Benchmarking: More aspects of High Performance Computing

by

Rahul Ravindrudu

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Ricky Kendall, Co-major Professor
Simanta Mitra, Co-major Professor
Mark Gordon

Iowa State University

Ames, Iowa

2004

Graduate College
Iowa State University

This is to certify that the master's thesis of

Rahul Ravindrudu

has met the thesis requirements of Iowa State University

_____

Co-major Professor

_____

Co-major Professor

_____

For the Major Program

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

IS-T 2196

CHAPTER 1. Introduction

The ever increasing need for faster and more powerful computers, coupled with the advent of fairly cheap microprocessors, has prompted considerable interest in massively parallel processing systems. Computational power has reached a plateau at the current state of technology for single processor systems, due to certain fundamental limits (i.e. the speed of light and the width of the atom being approached).

The past 15 years therefore, has been a renaissance in the high-performance computer architecture field. Virtually every reasonable parallel computer architecture has been implemented as a prototype or commercial product with most of them aimed at solving scientific computational problems. The variety of high-performance architectures ranges from large vector computers with a limited number of processors that share a common memory to machines with thousands of very simple processors and distributed memories. A problem with almost all of these machines the enormous performance range, sometimes potentially a factor of hundred or more, depending on the suitability of a certain piece of code for the underlying architecture. Peak computational rates are often achieved by executing in special modes, exploiting novel architectural features like vector hardware or multiple CPUs. Although this increases the potential power of a system, it adds a level of difficulty to performance evaluation methods, which must consider the relative values and contributions of the various components. This complex structure makes the space of design decisions too large and complicated for analytical prediction and validation.

In this context, the benchmarking of high performance computer systems has rightly become an active area of investigation. Implicit in every well-known scientific benchmark

is the suggestion that the benchmark somehow captures the essence of many important scientific computations and applications [1]. Just what are the important scientific computations and in what sense is their essence represented by a benchmark are questions that are typically at the center of any benchmark controversy. Generally, investigating the performance of a system through benchmarks has three major objectives 1) provide input for improving the design of future advanced computer architectures 2) permit manufacturers to state the capabilities of their systems in a comparable fashion 3) access the suitability of a given architecture for a class of applications.

High Performance Linpack, (HPL), is a software package that solves a (random) dense linear system in double precision arithmetic on distributed-memory computers. It can thus be regarded as a portable as well as freely available implementation of the High Performance Computing Linpack Benchmark [2, 3, 4]. The **original algorithm** used by HPL can be summarized by the following keywords: Two-dimensional block-cyclic data distribution - Right-looking variant of the LU factorization with row partial pivoting featuring multiple look-ahead depths - Recursive panel factorization with pivot search and column broadcast combined - Various virtual panel broadcast topologies - bandwidth reducing swap-broadcast algorithm - backward substitution with look-ahead of depth 1. The HPL package provides a testing and timing program to quantify the accuracy of the obtained solution as well as the time it took to compute it. The best performance achievable by this software on your system depends on a large variety of factors. Nonetheless, with some restrictive assumptions on the the interconnection network, the algorithm described here and the implementation are scalable in the sense that their parallel efficiency is maintained constant with respect to the per processor memory usage. The parallel efficiency of the entire algorithm is estimated according to the following machine model.

Distributed-memory computers consist of processors that are connected using a message passing interconnection network. Each processor has its own memory called the

local memory, which is accessible only to that processor. The interconnection network of this machine model is static, meaning that it consists of point-to-point communication links among processors. This type of network is also referred to as a direct network as opposed to dynamic networks. The latter are constructed from switches and communication links. The model assumes that a processor can send or receive data on only one of its communication ports at a time (assuming it has more than one). In the literature [18], this assumption is also referred to as the one-port communication model. Finally, the model assumes that the communication links are bi-directional. In particular, a processor can send a message while receiving another message from the processor it is sending to at the same time.

HPL requires an implementation of the Message Passing Interface (MPI) [5, 6] standard and the Basic Linear Algebra Subprograms (BLAS) [7]. These implmentations are how vendors are able to tune the performance of HPL on their system offerings.

CHAPTER 2. Motivation

A benchmark was originally a mark on some permanent object indicating elevation. The mark served as a point of reference from which measurements could be made in topological surveys and tidal observations. Contemporary usage indicates an object that serves as a standard by which others can be measured. Analogously, benchmarking of computer systems is intended to measure new systems relative to a reference point on current systems. In particular, benchmarks are standardized computer programs for which there is history of measurement data for executions of the programs (typically timings) with specifically defined input and reproducible output that allow the comparisons for a wide range of computer systems. What distinguishes a benchmark from an ordinary program is a general consensus of opinion within the industry and research communities that the benchmark exercises a computer well.

Historically, benchmarking has mainly been employed for system procurements. It will certainly maintain its value in that area as it expands to become the experimental basis for a developing theory of supercomputer and multiprocessor performance evaluation. The number of benchmarks currently used is growing day by day. Every new benchmark is created with the expectation that it will become a standard of the industry and that manufacturers and customers will use it as the definitive test to evaluate the performance of computer systems with similar architectures. Most procurements use a variety of these "standard" benchmarks and applications specific to the purchasing site.

Statistics on high-performance computers are of major interest to manufacturers, users, and potential users. These people wish to know not only the number of systems in-

stalled, but also the location of the various supercomputers within the high-performance computing community and the applications for which a computer system is being used. Such statistics can facilitate the establishment of collaborations, the exchange of data, software and expertise as well as provide a better understanding of the high-performance computer market.

Statistical lists of supercomputers are not new. Every year since 1986 Hans Meuer has published system counts [8] of the major vector computer manufacturers, based principally on those at the Mannheim Supercomputer Seminar. However, statistics based merely on the name of the manufacturer are no longer useful. New statistics are required that reflect the diversification of supercomputers, the enormous performance difference between low-end and high-end models, the increasing availability of massively parallel processing (MPP) systems, and the strong increase in computing power of the high-end models of workstations with symmetric multiple processors (SMP).

2.1 Top 500

To provide this new statistical foundation, the Top 500 project was started in 1993 to assemble and maintain a list of the 500 most powerful computer systems. This list has been compiled twice a year since June 1993 with the help of high-performance computer experts, computational scientists, manufacturers, and the Internet community in general who responded to a questionnaire that was sent out. The project has also used parts of statistical lists published by others for different purposes.

In the present list (which is commonly called the TOP500), computers listed on it are ranked by their performance on the LINPACK Benchmark. The Linpack benchmark is run using HPL, A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. This list is updated half-yearly to keep track with the evolution of computers. The list is freely available at *http://www.top500.org/*

where the users can create additional sublists and statistics out of the TOP500 database on their own.

2.1.1 TOP500 Description

The TOP500 table shows the 500 most powerful commercially available computer systems known to us. To keep the list as compact as possible, only a part of the information is shown here:

- N_{world} - Position within the TOP500 ranking
- Manufacturer - Manufacturer or vendor
- Computer - Type indicated by manufacturer or vendor
- Installation Site - Customer
- Location - Location and country
- Year - Year of installation/last major update
- Field of Application
- #Proc. - Number of processors
- R_{max} - Maximal LINPACK performance achieved
- R_{peak} - Theoretical peak performance
- N_{max} - Problem size for achieving R_{max}
- $N_{\frac{1}{2}}$ - Problem size for achieving half of R_{max}

In addition to cross checking different sources of information, a statistical representative sample is randomly selected from the first 500 systems of the database. For these

systems, the given information is verified, by asking the supplier of the information to establish direct contact between the installation site and the verifier. This gives them basic information about the quality of the list in total.

As the TOP500 should provide a basis for statistics on the market of high-performance computers, the number of systems installed at vendor sites is limited. This is done for each vendor separately by limiting the accumulated performance of systems at vendor sites to a maximum of 5% of the total accumulated installed performance of this vendor. Rounding is done in favor of the vendor in question.

In the following table, the computers are ordered first by their R_{max} value. In the case of equal performances (R_{max} value) for different computers, they are ordered by R_{peak}. For sites that have the same computer, the order is by memory size and then alphabetically. The table provided is just a sample of the Top500 list and not the complete list.

2.2 Linpack

As a yardstick of performance we are using the "best" performance as measured by the LINPACK Benchmark [3, 4]. LINPACK was chosen because it is widely used and performance numbers are available for almost all relevant systems.

The LINPACK Benchmark was introduced by Jack Dongarra. A detailed description as well as a list of performance results on a wide variety of machines is available in postscript form from netlib. A parallel implementation of the Linpack benchmark (HPL) and instructions on how to run it can be found at *http://www.netlib.org/benchmark/hpl/*.

The benchmark used in the LINPACK Benchmark is to solve a dense system of linear equations. More details about solving systems of linear equations can be found in [10]. For the TOP500, the version of the benchmark used is the one, that allows the user to scale the size of the problem and to optimize the software in order to achieve the

Rank	Site Country/Year	Computer/Processors Manufacturer	Computer Family Model	Installation Type/Area	R_{max} R_{peak}	N_{max} $n_{\frac{1}{2}}$
1	Earth Simulator Center Japan/2002	Earth-Simulator / 5120 NEC	NEC Vector SX6	Research	35860 40960	1.0752e+06 266240
2	Lawrence Livermore National Laboratory United States/2004	Thunder Intel Itanium2 Tiger4 1.4GHz - Quadrics / 4096 California Digital Corporation	NOW - Intel Itanium Itanium2 Tiger4 Cluster - Quadrics	Research	19940 22938	975000 110000
3	Los Alamos National Laboratory United States/2002	ASCI Q - AlphaServer SC45, 1.25 GHz / 8192 HP	HP AlphaServer Alpha-Server-Cluster	Research	13880 20480	633000 225000
4	IBM - Rochester United States/2004	BlueGene/L DD1 Prototype (0.5GHz PowerPC 440 w/Custom) / 8192 IBM/ LLNL	IBM BlueGene/L BlueGene/L	Vendor	11680 16384	331775
5	NCSA United States/2003	Tungsten PowerEdge 1750, P4 Xeon 3.06 GHz, Myrinet / 2500 Dell	Dell Cluster PowerEdge 1750, Myrinet	Academic	9819 15300	630000

Table 2.1 TOP500 List for June 2004: Only the top 5 are included here

best performance for a given machine. This performance does not reflect the overall performance of a given system, as no single number ever can. It does, however, reflect the performance of a dedicated system for solving a dense system of linear equations. Since the problem is very regular, the performance achieved is quite high, and the performance numbers give a good correction to peak performance. Table 2.1 shows the 5 most powerful systems from the Top500 list.

By measuring the actual performance for different problem sizes N, a user can get not only the maximal achieved performance R_{max} for the problem size N_{max} but also the problem size $N_{\frac{1}{2}}$ where half of the performance R_{max} is achieved. These numbers together with the theoretical peak performance R_{peak} are the numbers given in the TOP500. In an attempt to obtain uniformity across all computers in performance reporting, the algorithm used in solving the system of equations in the benchmark procedure must confirm to the standard operation count for LU factorization with partial pivoting. In particular, the operation count for the algorithm must be $\frac{2}{3}N^3 + O(N^2)$ floating point operations. This excludes the use of a fast matrix multiply algorithm like "Strassen's Method" [11, 12]. This is done to provide a comparable set of performance numbers across all computers. If in the future a more realistic metric finds widespread usage, so that numbers for all systems in question are available, the benchmark authors may convert to that performance measure.

2.3 Basic Linear Algebra Subprograms

BLAS are a set of subroutines written in Fortran which provide a standard API for simple linear algebra operations. BLAS are split in 3 levels:

- level 1: vector operations,
- level 2: matrix-vector operations,

- level 3: matrix-matrix operations and triangular solve.

BLAS contains subprograms for basic operations on vectors and matrices. BLAS was designed to be used as a building block in other codes, for example LAPACK. The source code for BLAS is available through Netlib. However, many computer vendors will have a special version of BLAS tuned for maximal speed and efficiency on their computer. This is one of the main advantages of BLAS: the calling sequences are standardized so that programs that call BLAS will work on any computer that has BLAS installed. If you have a fast version of BLAS, you will also get high performance on all programs that call BLAS. Hence BLAS provides a simple and portable way to achieve high performance for calculations involving linear algebra. LAPACK is a higher-level package built on the same ideas.

2.3.1 Level 1

Lawson [19], proposed a number of basic linear algebra subroutines which are generally accepted as the so-called BLAS subroutines. In most vector computers efficient implementations of these subroutines are available, which fully utilize the vectorial and parallel properties of these computers. In additon, the source of these subroutines is available so that they can be implemented at any other computer. These consist mainly of vector-scalar, norm, inner-product and rotation operations. These are $O(N)$ operations.

2.3.2 Level 2

In 1985 and 1988 the level two and three BLAS-subroutines [20] [21, 22], respectively, have been introduced. The level two routines consist of matrix-vector routines and Special Matrix Solvers. Many of these routines are also optimized for upper-triangular and diagonal matrices and others. These are $O(N^2)$ operations.

2.3.3 Level 3

These routines consist of matrix-matrix operations. These are $O(N^3)$ operations.

CHAPTER 3. Background on Linear Algebra

Chapter 1 mentions that HPL solves a dense linear system. This chapter provides some basic knowledge on how to solve a Linear Equation of the form $Ax = b$.

3.1 Gausssian Elimination

The most commonly used direct method for solving general linear systems is Gaussian elimination with partial pivoting, which in modern terms is called LU decomposition with pivoting, or LU factorization with pivoting [27]. Gaussian elimination is commonly taught by adjoining the right hand side vector b to the matrix, then performing row combinations that would zero out the subdiagonal entries of the matrix A. LU decomposition (or "LU factorization") does the same operations, but ignores the right hand side vector until after the matrix has been processed. In particular: Let A be an NxN, nonsingular matrix. Gaussian elimination with partial pivoting gives the factorization $PA = LU$, where

- P is a permutation matrix, that is, it has exactly one 1 in each row and column, and zeros elsewhere. $P^{-1} = P^T$, and can be stored as an integer vector of length N.
- L is unit lower triangular matrix (ones on main diagonal, zero above main diagonal)
- U is upper triangular matrix (zeros below the main diagonal)

Solving $Ax = b$ then becomes $LUx = Pb$, since $PAx = Pb$. Three steps can be taken in solving $LUx = Pb$.

1. Set $d = Pb$, either by shuffling entries of b, or by accessing via indirect addressing using a permutation vector. In setting $d = Pb$, we can sometimes overwrite b with its permuted entries, depending on how P is represented.

2. Solve $Ly = d$ (unit lower triangular system)

3. Solve $Ux = y$ (upper triangular system)

We look later in some detail at all three stages. However, note that the triangular solves can be implemented by overwriting the right hand side vectors with the solution as we go along, rather than introducing new vectors y and d. In that case the three steps look more parsimonious with space:

1. Set $x = Pb$

2. Solve $Lx = x$ (unit lower triangular system). The x on right-hand side is replaced with the solution. Hence the same vector x on both sides.

3. Solve $Ux = x$ (upper triangular system). As in the previous step, the right-hand side is replaced with the solution.

and if the original right hand side vector b is not needed later on, x can be replaced with b in all three steps.

3.2 LU Factorization

3.2.1 Special Matrices

Three types of matrices will be used below. The matrix U is an upper triangular matrix if the elements of U, $u_{ij} = 0$ if $i > j$. Similarly, a lower triangular matrix L,

with elements $l_{ij} = 0$ if $i < j$ and with the additional condition that $l_{i,j} = 1$ if $i = j$. A third type of matrix is also required below which is a permutation matrix, P. P has the property that $PP = I$. P contains the row exchange information. A sample P matrix:

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

There are two types of representations for the above matrix using a single integer vector. One is to use a permutation vector and the other is to use a pivot vector.The first stores P as a set of integers p_i that represent position of x_i in $y = Px$:

$$\begin{array}{lccccc} i: & 1 & 2 & 3 & 4 & 5 \\ P_i: & 3 & 2 & 1 & 5 & 4 \end{array}$$

gives $y = (x_3, x_2, x_1, x_5, x_4)^T$. We can compute y by

```
for i = 1:n
    y(i) = x(p(i))
end for i
```

A second way of representing permutation matrices P is with pivot vectors, and it is these which we use in Gaussian elimination. This is an integer array *piv* of length n, applied to a vector x in $y = Px$ using

```
y = x
for k = 1:n
```

```
        swap y(k) and y(piv(k)) in the vector y
    end for
```

This can of course be better done by overwriting and avoid using another vector y.

