Must Linear Algebra be Block Cyclic? and Other Explorations into the Expressivity of Data Parallel and Task Parallel Languages

by

Harish Peruvamba Sundaresh

Submitted to the School of Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Computation for Design and Optimization

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2007

Author...
School of Engineering
May 10, 2007

Certified by.. ....
Alan Edelman
Professor of Mathematics
Thesis Supervisor

Accepted by...
[illegible]
Jaime Peraire
Professor of Aeronautics and Astronautics
Co director, Computation for Design and Optimization Program

Must Linear Algebra be Block Cyclic? and Other Explorations into the Expressivity of Data Parallel and Task Parallel Languages

by

Harish Peruvamba Sundaresh

Submitted to the School of Engineering
on May 10, 2007, in partial fulfillment of the
requirements for the degree of
Master of Science in Computation for Design and Optimization

Abstract

Prevailing Parallel Linear Algebra software block cyclically distributes data across its processors for good load balancing and communication between its nodes. The block cyclic distribution schema characterized by cyclic order allocation of row and column data blocks followed by consecutive elimination is widely used in scientific computing and is the default approach in ScaLAPACK. The fact that we are not familiar with any software outside of linear algebra that has considered cyclic distributions for their execution presents incompatibility. This calls for possible change in approach as advanced computing platforms like Star-P are emerging allowing for interoperability of algorithms.

This work demonstrates a data parallel column block cyclic elimination technique for LU and QR factorization. This technique yields good load balance and communication between nodes, and also eliminates superfluous overheads. The algorithms are implemented with consecutive allocation and cyclic elimination using the high level platform, Star-P. Block update tenders extensive performance enhancement making use of Basic Linear Algebra Subroutine-3 for delivering tremendous speedup.

This project also provides an overview of threading in parallel systems through implementation of important task parallel algorithms: prefix, hexadecimal Pi digits and Monte-Carlo simulation.

Thesis Supervisor: Alan Edelman
Title: Professor, Department of Mathematics

Acknowledgements

I would like to thank and express my gratitude to a number of individuals for their contribution to this thesis and my life at MIT.

First and foremost, I would like to thank my advisor, Alan Edelman, for the hours of guidance, motivation and constant encouragement during my stay at MIT. He inspired me and made me feel like family. It is also really special that Alan is going to be my brother's supervisor. It has been an amazing experience working under his supervision and I will never ever forget his contribution in this very important juncture of my life.

I would like to thank the CDO program at MIT for giving me this wonderful opportunity to work with revered faculty and providing me the expertise to work on this Thesis.

I would like to express my gratitude to Sudarshan at ISC for being extremely friendly and answering all my queries with regards to using Star-P. I would like to thank DOE Petascale Application Development grant to Alan Edelman for their financial support.

I would also like to thank my friends Ram, Sandeep and Varun whose friendship I will value forever. I would like to thank Ashish, Ganesh, Lavana, Mahesh, Sudhir and Viswajith with whom I have had some wonderful and fun-filled experiences. I want to thank my friends Ajay and Prashanth for being the best buddies ever.

I owe profound thanks to my parents, Latha and Sundaresh, my brother, Vignesh and my darling and cutest kid sister, Pooja for their undeterred support and inspiration, and constantly being there for me in times of paramount stress and spurring me on to achieve greater things in life. I am extremely grateful to them for being the best family ever and I owe them more than I can express in words.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallel and high end computing [12] have been motivated by complex simulations taking severely long execution times. It is the natural evolution of serial computing that attempts concurrent computation of interrelated events within a sequence. Parallel computing draws upon additional resources to optimize time as a commodity.

1.1 Understanding a Parallel Computer

Interpreted physically, a parallel computer [12] is the synchronized use of multiple compute nodes to achieve faster solution to a problem. A serial program is broken into discrete parts that are solved concurrently or parallel on different CPU's. Each processor is assigned data and a set of instructions at the same time, so that they all execute simultaneously without substantial delay by any single processor. This task is performed by the master processor and is achieved in part by adhering to good standards of load balancing of data across all the processors.

Parallel Computing thus has its advantages:

1. speedup based ideally on the number of compute nodes with a goal of highly efficient mode of scalability.
2. often there is additional space available on a parallel computer eliminating memory constraints.

1.1.1 Memory Management in a Parallel Computer

Memory and CPU speed are the two most important physical attributes of computer. Whilst processor speed is of utmost significance in computation, memory is exploited for handling large magnitudes of data. Shared and distributed memory architectures are the two types of parallel computer memory architectures. Shared memory parallel computers have a global addressed space. Multiple computers can operate independently and share the same memory being informed of all changes being made in the memory by other machines. This concept of global address space provides a user-friendly programming environment to memory. Data sharing between processors is fast and efficient. The most significant disadvantage of this mode has often been the lack of scalability between memory and CPU's.

Distributed memory systems need a communication system to connect inter-processor memory. Each processor in the network has its' own memory and there is no concept of global addressing. The processor along with its memory operates independently and thus needs detailed instructions from the user on how to access information on the memory of another processor. Like shared memory, the distributed memory system also has its own advantages and disadvantages. The memory increases with the number of processors added to the network, hence memory is scalable. Each processor's memory acts like a cache without any interference from any other node. The main disadvantage of this system is that it holds the programmer responsible for communication and distribution of data and instructions - synchronization is the duty of the programmer and also the unavailability of a global address structure makes mapping of data between processors quite difficult.

1.1.2 Thread Configurations in a Parallel Computer – Understanding Task/Data Parallelism

It is very important to understand the nature of the algorithm the user is trying to program with a parallel computer. There are inherently two types of parallel models that can fit the algorithm:

1.1.2.1 Data Parallelism

Data parallelism implies that the same independent operation is applied repeatedly to different data. It has been generalized perhaps incorrectly to convey the notion of global array processing. Image processing algorithms that apply a filter to each pixel are a common example of data parallelism. It involves spreading matrix data over all compute nodes in the machine and having each processor manipulate a portion of the data. Each processor usually executes a small number of instructions between communication cycles and then communicates the results to the other processors before the next operation.

1.1.2.2 Task Parallelism

The task-parallel model is chosen when independent threads can readily service separate functions. Task-level concurrency calls for independent work encapsulated in functions to be mapped to individual threads, which can execute asynchronously. This model can perform longer, non-communicating computations on independent nodes. Task parallel operations can execute on separate threads on a serial computer.

One must consider the following as a "rule of thumb" to decide whether the problem is data/task parallel based on the matrix size used:

1. If the matrix size is less than 100kB parallel computing need not be used.
2. If the matrix size is between 100kB and 100MB either data parallel or task parallel can be used depending on the characteristic nature of the algorithm.
3. If the matrix size is over 100MB data parallel would be the best way to do it.

1.2 Motivation for the current problem

Each processor in a distributed memory parallel computer is pre-allocated with data and a set of instructions at the same time for simultaneous execution and delivery of results. This is possible only if all the processors are distributed with equivalent and balanced data load. Thus, data distribution with good load balancing schemas is a core attribute in parallel computing systems. This distribution plays a key role in determining the load on each processor, mapping algorithms, blocking factor and communication between nodes.

Several ways have been identified in which data can be physically allocated across processors, of which the blocked, cyclic and the block cyclic versions find extensive use in computing systems. Some of the schemas are reviewed below.

1.2.1 Data distribution in a Parallel Computer

The column block cyclic algorithm [4] allocates cyclically, column blocks to nodes followed by consecutive column wise elimination to allow good load balance among the processors. This alleviates the problem confronted by various other data layouts described below.

The 1D block column distribution assigns a block of adjacent columns to consecutive processors, thus each processor is allocated only a single block during the factorization of the matrix. Column k is stored on process $[\frac{k}{t}]$ where $t = \left\lceil\frac{N}{P}\right\rceil$, the maximum number of columns stored per process. This layout performs badly for Gaussian Elimination because the idle time associated with each processor is very high, resulting in bad load balance amongst the nodes.

The 1D cyclic column distribution allots a single column to each processor in a cyclic manner assigning column k to process (k-1) mod P if the processors are numbered starting with zero. This method can realize good load balance, but since single columns are pivoted the performance of BLAS (basic linear algebra subroutine) related speedup is affected terribly.

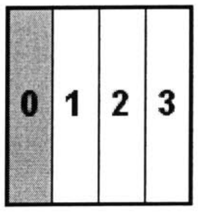

Figure 1-1: Column Blocked Data Layout

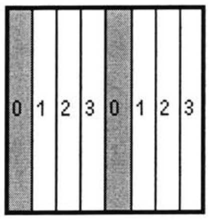

Figure 1-2: Cyclic Column Data Layout

The 1D block cyclic column distribution is a combination of the advantages of the two methods discussed above. Block of data from the array is distributed evenly over all nodes. A block of size *NB* is picked, the columns are resized into groups of columns of size NB, and then the groups are distributed in a cyclic manner. This means column k is stored in process $\left\lceil\frac{k-1}{NB}\right\rceil mod\ P$. Thus, in block cyclic allocation, matrix columns are consumed cyclically, uniformly and evenly and thereby relieving the problem of unbalanced load. Blocking permits the use of BLAS-3 operations to achieve significant speed-up in computation.

For *NB* larger than 1, this has bad balance than the one-dimensional cyclic column distribution, but can use higher levels of basic linear algebra subroutines for computations. For *NB* less than *t*, it has a better load balance than the one-dimensional block column distribution.

In [2] Johnsson showed that the processor utilization increased by a factor of three in block cyclic distribution than consecutive allocation and consecutive elimination. The block cyclic data distribution used by ScaLAPACK is justified in part by the analysis of many algorithms intended for linear algebra calculations and by the goal of achieving good load balancing. ScaLAPACK uses the routine PDGETRF to perform the LU factorization and the DGEQRF to perform its QR factorization. Coleman and Van De Geijn also used consecutive elimination and cyclic allocation for their multiprocessor and LU factorization implementations respectively.

1.2.2 Problem Introduction

Today, it is widely viewed that parallel dense linear algebra is somehow synonymous with block cyclic data distributions. The widely used ScaLAPACK software [4] has made this choice based in part on analysis [5]-[7] and also in part by history [8]-[9]. However it is time to re-examine the very importance of either pre-allocating or re-allocating dense matrices in block cyclic format. This algorithm not only employs exhaustive mapping and remapping techniques to track data in the processors and redistribution algorithms to adjust blocking factor, the fact that we are not familiar with any software other than linear algebra software that has considered cyclic distributions for their execution presents uncertainty on the scalability of block cyclic distribution to non-linear algebra. It also causes a situation of poor and rather complex communication exchange between nodes.

This question is also timely as high level general purpose high performance computing platforms such as Star-P by Interactive Supercomputing [10], are maturing. Such systems provide solutions to a vast space of parallel problems - yet only dense linear algebra has been screaming out for block cyclic distributions - only dense linear algebra. One can then wonder if dense linear algebra really needs block cyclic data distributions as well.

Lichtenstein and Johnsson [1] showed that block cyclic order elimination can be achieved for LU and QR factorization on their Connection Machine System. They used a cyclic elimination scheme as the Connection Machine compilers could by default only accept consecutive allocation or hypercube. This was shown to be an efficient and optimal alternative to cyclic allocation and elimination, yielding good load balance for solution of linear systems with dense equations. These authors implemented the cyclic elimination scheme using Level1 and Level2 BLAS using FORTRAN and also hinted that a BLAS-3 version is possible.

This work demonstrates block cyclic elimination for LU and QR factorization for two-dimensional arrays using Star-P. A description of the cyclic elimination method is provided along with relevant performance indices. Performance improvements with problem and machine size are also illustrated. Blocking and BLAS-3 operations present a vast performance enhancement measure.

1.3 Hardware and Software Configurations

Parry Husbands, Alan Edelman and Charles Isbell [11] devised a novel architecture for a linear algebra server that operates in parallel on extremely large matrices. In this implementation matrices are created by the server and distributed across many machines, thereby ensuring that all operations take place in parallel. Star-P is a high level language parallel simulation interface to MATLAB® and Python developed by Interactive Supercomputing whose infrastructure is based on the concept described above.

The Star-P computing platform allows desktop simulation software to operate on High Performance Computers. The Star-P client acts as a conduit between MATLAB® and Python on the desktop and server computer thereby providing the framework to operate with massive amounts of data.

