the subsize parameter, as well as the use of multiple nodes. The subsize parameter seems to have little effect on the time, whilst the number of nodes has a massive impact. Using two nodes with four processes is approximately half as expensive as using four nodes.

Finally, we have benchmarked the pivot operation on one node. This is the most computationally heavy operation, and a limiting factor. Table 4 shows the time spent to compute a full matrix using the pivot operation. The peak performance was measured for the largest matrix, 4096×4096, where the algorithm performed 4.6 GFLOPS. As a comparison, we timed the ATLAS implementation used in MATLAB, which achieved 3.5 GFLOPS on the same problem size.

Table 1: Variation of the subsize parameter, as well as the impact of several nodes. The number of processes is 4, and the times are in seconds.

-		Nodes			
n	Subsize	1	2	3	4
128	8	0,20607	0,14006	0,13756	0,28482
	16	0,23247	0,11918	0,14593	0,28739
	32	0,19208	0,10213	0,11030	0,27609
	64	0,13572	0,09238	0,08232	0,24506
512	32	0,54457	0,25811	0,28454	1,11726
	64	0,49388	0,24161	0,26360	0,78500
	128	0,32307	0,23648	0,24194	0,64518
	256	0,24012	0,20242	0,21688	0,36138
2048	128	3,17257	2,43952	2,95311	3,19620
	256	3,07729	2,43028	2,95513	3,15248
	512	2,88925	2,41467	2,87859	3,05907
	1024	2,59612	2,39955	2,68849	2,93310
4096	256	13,76410	13,03520	14,77890	15,26550
	512	13,70820	13,18710	14,74090	15,78910
	1024	13,59550	13,59640	14,73430	16,26440
	2048	14,62520	14,45370	14,50600	16,66760

Table 2: Variation of the number of processes. The number of nodes is four, and the times are in seconds.

Procs	Subsize	Time	Network time %
4	256	97,78950	42
	512	98,19350	36
	1024	99,65560	31
	2048	102,98800	30
16	256	86,30310	37
	512	88,69330	35
	1024	89,53110	35
	2048	95,1656	33
64	128	122,72900	24
	256	124,48000	23
	512	120,79000	23
	1024	124,32000	21

Table 3: The time spent transmitting data. The number of processes is four and the problem size is 2048, while the number of nodes is varied. This shows the impact of the network communication.

-	Nodes			
Subsize	1	2	3	4
128	0,57228	3,18597	5,70082	6,30781
256	0,59500	3,14385	5,72201	5,73072
512	0,62175	3,13140	5,64543	5,65009
1024	0,69741	3,02938	5,37086	5,38025

Table 4: The time spent computing using only a single node where subsize = n. The times are in seconds.

n	Time
64	0,0284489
256	0,0491337
1024	0,280545
2048	1,44955
4096	10,051

4.2 Analysis

Our global algorithm had a maximum measured performance of 4.2 GFLOPS using four nodes, while our local algorithm showed a promising 4.6 GFLOPS. The network communication could account for at least 20% of the total runtime. However, because of the way the presented algorithm is executed, most of the processes simply idle, waiting for data. This is the largest bottleneck, but there are some solutions.

Using a look-ahead strategy, as used in the HPL [1] algorithm, will increase the workload per node, and decrease the idling. In addition, restructuring the computation into smaller parts, so that pivot, eliminate, normalize and reduce are split into smaller subproblems, will also decrease the time spent idling per node.

We have not been able to show the full potential of this algorithm, because we have only have had four nodes at disposal. Having only four nodes makes almost all the computation execute serially, because we only have one node per operation at each global time-step. This parallelizes the computation of normalize and eliminate only. Using more nodes, will parallelize the reduction step of the algorithm as well, and probably speed up the total computational speed.

5 Conclusions and further research

We have presented a new way of computing the PLU factorization of a matrix, by using the GPU on a cluster of nodes. We have shown that the algorithms computed locally are efficient, even outperforming ATLAS. Our global algorithm, however, is less efficient. We have pointed to a slow network link, a lot of idling of nodes, and the use of only four nodes as the main reasons.

A faster network link will decrease the impact of the network communication in our algorithm. It is also possible to lessen the issue with idling of nodes by using techniques such as look-ahead, or splitting up the computation further.

It is possible to extend our algorithm to include forward and backward substitution, as the HPL algorithm does. The computation of the forward substitution will be virtually free, while the backward substitution will require more global passes. Including the forward and backward substitution in the algorithm will fulfill the complexity demands for the Top500 benchmark [2].

6 Acknowledgements

We would like to thank J. Hjelmervik for first proposing this project to us, and our supervisors K.-A. Lie and T. R. Hagen for their helpful guiding and notes on our white paper. We would also like to thank G. W. Ma for all his assistance, and our fellow master students at SINTEF ICT for insightful discussions.

References

[1] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. Hpl. http://www.netlib.org/benchmark/hpl/. [accessed 2007-04-18].

[2] University of Mannheim, University of Tennessee, and NERSC/LBNL. Top 500 supercomputer sites. http://www.top500.org/about. [accessed 2007-04-18].

[3] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[4] Neoptica. Programmable graphics – the future of interactive rendering. Online; http://www.neoptica.com/NeopticaWhitepaper.pdf, 2007. [accessed 2007-04-23].

[5] Khronos Group. OpenGL – the industry's foundation for high performance graphics. Online; http://www.opengl.org, 2007. [accessed 2007-04-25].

[6] Microsoft Corporation. Microsoft DirectX. Online; http://www.microsoft.com/directx, 2007. [accessed 2007-04-25].

[7] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 3, Washington, DC, USA, 2005. IEEE Computer Society.

[8] MPI Forum. Mpi documents. Online; http://www.mpi-forum.org/docs/docs.html. [accessed 2007-04-18].

[9] Argonne National Laboratory. Mpich2. Online; http://www-unix.mcs.anl.gov/mpi/mpich2/. [accessed 2007-04-18].

[10] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.

[11] A. Moravánszky. Dense matrix algebra on the GPU. Online; http://www.shaderx2.com/shaderx.pdf, 2003. [accessed 2006-05-11].