3.2.2 Factorization Details

The classical "pointwise" algorithm is presented here and later extended to a block oriented algorithm. The classical algorithm follows a three phase sequence on each step: find the pivot, interchange rows to bring the pivot to the diagonal position, scale the subdiagonal entries in the column, then update the rest of the matrix. Carrying out these operations with overwriting of the array A gives the L and U factors in the corresponding parts of A. L is unit lower triangular, so it's diagonal is not stored and both upper and lower triangular matrices fit nicely. Less obscurely, the algorithm without pivoting is:

LU factorization: Version 1

```
for k = 1:n-1
  for i = k+1:n
      A(i,k) = A(i,k)/A(k,k)
  end for
  for i = k+1:n
      for j = k+1:n
        A(i,j) = A(i,j) - A(i,k)* A(k,j)
      end for
  end for
end for
```

The above notation for the algorithm can be shortened by following the Matlab notation. Briefly, the notations are

1. $A(r:s,i)$ refers to rows r through s in column i,

2. $A(r, i:j)$ refers to columns i through j in row r,

3. $A(r:s, i:j)$ refers to rows r through s in columns i through j.

Note that (1) is a column vector, (2) is a row vector, while (3) is a 2D submatrix. Now the algorithm above can be stated more succinctly as

LU Factorization: Rank-1 Update Form

```
for k = 1:n-1
  A(k+1:n,k) = A(k+1:n,k)/A(k,k)
  A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - A(k+1:n,k)* A(k,k+1:n)
end for
```

The last form more clearly shows what the operations involved are. Each iteration is a vector scaling of the subdiagonal part of column k, followed by a $rank-1$ update of the trailing part of the matrix. Graphically, the **LU rank-1 update form** procedure is:

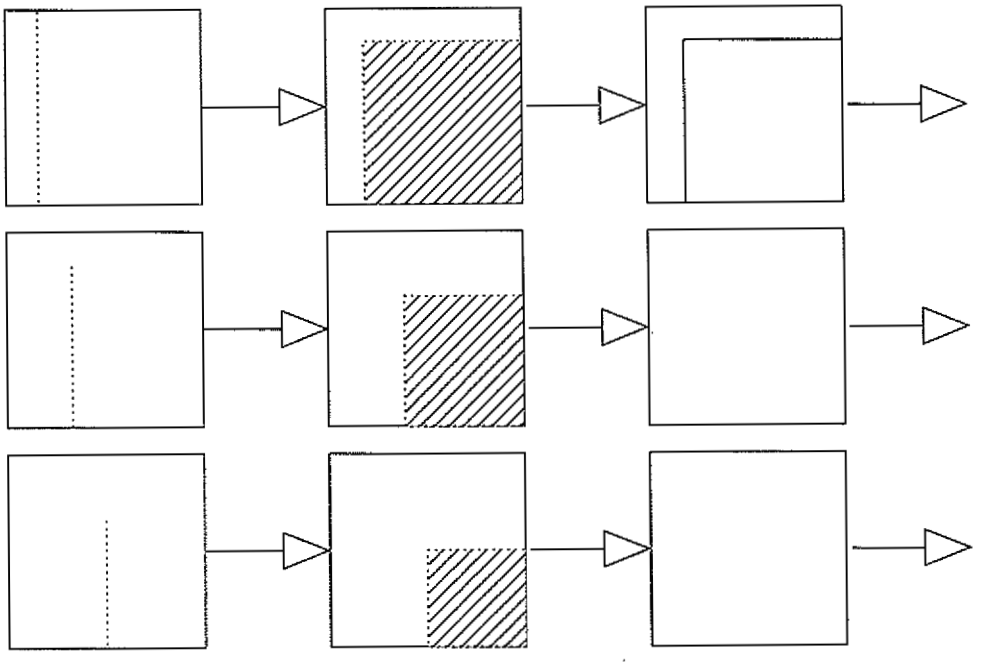

Figure 3.1 LU Factorization: Rank-1 Update

The $rank - 1$ update has a high memory reference to flop ratio, hence does not perform well as a computational kernel. A more efficient version uses a common idea in computer science, called lazy evaluation. In numerical linear algebra it has a longer history, and is called "deferred updates". This means apply the updating information to a part of the matrix only when it is actually needed. The algorithm results by equating parts of A, L, and U in a partitioned way. Graphically, we have

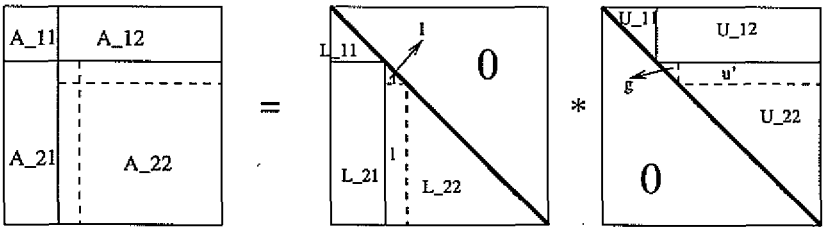

Figure 3.2 LU Factorization Derivation

Known values: $A\_ij, L\_11, L\_21, U\_11, U\_12$

Find on step k: l, g, u'

Unknown until later steps: L_22, U_22

First row of $A\_22 =$ (first row of $L\_21) * U\_12 + 1 * (g, u')$

First col of $A\_22 = L\_21 *$ (first col of $U\_12) + g * \begin{pmatrix} 1 \\ l \end{pmatrix}$

The diagonal entry in column k of L is known to be one, of course. Note that by sequencing the finding of g, u', and l as shown above defines an algorithm which recursively works on the trailing subblock. The resulting algorithm is

LU Factorization: Matrix-Vector Product Form

```
for k = 1:n-1
   A(k:n,k) = A(k:n,k) - A(k:n,1:k-1)*A(1:k-1,k)
   A(k,k+1:n) = A(k,k+1:n) - A(k,1:k-1)*A(1:k-1,k+1:n)
   A(k+1:n,k) = A(k+1:n,k)/A(k,k)
end for
```

The computational kernels are (in order) matrix-vector product, vector-matrix product, and vector scaling. So the rank-1 update has been replaced by matrix-vector products, which have half the memory reference to flop ratio of a $rank-1$ update. This is a big win on a cache-based machine. Minor point performance-wise, but big in terms of correctness: there is a final update of $A(n,n)$ which has been left off above.

3.2.3 Triangular Systems

Once we have the LU factors we need to solve two triangular systems. Consider lower triangular systems; upper triangular systems can be handled similarly. Also, in Gaussian elimination the lower triangular matrices are always unit lower triangular, meaning that they have ones on the diagonal. Now let's solve a particular lower triangular system:

$$\begin{pmatrix} 2 & 0 & 0 & 0 \\ 1 & 3 & 0 & 0 \\ -3 & 2 & 5 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} -2 \\ 2 \\ 15 \\ 0 \end{pmatrix}$$

Everybody would begin with $2x_1 = -2$ giving $x_1 = -1$, and then solve in order for x2, x3,... .

Two possibilities occur next: Plug in already-known values as you need them in order to solve for x_i, or when you find a value, plug it into all the remaining equations before going on to find the next value. Gives two algorithms:

Row-oriented Lower Triangular Solve

```
for i = 1:n
    for j = 1:i-1
        b(i) = b(i) - l(i,j)*x(j)
    end for j
    x(i) = b(i)/l(i,i)
end for i
```

This version has an inner product as innermost loop and accesses rows of L. The second version is

Column-oriented Lower Triangular Solve

```
for j = 1:n
    x(j) = b(j)/l(j,j)
    for i = j+1:n
        b(i) = b(i) - l(i,j)*x(j)
    end for i
end for j
```

This version has a daxpy as innermost loop (bad), and accesses columns of L.

Algorithm 1 is sometimes called a row sweep algorithm, and Algorithm 2 is called a column sweep algorithm. Both algorithms have a block version possible by simply treating l_{ij} as an m x m block L_{ij} [15]. Then

- $x_j = b_j/l_{jj}$ is replaced by: solve $L_{jj}x_j = b_j$, where x_j, b_j are now vectors.
- $b_i = b_i - L_{ij}x_j$ becomes a matrix*vector (BLAS-2) operation.

Block Row-oriented Lower Triangular Solve

```
for i = 1:n
    for j = 1:i-1
        b(i) = b(i) - L(i,j)*x(j)
    end for j
    solve L(i,i) x(i) = b(i) for x(i).
end for i
```

Block Column-oriented Lower Triangular Solve

```
for j = 1:n
    solve L(j,j) x(j) = b(j) for x(j)
    for i = j+1:n
        b(i) = b(i) - L(i,j)*x(j)
    end for i
end for j
```

The minimal number of memory references are to read L and b and write x. This gives $n(n+1)/2 + n + n = (n^2 + 5n)/2$ memory references. The number of flops involved is $sum_{j=1:n}[2*(n-j)+1] = n^2$. So the ratio of memory references to flops is $1/2+(5/2)n$, a typical BLAS-2 number.

Performing triangular solves is not too hard, and much time should not be spent optimizing them. The reason is simple: computing the LU factorization requires $(2/3)n^3$ flops, while the triangular solves require only $4n^2$ ($2n^2$ each for L and U). So our efforts should go for the most expensive step: LU factorization.

3.2.4 Pivoting in LU Factorization

When using either of the two versions ($rank-1$ update and matrix-vector product) of LU factorization, the division by $A(k,k)$ in the scaling operation can cause problems if $A(k,k)$ is small in absolute value. Pivoting means bringing a larger element into that position by swapping rows or columns in the matrix. Partial pivoting means doing only row swaps; LU factorization is "usually stable" when partial pivoting is used [14]. However, there are classes of problems (some two point boundary value problems in ODE's is one example) for which it can fail. For the $rank-1$ version this gives

LU Factorization: Rank-1 Update with Pivoting Form

```
for k = 1:n
  p = index of max element in |A(k:n,k)|
```

```
  piv(k) = p
  swap rows k and p of A: A(k,:) <---> A(p,:)
  A(k+1:n,k) = A(k+1:n,k)/A(k,k)
  A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - A(k+1:n,k)* A(k,k+1:n)
end for
```

Note that the first step picks out the largest element in the k^{th} column among the remaining unreduced part of the matrix. Also, note that we swap the entire rows k and p of the array A; this will also swap the corresponding parts of L and U since we are overwriting A with those factors. It is another magic feature of LU factorization that this makes the factors come out "in the right order" as well.

Next, here is a matrix-vector product formulation with pivoting, but stated for an m by n matrix B, with $m >= n$.

LU Factorization: Matrix-Vector Product with Pivoting for mxn Matrix

```
for k = 1:n-1
   B(k:m,k) = B(k:m,k) - B(k:m,1:k-1)*B(1:k-1,k)
   p = index of max element in |B(k:m,k)|
   piv(k) = p
   swap rows k and p of B: B(k,:) <---> B(p,:)
   B(k,k+1:n) = B(k,k+1:n) - B(k,1:k-1)*B(1:k-1,k+1:n)
   B(k+1:m,k) = B(k+1:m,k)/B(k,k)
end for
B(n:m,n) = B(n:m,n) - B(n:m,1:n-1)*B(1:n-1,n)
B(n+1:m,n) = B(n+1:m,n)/B(n,n)
```

Note the last two steps are new ingredients - they consist of applying the remaining updates to the rest of the matrix. Also note that we can do another pivot step in case $B(n,n)$ is small.

3.2.5 Blocked LU Factorization

Moving from the $rank - 1$ update version of LU factorization to a matrix-vector version was done to get a better BLAS-2 kernel (in terms of load/store analysis) as the workhorse. But since the best BLAS kernel is matrix-matrix multiplication, we should try to get a BLAS-3 version. This can be done by blocking [15].

Blocked versions of LU factorization can be derived in the same way as was done to get the matrix-vector product version. Partition the matrix into block columns, each of width v. We can find v columns of L and v rows of U on each "iteration". Suppose that k columns/rows have already been found. The picture is

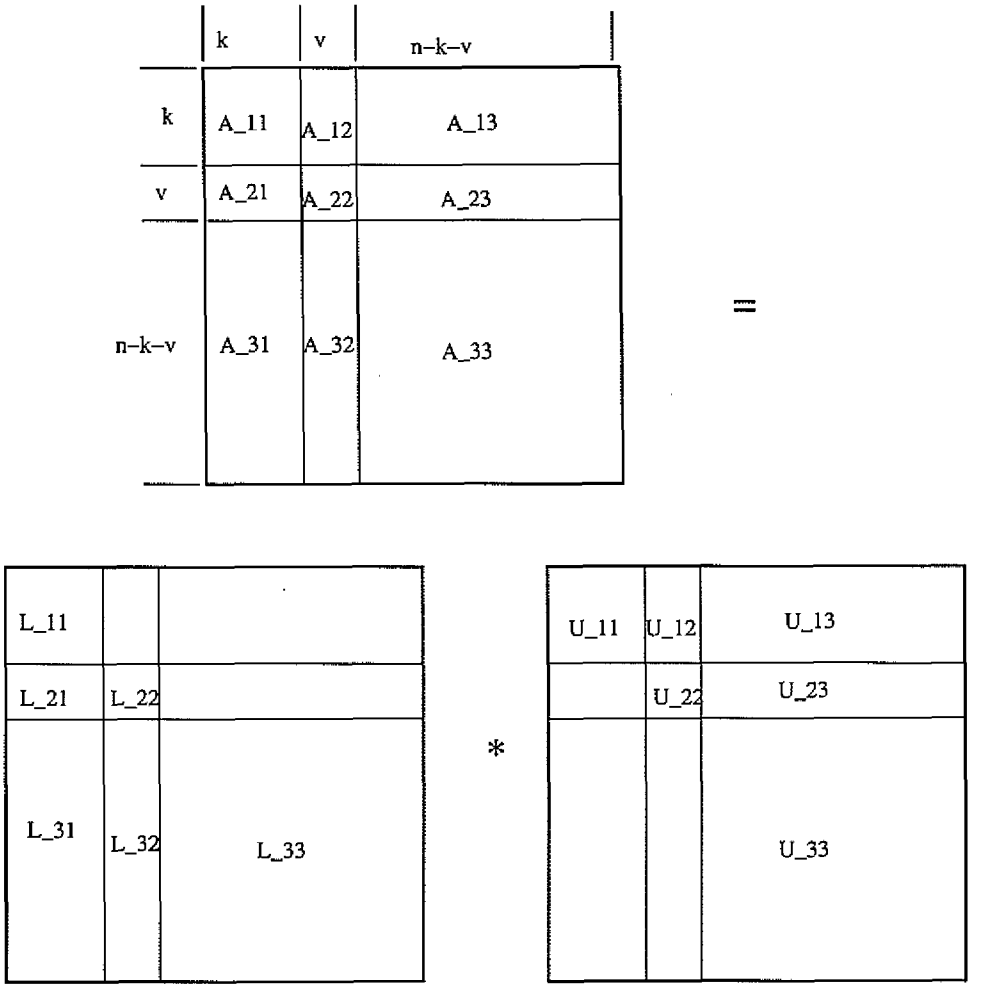

Figure 3.3 LU Factorization Blocked

where

```
Known Values: A, L_11, L_21, L_31, U_11, U_12, U_13
Values to find on this step: L_22, L_32, U_22, U_23
Values to find on later steps: L_33, U_33
```

This is the same as the last diagram, but with block columns replacing the unknown row/column to find at this step. In more detail showing the sizes and shapes of the systems,

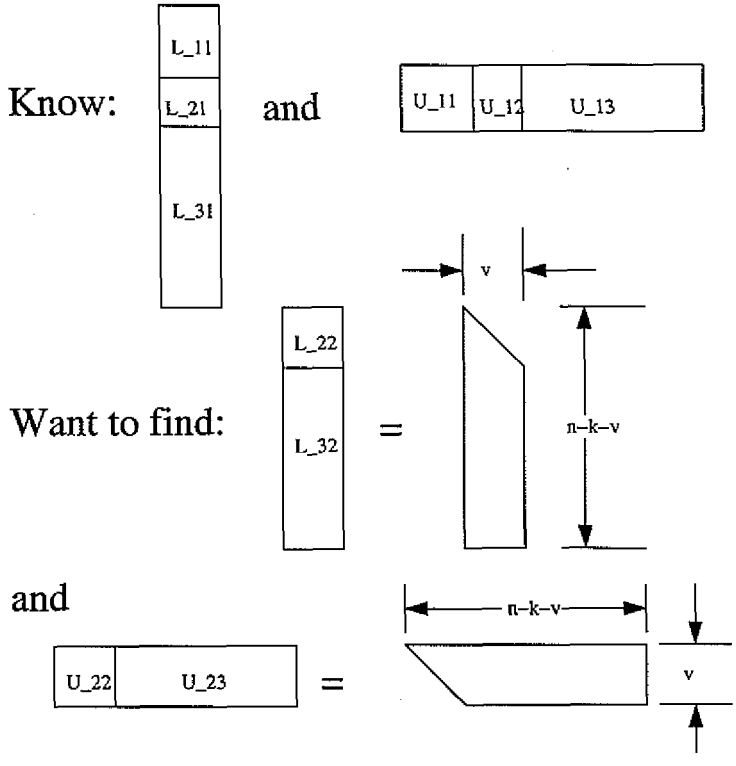

Figure 3.4 LU Factorization Blocked Details

Sequencing the known and unknowns gives:

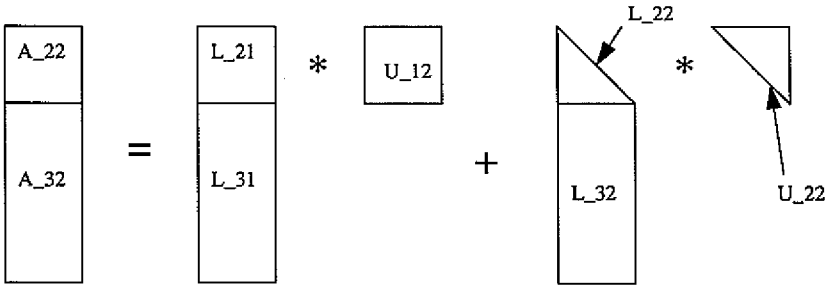

Figure 3.5 Result of multiplying L*U in block form shown above and equating with the corresponding blocks of A

Collecting unknowns on left hand side and knowns on right hand side gives a two step process for L_{22}, L_{32}, U_{22}:

1. Update $\begin{bmatrix} A_{22} \\ -- \\ A_{32} \end{bmatrix} = \begin{bmatrix} A_{22} \\ -- \\ A_{32} \end{bmatrix} - \begin{bmatrix} L_{21} \\ -- \\ L_{31} \end{bmatrix} * [U_{12}]$

2. Perform LU factorization on the $(n-k)$ by v matrix $\begin{bmatrix} A_{22} \\ -- \\ A_{32} \end{bmatrix}$

   Still need to find U_{23}. Equating (2,3) block in $A = LU$ gives

$$A_{23} = L_{21}U_{13} + L_{22}U_{23}$$

   and two more steps in the factorization:

3. Update $A_{23} = A_{23} - L_{21}U_{13}$

4. Solve the v by v lower triangular system with $n-k-v$ right hand sides:

$$L_{22} * U_{23} = A_{23}$$

   This step is carried out with overwriting of A_{23}

The reason for using a matrix-vector product earlier was that it gave better computational kernels. However, when using a blocked version the $rank-1$ update becomes a $rank-v$ update and so becomes matrix-matrix multiplication. That kernel is the best among the BLAS in terms of its ratio of memory references to flops, so we can safely use it in a high-performance version of LU factorization, instead of the version above based on the blocked matrix-vector version. Finally, all this does is produce the LU factorization. Then comes the triangular solver.

After LU factorization is complete, $Ax = b$ may be solved as mentioned above. The equation $Lc = b'$ may be solved by a method referred to as a forward solve, as shown:

$$\begin{aligned}
1 * c_1 + 0 * c_2 + 0 * c_3 &= b'_1 \\
l_{2,1} * c_1 + 1 * c_2 + 0 * c_3 &= b'_2 \\
l_{3,1} * c_1 + l_{3,2} * c_2 + 1 * c_3 &= b'_3 \\
\Rightarrow c_1 &= b'_1 \\
\Rightarrow c_2 &= b'_2 - l_{2,1} * c_1 \\
\Rightarrow c_3 &= b'_3 - l_{3,1} * c_1 - l_{3,2} * c_2
\end{aligned}$$

Once c is obtained, the equation $Ux = c$ may be solved in a similar way.

3.2.6 LU Decomposition Variants

Three natural variants occur for block LU decomposition: right-looking, left-looking and Crout [27]. The left-looking variant computes on block column at a time using previously computed columns. The right-looking variant computes a block row column at each step and used them to update the trailing submatrix. The right-looking is also known as the recursive algorithm. The terms right and left refer to the regions of data access, and Crout represents a hybrid of the left- and right- looking versions. The

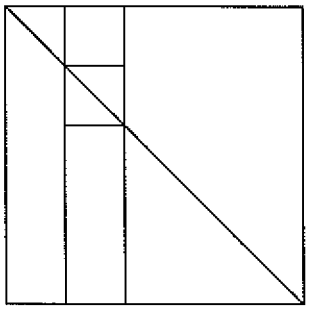

Figure 3.6 Left-Looking LU Algorithm

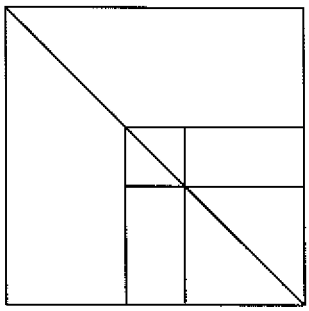

Figure 3.7 Right-Looking LU Algorithm

graphical representation of the algorithms are in Figures 3.6, 3.7, and 3.8.

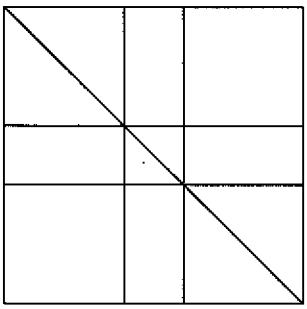

Figure 3.8 Crout LU Algorithm

CHAPTER 4. I/O in High Performance Computing

High performance applications have unique needs that commodity hardware and software developed for larger markets do not always meet. These needs vary widely between different kinds of applications, so before developing sophisticated I/O techniques, the demands of the application must first be understood [24, 26].

Until recently, most applications developed for parallel machines avoided I/O as much as possible (distributed databases have been a notable exception). Typical parallel applications (usually scientific programs) would perform I/O only at the beginning and the end of execution with the possible exception of infrequent checkpoints. The paper Oldfield [30] surveys various scientific applications using parallel I/O. This has been changing: I/O-intensive parallel programs have emerged as one of the leading consumers of cycles on parallel machines. This change has been driven by two trends. First, parallel scientific applications are being used to process larger datasets that do not fit in memory. Second, a large number of parallel machines are being used for non-scientific applications, for example databases, data mining, web servers for busy web sites (e.g. Google, Yahoo and NCSA). Characterization of these I/O intensive applications is an important problem that has tremendous effects on the design of I/O subsystems, operation systems and filesystems. Papers by Rosti [28] and Smirni [29] provide an idea of the I/O requirements of scientific applications.

4.1 High Performance I/O Requirements

Three categories of applications that demand good I/O performance are database management systems (DBMSs), multimedia applications, and scientific simulations. DBMSs manage collections of data records, usually stored on disk. The collections can be quite large, containing millions of records or more, and the DBMS must often search for individual records that meet certain criteria. Depending on the task at hand, the DBMS may examine every record in the database, or it may look at a subset of records scattered across the database. Reading or writing data in small pieces, roughly less than 1000 bytes, is called fine-grained access, and it is less efficient than accessing data in larger pieces.

Multimedia applications process sound, still images, and video data. Unlike a DBMS, a multimedia application can often access large blocks of data in a predictable sequence. For example, if the application is presenting a video, it can determine well in advance what data to read, so it can schedule disk accesses in an efficient sequence. The user may search forward and backward through the video or take different branches through an interactive story, but even then the program can usually access data in large blocks. However, unlike many other applications, multimedia programs often require the I/O system to read the data at no less than a specified minimum rate; this rate must be sustained to make the sound and pictures move smoothly.

In scientific applications, the granularity may be coarse or fine, and the access patterns may be predictable or random. Many scientific applications read and write data in well-defined phases: the program will read some data, compute for a while, then write some data. Usually, a program will continue computing in many steps, writing data each time the program has completed a specified number of steps. In a parallel application, the individual tasks within a job will often write their data at the same time. Parallel I/O libraries can take advantage of this synchrony to improve performance. As example

is a paper by Dror Feitelson [31].

An important difference between scientific applications and database or multimedia applications is that the latter two are often designed specifically to do I/O. The application designers recognize that accessing data in external storage is essential to the program's performance. In scientific applications, on the other hand, the central task is usually a numerically intensive computation involving data that is already in memory. Moving the data between memory and external storage is a secondary problem. Therefore, designers of scientific codes may be less interested in I/O issues than in issues of numerical accuracy and computational efficiency. Also, many database and multimedia applications focus on reading data, while many scientific applications focus on writing and reading data.

The two main requirements for most I/O-intensive applications are I/O speed and storage capacity. Like other computer components, such as processors, memory, and interconnection networks, external storage devices continue to improve at a phenomenal rate. However, these components are all improving at different rates, which creates an imbalance between the performance of different computer subsystems.

Ideally, computer architects and system programmers could improve I/O performance without forcing application developers to change their programs. However, tuning the overall performance of an application requires an understanding of all the major parts of a computer's architecture, including not only the CPU, cache, and memory, but also the I/O system. The I/O system includes storage devices, interconnection networks, file systems, and one or more I/O programming libraries.

Computers store data on a variety of media, including electronic memory, magnetic disk, optical disk, and tape. These media are often classified in a three-level hierarchy, which distinguishes then according to their volatility, cost, access time, and typical use. Primary storage, the top level of the hierarchy, includes all types of electronic memory. Computers use primary storage to hold data and instructions for programs that are

currently running. Primary storage is also called main memory, this usually excludes cache memory. Secondary storage includes rigid magnetic disks and sometimes optical media. It is nonvolatile, and data can be retrieved from it in a matter of milliseconds. Tertiary storage includes magnetic tape, and some optical media. [32] shows how UNIX I/O performance measurement methodologies are applied to various storage technologies.

4.2 File Systems

Most storage devices have no notion of files, directories, or the other familiar abstractions of data storage; they simply store and retrieve blocks of data. A file system is the software that creates these abstractions, including not only files and directories but also access permissions, file pointers, file descriptors, and so on. Files systems are also responsible for moving data efficiently between memory and storage devices, coordinating concurrent access by multiple processes to the same file, allocating data blocks on storage devices to specific files, and reclaiming those blocks when files are deleted and recovering as much data as possible if the file system becomes corrupted. The paper by Thomas Ruwart [33] describes how various file systems can be compared by file system benchmarks.

Disk drives read and write data in fixed-size units. File systems allocate space in blocks, which consist of a fixed number of contiguous disk sectors. Obviously, most files don't fit exactly into a whole number of blocks, and most read and write requests from applications don't transfer data in block-sized units. File systems use buffers to insulate users from the requirement that disks move data in fixed-size blocks. Buffers also give the file systems several ways to optimize data access. File systems allocate their buffers in units the same size as a disk block. The most important benefit of buffers is that they allow the file system to collect full bocks of data in memory before moving it to the disk. If a file system needed to write less than a full block of data, it would have

to perform a expensive read-modify-write operation. Similarly, when a file system reads data, it must retrieve a full block at a time. Even if the application program hasn't asked for all the data in the block, the file system will keep the entire block in memory, since the application may later request more data from the same block.

This technique is called file caching. If a file system detects that an application is reading data sequentially from a file in small steps, it may use prefetching, also called read ahead, to improve performance further: the file system reads not only the block that contains requested data but also one or more subsequent blocks in the file. The extra cost of reading the additional blocks in a single request is usually less than the cost of reading two or more blocks separately. Prefetching reduces the apparent data access time for a disk, since the cost of reading the second and subsequent blocks is hidden from the application. However, prefetching works poorly when an application's read requests don't follow a simple, predictable pattern. In that case, the file system may waste time prefetching blocks that the application doesn't need right away.

The file system uses the same pool of memory for both buffering and caching. This allows it to keep the data consistent when the application writes and then reads back the same file location. These accesses will be very efficient because neither request will require access to the disk. One disadvantage of buffering data is that most buffer memory is volatile. A user who has saved data to a file may think the data is safe in the event the computer crashes, but if a crash happens before the file system has written its buffers to disk, the data in those buffers will be lost.

4.2.1 Nonblocking I/O

Caching and buffering improve performance in two ways: by avoiding repeated accesses to the same block on disk and by allowing the file system to smooth out bursty I/O behavior. The smoothing happens because the application can quickly write a large amount of data into file system buffers without waiting for the data to be written to

disk. The file system can write these blocks to disk at a slower, steady rate while the application continues with other work that doesn't require I/O. This delayed writing can make the file system's instantaneous transfer rate much higher than its sustained rate. Not all file systems implement nonblocking I/O. Those that do often use the term asynchronous I/O for these operations. All forms of synchronous I/O work best when the computer has hardware, such as DMA than can move the data at the same time as it computes. If a CPU manages these transfers, then it has fewer available cycles to devote to computation. Likewise, if data moving between primary and secondary storage travels over the same bus that carries data between memory and the CPU or cache, the file access and the computation have to share the available bandwidth.

A common use of nonblocking I/O is for double buffering. Double buffering can improve performance when a program repeatedly reads data and then processes it, or produces data and then stores it. Double buffering, and asynchronous I/O in general, can improve performance by no more than a factor of two. This optimum improvement happens when the background I/O request takes exactly as long as the computation it overlaps.

4.2.2 Fault Tolerance

If a disk drive fails or power is lost while an operation is in progress, the data structures that organize the disk blocks into files may be left in an inconsistent state. To reduce the chance of data loss, file systems include a number of features to maintain data in a consistent state.

4.2.3 Distributed File Systems

All modern file systems handle these tasks, whether they run on parallel or sequential computers. A parallel file system is especially concerned with efficient data transfer and coordinating concurrent file access. File systems use caching and buffering to improve

performance, especially for accesses to small amounts of data and for bursty access patterns. Several processes may access a file concurrently, but the file system guarantees sequential consistency. It usually does this by preventing any process from writing a file at the same time as another process is either reading or writing the file. Distributed file systems are designed to let processes on multiple computers access a common set of files. Although distributed file systems have some features in common with parallel file systems, they are not a complete solution for parallel I/O.

The best known distributed file system is NFS (Network File System). NFS allows a computer to share a collection of its files with other computers on the network. The computer where the collection of files resides is called a server, and a computer that remotely accesses these files is a client. In NFS, a computer can be a server for some files and a client for others. Clients mount a collection of files - a directory on the server and all its subdirectories - at a particular location in their own directory hierarchy. The remote files appear to be part of the client's directory hierarchy, and programs running on the client can access them using the standard Unix naming conventions. When a client program reads a file that resides on the server, the client's file system sends a request to the server, which gets the file and sends it back to the client. The operation is invisible from the application's point of view, except that accessing a remote file takes longer than accessing a local one. Users often don't know which directories in their system are local and which are remote.

4.2.4 Parallel File Systems

A distributed file system does only part of what a parallel file system needs to do. Distributed file systems manage access to files from multiple processes, but they generally treat concurrent access as an unusual event, not a normal mode of operation. Parallel file systems do handle concurrent accesses, and they stripe files over multiple I/O nodes to improve bandwidth. Sequential Unix-based file systems have traditionally

defined the semantics of read and write operations in a way that makes concurrent file accesses by separate processes appear to occur in a well-defined order. Maintaining these semantics in parallel and distributed file systems is difficult, so some systems relax the traditional semantics to improve performance. Other systems use various techniques to maintain standard Unix consistency semantics while endeavoring to offer good parallel performance.

Computer vendors and researchers have developed many parallel file systems, some with novel programming interfaces. The trend in current commercial parallel file systems appears to be toward offering standard Unix semantics rather than specialized parallel I/O interfaces.

4.2.4.1 General Parallel File System (GPFS)

GPFS was introduced in 1998 as a successor to PIOFS (Parallel I/O File System) [53]. GPFS is based on another IBM file system called Tiger Shark. Tiger Shark was designed specifically for multimedia applications, such as video-on-demand. It includes a number of features to help it stream audio and video data at high, guaranteed transfer rates. It also has a number of special features to ensure data integrity.

GPFS represents an interesting departure from the trend that other parallel file systems established. Instead of offering a standard Unix interface for compatibility and specialized extensions for high performance, GPFS has only a standard interface. All its special features to support high performance concurrent I/O lie below the interface, essentially invisible to the user. Its designers apparently determined that GPFS could offer strict Unix semantics without reducing performance unacceptably. The exact cost of this trade-off cannot be measured since there is no way to relax Unix semantics.

Like other file systems for distributed memory computers, GPFS accesses data through I/O nodes. It uses client buffering with a distributed locking mechanism to maintain cache coherence. Although GPFS doesn't implement server buffering directly,

it is designed to access storage devices through IBM's Virtual Shared Disk (VSD) software, which does its own buffering. This software runs on the I/O nodes and controls multiple physical storage devices; it presents a uniform view of these devices to higher-level software like GPFS.

GPFS differs from Tiger Shark mainly in its support for genera-purpose computing. Although Tiger Shark has a standard Unix interface, it is optimized specifically for multimedia workloads, which tend to read long streams of data and do little concurrent writing. GPFS added byte range locking to Tiger Shark's file locking, and it uses more sophisticated algorithms for prefetching data.