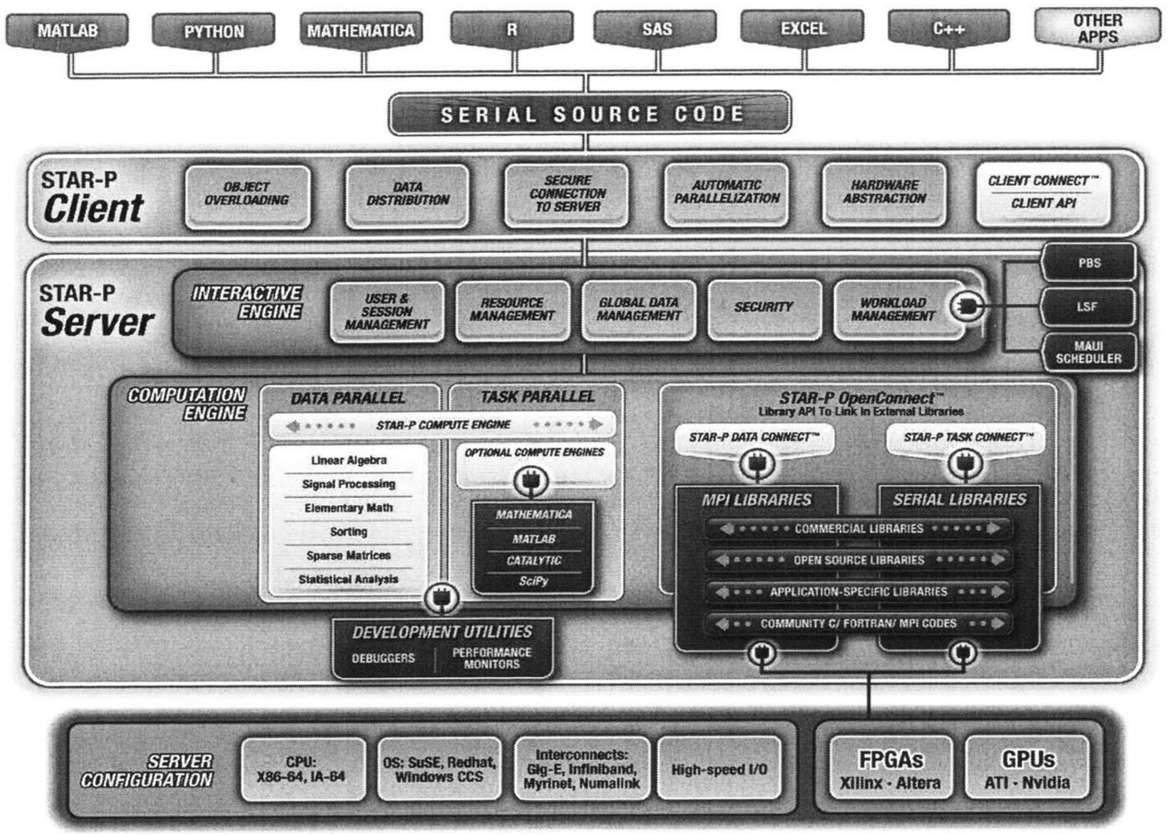

Figure 1-3: STAR-P Functionality overview diagram

The functional overview diagram of STAR-P is illustrated above. The client connects the desktop application to the parallel server, outsources to it the most computationally-intensive operations and controls loading of large data sets from high-speed parallel disks directly into the servers' distributed memory. The engine runs on top of the server operating system, and manages the multiple interactive sessions in a multi-user environment, giving multiple client applications simultaneous interactive access to the server's processors, memory, and file system. The computation engine performs the three key operations – data parallel, task parallel computations and a link to the Open Connect Library API.

The software was installed on an SGI Altix system at the Massachusetts Institute of Technology consisting of six nodes, each with two 1.3GHz Itanium 2's and 2GB of RAM, for a total of 12 processors and 12GB of RAM.

Chapter 2

Description of the Method

Block cyclic elimination is a significantly advantageous alternative to the widely used block cyclic data allocation technique. A brief summary of the original procedure is provided, followed by the description of the technique. The disadvantage and advantage of using both methods is also discussed.

2.1 Block Cyclic Distribution

The following conventions are used interchangeably in the chapters that follow. Block cyclic in time refers to consecutive allocation of data followed by cyclic elimination and block cyclic in space refers to cyclic allocation of data and consecutive elimination. The block cyclic distribution algorithm is illustrated next in section 2.1.1.

2.1.1 Block Cyclic Distribution for LU and QR Factorization

The LU factorization algorithm traverses through the diagonal of a matrix, subtracting a multiple of a row, the pivot row, from the remaining rows. In QR, a normalized linear combination of columns is subtracted from each of them to make them orthonormal to each other. In both the algorithms a matrix A is factored such that $A = LU$ in LU factorization and $A = QR$ in QR factorization.

When the matrix is factorized with a consecutive allocation and elimination scheme, after first block of (N/p) columns (N is the dimension of the matrix and p is the number of nodes) have been selected and factorized, the first node would have completed its assigned work for the cycle before the other nodes and this will aggregate its idle time.

The cyclic data allocation scheme is well-designed for load balancing. In the cyclic data allocation scheme the first column of the matrix is assigned to the first node, the second column of the matrix to the second node, and so on until each matrix column has been allotted to a node for processing. Then, the (p+1)th column of the matrix is assigned to the first node. Thus the cyclic distribution method ensures uniform and even load across processors minimizing to a great extent the idle time associated with each processor but its operations are not BLAS-3 vectorized.

The block cyclic in space algorithm proceeds through configuration and distributive statements that illustrates the mapping between the matrix and the processors. The matrix is first assigned to templates, and templates are distributed across the processors.

Consider a one dimensional matrix or array A of order $1\ x\ N$ to be mapped across p processors in a block cyclic convention. The elements in the array and the processors are numbered 0 through N, thus the element $A(i)$ is assigned to processor $\left[\frac{i}{k}\right] mod\ p$ where k is the block size assigned to each processor, [] represents the greatest integer function and p is the number of processes used.

Let Row(i), Proc(i) and Loc(i) [22] describe the location of matrix element A(i) in the processor. Let Proc(i) denote the processor holding the element, Row(i) is the row of blocks within processor Proc(i) holding the element A(i) and Loc(i) is the location of A(i) within the row of block Row(i). This layout is depicted by Figure 2-1. The functions are described below:

$$Proc(i) = \left[\frac{i}{k}\right] mod\, p$$

$$Row(i) = [i\, mod\, pk]$$

$$Loc(i) = \ i\, mod\, k$$

Hence the mapping between the matrix element A(i) and processor functions Proc, Row and Loc is given by

$$i = pk.Row(i) + \ k.Proc(i) + \ Loc(i)$$

The local location of elements within each processor is given by

$$M(i) = k.Row(i) + \ Loc(i)$$

Rows	Processor 0			Processor 1			...	Processor p-1		
0	0	...	k-1	k	...	2k-1	...	(p-1)k	...	pk-1
1	pk	...	(p+1)k-1	(p+1)k	...	(p+2)k-1	...	(2p-1)k	...	2pk-1
.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	...	.	.	.
Offset	0	...	k-1	0	...	k-1	...	0	...	k-1

Figure 2-1: Block cyclic data distribution. Each region is a memory cell and the number in the cell is the array element it holds.

This mapping procedure can be easily extended to multi-dimensional matrices. After selecting the processors to be used, the user's data is mapped (block cyclic format) onto the processors, the parallel algorithm is then executed, the data then has to be reverse mapped and then the solution is returned to the user.

Thus for referencing each array element which has been mapped using block cyclic distribution onto a processor, separate computations of the processor number Proc, row number Row and local address M are required. Similar computations need to be performed for reverse mapping also. These computations carry unacceptable floating point operation cost and exponentially increase with size of data.

Processor 0			Processor 1			Processor 2		
0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35

Figure 2-2: Block cyclic mapping of data from array of order 36x1 to processors with p = 3, k = 3.

The positioning of an array to a template and the distribution of the template across the processes determines the final mapping of the array. An algorithmic blocking factor is used to align an array into a template (adjust granularity of the subtasks to maximize the efficiency of the hardware resources) and a distribution blocking factor is used to distribute the template across the processors. Most parallel linear algebra codes assume the factors to be identical but the optimal values of the blocking factors are usually different. For this purpose redistribution and scheduling algorithms are necessary. This is yet another overhead. Thus block cyclic in space may come off as complex and as involving unnecessary computational overheads involving mapping, remapping, scheduling and redistribution.

2.1.2 Disadvantages of Block Cyclic Distribution

Block cyclic distribution can be categorized as an algorithm with disputable computational overheads. These demerits are summarized below.

1. Requires mapping and remapping between the processor and the matrix.
2. Associated mainly with Linear Algebra software; hence impact of interaction with non linear algebra algorithms is not obvious.
3. Redistribution algorithms are necessary for good load balancing.
4. Poor communication between nodes.

Whilst the cyclic data distribution method actually permutes data itself across the processors the cyclic elimination method uses smart indexing to access the data in-place. While it is factual that the implementation of such a schema is complex and depends heavily on indexing (subsref) its advantages certainly offset this shortcoming. Block cyclic elimination is elucidated in section 2.3.

2.2 Blocking and its impact on performance

Memory is one of the most important attributes in supercomputing platforms [1]. There are numerous floating point computations in linear algebra and they mostly occur as add-multiply configurations. The fact that BLAS is used to execute several operations in the solution of linear systems of equations, Eigen analysis, optimization and the solution of partial differential equations is testimonial to this. These require large memory bandwidth for large computations; hence BLAS-2 and BLAS-3 levels have gained prominence for better performance.

The BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. The BLAS functionality is divided into three levels: 1, 2 and 3 based on their operations. The Level 1 BLAS [4] carries out scalar, vector and vector-vector operations, the Level 2 BLAS does matrix-vector operations, and the Level 3 BLAS perform matrix-matrix operations.

The Level 3 BLAS are a highly optimized set of BLAS instructions used for executing matrix multiplication. This level contains matrix-matrix operations of the form $C \leftarrow \alpha AB + \beta C$ and allows for a significant reduction in memory. Blocking helps to achieve matrix-matrix operations as a consequence of which the calculations are performed at Level 3 BLAS. Level 3 BLAS results in remarkable speedups in comparison to Level 1 and Level 2 and this is described later in this section.

Level 1 and Level 2 BLAS are described with an example. In Gaussian Elimination, the column elements below the diagonal are zeroed out to convert the original matrix into an upper triangular matrix by adding a multiple of the current pivot row 'I' to row 'j'. The obvious candidate for the row update is BLAS-2 vector matrix multiplication as shown below.

```
$\boldsymbol{for\ 1 = 1{:}n - 1}$
        $A(i+1{:}n, i) = \frac{A(i+1{:}n, i)}{A(i, i)}$                ....$Blas1\ Update$
        $A(i+1{:}n, i+1{:}n) = A(i+1{:}n, i+1{:}n) - A(i+1{:}n, i) * A(i, i+1{:}n)$
                                                         ....$Blas2\ Update$
```

Figure 2-3: An example describing Level 1 and Level 2 BLAS.

A Blas-1 vector update is performed to obtain the multipliers for each row corresponding to the pivot row 'I'. The multiplier vector is then multiplied with elements of the pivot row 'I' before subtracting them from subsequent rows 'j' to give zeros below the diagonal. This is a Blas-2 update. Implementation procedure with Blas 3 update is discussed later.

A simple matrix product test was used for the comparison of execution rate between the linear algebra subroutines in MATLAB®. The number of floating point operations in the product of 2 matrices of order $(\mathrm{n \times n})$ is $2\mathrm{n}^3 - \mathrm{n}^2$ contributed by n^3 floating point multiplications and $(\mathrm{n}-1)\mathrm{n}^2$ number of additions.

Figure 2-4 describes the execution rate of matrix multiplication in 10^8 Flops for Blas-3 and Blas-2 formulations. It is obvious that Blas-3 performs notably superior than Blas-2 and is at least 150 MFlops faster than Blas-2 for all orders of matrices. Hence there is an obvious need to try and extend Gaussian Elimination to use Blas-3 matrix-matrix operations as much as possible.

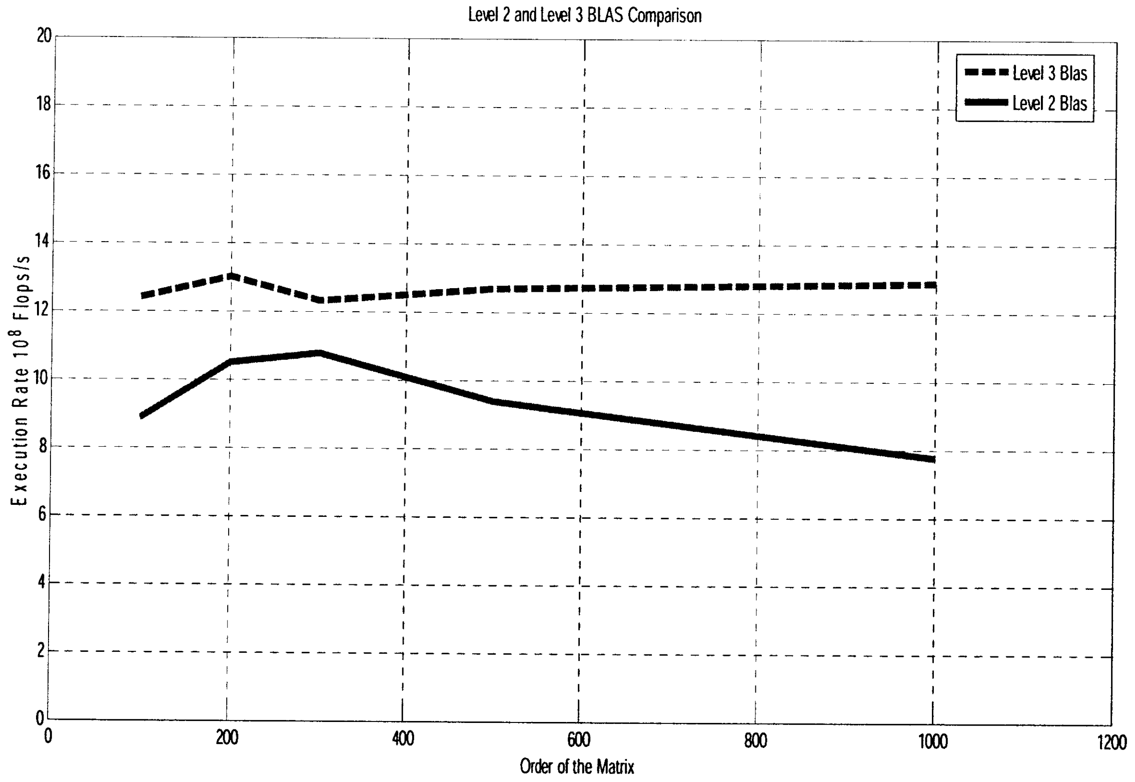

Figure 2-4: Blas 2 vs. Blas 3 performance comparison for Matrices of different Order

The most important characteristic and necessity in Blas-3 operations is to find the appropriate size of column blocks which would allow us to perform matrix-matrix operations optimally. The block size is extremely critical as it must be small enough so that the active sub-matrix consisting of a column block of the original matrix fits in the cache memory and at the same time be large enough to make BLAS3 fast. This can be performed by delayed updating procedure. This involves saving updates from several consecutive Blas-2 operations and applying the updates simultaneously in one Blas-3 operation. This procedure is described in section 2.3.2.

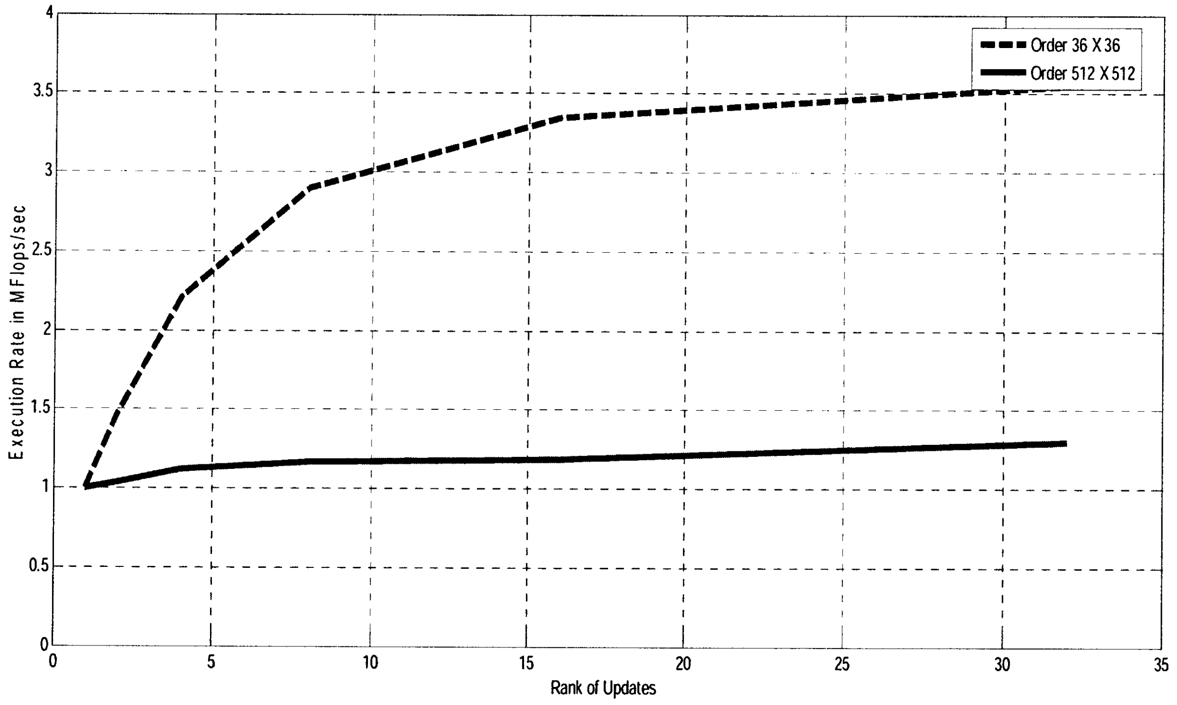

Figure 2-5: Speedup of higher rank updates over rank 1 update for small and large order matrix

Figure 2-5 shows the speedup and performance gain of Higher Rank Updates relative to Rank 1 updates. The speedups are shown for matrices of size 36 x 36 and 512 x 512. It can be seen that blocking yields significantly better performance for smaller matrices when compared to larger matrices. Higher the order of the update higher the execution rate achieved.

2.3 Introduction to Block Cyclic Elimination

Until recently there was no really effective high level distributed computing platform in a high level language. Distributed computing was by and large synonymous with low level languages like C and FORTRAN combined with MPI. But, with increased and extensive use of high level languages like MATLAB® or Python and STARP in academia, industry and research whose functionalities are not restricted to dense linear algebra alone the use of a spatial block cyclic algorithm needs to be rethought.

According to Johnson [3] the space (columns) and time (pivot selection) dimensions are interchangeable. This means that block cyclic elimination will achieve the exact same results as that of block cyclic allocation and will offer enhanced performance in comparison to it. I have consequently chosen this approach for implementation.

In the earlier section we saw that data was mapped to each of the processes in a block cyclic distributive method and eliminated continuously. The computational overheads originating from block cyclic distribution were also studied. In this technique instead of distributing columns cyclically over the nodal array, pivot columns can be selected cyclically by smart indexing in the factorization code. Blocked columns are uniformly distributed across each of the processors and are followed by cyclical elimination using an in-place factorization schema. The cyclic elimination is also shown to yield good load balance for solution of systems with dense equations.

This work explores the implementation of the cyclic elimination algorithm in Star-P and presents a very niche message to the high level language parallel computing world. The block cyclic elimination procedure is described next.

2.3.1 Block Cyclic Elimination Procedure

Consider any matrix ‘A’ of the order (m X n) where ‘m’ represents the number of rows and ‘n’, represents the number of columns of the matrix. Let Matrix ‘A’ be divided into ‘t=n/q’ (q>t) number of sub-matrices or bricks each with number of columns ‘q’ and rows ‘m’. Let ‘b’, (b<q and b, factor of q) represent the number of contiguous columns processed by a node at each time. Thus (q/b) will represent the number of blocks ‘h’ in each sub-matrix or brick and n/q will represent the number of bricks.

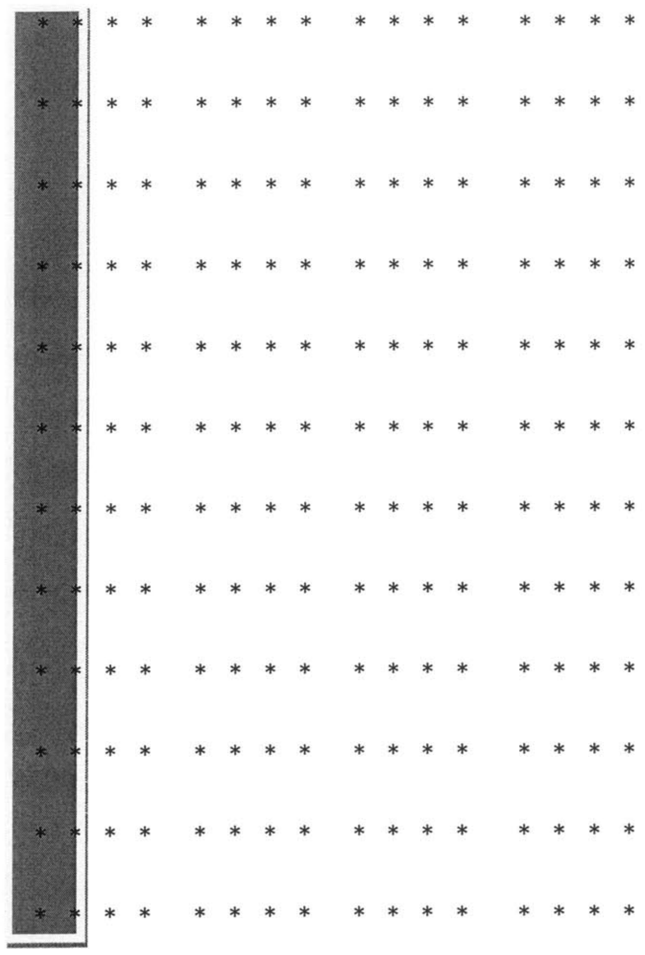

Figure 2-6: Partitioning of the domain into blocks and bricks. A typical MxN matrix with block size of 2 and brick size 4

If 'A' is a matrix of order 16 x 16 and if 'A' is split into 2 sub-matrices of order 16 x 8 or 4 sub-matrices of order 16 x 4, then 'q' in each of the cases would be 8 and 4, 't' will be 2 and 4 respectively. In block cyclic elimination order (Fig 2.7) by columns, a block of columns corresponding to each brick is distributed in a consecutive manner to the nodes and then cyclically eliminated according to block size i.e. block columns of size 'b' are pivoted and factorized. For block size 'b' first eliminate columns 1...b then 'b' columns of the next brick and so on to ultimately obtain a triangular matrix which is easy to solve.

In LU after a block of columns are pivoted, the multipliers for the corresponding rows need to be calculated and stored below the diagonal entry of that row. An LU factorization is performed on the sub-matrix with pivoted columns to obtain the multipliers for each row. Partial Pivoting is performed to make the diagonal entry the largest entry in the column. After this step a level 3 BLAS update is performed on the matrix consisting of all the non-factorized diagonal entries to the right and left of the pivoted block. BLAS 3 update is described in the next section.

In QR the original matrix A is factorized into an orthogonal matrix Q and an upper triangular matrix R. A block of columns are pivoted and normalized. The dot product or the projection of the normalized column along other columns in matrix A is calculated and stored in matrix R. The remainder of the columns in matrix A are made orthogonal to the pivoted columns by means of the Gram-Schmitt procedure and the algorithm proceeds until all orthogonal projections are calculated and each column in the matrix Q is orthogonal to one another.

From Fig 2-7 it can be seen that the outcome of the factorization is a block-cyclic triangle which can be easily restructured to obtain solution of dense linear system of equations. The demanding development time and optimization of the block cyclic elimination order code is offset by its recompense.

Example 1: A = 8 x 8 array p=q = 4, b = 2

Step 1

1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	*	*	*	*	*	*
1	1	*	*	*	*	*	*
1	1	*	*	*	*	*	*
1	1	*	*	*	*	*	*
1	1	*	*	*	*	*	*
1	1	*	*	*	*	*	*

Step 2

*	*	*	*	*	*	*	*
0	*	*	*	*	*	*	*
0	0	2	2	2	2	2	2
0	0	2	2	2	2	2	2
0	0	*	*	2	2	*	*
0	0	*	*	2	2	*	*
0	0	*	*	2	2	*	*
0	0	*	*	2	2	*	*

Step 3

*	*	*	*	*	*	*	*
0	*	*	*	*	*	*	*
0	0	*	*	*	*	*	*
0	0	*	*	0	*	*	*
0	0	3	3	0	0	3	3
0	0	3	3	0	0	3	3
0	0	3	3	0	0	*	*
0	0	3	3	0	0	*	*

Step 4

*	*	*	*	*	*	*	*
0	*	*	*	*	*	*	*
0	0	*	*	*	*	*	*
0	0	*	*	0	*	*	*
0	0	*	*	0	0	*	*
0	0	0	*	0	0	*	*
0	0	0	0	0	0	4	4
0	0	0	0	0	0	4	4

Step 5

$$\begin{array}{cccc@{\qquad}cccc}
* & * & * & * & * & * & * & * \\
0 & * & * & * & * & * & * & * \\
0 & 0 & * & * & * & * & * & * \\
0 & 0 & * & * & 0 & * & * & * \\
\\
0 & 0 & * & * & 0 & 0 & * & * \\
0 & 0 & 0 & * & 0 & 0 & * & * \\
0 & 0 & 0 & 0 & 0 & 0 & * & * \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & *
\end{array}$$

Figure 2-7: Step by step column block cyclic LU elimination procedure of an 8x8 matrix

2.3.2 Blas-3 Update Procedure

To allow for the use of level 3 BLAS, blocked columns are used on each node. In LU factorization a blocking of the operations on b columns means that b rows are eliminated at a time from all the other rows. Let the block size used be 'b' and pivot element be$A(i,j)$. Identify the pivot column block of size $(n-i+1)\, x\, b$ that is the block $A(i{:}n, j{:}j+b-1)$ and apply a three parameter LU factorization on the block such that [L,U,f] = LU($A(i{:}n, j{:}j+b-1)$) returns unit lower triangular matrix L, upper triangular matrix U, and permutation matrix f so that f* $A(i{:}n, j{:}j+b-1)$ = L*U. This operation helps to obtain the multipliers for the block column below the pivot diagonals. This operation also performs the partial pivoting that is necessary to establish stability to the algorithm by making the diagonal element the largest in the column. This is essential to prevent divide by zero or by a very small number which may result in round off error.

Now obtain the strict lower triangular part 'l' of the block $A(i{:}i+b-1, j{:}j+b-1)$ of size$(b\, x\, b)$. In Fig (2-8) shown below, the matrix has a block size of b = 2 and 'l' is the block shown in region 1. Let 'LL' denote the matrix obtained by the following operation:

$$LL = l + I \quad (I\ is\ an\ identity\ matrix\ of\ size\ b\ x\ b)$$

The sub-matrix in region 4 is multiplied with the inverse of LL before performing a Blas-3 update on the block represented by region 2. To perform the Blas-3 update, subtract the product of block $A(i+b{:}n, j{:}j+b-1)$ represented by region 3 and the block represented by region 4, from each of the rows of the block represented by region 2. The updates must be applied forward and backward of the pivot column block for those sub-matrices below region 4.

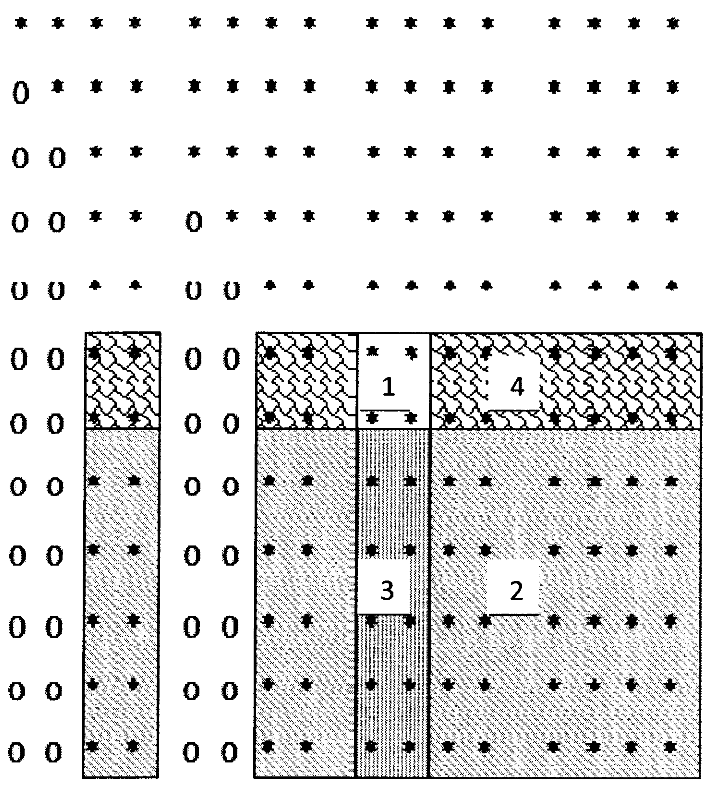

Figure 2-8: Block Cyclic elimination procedure. Regions are described as 1 - blank space, 2 – all sub-matrices represented with diagonal lines, 3 – all sub-matrices with vertical lines and 4-all sub-matrices with mesh.