4.3 Programming Interfaces

There are two types of programming interfaces, one is the low-level programming interface of parallel I/O and the other is the higher-level programming libraries for scientific applications. The low-level I/O interfaces, unlike standard sequential Unix I/O, allow applications to implement optimizations like collective I/O and hints. MPI-IO, LLAP, POSIX are some the commonly used standards [50]. NetCDF(Network Common Data Form) [51], HDF (Hierarchical Data Format) [52], are two examples for higher-level programming libraries. Both libraries use more sophisticated data models than lower-level interfaces. These models allow application programmers to specify data access operations on complex data structures rather than blocks of bytes.

4.4 Special Purpose I/O Techniques

There are two special-purpose I/O techniques commonly used in high performance computing: out-of-core data access and checkpointing. Considerably research has been done on both of these areas. Standard I/O interfaces could be used for both purposes, but specialized libraries and algorithms could offer important benefits.

An out-of-core computation is one whose data set is larger than the available primary storage. These computations require the program or the operating system to move portions of the data set from secondary to primary storage and back as the data is needed. Out-of-core techniques [34, 35] date to the early days of electronic computers, when programs often needed to stage both data and instructions into primary storage.

Virtual memory systems have moved most of the responsibility for data staging from the application developer to the operating system and the hardware. However, standard virtual memory algorithms that are adequate for general-purpose computing perform poorly in some very large computations: they do not prefetch data, and they may access data in inefficient patterns. Some users avoid the performance problems inherent in virtual memory by sizing their computations to fit in the primary storage of the computer they will be using. Indeed, some Cray supercomputers did not support virtual memory even after the technique had become common in general-purpose computers.

For users whose applications cannot fit into primary storage, a number of techniques and specialized I/O systems have been developed to improve performance [36, 37, 38]. All these techniques are based on two familiar optimizations: hiding the disk access time and minimizing the number of accesses. For out-of-core applications, staging libraries optimize common I/O requests, such as reading and writing sections of distributed arrays. Moreover, out-of-core algorithms that explicitly recognize the role of secondary storage can improve performance compared to using standard virtual memory to simulate a very large primary storage space [39, 40].

Checkpointing techniques fall into two categories: those initiated within the application and those initiated by the system. The former are generally easier to implement because the application can initiate requests at times when it is in a consistent and easily recorded state. However, system-initiated checkpointing eliminates the need for users to include checkpointing requests explicitly in their code. All checkpointing software represents a trade-off between three competing goals: minimizing the application pro-

grammer's effort, minimizing the size of the checkpoint file, and making the checkpoint data portable.

This chapter has introduced some of the important problems in storing data for high performance computers, along with several I/O programming interfaces designed for performance, convenience, or both. Applications make a variety of demands on I/O systems and parallel I/O can help meet these demands by increasing both capacity and I/O speed. Moving data between memory and disk presents the most pressing I/O problems, and the majority of I/O research has been directed at these problems.

CHAPTER 5. HPL Algorithm

This chapter provides a high-level description of the algorithm used in this package. As indicated in the following sections, HPL contains in fact many possible variants for various operations. Defaults could have been chosen, or even variants could be selected during the execution. Due to the performance requirements, it was decided to leave the user with the opportunity of choosing, so that an "optimal" set of parameters could easily be experimentally determined for a give machine configuration. From a numerical accuracy point of view, all possible combinations are rigorously equivalent to each other even though the result may differ slightly (bit-wise).

5.1 Main Algorithm

This software package solves a linear system of order n: $Ax = b$ by first computing the LU factorization with row partial pivoting of the n-by-$n+1$ coefficient matrix $[Ab] = [[L, U]y]$. Since the lower triangular factor L is applied to b as the factorization progresses, the solution x is obtained by solving the upper triangular system $Ux = y$. The lower triangular matrix L is left unpivoted and the array of pivots is not returned.

The data is distributed onto a two-dimensional P-by-Q grid of processes according to the block-cyclic scheme to ensure good load balance as well as the scalability of the algorithm. The n-by-$n+1$ coefficient matrix is first logically partitioned into nb-by-nb blocks, that are cyclically dealt onto the P-by-Q process grid. This is done in both dimensions of the matrix.

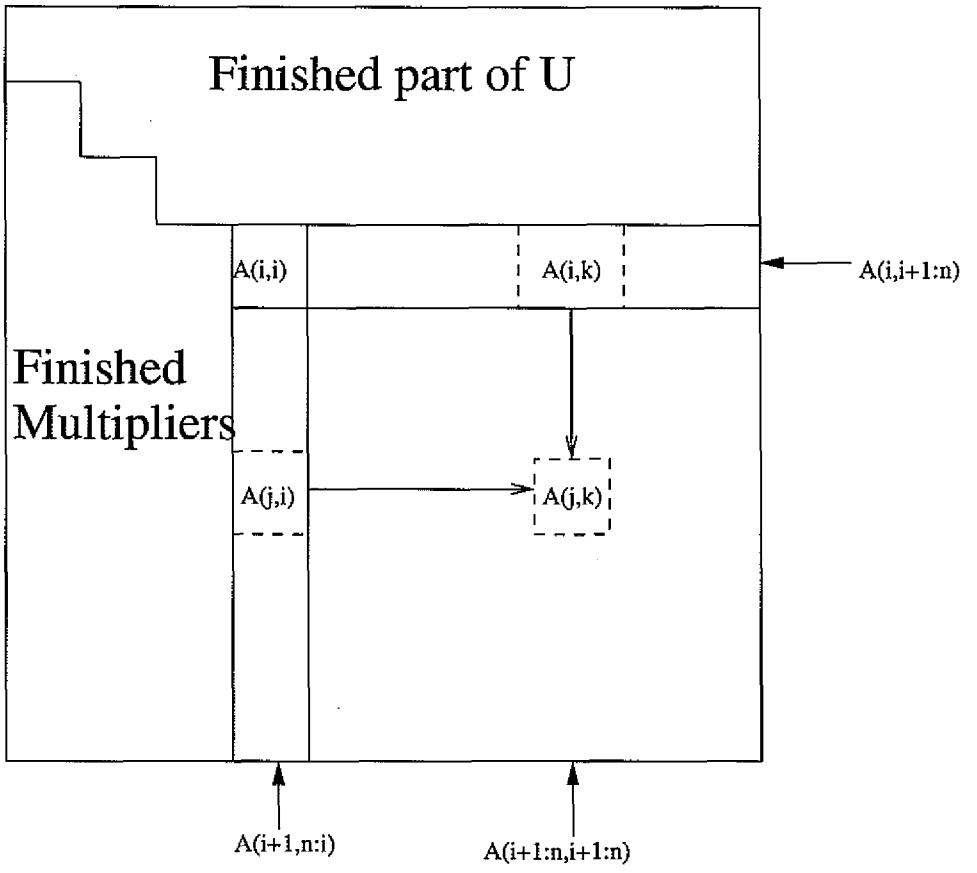

Figure 5.1 HPL Algorithm

The right-looking variant has been chosen for the main loop of the LU factorization. This means that at each iteration of the loop a panel of nb columns is factorized, and the trailing submatrix is updated. Note that this computation is thus logically partitioned with the same block size *nb* that was used for the data distribution.

5.2 Panel Factorization

At a given iteration of the main loop, and because of the cartesian property of the distribution scheme, each panel factorization occurs in one column of processes. This particular part of the computation lies on the critical path of the overall algorithm. The user is offered the choice of three (Crout, left- and right-looking) matrix-multiply based

recursive variants. The software also allows the user to choose in how many sub-panels the current panel should be divided into during the recursion. Furthermore, one can also select at run-time the recursion stopping criterion in terms of the number of columns left to factorize. When this threshold is reached, the sub-panel will then be factorized using one of the three Crout, left- or right-looking matrix-vector based variant. Finally, for each panel column the pivot search, the associated swap and broadcast operation of the pivot row are combined into one single communication step. A binary-exchange (leave-on-all) reduction performs these three operations at once.

5.3 Panel Broadcast

Once the panel factorization has been computed, this panel of columns is broadcast to the other process columns. There are many possible broadcast algorithms and the software currently offers 6 variants to choose from. These variants are described below assuming that process 0 is the source of the broadcast for convenience. "$\longrightarrow$" means "sends to".

Increasing Ring $0 \longrightarrow 1$; $1 \longrightarrow 2$; $2 \longrightarrow 3$ and so on. This algorithm is the classic one; it has the caveat that process 1 has to send a message.

Increasing Ring (modified) $0 \longrightarrow 1$; $0 \longrightarrow 2$; $2 \longrightarrow 3$ and so on. Process 0 sends two messages and process 1 only receives one message. This algorithm is almost always better, if not the best.

Increasing 2-Ring The Q processes are divided into two parts: $0 \longrightarrow 1$ and $0 \longrightarrow$ Q/2; Then processes 1 and Q/2 act as sources of two rings: $1 \longrightarrow 2$, Q/2 $\longrightarrow$ Q/2+1; 2 $\longrightarrow 3$, Q/2+1 $\longrightarrow$ to Q/2+2 and so on. This algorithm has the advantage of reducing the time by which the last process will receive the panel at the cost of process 0 sending 2 messages.

Increasing 2-Ring-modified As one may expect, first 0 $\longrightarrow$ 1, then the Q-1 processes left are divided into two equal parts: 0 $\longrightarrow$ 2 and 0 $\longrightarrow$ Q/2; Processes 2 and Q/2 act then as sources of two rings: 2 $\longrightarrow$ 3, Q/2 $\longrightarrow$ Q/2+1; 3 $\longrightarrow$ 4, Q/2+1 $\longrightarrow$ to Q/2+2 and so on. This algorithm is probably the most serious competitor to the increasing ring modified variant.

Long (bandwidth reducing) as opposed to the previous variants, this algorithm and its follower synchronize all processes involved in the operation. The message is chopped into Q equal pieces that are scattered across the Q processes.

The pieces are then rolled in Q-1 steps. The scatter phase uses a binary tree and the rolling phase exclusively uses mutual message exchanges. In odd steps 0$\longleftrightarrow$1, 2 $\longleftrightarrow$ 3, 4 $\longleftrightarrow$ 5 and so on; in even steps Q-1 $\longleftrightarrow$ 0, 1 $\longleftrightarrow$ 2, 3 $\longleftrightarrow$ 4, 5 $\longleftrightarrow$ 6 and so on.

More messages are exchanged, however the total volume of communication is independent of Q, making this algorithm particularly suitable for large messages. This algorithm becomes competitive when the nodes are "very fast" and the network (comparatively) "very slow".

Long (bandwidth reducing modified) same as above, except that 0 $\longrightarrow$ 1 first, and then the Long variant is used on processes 0,2,3,4 .. Q-1.

The rings variants are distinguished by a probe mechanism that activates them. In other words, a process involved in the broadcast and different from the source asynchronously probes for the message to receive. When the message is available the broadcast proceeds, and otherwise the function returns. This allows to interleave the broadcast operation with the update phase. This contributes to reduce the idle time spent by those processes waiting for the factorized panel. This mechanism is necessary to accommodate for various computation/communication performance ratios.

5.4 Look Ahead

Once the panel has been broadcast or say during this broadcast operation, the trailing submatrix is updated using the last panel in the look-ahead pipe: as mentioned before, the panel factorization lies on the critical path, which means that when the kth panel has been factorized and then broadcast, the next most urgent task to complete is the factorization and broadcast of the k+1 th panel. This technique is often referred to as "look-ahead" or "send-ahead" in the literature [27]. This package allows to select various "depth" of look-ahead parameters. By convention, a depth of zero corresponds to no lookahead, in which case the trailing submatrix is updated by the panel currently broadcast. Look-ahead consumes some extra memory to essentially keep all the panels of columns currently in the look-ahead pipe. A look-ahead of depth 1 or 2 is likely to achieve the best performance gain.

5.5 Update

The update of the trailing submatrix by the last panel in the look-ahead pipe is made of two phases. First, the pivots must be applied to form the current row panel U. U should then be solved by the upper triangle of the column panel. U finally needs to be broadcast to each process row so that the local rank-nb update can take place. We choose to combine the swapping and broadcast of U at the cost of replicating the solve. Two algorithms are available for this communication operation.

5.5.1 Binary-Exchange

This is a modified variant of the binary-exchange (leave on all) reduction operation. Every process column performs the same operation. The algorithm essentially works as follows. It pretends reducing the row panel U, but at the beginning the only valid copy is owned by the current process row. The other process rows will contribute rows of A

they own that should be copied in U and replace them with rows that were originally in the current process row. The complete operation is performed in log(P) steps. For the sake of simplicity, let assume that P is a power of two. At step k, process row p exchanges a message with process row $p + 2^k$. There are essentially two cases. First, one of those two process rows has received U in a previous step. The exchange occurs. One process swaps its local rows of A into U. Both processes copy in U remote rows of A. Second, none of those process rows has received U, the exchange occurs, and both processes simply add those remote rows to the list they have accumulated so far. At each step, a message of the size of U is exchanged by at least one pair of process rows.

5.5.2 Long

This is a bandwidth reducing variant accomplishing the same task. The row panel is first spread (using a tree) among the process rows with respect to the pivot array. This is a scatter (V variant for MPI users). Locally, every process row then swaps these rows with the the rows of A it owns and that belong to U. These buffers are then rolled (P-1 steps) to finish the broadcast of U. Every process row permutes U and proceed with the computational part of the update. A couple of notes: process rows are logarithmically sorted before spreading, so that processes receiving the largest number of rows are first in the tree. This makes the communication volume optimal for this phase [41].

Finally, before rolling and after the local swap, an equilibration phase occurs during which the local pieces of U are uniformly spread across the process rows. A tree-based algorithm is used which may not necessarily be optimal for a given compouter. This operation is necessary to keep the rolling phase optimal even when the pivot rows are not equally distributed in process rows. This algorithm has a complexity in terms of communication volume that solely depends on the size of U. In particular, the number of process rows only impacts the number of messages exchanged. It will thus outperform the previous variant for large problems on large machine configurations.

The user can select any of the two variants above. In addition, a mix is possible as well. The "binary-exchange" algorithm will be used when U contains at most a certain number of columns. Choosing at least the block size nb as the threshold value is clearly recommended when look-ahead is on.

5.6 Backward Substitution

The factorization has just now ended, the back-substitution remains to be done. For this, we choose a look-ahead of depth one variant. The right-hand-side is forwarded in process rows in a decreasing-ring fashion, so that we solve Q * nb entries at a time. At each step, this shrinking piece of the right-hand-side is updated. The process just above the one owning the current diagonal block of the matrix A updates first its last nb piece of x, forwards it to the previous process column, then broadcasts it in the process column in a decreasing-ring fashion as well. The solution is then updated and sent to the previous process column. The solution of the linear system is left replicated in every process row.

5.7 Checking The Solution

To verify the result obtained, the input matrix and right-hand side are regenerated. Three residuals are computed and a solution is considered as "numerically correct" when all of these quantities are less than a threshold value of the order of 1.0. In the expressions below, ε is the relative (distributed-memory) machine precision.

$$
\begin{aligned}
r_n &= \frac{\|Ax-b\|_\infty}{\|A\|_1.n.\varepsilon} \\
r_1 &= \frac{\|Ax-b\|_\infty}{\|A\|_1.\|x\|_1.\varepsilon} \\
r_\infty &= \frac{\|Ax-b\|_\infty}{\|A\|_\infty.\|x\|_\infty.\varepsilon}
\end{aligned}
$$

CHAPTER 6. Design of HPL

HPL performs a series of tests given a set of parameters such as the process grid, the problem size, the distribution blocking factor. This testing routine generates the data, calls and times the linear system solver, checks the accuracy of the obtained vector solution and writes this information to the specified output medium. This linear system is solved first by factoring a N+1 by N matrix using LU factorization with row partial pivoting. The main algorithm is the "right looking" variant with or without look-ahead. The lower triangular factor is left unpivoted and the pivots are not returned. The right hand side is the N+1 column of the coefficient matrix.

A new panel data structure is initialized to start the factorization. The main loop factorizes NB columns at a time, which is the same as the blocking factor used in generating and storing the input data. This 1-dimensional panel of columns is then recursively factorized. The RPFACT function pointer specifies the recursive algorithm to be used, either Crout, Left looking or Right looking. NBMIN allows to vary the recursive stopping criterion in terms of the number of columns in the panel, and NDIV allow to specify the number of subpanels each panel should be divided into. Finally PFACT is a function pointer specifying the non-recursive algorithm to be used on at most NBMIN columns. One can also choose here between Crout, Right looking or Left looking.