The block cyclic elimination algorithm requires no permutation of data and factorization can be achieved by performing the above operations forward and backward of the pivot column block by appropriate indexing in the code.

Chapter 3

Results and Inference

Block cyclic order elimination algorithms for LU and QR factorization for rectangular matrices of arbitrary shape are implemented. The execution time and performance characteristics for various matrix order and block sizes are described in this chapter. The algorithm was validated against MATLAB®'s regular backslash operator to check the accuracy and precision of the solution. Graphical interpretation of performance is also made in this section. These timings were done on a preliminary version of STAR-P that was sometimes on a busy machine. Hence there is greater scope for performance with the future editions than what is presented in this project. As part of future work we plan to redo these timings. We found the absolute performance numbers not as valuable as the comparative trends.

3.1 LU Factorization Performance Characteristics

The LU factorization algorithm factors matrix A in $Ax = B$ using a block cyclic elimination order with partial pivoting. Partial Pivoting, a technique used to ensure numerical stability of the algorithm progresses through the diagonal elements making certain that the pivot element is the largest element in that column and essentially non-zero. Partial pivoting is not necessary in diagonally dominant systems. A triangular solve is then performed to obtain the solution to the dense set of linear equations. The algorithm can be used for all rectangular matrices of arbitrary sizes but one must make sure that the block size is a factor of the brick size during elimination.

Several assessment runs were carried out for a range of blocking factors and matrix order. The results are tabulated for single right-hand side. The BLAS algorithms used in LU factorization typically consist of multiplication and addition floating point operations. Typically the number of floating point operations in LU factorization is equivalent to $\frac{2}{3}n^3$ and the execution speed is calculated taking into account the number of floating point operations. The outcomes of the execution are summarized in Table 3-1 and graphically represented in Figure3-1. The block sizes used in the tabulation are 1, 2, 4 and 8 for matrix sizes in the range of 64 and 1024.

Matrix Size	Block Size	Execution Rate (Mflops/s)
64	1	**0.003**
128	1	**0.007**
256	1	**0.020**
512	1	**0.046**
1024	1	**0.120**
64	2	**0.006**
128	2	**0.021**
256	2	**0.072**
512	2	**0.173**
1024	2	**0.224**
64	4	**0.010**
128	4	**0.036**
256	4	**0.088**
512	4	**0.348**
1024	4	**0.599**
64	8	**0.021**
128	8	**0.088**
256	8	**0.171**
512	8	**0.643**
1024	8	**1.031**

Table 3-1: Execution Rate for LU factorization as a function of block size and matrix order and fixed number of Right hand side = 1.

Higher order matrices offer significant speed-up than lower order matrices. The execution rate increases by a factor of over 40 for an increase in matrix dimension N by a factor of 16 and block size 1. For block size of 16 the execution rate increases by a factor of 50.

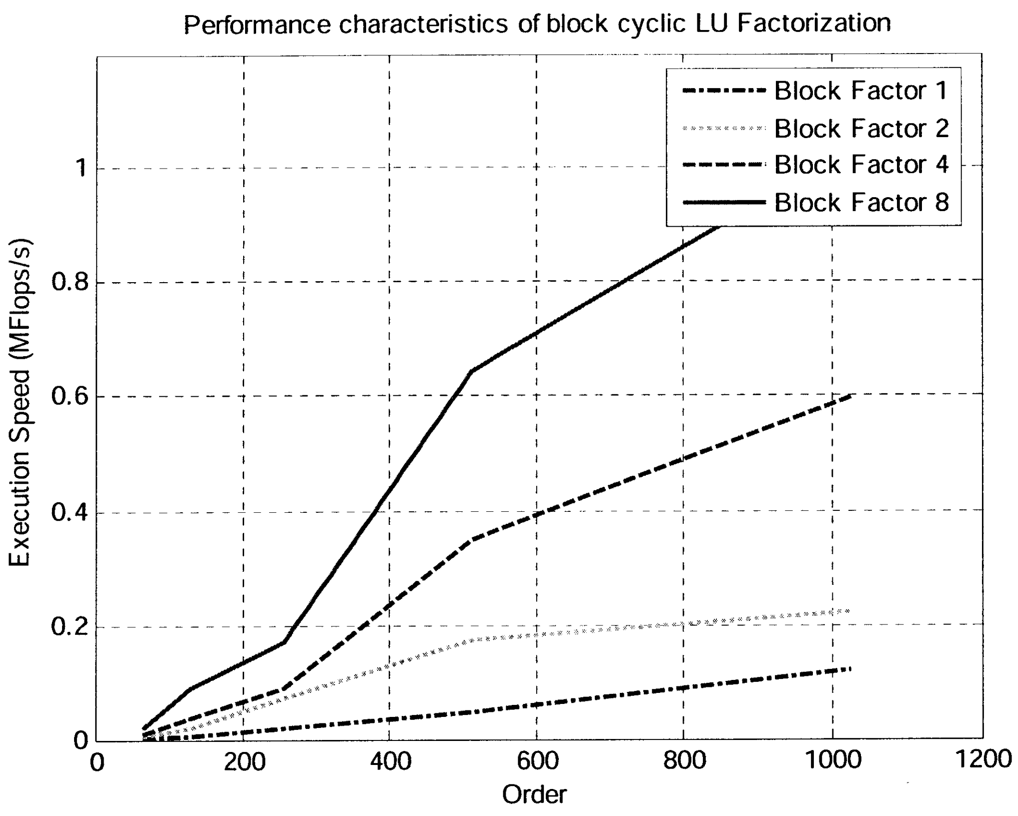

Figure 3-1: Execution Rate vs. Order of matrix for different blocking factors.

Similarly blocking also plays a crucial role in performance improvement. This can be observed in Fig 3-1. The execution rate increases by a factor of 7 for matrices of order 64 for increase in blocking by a factor of 8. For matrix order of 1024 the execution increases by a factor of 11.

3.2 QR factorization Performance Characteristics

In QR factorization, the matrix A is factored into Q and R matrices of the same order of A, R is an upper triangular matrix and Q is an orthonormal matrix. As in LU factorization, partial pivoting must be implemented for stability. QR involves a vector-matrix product followed by a BLAS-3 matrix update. The vector matrix product doubles the number of floating point operations compared to that in LU to $\frac{4}{3}n^3$.

Assessment runs were carried out for a range of blocking factors and matrix order. The results are tabulated for dense linear equations with a single right hand side. Similar conclusions to that of LU can be drawn based on these results. The outcomes of the test runs from QR factorization are summarized in Table 3-2 and are graphically represented in Figure 3-2. The block sizes used in the tabulation are 2, 4 and 8 for matrix sizes in the range of 64 and 4096.

Block Size	Matrix Order n	Execution Rate (Mflops)
2	64	**0.03**
2	128	**0.12**
2	256	**0.48**
2	512	**1.74**
2	1024	**4.72**
2	2048	**6.23**
2	4096	**9.65**
4	64	**0.03**
4	128	**0.12**
4	256	**0.52**
4	512	**1.74**
4	1024	**4.83**
4	2048	**6.62**
4	4096	**9.98**
8	64	**0.04**
8	128	**0.13**
8	256	**0.53**
8	512	**1.77**
8	1024	**4.94**
8	2048	**6.97**
8	4096	**10.22**

Table 3-2: Execution Rate table for QR factorization as a function of block size and matrix order and fixed number of Right hand side = 1.

It is quite evident that QR factorization performs much better than the LU factorization. There is a significant performance improvement with matrix order. The execution rate increases by a factor of over 320 for an increase in matrix dimension N by a factor of 64 and block size 2. The factor increased by 150 times for an increase in matrix order by a factor of 16. It performed even better for block size 8. The execution rate increases by a factor of over 340 for an increase in matrix dimension N by a factor of 64 and by a factor of 165 for a factor of 16.

Blocking is not as significant in QR as in LU. The execution rate increases by a factor of 1.1 for each step increase in blocking by a factor of 2. Hence it is fair to comment that whilst LU is dependent on both blocking and order, QR is highly reliant on matrix order for better performance. Figure 3-2 depicts the performance characteristics of QR factorization for various blocking factors.

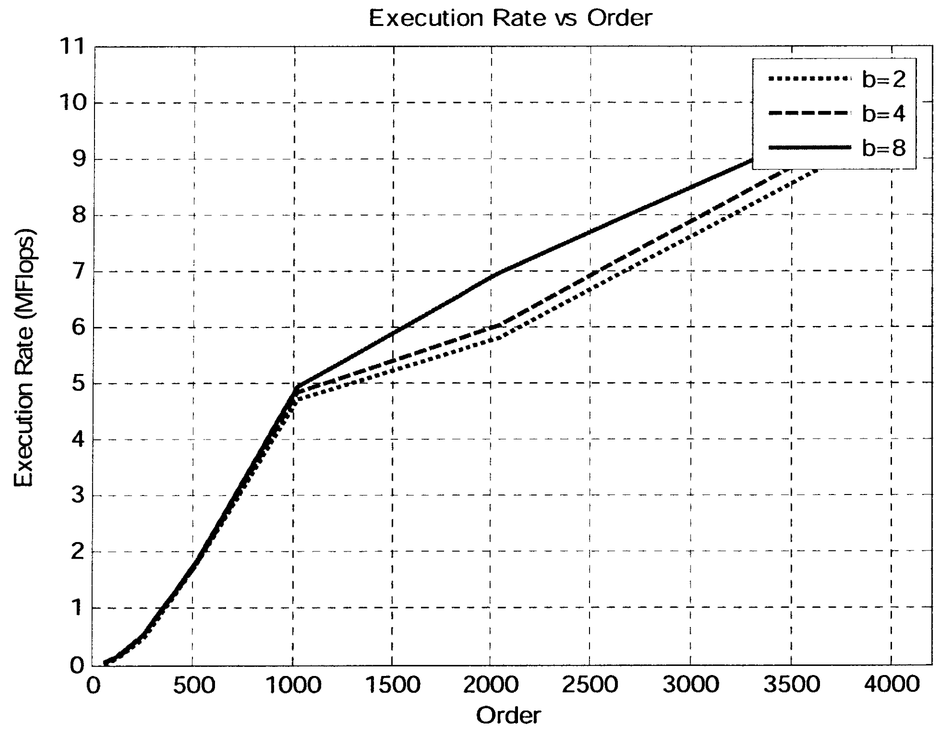

Figure 3-2: Performance characteristics of QR factorization. Execution Rate vs. Order of matrix for different blocking factors

Figures 3-1 and 3-2 illustrate the performance characteristics of LU and QR algorithm respectively as a function of block size and order for a constant right-hand side size = 1. Both curves slope upwards which demonstrates that the execution speed increases with increase in matrix order. This can be addressed by the fact that the significance of the $O(N^2)$ term reduces with increase in matrix order.

3.3 Discussions and Inference

Block cyclic order LU and QR factorization algorithms are described for rectangular matrices of arbitrary sizes. Results are tabulated for a single hand-side; also solutions can be generated for numerous numbers of right-hand sides. The LU and QR algorithms are implemented with partial pivoting for numerical stability. A brief description about the potential advantages and disadvantages of using block cyclic elimination over block cyclic distribution was also presented. The impact of blocking was also discussed.

Matrix order plays a momentous role in upgrading the performance characteristics of the algorithm. The execution rate increased by a factor of around 45 for LU factorization and a factor of 150 for QR factorization with increase in order by a factor of 16. Level-3 BLAS Block update provided significant performance improvement in the order of 10 for LU and around 1.2 for QR factorization. The block cyclic elimination algorithm is an analogous illustration of data parallelism. The next chapter addresses task parallelism.

Chapter 4

Task Parallel Algorithms

The block cyclic elimination algorithm described in the earlier chapter employs data parallel threading. This chapter delves into the expressivity of task parallel algorithms. Task Parallelism is extensively used when independent algorithms work on different data sets to produce a result which is later combined. Thus there is almost no communication between the processors. The algorithms discussed in this chapter are the parallel prefix, Monte-Carlo and an algorithm which generates millions of digits of Pi.

4.1 Parallel Prefix Algorithm

Parallel prefix [19]-[21] is an important data parallel algorithm with immense engineering applications especially in circuit design. The prefix algorithm takes an associated binary operator ℗ and an ordered set $[a_1, a_2, \dots a_n]$ of n elements and returns the ordered set,

$$[a_1,\ (a_1 ℗ a_2), \dots\dots\dots,\ (a_1 ℗ a_2 ℗ \dots . ℗ a_{n-1} ℗ a_n)]$$

The prefix algorithm has several important applications; the most important ones being the segmented scan, Csanky's matrix inversion, Carry Look Ahead Addition and segmented scan which is extensively used in VLSI circuit design. The serial prefix algorithm is of order O(n) as it requires $(n-1)$ serial ℗ operations on an n-element array each element being a 2-D array of dimension $m\ x\ m$ whilst the parallel prefix algorithm is of order O(log n) assuming n processors exist for computation or of order O(n/p + log p) for p processors.

In this implementation we consider that each matrix resides in an individual processor and the recursive algorithm to compute prefix is as follows:

Function $\boldsymbol{prefix\_matmul(a)}$

1. Obtain matrix size along the third dimension.
2. Compute pair-wise product, communicating with adjacent processor. This can be done in Star-P using the reshape command to get array 'w'.
$$w_i = a_i * a_{i-1} \qquad (i\ is\ even)$$
3. Recurse through the array obtained from Step 2 to compute even entries of the output.
$$w = prefix\_matmul(w)$$
4. Compute the odd entries with a pair-wise sum between the odd entries of matrix A and a zero appended 'w' matrix.
$$y_i = a_i + w_{i-1} \qquad (i\ is\ odd)$$
$$y_i = w_i \qquad (i\ is\ even)$$
5. End

The prefix algorithm forms a structure closely resembling a tree in which two actions need to be performed to complete the prefix algorithm: Going up the tree and going down the tree.

Consider vector a = {1,2,3,4,5,6,7,8} for illustrating the tree structure in the recursive implementation of the algorithm. Whilst going up the tree (Figure 4-1) the algorithm simply performs pair-wise addition and while coming down the corresponding odd positions are updated according to steps 3 and 4. Thus for even positions the value of the parent node w_i is used. For odd positions, the value on the left of the parent node $w_{i-1}$is added to the original matrix a_i.

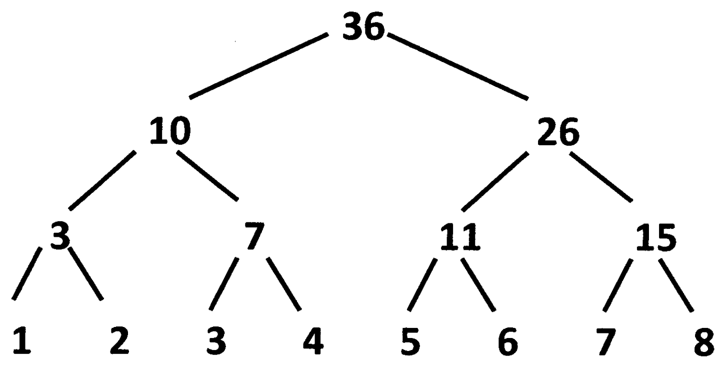

Figure 4-1: Going up the tree structure in the recursive parallel prefix implementation

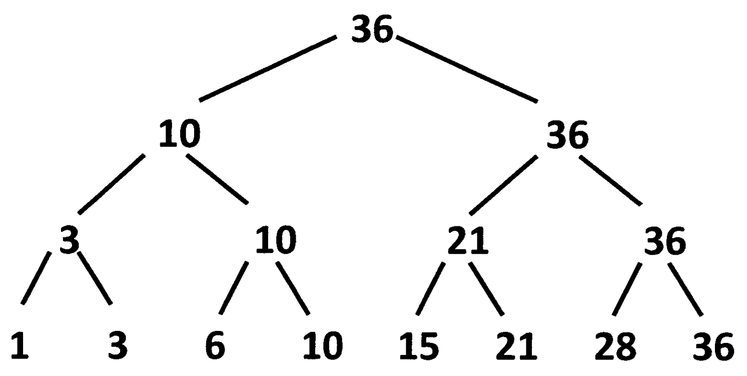

Figure 4-2: Going down the tree in the recursive framework.

4.1.1 Run time Calculation

Consider an array with n elements. Suppose we have enough processors to store each element we can process all the elements in the same step. The number of items in the array is halved in each step i.e. $n \to n/2 \to n/4 \to \cdots \to 1$. Hence O(log *n)* steps are required for adding *n* numbers and the processor requirement is O (*n*). Thus if we started with eight elements it will take three steps for termination of the algorithm and obviously computation in each of the steps is being performed in a parallel manner.

Sometimes the data matrix is huge and the number of processors is very limited. Hence there will be more than 1 element associated with each processor. Suppose we have n elements and p processors, and define k = n/p. Then the procedure to compute the scan is:

1. At each processor i, compute a local scan serially, for n/p consecutive elements whose outcome is $[d_1^i, d_2^i, \ldots\ldots, d_k^i]$.

2. Use the parallel prefix algorithm to compute

$$scan([d_k^1, d_k^2, \ldots\ldots, d_k^p]) = [w_1, w_2, \ldots\ldots w_p]$$

3. At each processor i > 1, add $w_{i-1}$to all elements d_j^i.

$$T = 2\begin{pmatrix} time\ to\ add\ and\ store \\ \frac{n}{p} numbers\ serially \end{pmatrix} + 2(\log p)\begin{pmatrix} communication\ time\ up\ and \\ down\ the\ tree, and\ a\ few\ adds \end{pmatrix}$$

In the limiting case of p<<n , the (log p) message passes are an insignificant portion of the computational time, and the speedup is due solely to the availability of a number of processes each doing the prefix operation serially.

4.1.2 Implementing Recursive Parallel Prefix in StarP

Star-P provides a very simple way to perform the prefix algorithm in a parallel manner. Consider a 3-D matrix A of order nxnxl. The reshape function can be used to convert the 3-D to a 4-D matrix. This matrix is then split across the last dimension and a pair-wise product is computed using the ppeval command. The program is driven into a recursion exactly like the tree structure shown above, each time applying the task parallel ppeval command for parallel product of the matrices. Finally the odd entries are updated using ppeval.

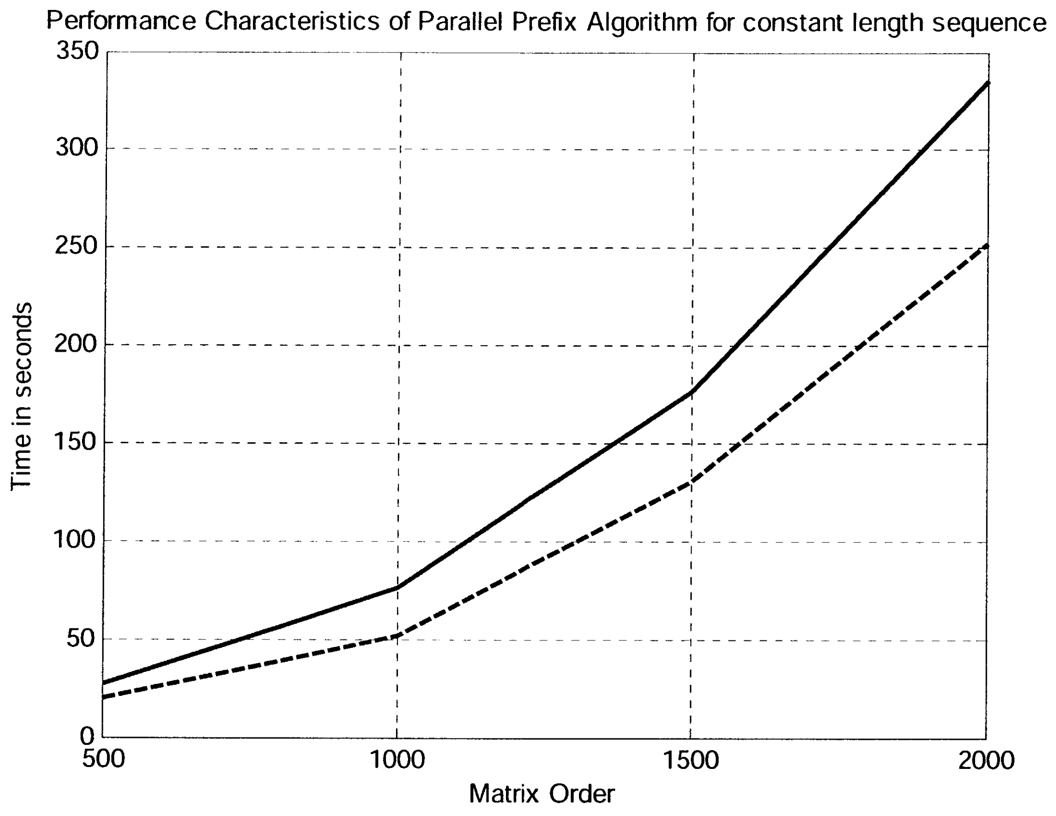

Figure 4-3: Time elapsed vs. Order of matrix for length sequence's 16 and 32.

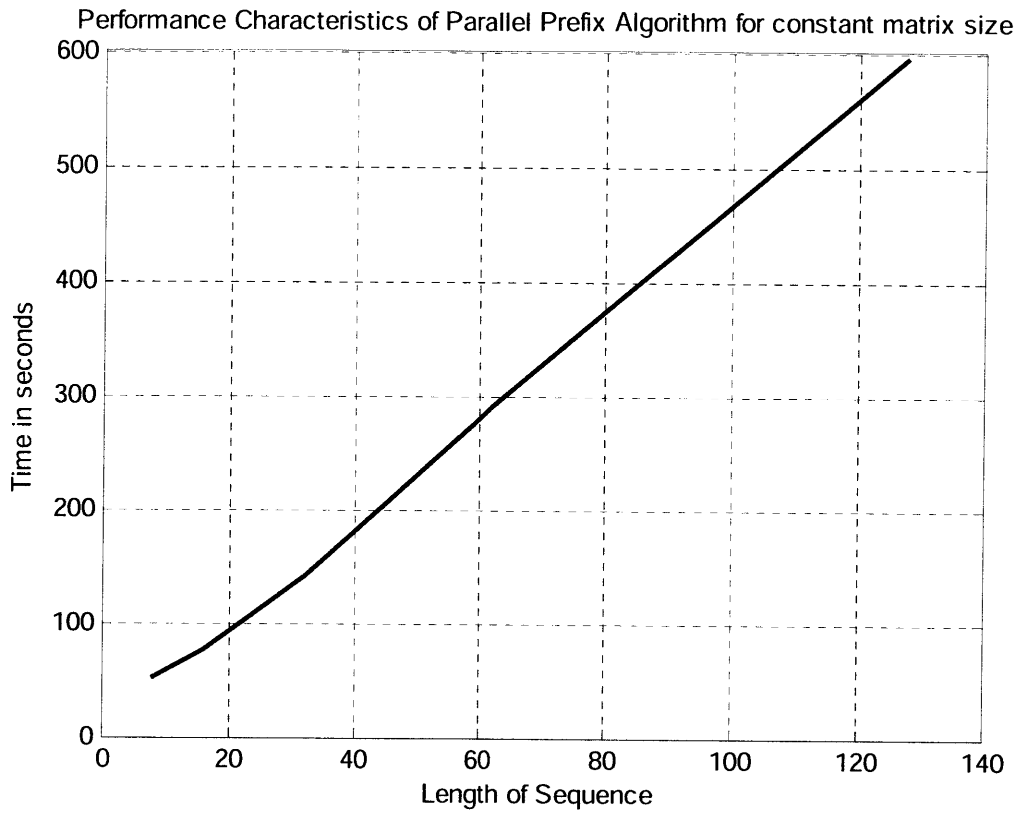

Figure 4-4: Time elapsed vs. length of sequence for constant matrix order of 1500.

4.1.3 Summary and Discussion

A recursive parallel prefix algorithm was implemented using Star-P and its performance was evaluated. The parallel prefix algorithm utilizes features of task parallelism to achieve speed up. Fig 4-3 characterizes the algorithm's performance for constant length sequences for varying matrix order using 6 nodes. The algorithm time increases approximately quadratic with increase in matrix order for all length sequences. Fig 4-4 analyses the algorithms performance for matrix of order 1500 and variable length of sequence. Fig 4-4 utilizes 8 nodes. It can be seen that the scale-up relationship is linear.

4.2 Generating Millions of Digits of PI

Pi has always been a fantasy for every mathematician [17]-[18] and enumerating its billionth and trillionth digits has always been associated with really fast parallel machines. Several algorithms have thus been established to calculate the n^{th} digit without the knowledge of the $(n-1)^{th}$ digit. These algorithms when executed on a parallel machine are embarrassingly parallel allowing wide scope in listing out a large number of these digits. The nature of these algorithms is purely experimental and methods have been only established to obtain these digits in binary and hexadecimal forms.

The earliest decimal approximate fractions of π are $\frac{22}{7}, \frac{333}{106}$ and $\frac{355}{113}$. To calculate the n^{th} digit with each of these fractions it would require the calculation of all the $(n-1)$ digits prior to it and empirical experiments to generate digits in base 10 never bore fruit.

4.2.1 Introduction

In 1996, Bailey, Borwein and Pluoffe [17] provided an empirical algorithm which produced the hexadecimal equivalent of a digit at any arbitrary position without needing to compute digits prior to it. The algorithm is as follows:

$$\Pi = \sum_{i=0}^{\infty} \frac{1}{16^i} \left[\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6} \right] \tag{1}$$

The embarrassingly parallel structure of the algorithm can be immediately observed from equation 1. This scheme can be easily reduced to a form which allows use of the binary algorithm for exponentiation to prevent memory overflow.

Equation 1 can be reduced to the following form

$$\{16^d\pi\} = \quad 4\{16^d S_1\} - 2\{16^d S_4\} - \{16^d S_5\} - \{16^d S_6\}$$

where

$$\sum_{i=0}^{\infty} \frac{1}{8i+j} \text{ is represented by } \{16^d S_j\}.$$

Now

$$\{16^d S_j\} = \left\{ \left\{ \sum_{k=0}^{d} \frac{16^{d-k}}{8k+j} \right\} + \sum_{k=d+1}^{\infty} \frac{16^{d-k}}{8k+j} \right\}$$

$$= \left\{ \left\{ \sum_{k=0}^{d} \frac{16^{d-k} mod\, 8k+j}{8k+j} \right\} + \sum_{k=d+1}^{\infty} \frac{16^{d-k}}{8k+j} \right\}$$

Here {.} is the fractional part of the quotient. 'mod k' has been inserted in the numerator because we require only the fractional part of the quotient and that piece can be calculated very rapidly by means of the binary algorithm for exponentiation. Combining all the four parts we obtain a fraction which when expressed in hexadecimal notation gives the hex digits of π in position (d+1). The above algorithm was implemented by Bailey et al. in C and FORTRAN-90.

4.2.2 Implementation of the algorithm

This work presents a novel matrix implementation of the above algorithm to get millions of digits of π in hexadecimal notation. Each row of the matrix represents the terms from the first part of the equation and is updated from the previous row; hence there is no issue with memory overflow.