Bidirectional exchange is used to perform the swap::broadcast operations at once for one column in the panel. This results in a lower number of slightly larger messages than usual. On P processes and assuming bi-directional links, the running time of this

function can be approximated by (when N is equal to N0):

$$N0 * log_2(P) * (lat + (2 * N0 + 4)/bdwth) + N0^2 * (M - N0/3) * gam2 - 3$$

where M is the local number of rows of the panel, lat and bdwth are the latency and bandwidth of the network for double precision real words, and gam2-3 is an estimate of the Level 2 and Level 3 BLAS rate of execution. The recursive algorithm allows indeed to almost achieve Level 3 BLAS performance in the panel factorization. On a large number of modern machines, this operation is however latency bound, meaning that its cost can be estimated by only the latency portion $N0 * log_2(P) * lat$. Mono-directional links will double this communication cost.

CHAPTER 7. Modifications to HPL

HPL is part of Highly Parallel Computing Benchmark of LINPACK. This benchmark is used in generating the Top500 Report from the LINPACK benchmark suite. This benchmark attempts to measure the best performance of a machine in solving a system of equations. The Top500 lists the 500 fastest computer systems being used today. The best Linpack benchmark performance achieved is used as a performance measure in ranking the computers. The TOP500 list has been updated twice a year since June 1993. As the report it self mentions, the timing information presented in the Top500 list should in no way be used to judge the overall performance of a computer system. The results reflect only one problem area: solving dense systems of equations.

The changes made to HPL can be categorized into two parts. The first step was to add an out-of-core feature to the existing algorithm. The second was to use a threads based programming model to parallelize some parts of the algorithm. Some very basic modification were made to the timings routine. The timing routine was modified to time more parts of the program/algorithm than originally planned. This also help in identifying the parts of the program which took the most time, and led us to take the appropriate steps to reduce the runtime for these parts of the program.

7.1 Out-of-Core Capability

The main goal of the modifications was to include the out-of-core feature to the HPL benchmarking tool. The making of modifications to this tools had a lot of issues, just

like any other software modifications. How much/what part of the software should be modified. One of the main criteria was that the algorithm should remain the same, i.e. it still solve a linear system using LU factorization.

This implied that the matrix in memory should still be in block-cyclic format. A design point for our implementation is should the matrix in the secondary store (on disk) be in block cyclic or in some other format? A literature survey was done on the different storage formats used by various out-of-core algorithms [42, 43, 44]. Further research was done on the various algorithms used in out-of-core computations [45, 46]. One common theme among all the out-of-core computation programs were their I/O operations were designed to give the best performance and the algorithms themselves were chosen that gave best performance for I/O. But this is not the case for HPL. In HPL the main loop for factorization is a **Right looking** algorithm, but according to published papers/documents [47, 48] a **Left looking** algorithm is good for factorization. This would require changing the whole HPL algorithm, and implement the algorithms from scratch, which was not the purpose of this project. The matrix data is stored in blocks in the original algorithm. Since the I/O operations involved would be significant compared to the floating point operations, the overhead would be reduced if the data format in main memory was the same as the data stored on disk. Before each computational block, namely the BLAS routines, the input data pointers were calculated using functions in the original HPL code. This meant that, the data could be transferred to/from main memory and disk transparent to the computational blocks. There would be no direct involvement between compute operations and I/O operations. Due to the data access pattern of the original HPL algorithm, each I/O operation required reading/writing a column of data blocks. However, due to the recursive nature of the factorization algorithm, the entire matrix needed to be accessed during a single iteration of the main loop. This resulted in polynomial I/O operations in relation to the data size N. Some minor changes were made in some loops to reduce the number of I/O transactions. This

made quite an impact on the I/O operations. The actual details are presented in the following chapter.

7.2 Threads in HPL

HPL uses MPI for the multiprocesseor/interprocess communication. Threads are used for performing some tasks, to take advantage of the SMP style architecture for the nodes like the IBM SP system *Seaborg*, at National Energy Research Scientific Computing Center (NERSC). HPL uses external libraries for performing the BLAS routines [49]. Since, these were the most computationally intensive operations in the benchmark suite, they made a very good starting point for parallelization. Some of the routines are memory intensive. This also was hoped to give some performance metrics. Using OpenMP threads was one of the thoughts on parallelization. Using pthreads to parallelize the BLAS algorithms should also be very similar. Parallelization using OpenMP should involve less work, at least from the programming perspective. The Pthreads interface requires much more code restructuring to obtain a similar implementation.

The HPL code provides some BLAS algorithms of its own but not all algorithms. This required writing some extra code to provide the same functionality. The details are provided in the next chapter. The use of this code rather than a third party library routines, provides a more accurate comparison of the result of including OpenMP threads in the BLAS algorithms and the overall performance. This should also include the overhead of creating multiple threads. The major parameters which are expected to influence the results/timings are the block size, problem size, number of threads. Plots and tables for these parameters are provided in the chapter on results. Furthermore, the modified version could readily use any thread based BLAS routines from the vendor offerings. We chose to go with straightforward source code implementations as a normalizing effect for comparison.

CHAPTER 8. Implementation

The implementation of the modifications proceeded in two stages. The first stage involved adding I/O and Out-of-Core memory management to HPL. The second part was adding Threads to computationally intensive parts of HPL. The following sections describe these steps in detail.

8.1 I/O Implementation

The I/O for HPL arises due to inclusion of Out-of-Core feature to HPL. The main preface to an Out-of-Core algorithm is that the data is more than that available in the main memory. In the case of HPL, the data was the generated input matrix for the N-variable linear equation. The in-core algorithm distributes the generated data in a block-cyclic pattern to all the processes involved, as mentioned in earlier chapters. In HPL each process was assumed to have its own memory and any type of communication between the processes was through MPI. Since each process had its own memory, the easiest way to add I/O would be to write the input data generated by each process to a file (secondary storage), and read from it when the data is needed. This brings us to the question of how the data is to be stored on disk or secondary storage.In order to answer the above question we would need to take a look at the high performance computing system on which the testing and debug runs were going to be based. A typical node on the target system was a 16 processor SMP node with 16 Gigabytes of aggregate memory for each node. This meant that the out-of-core data generated for each process was going

to be more than 1 Gigabyte. The final location where the data was going to be written should be large enough to store all data from each process at the same time. This model works for most MPP systems allowing for adequate secondary storage resources.

Once the amount of secondary storage needed was determined, then came question of how the data was going to be stored. There were various options to consider, the data could be written in a single file and all process access it at the same time or they could create and write files of their own. The latter option was chosen as it was the most straigtforward to implement. There would be no offsets and integrity checks required from the other processes. After looking at all the different methods available in [17], it was not clear on what kind of I/O to actually implement. There were various issues involved like, the level of abstraction to be used, the layers of storage to be used. External scientific interfaces like MPI-IO [50], netCDF [51] and HDF [52] were also considered. This also brought into the picture, the data structure needed to store the data on disk. The easiest option was to store the data in the same format as it is in the main memory. This would eliminate the overhead of transforming from one format to the other, both during reads and as well as writes. It would essentially mean that data generated would be written to the disk/secondary storage as fast as possible.

This would bring us to the actual I/O interface. Since this is a modular design, interface in HPL would not know the actual method used to perform the I/O from memory to disk. This enables us to use vendor specific features to improve the performance and maintain portability at the same time. This can be accomplished by using generic I/O for each process as one option and vendor specific implementation as another option. In the current implementation, since most of the testing was done on an IBM SP system, the vendor specific implementation pertains to an IBM SP system. The IBM SP system has separate I/O nodes running the distributed, parallel I/O system GPFS (General Parallel File System) [53]. The data distribution in memory will the sames as the data stored on disk. Also, as there will not be any overlapping I/O and computation, se-

quential access for each process would be a good starting point for the implementation. Since, most of the I/O is going to be in sequential and in discrete blocks for each process, the GPFS should provide a good performance. By using the GPFS for I/O, we could concentrate on data I/O in terms of blocks of bytes and not on the intricacies of parallel I/O between each process or node. The modular design also meant that the type of I/O used was not visible to the main program.

8.1.1 Matrix Pointer

The third party software being used were the ATLAS libraries [49] for the BLAS algorithms and the MPI libraries for interprocess communication. This implied that BLAS routines and communication routines should not be aware of any I/O involved. Since the basic HPL algorithm of solving an N-variable linear equation was supposed to remain the same, the only possible place where code changes could be made was the pointer/memory access to matrix data and coefficients.

In the original algorithm, the pointer to the matrix data was obtained by using the required indices and a base pointer to the matrix, as shown below:

```
/*
 * Mptr returns a pointer to a_( i_, j_ ) for readability reasons and
 * also less silly errors ...
 */
#define     Mptr( a_, i_, j_, lda_ ) ( (a_) + (i_) + (j_)*(lda_) )
```

The routine *Mptr* is now modified to implicitly call a function. The function returns the appropriate pointer as it does previously, at the same time keeping the I/O involved invisible to HPL algorithm.

```
/** Mptr modified to call a function **/
#define     Mptr HPL_oocptr
```

As it is evident from the above code, the input matrix as well as all the intermediate results are all stored in a column major format. To get a better understanding of the problem involved in converting from in-core to out-of-core, we need to take a brief look at the data arrangement in the original code.

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

	0		1	
0	$A_{0,0}$	$A_{0,2}$	$A_{0,1}$	$A_{0,3}$
	$A_{2,0}$	$A_{2,2}$	$A_{2,1}$	$A_{2,3}$
1	$A_{1,0}$	$A_{1,2}$	$A_{1,1}$	$A_{1,3}$
	$A_{3,0}$	$A_{3,2}$	$A_{3,1}$	$A_{3,3}$

The above tables show the 2-D block cyclic data distribution used by HPL. The data is distributed onto a two-dimensional grid of processes according to the block-cyclic scheme to ensure good load balance as well as the scalability of the algorithm. The grid dimensions are P by Q. The N by N+1 coefficient matrix is logically partitioned into blocks, that are cyclically dealt onto the P by Q process grid. Each block is N_B by N_B, with N_B being referred as the blocking factor. This cyclic distribution is done in both dimensions of the matrix.

In the example above, the process grid is a 2 by 2 grid (P=2 and Q=2), 4 processes total. The first process being denoted by (0,0), the second by (0,1) and so on. The number of subblocks is 4 in both dimensions. The first table shows the position of each

subblock in relation to the whole coefficient matrix. The second table shows the final subblocks distributed to each process, indicated by each quadrant.

As mentioned earlier each individual coefficients in a block (subblock) is accessed in a column major format. Then each subblock on a process, is accessed again in a column major format. The storage on disk (secondary storage) would mirror the above structure to avoid rearrangement overhead.

8.1.2 Main Memory Requirement

After the data structure for storage of matrix data on disk had been decided, then came the question of how much of data is going to reside in memory and how to go about actually transferring data to and from secondary storage.

This step required analyzing the memory access pattern of HPL in minute detail. HPL used the right-looking variant for the main loop of the LU factorization, in solving the linear equation. At each iteration of the loop a panel of N_B columns is factorized, and the trailing submatrix is updated. An important point to note, is that, N_B above, is the same value used in partitioning the coefficient matrix into blocks and later used for data distribution.

The following were the observations on the memory access patterns:

- At each iteration of the main loop, factorization was performed on a panel at a time.
- The width of each panel was N_B, the same as the block size in data distribution.
- Following panel factorization, each panel of columns was broadcast to other processes performing a similar operation.
- After the factorization and broadcast, the trailing submatrix was updated. The trailing submatrix includes all the columns (panels) yet to be factorized.

- Finally after the entire matrix was factorized, the triangular solve was done, which involved the entire matrix data.
- Then the results are verified by computing the scaled residuals which involved the entire matrix again, as the input data was regenerated.

In addition, the parameters to MPI communications and BLAS routines were also observed. MPI messages required two distinct locations for sending and receiving messages. The BLAS had varying memory requirements. Some BLAS routines, usually that involved vectors required just a single pointer location, where some BLAS routines required as many as three distinct pointer locations. Upon closer inspection it was found that the *three* pointer locations corresponded to just *two* panels.

From all the above observations, it was gathered that having just *two* panels of blocks in memory at a given time, would be enough to correctly execute the HPL code. The option of being able to add more panels of blocks during runtime was also taken into consideration. This also means that having more panels in memory would reduce the maximum problem size that can be run by HPL. The panels were also included in a priority queue to avoid erasing a panel currently being used and maybe even reducing the number of reads and writes needed.

8.1.3 Data Integrity

Now that the data format to be stored on disk, and the matrix data to be retained in main memory have been decided; this brings us to the hardest part yet. The maintenance of coherence between the data on disk and the data in main memory. There are two pointers to keep in mind, one is the location of the current matrix data element being accessed relative to panel of blocks, and the second is the location of the same data element relative to the entire data distributed to the process. For the original HPL code, these two pointers are the same, as all the matrix data resides in main memory.

But for the out-of-core implementation, the location in the panel of blocks corresponds to the data element in main memory, and the location relative to data distribution corresponds to the matrix data in secondary storage.

The access and/or modification of matrix data elements was just a single operation in the original HPL code. But the new HPL code has obviously become much more complex for even a simple data element access. In fact data modification involves less work than data access in HPL with I/O. In HPL with I/O, when a data element is to be accessed, the element is first checked in the two(or more) data panels in main memory. If neither of the two data panels contain the element, then one of the panels in memory is replaced by the panel of blocks containing the required element. The panel being replaced is written back to the disk first, of course. The appropriate pointer to the data element in the panel of blocks is then returned by *Mptr*. In the case of data modification, the appropriate panel of blocks must already be in memory. It simply involves modifying the data in memory. This remains in memory until the panel containing it, is going to be replaced, which is then written back to the secondary storage. Changes to data are cached in-core thus avoiding as many secondary storate I/O operations as possible.

The above changes would have been enough if *Mptr* was used only for accessing the matrix data elements. Unfortunately, *Mptr* was used to also reference temporary data from panel factorization and broadcast. If the two data elements could be easily differentiated, then the solution would have been simple. The original *Mptr* could be used for the temporary data, and the modified version for all matrix data elements. But the distinction was not so clearcut. This was not an issue for the original code as everything was in main memory and it could be easily represented by pointers. This need for differentiation between in-core and out-of-core (secondary storage) data elements, created a need for another parameter to indicate the type of pointer/data element being accessed by the routine *Mptr*. The data types and data structures involved above can be seen in more detail in Appendix B.

8.2 Threads With OpenMP

As mentioned in the previous chapter, HPL uses third party BLAS routines like ATLAS. These routines are optimized for the specific architecture and there is no provision for including OpenMP in ATLAS. However, HPL provides reference code for BLAS routines which are not optimized for any particular architecture. This code is only executed when VSIPL (Vector Signal Image Processing Library) is used instead of BLAS.

This created considerable modifications in the preprocessor to call the above unoptimized routines in place of the BLAS routines. Since these routines were tightly linked to the VSIPL library, the routines had to be replicated and preprocessed separately. These unoptimized routines were then parallelized using OpenMP and compared with the sequential versions. Some parallelizations were direct while, the others required modifications to the loops. A few examples are presented in the Appendix. As a result of these modifications there are four versions of the basic BLAS routines available to HPL. There are the two in original code, which include the vendor supplied Fortran BLAS routines, external BLAS routines in C like ATLAS, and then there are the unoptimized versions of these routines with and without threads. These options are available at compile time only.

CHAPTER 9. Results

This chapter presents the timing and performance results of HPL with out-of-core I/O support. The comparisons with the original in-core HPL are also presented. The plots and timings tables analyzed and conclusions are drawn from the figures presented.

Appendix A contains a sample input file used for the runs performed by HPL. This sample input file has 3 additional inputs from the original HPL input file. These are:

- *Data directory*: This contains destination directory where the out-of-core data is to the written. The default is *data*, which is a subdirectory where the executable is being run.

- *Data file prefix*: This is optional prefix to the out-for-core files being written in the above directory. The suffix is a combination of the row process id and column process id. This helps in keeping the data file names generic.

- *Buffer size*: This the number of panels of columns to be used for keeping the data in main memory. The minimum is 2.

Jobs are run with various input parameters and the results are plotted and analyzed. Not all parameters are tested, as they do not directly affect the I/O. The results are also presented in two sections, the first is just involving the out-of-core modifications. The second section includes the performance of OpenMP threads.