The matrix is as described below with 'd' representing the rows and 'k' representing the columns('d' and 'k' are terms from equation above). Suppose we would like to find the first 4 terms of π the corresponding matrix will resemble the following structure.

$$\begin{array}{c|cccc|c|}
d\backslash k & 0 & 1 & 2 & 3 & \\
\hline
0 & 16^0 mod1 & 0 & 0 & 0 & 1 \\
1 & 16^1 mod1 & 16^0 mod9 & 0 & 0 & 9 \\
2 & 16^2 mod1 & 16^1 mod9 & 16^0 mod17 & 0 & 17 \\
3 & 16^3 mod1 & 16^2 mod9 & 16^1 mod17 & 16^0 mod25 & 25
\end{array}$$

j = 1

$$\begin{array}{c|cccc|c|}
d\backslash k & 0 & 1 & 2 & 3 & \\
\hline
0 & 16^0 mod4 & 0 & 0 & 0 & 4 \\
1 & 16^1 mod4 & 16^0 mod12 & 0 & 0 & 12 \\
2 & 16^2 mod4 & 16^1 mod12 & 16^0 mod20 & 0 & 20 \\
3 & 16^3 mod4 & 16^2 mod12 & 16^1 mod20 & 16^0 mod28 & 28
\end{array}$$

j = 4

$$\begin{array}{c|cccc|c|}
d\backslash k & 0 & 1 & 2 & 3 & \\
\hline
0 & 16^0 mod5 & 0 & 0 & 0 & 5 \\
1 & 16^1 mod5 & 16^0 mod13 & 0 & 0 & 13 \\
2 & 16^2 mod5 & 16^1 mod13 & 16^0 mod21 & 0 & 21 \\
3 & 16^3 mod5 & 16^2 mod13 & 16^1 mod21 & 16^0 mod29 & 29
\end{array}$$

j = 5

$$\begin{array}{c|cccc|c|}
d\backslash k & 0 & 1 & 2 & 3 & \\
\hline
0 & 16^0 mod6 & 0 & 0 & 0 & 6 \\
1 & 16^1 mod6 & 16^0 mod14 & 0 & 0 & 14 \\
2 & 16^2 mod6 & 16^1 mod14 & 16^0 mod22 & 0 & 22 \\
3 & 16^3 mod6 & 16^2 mod14 & 16^1 mod22 & 16^0 mod30 & 30
\end{array}$$

j = 6

Figure 4-5: Matrix structure for j = 1, 4, 5 and 6 processed by different nodes.

The first part of the equation $\left\{\sum_{k=0}^{d} \frac{16^{d-k} mod\, 8k+j}{8k+j}\right\}$ is represented as above in form of a matrix. All the four matrices for j = {1, 4, 5, 6} are shown above. Since the calculations are independent of each other, each matrix can be calculated with a single processor in an embarrassingly parallel manner after which the results can be combined. The second part of the equation requires utmost 20 digits i.e. $\mathrm{k} = (\mathrm{d}+1)\mathrm{to}\ (\mathrm{d}+20)$ to converge. The matrix method performs significantly faster than the conventional implementation. This is a BLAS-2 implementation involving matrix vector product.

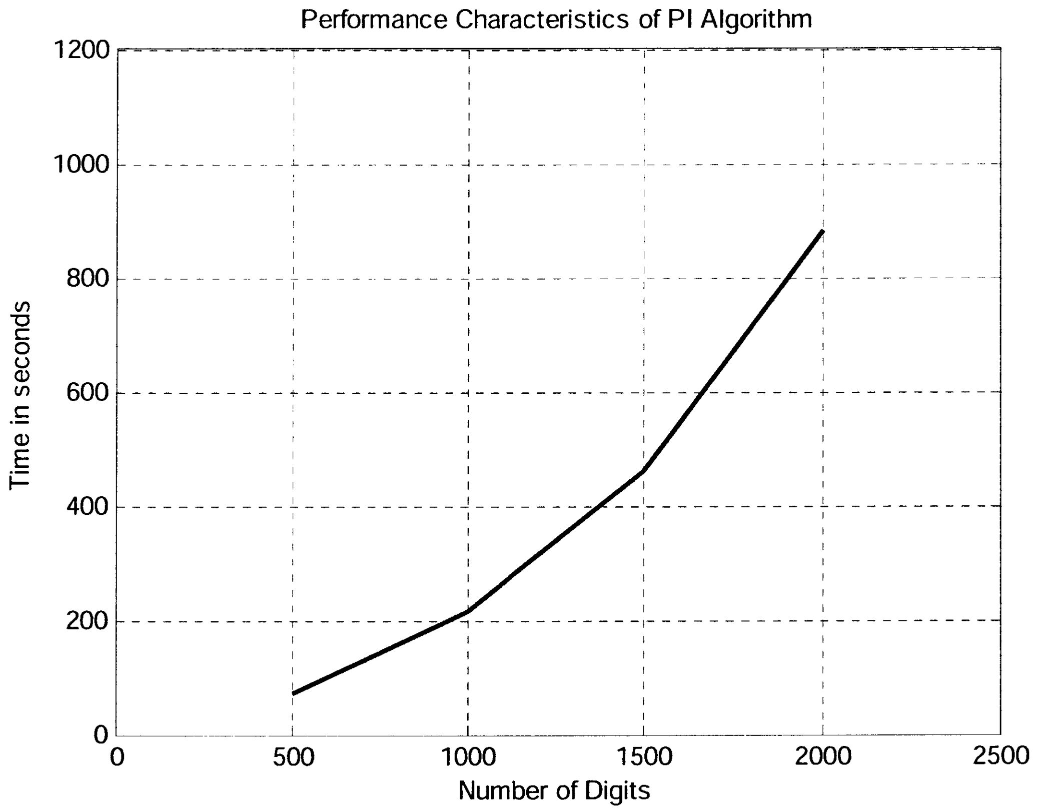

Figure 4-6: Performance characteristics of PI algorithm describing Time elapsed vs. Number of digits. In this experiment, a total of 4 processors were used.

4.2.3 First 1000 Hexadecimal Digits of PI

```
2 3 2 0 2 9 A 7 F 8 5 5 4 5 F 0 D 5 4 B 8 C 1 C 6 1 E 6 A 4 5 1 9 3 F 3 A 0 6 D
4 7 E 1 9 1 6 B 1 0 7 7 A 3 0 6 7 6 8 1 6 2 6 7 B D C 2 C 8 E A 6 7 1 0 5 7 2 3
3 0 F 3 B 7 9 8 2 1 4 4 E 9 C 0 1 0 9 0 A A 9 A 4 8 8 3 C 2 2 B 0 A 6 E C 3 3 7
F 7 A 7 7 9 8 E C F E 8 E 2 A 3 5 5 8 B F 2 B 3 B 0 0 0 5 A 1 D F 3 5 8 3 4 6 D
6 3 9 7 C 2 D 1 7 2 6 F 7 A 4 A 7 C 6 6 7 B 8 2 B 9 3 2 0 4 C 3 A B 1 8 3 0 3 0
A 4 8 B 9 1 F A F E 9 7 B F 1 1 7 6 2 B C A 7 5 9 C 2 E F 8 6 8 7 E D 8 B 1 F D
8 4 E E 7 6 B F 9 2 A 2 5 2 7 8 C 0 6 4 7 9 9 3 A C E B 6 4 6 8 2 4 3 C 8 A 7 7
8 A C 5 C D 5 E 9 8 4 8 4 6 9 0 1 E 3 C 2 C 3 8 F F F 6 D 2 8 F 8 B 9 E B 4 7 2
8 4 4 4 5 5 A D 2 5 5 E A 0 1 E B 6 E C E 5 1 1 C B 8 5 6 0 4 0 A A A E 5 4 0 4
5 0 E 6 0 D C 6 4 8 8 B 4 1 8 6 D 5 8 5 9 5 E 2 4 2 4 1 F 0 2 6 B 3 F 8 E 9 6 D
A 9 6 6 D 9 2 A A E F 6 1 3 B C 3 5 1 C 9 D A 8 B 1 5 B F 4 F A 5 B 0 6 B F 1 0
3 3 C C D 8 F 2 1 F E 5 D C 8 9 1 2 4 3 3 7 F 9 F A D 8 3 6 6 5 1 F 1 1 E 5 B 0
0 8 8 F 3 9 F 6 9 C A 8 C 5 D E 4 5 4 4 B 4 D 5 E 9 5 8 8 9 E 1 3 0 7 9 E 6 F A
8 2 9 3 F 7 D 7 9 1 3 7 2 D B 0 B F 0 1 3 1 6 8 8 9 D 2 3 C 9 A 3 5 6 4 0 C E 1
D 2 4 4 8 9 7 E 4 6 F 1 5 1 3 E 2 3 5 1 E 8 B 6 1 1 E 3 F 8 6 0 A 0 6 5 6 1 D 2
3 2 5 E 4 F 2 9 7 6 4 8 A B 8 8 7 A 5 4 E 3 A 7 B 4 9 8 4 F C D 3 7 6 6 F 6 F 4
1 9 2 9 D B D 6 B 3 9 B 5 0 E B 7 A C 1 1 1 3 7 6 8 8 9 4 0 9 2 6 E C F 7 A 7 8
3 9 8 0 5 1 B B 3 6 3 C 9 2 F B 8 5 A E 4 F 3 3 6 7 5 3 2 4 A D E F A 9 5 A 2 D
1 F 2 C B B D A 9 9 3 D B 3 8 0 A 5 3 8 1 6 6 B 2 C 7 E 3 A 6 8 E B 5 F D 6 4 B
9 3 1 6 5 D 0 7 1 2 D 5 5 2 E 1 F A 9 C 1 C C 8 8 A 5 8 9 9 7 5 F 2 9 B 8 4 2 0
8 1 E C B 1 1 C 6 0 7 8 9 8 7 E 2 B 6 E 6 E 2 F 2 C B 1 2 E 0 4 0 A 3 4 8 E 9 F
A D 6 C 5 3 A 9 C D E 8 C 6 9 8 F 9 A A 3 5 4 4 1 6 1 D E 1 C 2 B 9 E 7 5 D B E
2 0 3 0 4 1 D 0 F 8 0 2 3 0 D A D 4 2 1 6 C C 8 9 0 D 3 0 F 9 F 6 8 8 D C 3 0 A
E 0 8 A 7 0 F 4 7 7 D 1 0 8 C 3 A 5 A 5 F 3 F 9 3 5 C 9 B 9 C 6 C A 2 8 1 A 2 D
0 8 D C 0 B B 5 0 1 9 5 D 5 B E 5 7 A 4 B E 5 8 6 D 2 6 4 B 6 8 1 1 4 4 2 A 3 3
```

Note: Read through the rows of the matrix, Column 1 contains the first 25 digits.

4.2.4 Summary and Discussion

An empirical relationship which generates individual digits of PI was implemented with a unique matrix construction to extract millions of digits. The method was described and the correctness of the digits was also checked. The matrix becomes gigantic for generating a large number of digits soliciting the necessity for use of large back end nodes to perform the role. From Fig 4-6, it is evident that the Time-digits scaling is quadratic i.e. the running time increases quadratically with increase in number of digits.

4.3 Parallel Monte-Carlo Simulation for derivative Pricing

Intense competition amongst various Investment Banks and highly complex quantitative strategies in use has led to an enormous increase and necessity for incredible speed of computation. The Monte Carlo algorithm is one of the most extensively used approaches to price derivatives in the financial industry. The structure of the algorithm makes it a perfect candidate for the task parallel feature in Star-P. In this project, optimal use of the task parallel Monte Carlo algorithm structure is made to propose an efficient algorithm.

4.3.1 Derivative Pricing and Total Return Swaps

A derivative is a financial instrument [14] derived from some other asset where market participants enter into an agreement to exchange money, assets at some future date based on the performance of the underlying asset. There are many types of financial instruments classified under derivatives of which options, futures and swaps are most common.

In swaps [15] one party agrees to swap cash flows with another. The exchange of fixed-rate and variable-rate loans between established businesses based on the preference for the other type of loan falls under this category. Rather than cancelling their existing loans, the two businesses can swap cash flows: the first pays the second based on floating rate and the second business pays the first based on a fixed-rate loan.

Total return swap is a contract in which one party receives interest payments on a reference asset plus any capital gains and losses over the payment period, while the other receives a specified fixed or floating cash flow unrelated to the credit worthiness of the reference asset, especially where the payments are based on the same nominal amount. The reference asset may be any asset, index or basket of assets.

4.3.2 Problem Definition

In this example [13], a Total Return Swap agreement between SK securities and JP Morgan is considered. The swap agreement was entered into in 1997 with a one year maturity. SK securities were obligated to return $53{,}000{,}000\left[0.97 - Max\left(0, \frac{Y_T - Y}{Y_T}\right) + Max\left(-1{,}5 * \frac{B_T - B}{B_T}\right)\right]$ for the \$53,000,000 that they borrowed from JP Morgan. Here Y_T, B_T denote Yen and Baht vs. the Dollar respectively at the time of agreement. Y,B denote the Yen and Baht vs. the Dollar respectively at the time of maturity. We solve this problem using the Black-Scholes Brownian motion assuming that the Yen, Baht vs. Dollar exchange rates follow a log-normal distribution with volatilities σ_y, σ_Brespectively and correlation ρ. r_D, r_Y, r_B are the risk free interest rates in the US, Japan and Thailand for 1year.

The log-normal distributions of the Dollar/Yen and Dollar/Baht is given below:

$$Y_T = Ye^{-[\left(r_D - r_Y - \frac{\sigma_y^2}{2}\right)T + \sigma_Y\sqrt{T}\xi_Y]}$$

$$B_T = Be^{-[\left(r_D - r_B - \frac{\sigma_B^2}{2}\right)T + \sigma_B\sqrt{T}\xi_B]}$$

4.3.3 Implementation - Parallel Monte Carlo Algorithm

The Monte Carlo model includes any method of estimating a value by the random generation of numbers and statistical principles. As an approach to pricing swaps, Monte Carlo option models use a pseudo-random sequence that will be random enough to simulate a range of outcomes.

The Monte-Carlo algorithm can be implemented in two pieces, one part to call the algorithm to execute on the processor nodes and the other part, the algorithm itself. The algorithm is as follows:

Callmontecarlo.m

- $Iteration = [1{:}n].$
- $Value = ppeval('montecarlo', Iteration).$
- $Compute\ Mean\ of\ Value.$

Montecarlo.m

- $Get\ parameters\ Y, B, r_D, r_Y, r_B, \sigma_y, \sigma_B.$
- $\xi_Y, \xi_B are\ random\ numbers\ from\ bivariate\ normal\ distribution\ with\ correlation\ \rho\ .$
- $Calculate\ Y_T, B_T\ from\ eqn. (2) and\ (3).$
- $Calculate\ mean\ of\ Y_T, B_T.$
- $Calculate\ Payment\ from\ (1)$
- $Calculate\ present\ value\ discounted\ by\ r_D.$

The callmontecarlo.m function executes the algorithm contained in montecarlo.m in a task parallel manner i.e. the same algorithm is distributed over different nodes to allow each processor to return a unique solution for the random bivariate variable associated with it. The results of all the N operations are returned to the main processor which then calculates the mean of all the trials. Since the N operations are performed in parallel the time taken for execution is incredibly fast. Often N>>p, where p is the number of processors, then N/p data is stored and processed by each node.

4.3.4 Summary and Discussion

In this project, a parallel Monte Carlo algorithm for financial derivatives pricing is shown. Experimental results show performance of the algorithm when up to 8 computing nodes are used. The embarrassingly parallel structure of the Monte Carlo algorithm makes use of the task parallel feature of STAR-P and performance gains approaching the number of parallel processors can be achieved. They are graphically represented below.

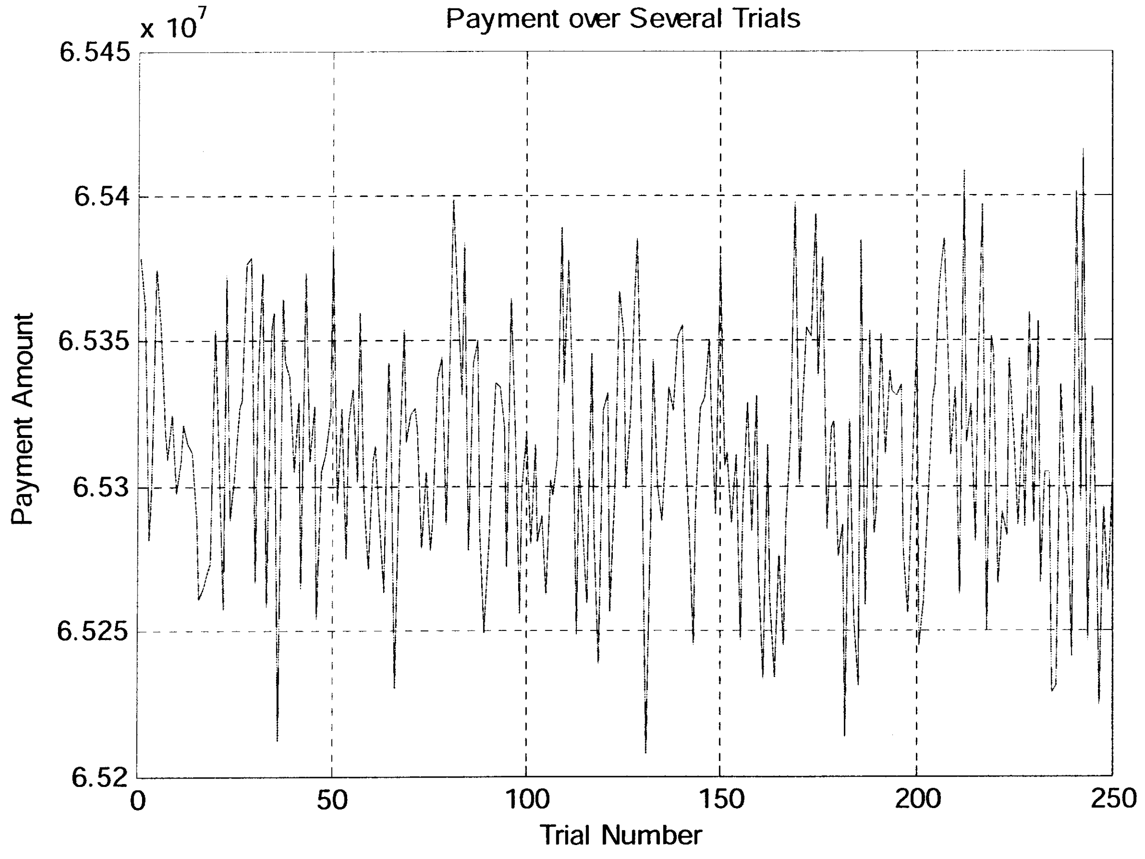

Figure 4-7: Present Value of the asset over several Monte-Carlo trials.

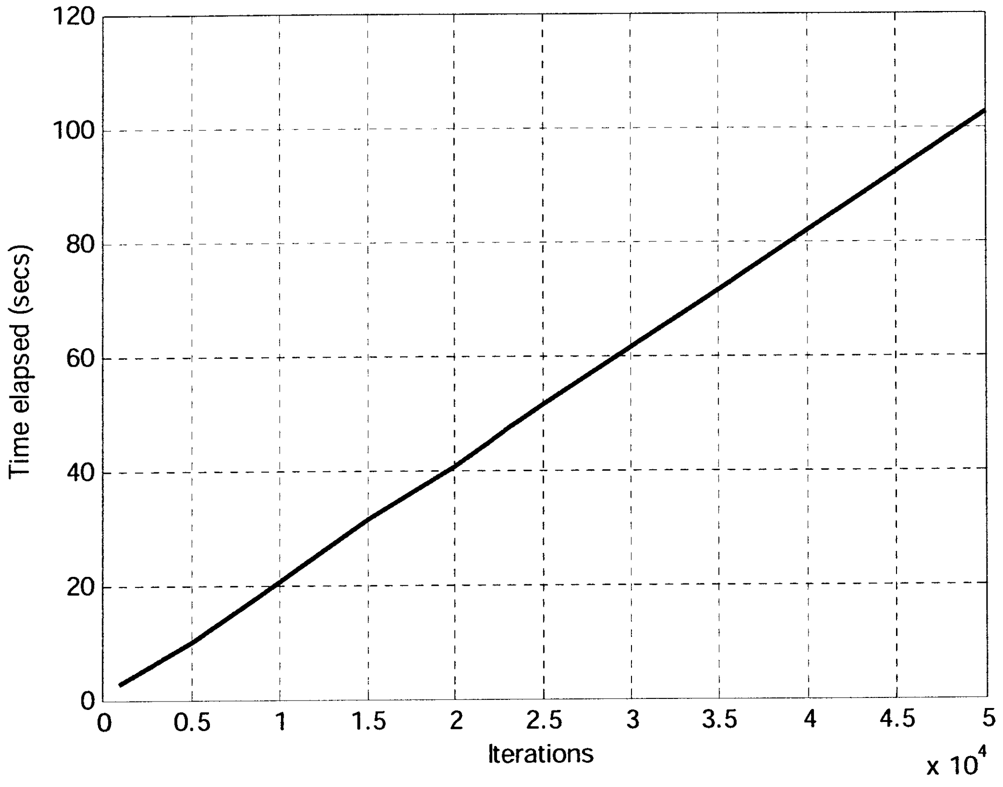

Figure 4-8: Task parallel speed up behavior for the problem defined in 4.4.2. In this experiment 4 nodes of computers are used. With increase in number of iterations the accuracy of the solution increases and the relationship is also linear.

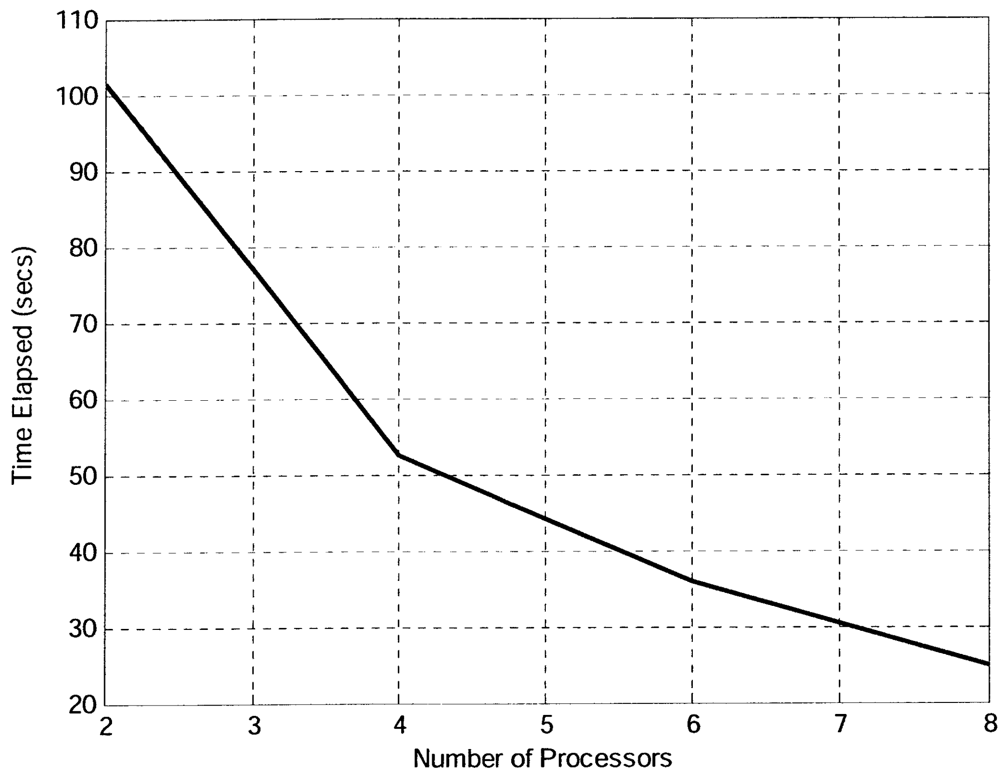

Figure 4-9: Speedup behavior of the algorithm versus number of processors for a fixed number of iterations = 25000.

Chapter 5

Scope and Future Direction

This project describes the implementation of a column block cyclic elimination algorithm. This project can be extended for better performance by incorporating the 2-D block cyclic elimination algorithm. It provides for superior load balance and communication between nodes when compared to the 1-D column block cyclic elimination but consumes significant amount of development time. Also, all programming codes have been written in Star-P and the results have been compared to performance of serial MATLAB®. A better performance indicator would have been to compare the results with MPI. Whilst it can be predicted that MPI would have performed much better in terms of execution rate one would have gained tremendously in Star-P from development time and ease of development.

In the end, the best linear algebra is likely neither block cyclic in distribution or elimination. Both are schemas to answer:

1. how does the data come in
2. where does the execution occur

With large high level inter-operable systems, the decision of how data comes in may no longer be made in isolation, nonetheless where execution occurs is worthy of study. As parallel structure becomes more extensible, likely neither approach will prove optimal as automated tuning software show the way.

Appendix A

LU and QR decomposition

The QR decomposition is used to solve large dense linear systems. In QR decomposition, a matrix A is factored into two matrices Q and R such that A = QR, Q being orthonormal and R an upper triangular matrix. Q is orthonormal and $Q^T Q = I$. There are several methods to compute the QR decomposition of a matrix - Gram-Schmidt procedure, Householder transformation, rotations. The new Gram-Schmidt algorithm is described below for a matrix A of order mxn with each column vector denoted by $a_1, a_2, \dots a_n$.

Given an arbitrary basis $\{a_1, a_2, \dots a_n\}$ for an n-dimensional inner product space V, the Gram-Schmidt algorithm constructs an orthogonal basis $\{v_1, v_2, \dots v_n\}$ for V :

$\boldsymbol{Step1}: Let\ v_1 = a_1$

$\boldsymbol{Step2}: Let\ v_2 = a_2 - proj_{W_1} a_2$

$$= a_2 - \frac{(a_2, v_1)}{\|v_1\|^2} v_1\ here\ W_1\ is\ the\ space\ spanned\ by\ v_1, and\ proj_{W_1} a_2\ is\ the\ orthogonal$$

$projection\ of\ a_2\ on\ W_1.$

$\boldsymbol{Step3}: Let\ v_3 = a_3 - proj_{W_2} a_3 =$

$$a_3 - \frac{(a_3, v_1)}{\|v_1\|^2} v_1 - \frac{(a_3, v_2)}{\|v_2\|^2} v_2,\quad where\ W_2\ is\ the\ space\ spanned\ by\ v_1\ and\ v_2$$

Continuing this process up to v_n, the resulting vectors $\{v_1, v_2, \dots v_n\}$consists on n linearly independent vectors forming an orthogonal basis. The Gram-Schmidt algorithm can be used to obtain orthonormal Q matrix by dividing each column vector by its respective norm before projection. The elements of R matrix are easy to obtain. The diagonal elements r_{ii}can be obtained at each pivot step as $\sqrt{a_i.a_i}$ and the off-diagonal elements are obtained as $a_j.v_i$ for all $j > i$ at each pivot column i.

LU factorization is a computationally expensive method which produces very accurate results in solving dense linear systems for a broad range of matrices. In LU factorization a matrix A is factored into two matrices L and U such that A = LU, L being a lower triangular and U an upper triangular matrix. The idea behind the LU factorization is to zero out all the entries below the pivot diagonal element by subtracting a multiple of the row containing the pivot from each of the rows below the pivot. The multipliers calculated simply by dividing the target row elements $a(i+1{:}m,i)$by the pivot element $a(i,i)$is stored in the corresponding position in matrix L to obtain the lower triangular matrix such that A = LU.

LU factorization applied to a strictly diagonally dominant matrix will never produce a zero pivot. To ensure that the diagonal is the largest entry partial pivoting is performed. Partial pivoting ensures numerical stability provided the matrix is non-singular. Partial pivoting can be implemented by swapping the diagonal row with the row which has an entry of maximum value in that column. The Crout and Doolittle algorithms perform LU factorization.