Table 9.1 Table for the Runtime(in seconds) for problem size N=20000 run on 4x4 processors. NB indicates the block size

NB	Left RT	I/O	Crout RT	I/O	Right RT	I/O
500	739.46	240.61	764.56	262.68	757.05	261.80
250	1203.14	532.42	1259.67	574.64	1245.89	580.16
100	2866.44	1548.79	2988.52	1619.48	2958.63	1631.48
75	3870.92	2110.21	4017.82	2203.60	3976.56	2173.77
50	6051.73	3461.24	6328.21	3624.35	6250.78	3628.12

9.1 Performance With Disk I/O

Left,Crout and Right in Table 9.1, indicate the type of factorization used to factor the individual panels. It can be clearly observed that the greater the block size the lesser the run time. This is primarily due to the decrease in disk reads and writes involved by increasing the block size. The process grid is a 4 by 4. So, each process gets 5000 by 5000 data chunk. When the block size is 50, the number of column panels for factorization is about 100, but when the block size is 500, the number of panels comes down to 10. That is a considerable change, hence the noticeable decrease in I/O time. It is to be kept in mind here even though the data read or written is the same in both the cases, it the number of reads/writes which make the difference due to the high latency for secondary storage. The performance in GFlops/s (Billions of floating point operations per second) is also provided to give an overall picture in figure 9.3. Even though I/O directly does not contribute to increase GFlops, the less time spent doing I/O means less total runtime, implying more effective GFlops per unit time.

Table 9.2, and figures 9.4 and 9.5 show similar curves in the plots when the problem size is increased for the same process grid. By looking at the actual timings, it can be seen that it took nearly 3 times longer even though the increase in problem size per process was a few thousand. This reiterates the polynomial time increase in the computation involved in matrix operations.

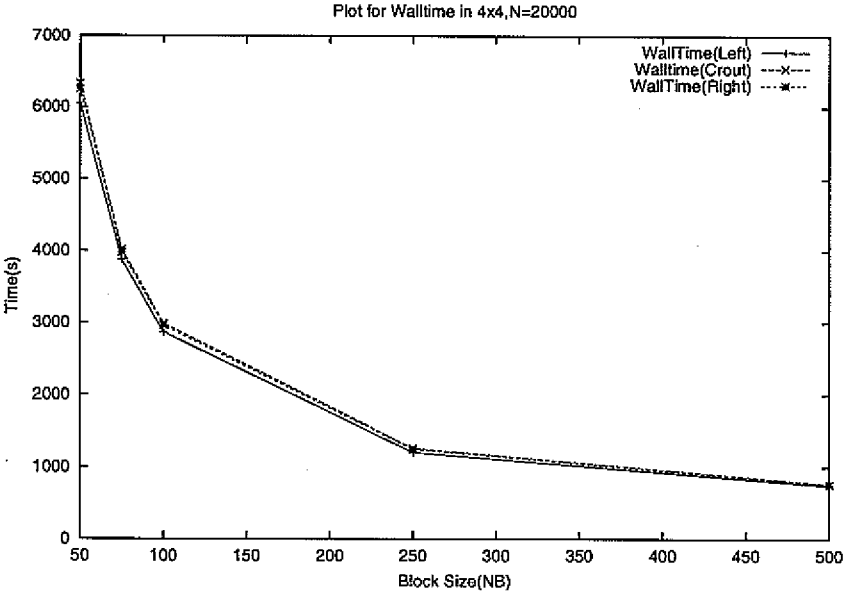

Figure 9.1 Runtimes (in seconds) for various block sizes, and varying the panel factorization. Problem size N=20000. Data from Table 9.1.

Table 9.2 Table for the Runtime (in seconds) on 4x4 processors for N=30000

NB	Left RT	I/O	Crout RT	I/O	Right RT	I/O
500	2428.46	847.60	2506.46	913.32	2440.96	874.27
250	4045.25	1827.46	4295.01	2015.94	4205.67	1987.30
100	9480.16	5145.75	10143.39	5742.90	9973.85	5640.96
75	12204.83	6784.75	13257.70	7501.88	13505.61	7715.63
50	19111.48	10981.55	20405.59	11853.87	20959.22	12297.79

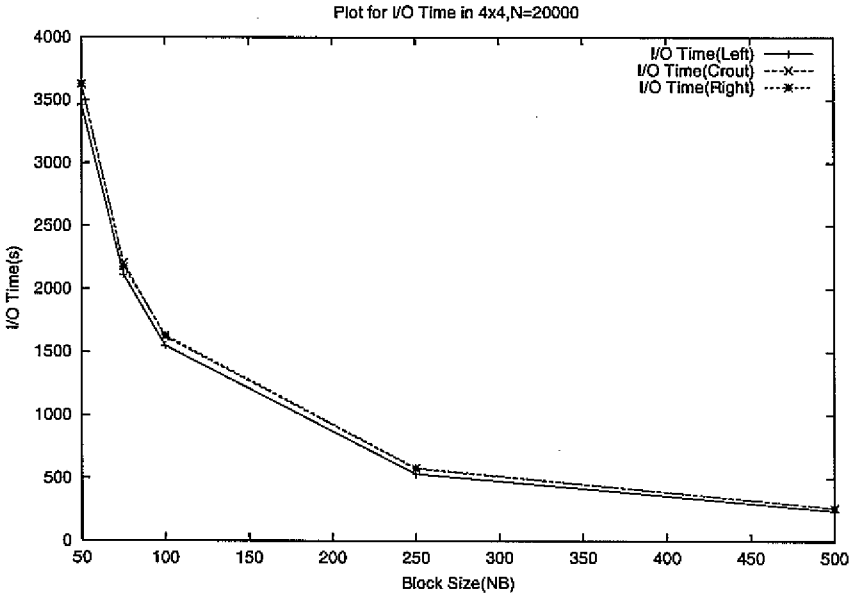

Figure 9.2 I/O Times (in seconds) for various block sizes, and varying the panel factorization. Problem size N=20000. Data from Table 9.1.

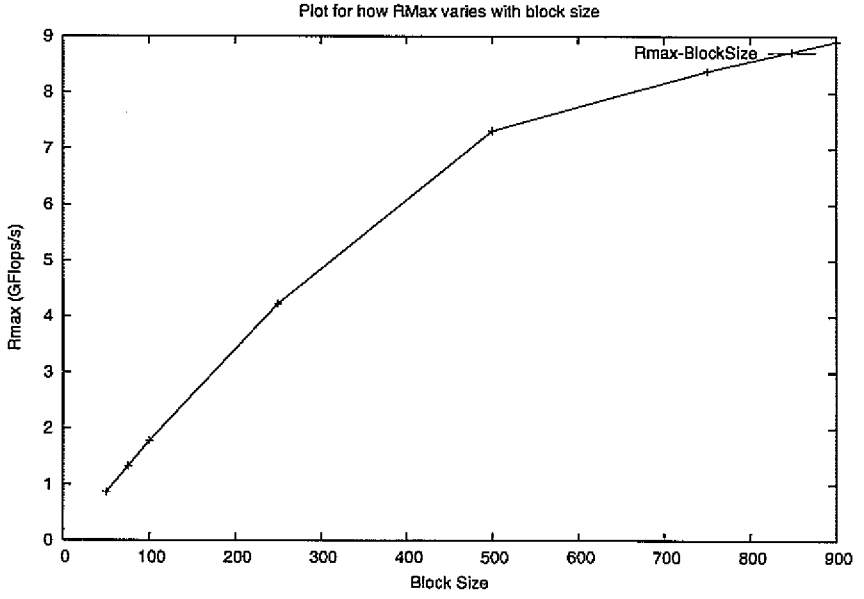

Figure 9.3 RMax values for 4x4 process node with varying block sizes. Problem size N=20000. Data from Table 9.1.

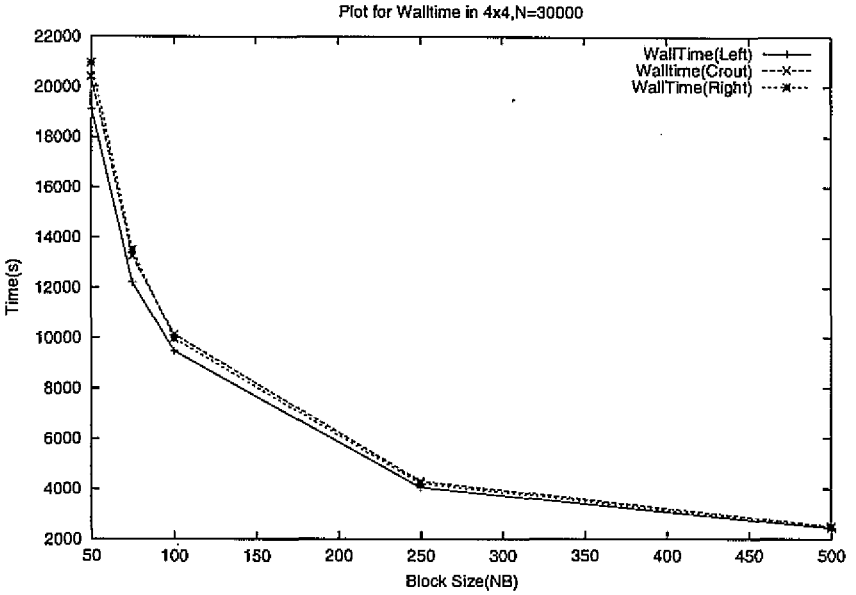

Figure 9.4 Runtime(in seconds) for 4x4 processors, N=30000. Data from Table 9.2.

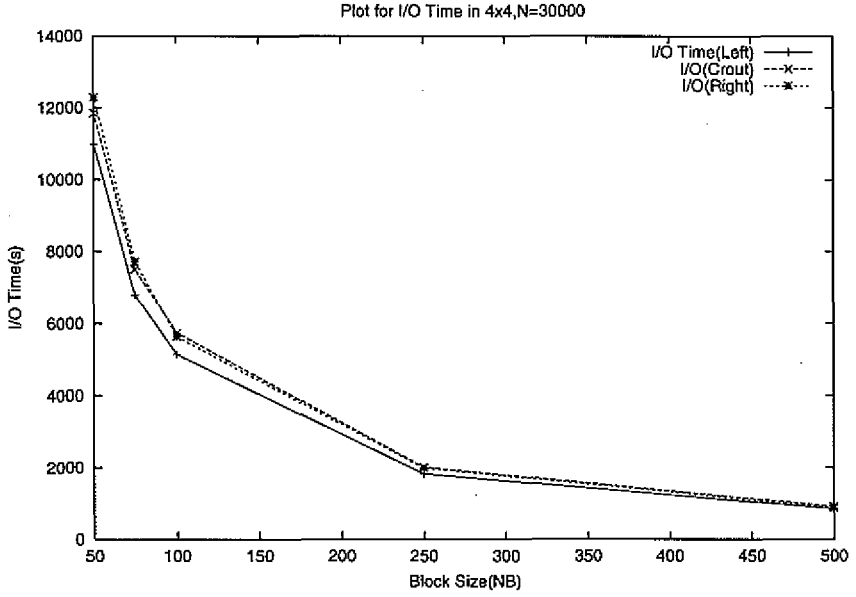

Figure 9.5 I/O Time(in seconds) for 4x4 processors, N=30000. Data from Table 9.2.

Table 9.3 Table for the Buffer size relationship with runtime(in seconds), for problem size N=50000 on an 8x8 grid, with block size NB=500.

Buffer size	Broadcast	RunTime(s)	I/O Runtime(s)
8	1RingM	2656.76	926.54
5	1RingM	3435.78	1231.05
4	1RingM	3443.19	1521.74
2	1RingM	3442.88	1451.22
8	LongM	2606.80	900.53
5	LongM	3031.73	1211.18
4	LongM	3413.62	1492.20

In Table 9.3 other broadcast/communication types are not given as they are basically a variation of the above two. Even though the overall runtime may not be considerably different, the I/O time is impacted due to the size of the messages being transmitted between processing nodes, and the number of messages sent. The reason there is not a significant decrease in I/O time is, that the number of reads/writes has not decreased much. The recursive nature of the algorithm, results in an update of the trailing sub-matrix at the end of each loop of the main loop. Having a buffer does not help in this case, as all the unfactorized panels need to be updated. The buffer does decrease the I/O some toward the end of the algorithm when there are only a few panels left to be factorized. The buffer is most efficient when all the panels of columns of the data are in memory, but that would mean that the entire matrix could fit in the memory, which is not the purpose of the changes in the first place. But in general, the more the number of panels in memory the better the performance. The runtimes and I/O times are plotted in Figures 9.6 and 9.7.

Figures 9.8 through ?? try to show to effect of modifying more than a single parameter at a time. Previously all the plots just changed the values of block sizes, or problem sizes or the buffer size. The following graphs present of effect of combinations of the above variations. From the plots it can the easily seen that the results were just as expected

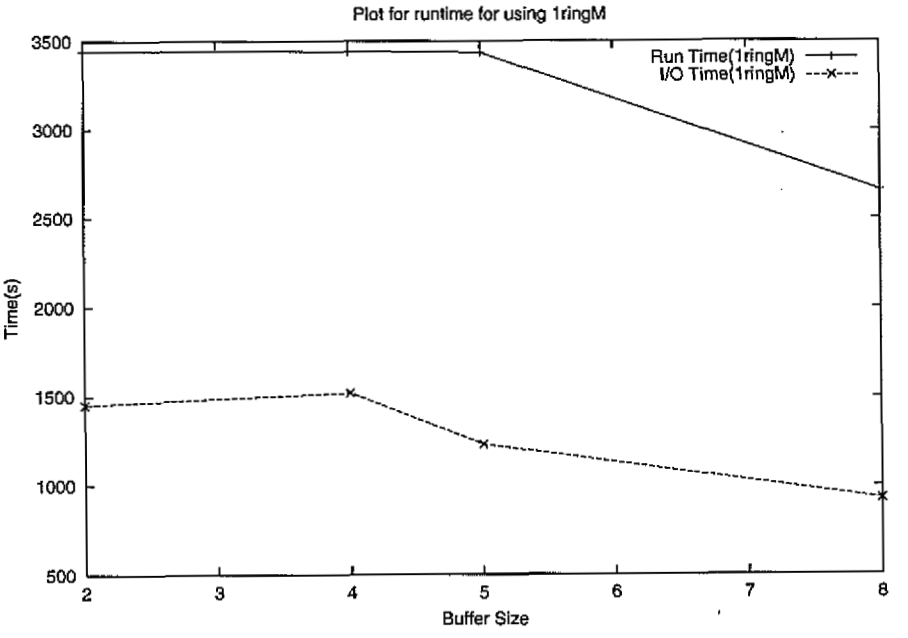

Figure 9.6 Runtime(in seconds) for problem size N=50000, running on 8x8 grid with block size 500 versus the buffer size. The broadcast type is 1Ring Modified

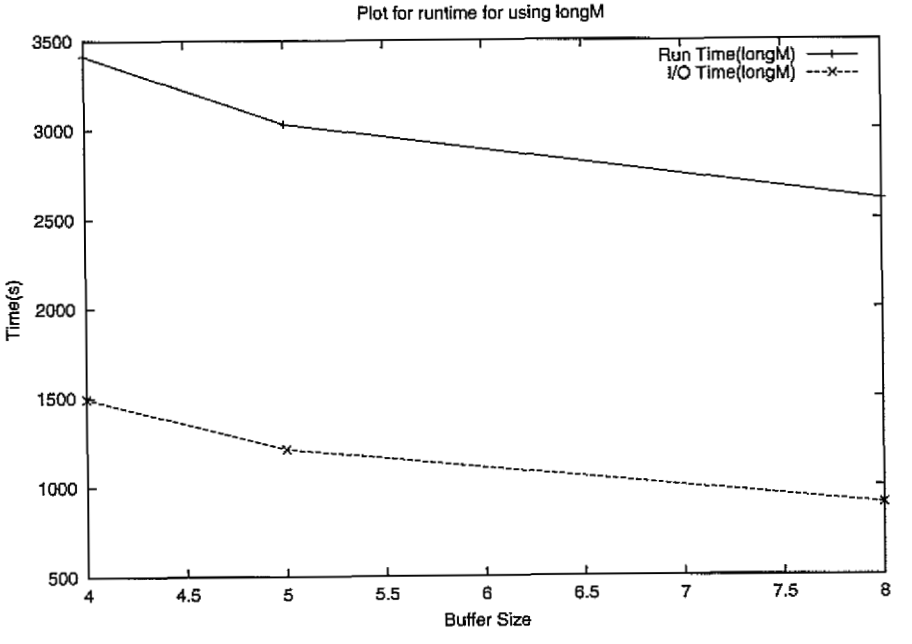

Figure 9.7 Runtime(in seconds) for problem size N=50000, running on 8x8 grid with block size 500 versus the buffer size. The broadcast type is Long Modified

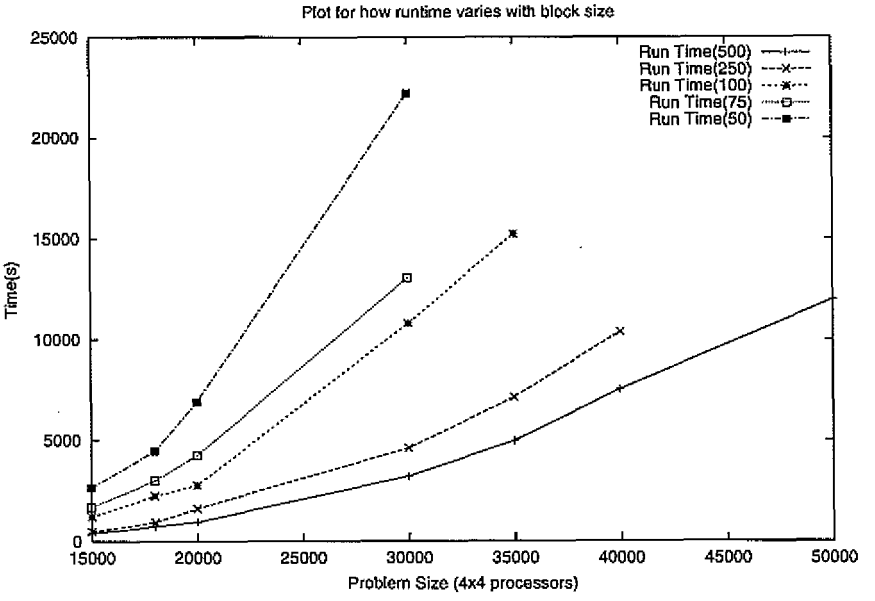

Figure 9.8 Runtimes (in seconds) for different problem sizes with varying block sizes on a 4x4 processor grid. Buffer size is 2.

without any unusual changes.