Given a dense linear system of equations $Ax = b$, A is factorized such that $LUx = b$. Then, the equation $Ly = b$ is solved for y and finally $Ux = y$ is solved for x.

Appendix B - PI Code

```
function solution = main_call(d)

sum1 =0; sum2 = zeros(d+1,4);
g = [1,4,5,6]; g = ppback(g);
z = ppeval('hexpi',split(g,2),bcast(d));  %%%% parallel call of function hexpi
for e = 0:d
   for k = d+1:d+20   %%%20 iterations enough for convergence
      b = 16^(d-k)./(8*k+[1 4 5 6]);
      sum1 = sum1 + (b-floor(b));
   end
   sum2(e+1,1:4) = sum1;
end
q = z + sum2; soln = 4*q(:,1)-2*q(:,2)-q(:,3)-q(:,4);
solution = floor(16*(soln - floor(soln))); %%%final solution

function u = hexpi(g,d)

A = eye(d+1,d+1); %%%% all matrices have ones on the diagonal
B = zeros(d+1,1); n = 1;
for m = g
    if (m == 1)
      A(1,1) =0;    %%% for matrix 1 entry a(1,1) = 0, as mod(1,1)=0
    end
    for j = 0:d
       B(j+1,1) = 1/(8*j+m);
          for i = j+1:d
             A(i+1,j+1) = mod(A(i, j+1)*16, 8*j+m); %%each row is updated from the previous row
          end
       A(1:d +1, j+1) = A(1:d +1, j+1)*B(j+1,1); %%% vector update
    end
    for i = 1:d+1
       f(i,n) = sum(A(i,:));
    end
n = n+1; u = f-floor(f);
end
```

Appendix C

Electrostatic interpretation of zeros of the Hermite polynomial

Consider a system of n unit charges distributed at variable points in the interval $[-\infty, +\infty]$. The potential of the system varies as the distance between the charges. Let D represent the discriminant of the distance matrix and is given by

$$D(x_1, x_2, \dots x_n) = \prod_{1<i\leq j<n} |x_i - x_j|$$

If the potential is logarithmic as is the case in 1-D the equation

$$-logD(x_1, x_2, \dots x_n) = -\sum_{1<i\leq j<n} \log\, |x_i - x_j|$$

represents the potential of the system. Let the configuration of charges be bound by the moment of inertia of the system such that it satisfies

$$n^{-1}(x_1^2 + x_2^2 + .. x_n^2) \leq L$$

where L is a positive number. This constraint is included to prevent the charges from moving to infinity. It can also be interpreted as the energy exerted by the presence of an imaginary spring.

The electrostatic equilibrium of this system is obtained at the point of minimum energy of the system and let $x_1, x_2, \dots x_n$ are the coordinates at which the charges will settle. The equilibrium points of the system can be obtained from the solution to the optimization problem defined by

$$min_x\ -\left[\sum_{1<i\leq j<n} \log\left|x_i - x_j\right| - n^{-1}(x_1^2 + x_2^2 + .. x_n^2)\right] \qquad (1)$$

The equilibrium positions can be obtained with a simple STAR-P program which enables use of data parallelism for constructing large matrices and use of inbuilt vector minimization functions. The simplest of minimization functions perturbs an element in vector x and calculates the energy of the system, repeating the process until the configuration with the lowest energy is obtained. Improved techniques use gradient information for minimization.

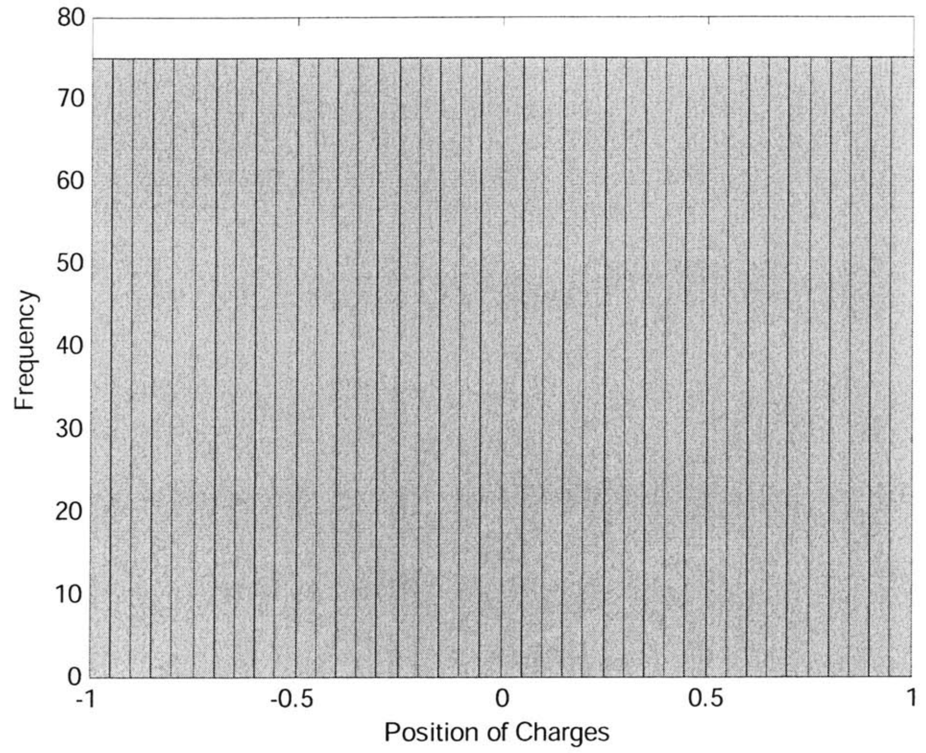

Figure C-1: Histogram of the Initial charge configuration.

3000 particles were distributed uniformly in linear space between $[-1, +1]$. Fig.1 shows the histogram of the initial position of the particles. Fig.2 describes the histogram of the equilibrium position of the particles. The histogram of the final position of the charges follows the semi-circle law. The semi-circle law was observed by Wigner for special cases of random matrices with entries that are normally distributed and symmetric.

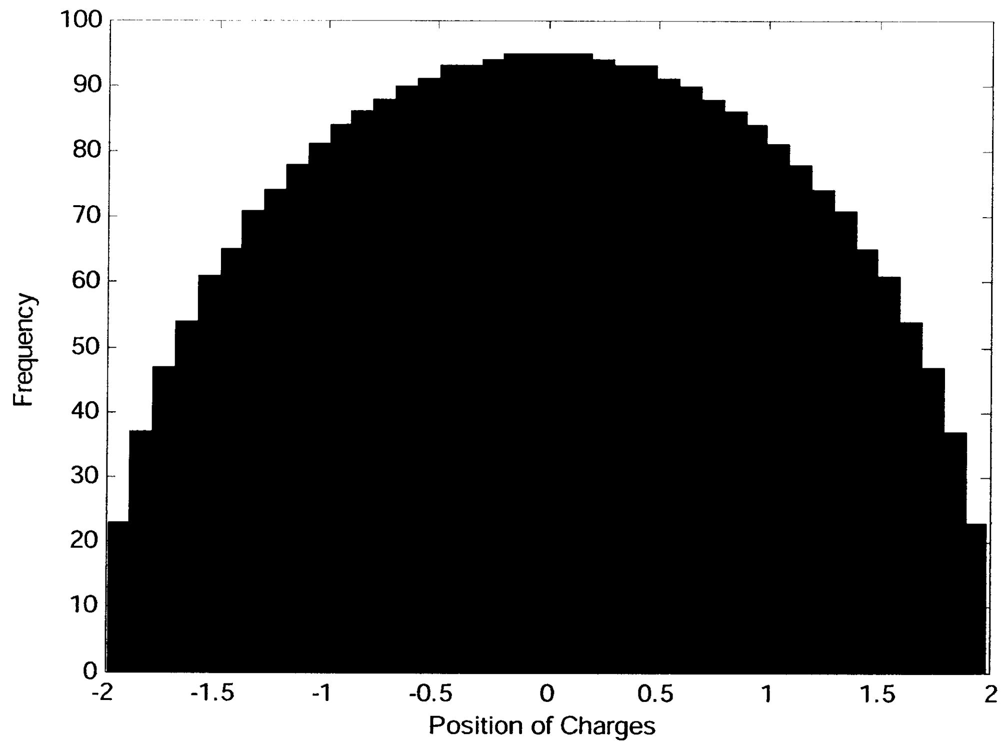

Figure C-2: Histogram of the Equilibrium charge configurations - state of minimum energy.

Another important observation is that the new position of charges are the zeros of the Hermite polynomial $H_N(c'x)$ where $c' = (2L)^{-1}\sqrt{n-1}$. This observation has been [23] proved and is revisited once again.

Consider a function $g(x)$ with zeros at $x_1, x_2, \dots x_{n-1}$ and let $f(x) = (x - x_0)g(x)$.

$$f'(x) = (x - x_0)g'(x) + g(x)$$
$$f''(x) = (x - x_0)g''(x) + 2g'(x)$$

Evaluating the equations at x_0,

$$f'(x_0) = g(x_0) \text{ and } f''(x_0) = 2g'(x_0)$$

Hence,

$$\frac{g'(x_0)}{g(x_0)} = \frac{f''(x_0)}{2f'(x_0)}$$

Hermite Polynomials solve the differential equation,

$$f''(x) - 2xf'(x) + 2nf(x) = 0$$

Evaluating the above equation at x_0,

$$f''(x_0) = 2x_0 f'(x_0)$$

Differentiating g(x) at x_0 after taking logarithms on both sides,

$$\frac{g'(x_0)}{g(x_0)} = \frac{1}{x_0 - x_1} + \frac{1}{x_0 - x_2} + \cdots \frac{1}{x_0 - x_{n-1}} = \frac{f''(x_0)}{2f'(x_0)} = x_0$$

This equation can be interpreted as the equilibrium condition of a charge at position x_0 that is attracted to the origin by a potential x_0^2 and is repelled by other charges by a potential of $-\sum_{j=1}^{n-1} log\, |x_0 - x_j|$. The equilibrium position can be determined by equating the derivative of the potential with respect to x_0 and equating the result to zero. This coincides with the equation above.

The roots of a Hermite Polynomial can be obtained from normalized eigenvalues of a symmetric tri-diagonal matrix taking the diagonal as zero and the square root of 1:(n-1) on the super and sub-diagonals and the histogram of the roots is a semicircle.

Bibliography

[1] Lennart Johnsson and Woody Lichtenstein. Block Cyclic Dense linear algebra. SIAM Journal on Scientific Computing, 1993

[2] S. Lennart Johnson. Communication efficient basic linear algebra subroutines on hypercube architectures. J. of Parallel Distributed Computing, April 1987

[3] S. Lennart Johnsson. Band matrix systems solvers on ensemble architectures. In Algorithms, Architecture, and the Future of Scientific Computation, pages 195 - 216. University of Texas Press, Austin, TX, 1985.

[4] J.Demmel, J.Dongarra et al, ScaLAPACK Users' Guide, Society for Industrial and Applied Mathematics, 1997

[5] J. Dongarra, R.van de Geijn, D.Walker. Scalability Issues in the Design of a Library for Dense Linear Algebra. Journal of Parallel and Distributed Computing, 22(3):523-537, 1994.

[6] J. J. Dongarra, R. Van De Geijn, And D. W. Walker, A Look At Scalable Dense Linear Algebra Libraries, In Proceedings Of The Scalable High-Performance Computing Conference, IEEE, Ed., IEEE Publishers, 1992, Pp. 372-379.

[7] B. Hendrickson And D. Womble, The Torus-Wrap Mapping For Dense Matrix Calculations On Massively Parallel Computers, SIAM J. Sci. Stat. Comput., 15 (1994), Pp. 1201-1226.

[8] C. Koebel, D. Loveman, R. Schreiber, G. Steele, And M. Zosel, *The High Performance* Fortran Handbook, MIT Press, Cambridge, Massachusetts, 1994.

[9] E.Anderson, J.Dongarra, A.Benzoni et al. LAPACK for distributed memory architectures: Progress Report. Parallel Processing for Scientific Computing, Fifth SIAM conference. SIAM 1991

[10] Ron Choy, Alan Edelman, John R. Gilbert, Viral Shah, David Cheng. Star-P: High Productivity Parallel Computing. June 9, 2004.

[11] Parry Husbands, Charles Isbell. The Parallel Problems Solver

[12] Blaise Barney. Introduction to Parallel Computing, Lawrence Livermore National Laborotary, 2006.

[13] Jin Suk Kim, Suk Joon Byun, A Parallel Monte Carlo Simulation on Cluster Systems for Financial Derivative Pricing. IEEE Conference, 2005.

[14] John Hull. Options, Futures and other Derivatives 5th Edition 2002

[15] Jason Fink. Monte Carlo Simulation for Advanced Option Pricing: A Simplifying Tool

[16] Interactive Supercomputing©. Getting started with STAR-P: Taking your first Test Drive

[17] Borwein, Bailey, Girgensohn. PI and Its Friends in Mathematics by Experiment: Plausible Reasoning in the 21st Century and Experiments in Mathematics: Computational Paths to Discovery. Sep 30, 2003.

[18] Charles Seife, Randomly Distributed Slices of PI. Science Magazine, Vol 293, 30 Aug 2001.

[19] Alan Edelman, Parallel Prefix. Math 18.337 MIT Lecture 3, Spring 2006.

[20] Ladner, Fischer, Parallel Prefix Computation. Journal of ACM, Vol 27, Oct 1980.

[21] Kuszmaul, A Segmented Parallel Prefix VLSI Circuit with Small Delays for Small Segments. SPAA'05, July 2005.

[22] Sidhartha Chaterjee, Gilbert. Et.al, Generating Local Addresses and Communication Sets for Data-Parallel Programs. ACM, USA 1993

[23] Szego, Orthogonal Polynomials – Chapter: Zeros of Orthogonal Polynomials, 1975