Figures 9.10 and 9.12 show how even the process grid can influence the performance. It can be observed that a grid with more processes in each row (P=2 x Q=8) performed better better than a grid with more processes in each column (P=8 x Q=2). With a bit of calculation we can check that in an 2x8 process grid there are effectively fewer panels of columns to compute than in an 8x2 process grid. This means that there will be fewer number of reads/writes in the 2x8 grid compared to an 8x2 grid. This again confirms that even though the data read is the same, the number of I/O operations make a difference in the performance. This is more clearly evident in the Figure 9.11 which plots the I/O timings for the two grids.

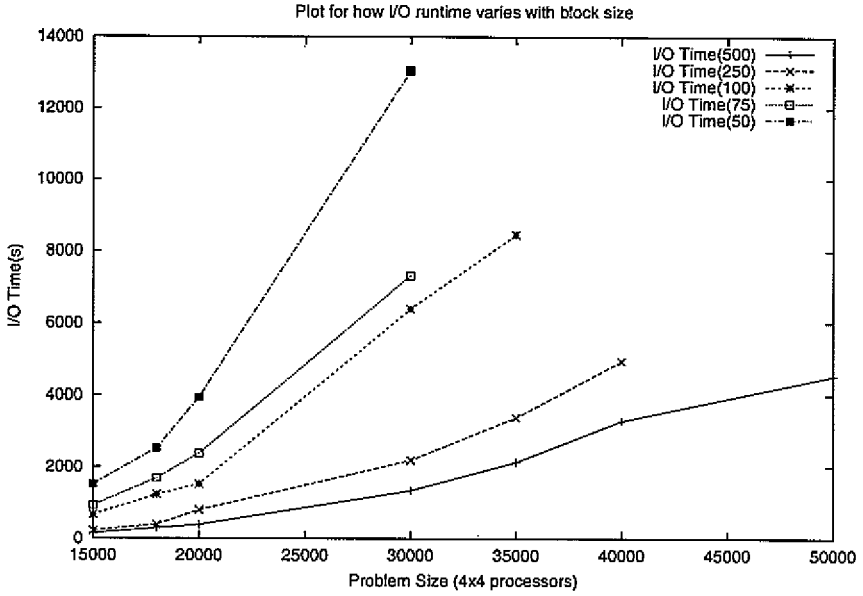

Figure 9.9 I/O Runtimes (in seconds) for different problem sizes with varying block sizes on a 4x4 processor grid. Buffer size is 2.

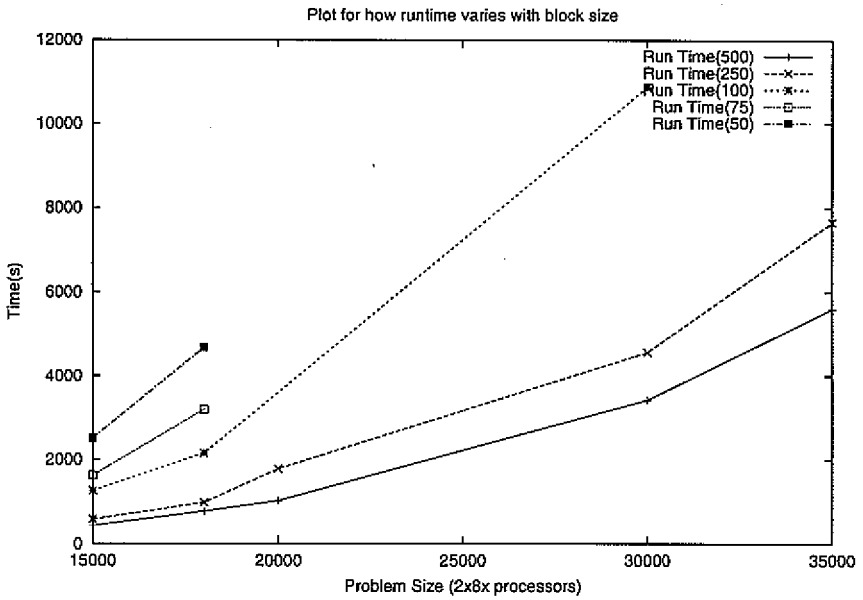

Figure 9.10 Runtimes (in seconds)for different problem sizes with varying block sizes on a 2x8 processor grid.Buffer size is 2. Communication is 1Ring Modified

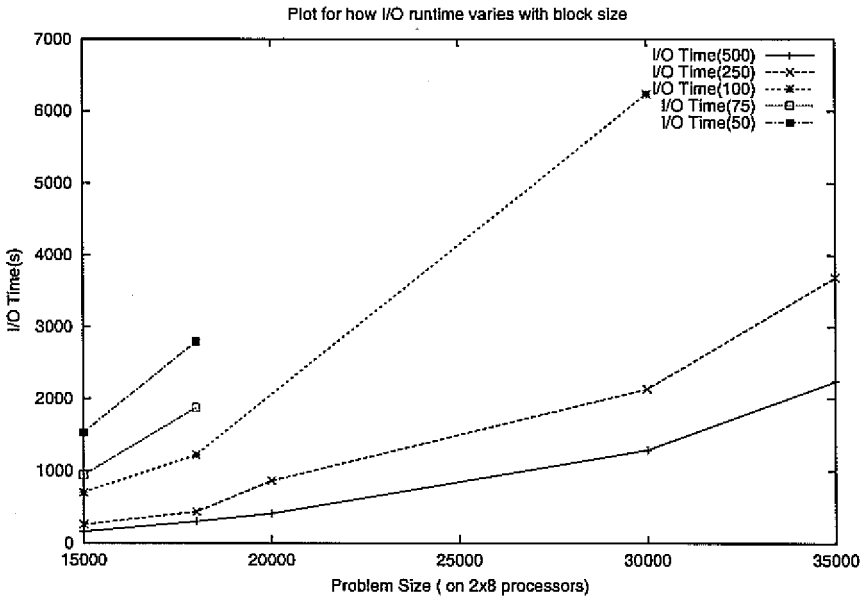

Figure 9.11 I/O Runtimes (in seconds) for different problem sizes with varying block sizes on a 2x8 processor grid. Buffer size is 2. Communication is 1Ring Modified

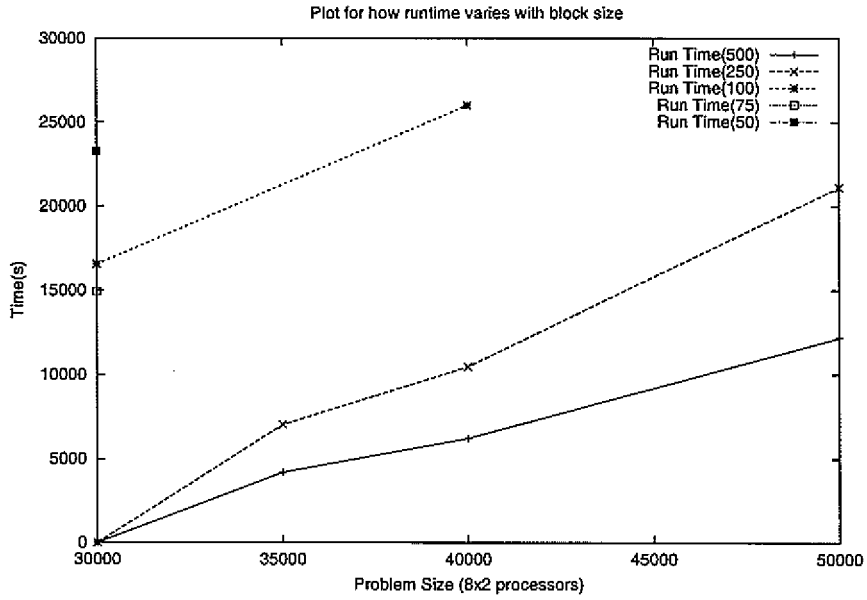

Figure 9.12 Runtimes(in seconds) for different problem sizes with varying block sizes on a 8x2 processor grid.Buffer size is 2.Communication is 1RingModified

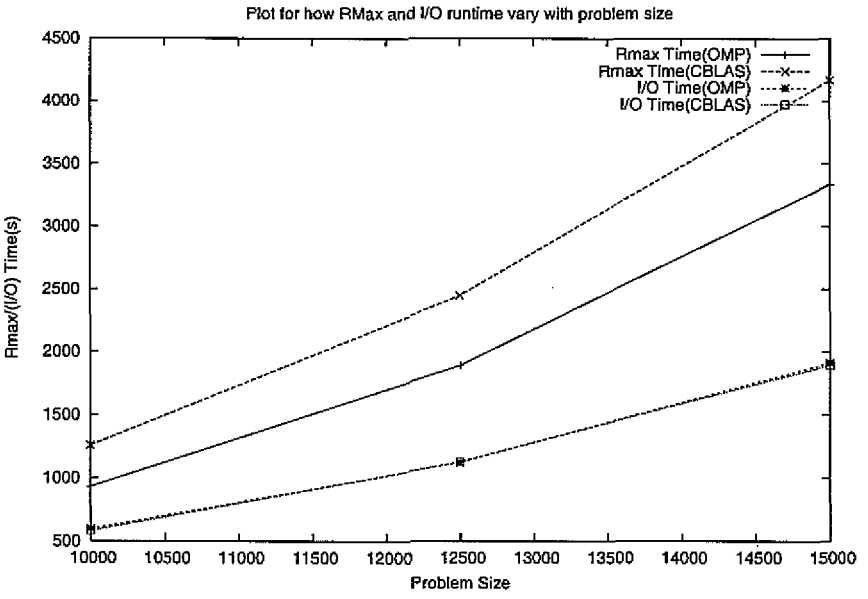

Figure 9.13 Plot for varying problem size, with fixed block size of 100. Using OpenMP

9.2 Performance Of OpenMP

The parameters that influence the timings of the thread based section of the code, are the parameters that are involved in the BLAS algorithms. These include primarily the Problem Size (N), Block Size (NB), and Panel factorization options.

The plot 9.13 shows the effect of increasing problem size on the runtimes and also the I/O time. The graph shows the values for HPL with OpenMP and using unoptimized BLAS. It can be easily seen that the I/O time is not influenced by the addition of OpenMP threads. This just verifies that the out-of-core part of HPL is independent of OpenMP threads. It can also be observed that thread performance marginally improves as problem size increases.

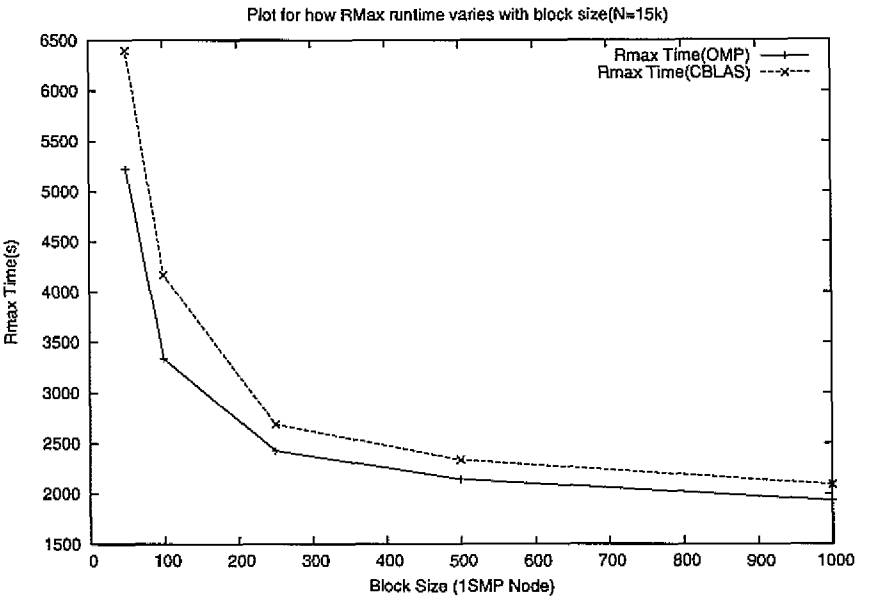

Figure 9.14 R_{max} times for problem size 15000 with increasing block size. Using OpenMP

The plot 9.14 shows the total runtimes (R_{max}) for problem size N=15000 and with block size (NB) as the x-axis. This plot shows that OpenMP performs better as the block size increases. This is expected as the block size directly determines the amount of work being done by the threads at a time. Another observation that can be made from the plot, is that the timings change rapidly for block sizes smaller than *256kb* and changes less for block sizes more than 256kb. This also turns out to be the block size for *Seaborg*, the system at NERSC that the program was executed.

The plot 9.15 shows total I/O times for the above runs. This shows that I/O is unaffected by OpenMP threads.

The plot 9.16 shows the effect of changing the number of columns for panel factorization. The runs are for two problem sizes 15000 and 12500, each with a block distribution size of 100. The figure shows that the performance is best at column size of 16. This is also the number of processors present on the SMP node that the program was run on. Increasing the number of threads more than the available processors does not provide

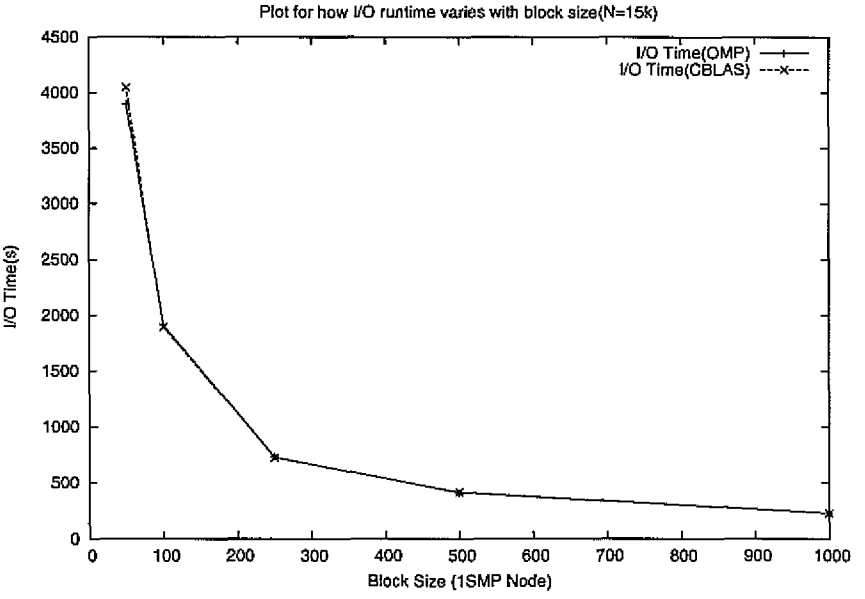

Figure 9.15 I/O times for problem size 15000 with increasing block size. Using OpenMP

any benefit; though the performance does not reduce by much for large values of column size(>64)

Figure 9.17 shows the performance of the BLAS routine *Dgemm* by increasing the column size, for a fixed problem size and block size. The figure shows the values for problem sizes 15000 and 12500 with block size of 100. Unlike figure 9.16, the above routine is largely unaffected by column size. This observation can be attributed to the fact that almost all the routine calls are made in the Update phase of the HPL algorithm which is after Panel Factorization.

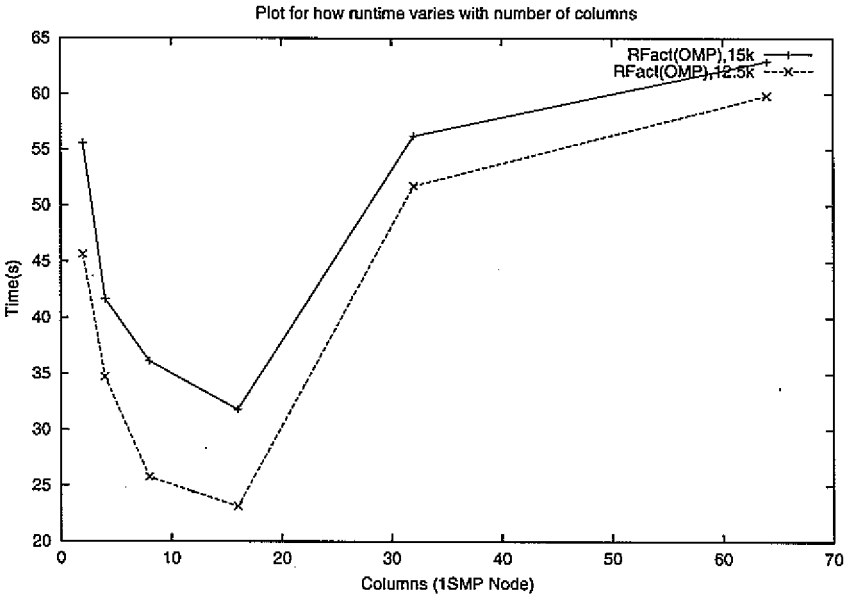

Figure 9.16 Factorization times for problem sizes 15k and 12.5k with block size 100. Using OpenMP

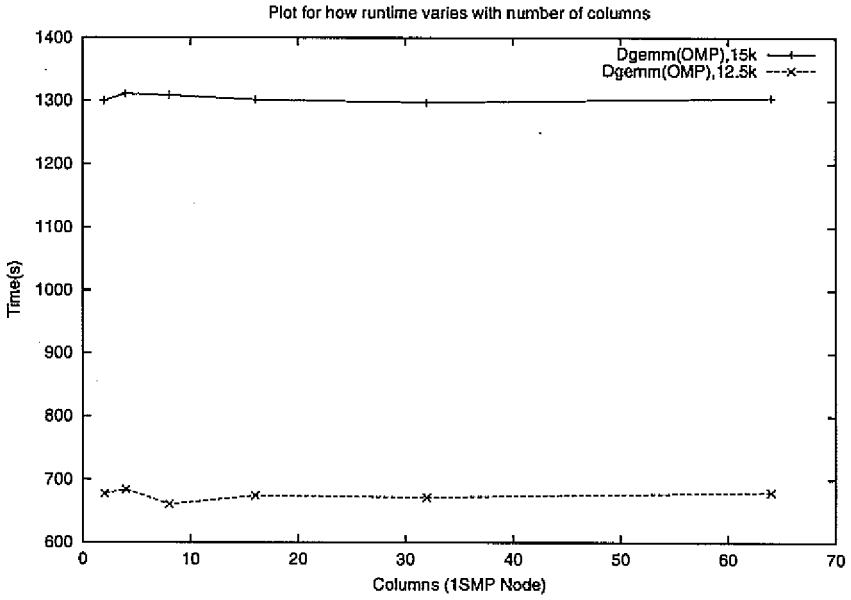

Figure 9.17 Dgemm times for problem sizes 15k and 12.5k with block size 100. Using OpenMP

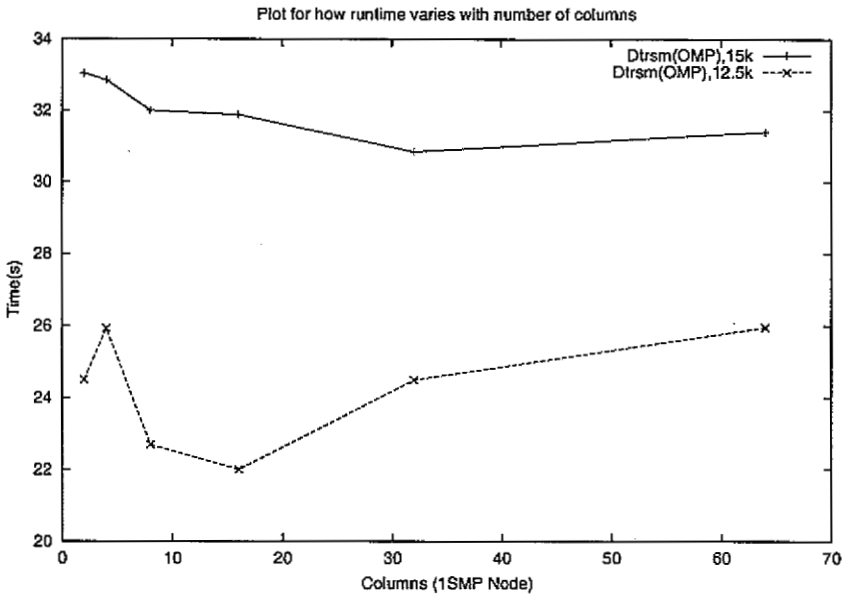

Figure 9.18 Dtrsm times for problem sizes 15k and 12.5k with block size 100. Using OpenMP

Figure 9.18 shows the performance of another BLAS routine *Dtrsm* for the same runs as mentioned for Dgemm. The changes are more noticeable here as the values are an order of magnitude smaller and number of calls to *Dtrsm* are significantly less than *Dgemm*. On average the performance is better when the number of columns is close to the number of processors on the SMP node.

Runs were also made by varying the number of OpenMP threads created during the execution of the benchmark. The values include both less than the available processors as well as more. The results have been plotted in 9.19. The figure shows that the performance increases exponentially until all the available processors are used. After peaking at one thread per processor, any extra threads allocated does not improve the performance. The performance is not far from optimal when the thread count is more than the SMP processors available.

The above algorithm is a cache optimized algorithm and is based on the work of Meng-Shiou [57]. The paper goes into detail on how, a cache aware algorithm as well as

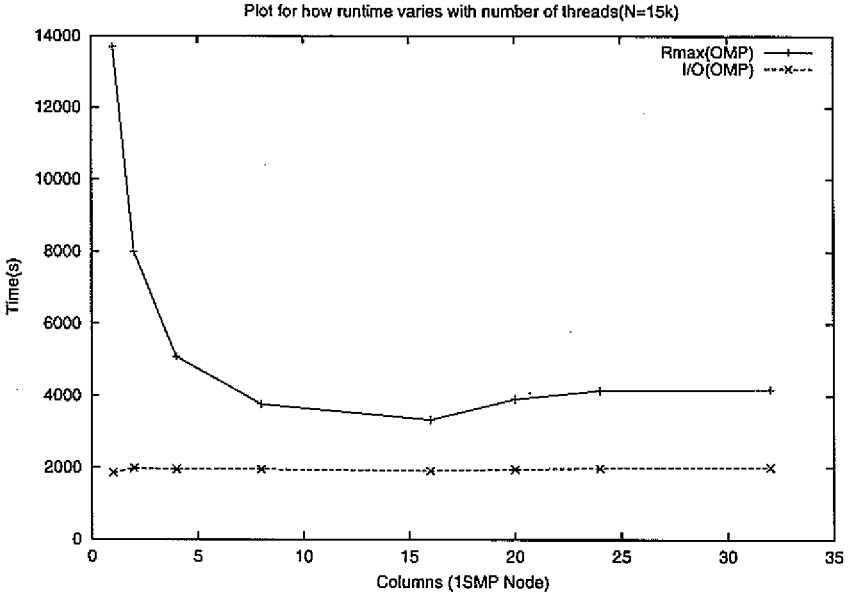

Figure 9.19 Runtimes for problem size 15000 with varying OpenMP threads

the use of threads, can impact the performance of a matrix multiply algorithm.

CHAPTER 10. Conclusions

The original HPL algorithm makes the assumption that all data can be fit entirely in the main memory. This assumption will obviously give a good performance due to the absence of disk I/O. However, not all applications can fit their entire data in memory. These applications which require a fair amount of I/O to move data to and from main memory and secondary storage, are more indicative of usage of an Massively Parallel Processor (MPP) System. Given this scenario a well designed I/O architecture will play a significant part in the performance of the MPP System on regular jobs. And, this is not represented in the current Benchmark. The modified HPL algorithm is hoped to be a step in filling this void.

The most important factor in the performance of out-of-core algorithms is the actual I/O operations performed and their efficiency in transfering data to/from main memory and disk. Various methods were introduced in the report for performing I/O operations. The I/O method to use depends on the design of the out-of-core algorithm. Conversely, the performance of the out-of-core algorithm is affected by the choice of I/O operations. This implies, good performance is achieved when I/O efficiency is closely tied with the out-of-core algorithms.

The out-of-core algorithms must be designed from the start. It is easily observed in the timings for various plots, that I/O plays a significant part in the overall execution time. This leads to an important conclusion, retro-fitting an existing code may not be the best choice. The right-looking algorithm selected for the LU factorization is a recursive algorithm and performs well when the entire dataset is in memory. At each stage of

the loop the entire trailing submatrix is read into memory panel by panel. This gives a polynomial number of I/O reads and writes. If the left-looking algorithm was selected for the main loop, the number of I/O operations involved will be linear on the number of columns. This is due to the data access pattern for the left-looking factorization. The right-looking algorithm performs better for in-core data, but the left-looking will perform better for out-of-core data due to the reduced I/O operations. Hence the conclusion that out-of-core algorithms will perform better when designed from start.

The out-of-core and thread based computation do not interact in this case, since I/O is not done by the threads. The performance of the thread based computation does not depend on I/O as the algorithms are in the BLAS algorithms which assumes all the data to be in memory. This is the reason the out-of-core results and OpenMP threads results were presented separately and no attempt to combine them was made. In general, the modified HPL performs better with larger block sizes, due to less I/O involved for out-of-core part and better cache utilization for the thread based computation.

APPENDIX A. HPL Input File

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
5            device out (6=stdout,7=stderr,file)
data       data directory (if any)
Adat1       data filename prefix (if any)
2            Buffer size (SIZE>=2)
1            # of problems sizes (N)
6 Ns
1            # of NBs
5      NBs
1            # of process grids (P x Q)
1 1 2  Ps
1 2 1  Qs
16.0         threshold
1            # of panel fact
0 1 2        PFACTs (0=left, 1=Crout, 2=Right)
1            # of recursive stopping criterion
2          NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
```

```
1            # of recursive panel fact.
0 1 2        RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
1            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
1            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
64           swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
```

APPENDIX B. HPL Out-of-Core Data Structures

```
#ifndef HPL_OCORE_H
#define HPL_OCORE_H
/*
 * ---------------------------------------------------------------------
 * Include files
 * ---------------------------------------------------------------------
 */

/*
 *   macro constants for out-of-core routines
 */
#define  MAX_BUFS 100

/*
 * ---------------------------------------------------------------------
 * #typedefs and data structures
 * ---------------------------------------------------------------------
 */
typedef enum
{
   A_PTR           = 401,
```

```
   WORK_PTR        = 402,
   L2_PTR          = 403,
   L1_PTR          = 404,
   DPIV_PTR        = 405,
   DINFO_PTR       = 406,
   U_PTR           = 407,
   IWORK_PTR       = 408,
   W_PTR           = 409,
   X_PTR           = 410
} HPL_DISK_PTR;


typedef struct HPL_S_pnlq
{
  int bufferno;
  int panelno;
  int offset;
  void *vptr;
  struct HPL_S_pnlq * prev;
  struct HPL_S_pnlq * next;
} HPL_T_pnlq;

typedef struct HPL_S_dkptr
{
   char fname[HPL_LINE_MAX];
   int lda;
   int panelsize;
```

```
   int lastpanelsize;
   int nblks;
   double **bufs;
   double *minptr1;
   double *minptr2;
   double *maxptr1;
   double *maxptr2;
   int posinmem1;
   int posinmem2;
   int curpos1;
   int curpos2;
   int panelinmem1;
   int panelinmem2;
   int blksinmem;
   int localcurpos;
   int maxpos;
   int offset;
   int p2rpl;
   int myrow;
   int mycol;
   HPL_DISK_PTR ptrtype;
   int fd;
   double * origptr;
} HPL_T_dkptr;
/*
 *---------------------------------------------------
 * global structures for easy access
```

```
 *---------------------------------------------------
 */
HPL_T_dkptr ptr4A;
HPL_T_pnlq * pnlqueue;
HPL_T_pnlq * pnlq_head;
/*HPL_T_pnlq * pnlq_tail=NULL;*/
char fpfix[HPL_LINE_MAX];
int         bufsize;
int         l2_pts_a;
int         bcast_a_l2[2];
int         bcast_pos[2];
int         bcast_plen[2];
int         bcast_ncol[2];
int       * bcast_rcol[2];
double *    bcast_ptr[2];
double *    bcast_pack_vptr[2];
int         bcast_pack_alloc[2];
HPL_DISK_PTR                bcast_buf_ptr_type[2];
/*
 * ---------------------------------------------------------------------
 * Function prototypes
 * ---------------------------------------------------------------------
 */
double *        HPL_oocptr
STDC_ARGS( (
   double *,
   const int,
```

```
    const int,
    const int,
    HPL_DISK_PTR,
    int *,
    const int
) );
```

APPENDIX C. OpenMP Without Modification

The following piece of code is from the Level 3 BLAS routine DGEMM. HPL_dgemm performs one of the matrix-matrix operations $C := alpha * op(A) * op(B) + beta * C$ where $op(X)$ is one of $op(X) = X$ or $op(X) = X^T$. Alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix, op(B) a k by n matrix and C an m by n matrix.

```
/* Original HPL Routine */
for( j = 0, jbj = 0, jcj  = 0; j < N; j++, jbj += LDB, jcj += LDC )
{
   HPL_dscal( M, BETA, C+jcj, 1 );
   for( l = 0, jal = 0, iblj = jbj; l < K; l++, jal += LDA, iblj += 1 )
   {
      t0 = ALPHA * B[iblj];
      for( i = 0, iail = jal, icij = jcj; i < M; i++, iail += 1, icij += 1 )
      { C[icij] += A[iail] * t0; }
   }
}
```

The same piece of code as above, now including OpenMP threads is given below.

```
/* Modified HPL Routine */
#pragma omp parallel for private(i,j,l,t,ia,ib,ic,factor)
                         shared(M,N,K,LDA,LDB,LDC,A,B,C,ALPHA,BETA)
```

```
for(j=0; j<N; j++) {
  ic = j*LDC;
  ib = j*LDB;
  for(t=0; t<M; t++) C[ic+t] *= BETA;
  for(l=0; l<K; l++) {
    ia = l*LDA;
    factor = ALPHA*B[ib+l];
    for(i=0; i<M; i++) {
      C[ic+i] += factor*A[ia+i];
    }
  }
}
```

APPENDIX D. OpenMP With Modification

The following piece of code is from the Level 3 BLAS routine DTRSM. HPL_dtrsm solves one of the matrix equations $op(A) * X = alpha * B$, or $X * op(A) = alpha * B$, where alpha is a scalar, X and B are m by n matrices, A is a unit, or non-unit, upper or lower triangular matrix and $op(A)$ is one of $op(A) = A$ or $op(A) = A^T$. The matrix X is overwritten on B. No test for singularity or near-singularity is included in this routine. Such tests must be performed before calling this routine.

```
/* Original HPL Routine */
for( j = 0, jaj = 0, jbj = 0; j < N; j++, jaj += LDA, jbj += LDB )
{
   for( i = 0, ibij = jbj; i < M; i++, ibij += 1 ) { B[ibij] *= ALPHA; }
   for( k = 0, iakj = jaj, jbk = 0; k < j; k++, iakj += 1, jbk += LDB )
   {
      for( i = 0, ibij = jbj, ibik = jbk; i < M; i++, ibij += 1, ibik += 1 )
      { B[ibij] -= A[iakj] * B[ibik]; }
   }
}
```

The above routine has then been changed to include OpenMP threads and the result follows. The original routine cannot be parallelized as is. The outmost loop is over the columns N, and the second inner loop is again over columns but only partially. This meant the parallelization had to be done over rows M, and not columns. That required

splitting the main loop into two parts and parallelizing them separately. The first loop involved scaling the columns and it was simpler to parallelize on columns. The second loop had to be exchanged with the inner loop to make the rows at the outer loop and enabling us to parallelize.

```
/* Modified HPL Routine */
#pragma omp parallel for private(j,t,jLDB) shared(N,M,LDB,B,ALPHA)
    for(j=0;j<N;j++) {
      jLDB = j*LDB;
      for (t=0;t<M;t++) { B[t+jLDB] *= ALPHA;  }
    }
#pragma omp parallel for private(k,i,j,kLDB,jLDA,jLDB) shared(N,M,A,B,LDB,LDA)
    for(i=0;i<M;i++) {
      for (j=0;j<N;j++) {
        jLDB = j*LDB;
        jLDA = j*LDA;
        factor = B[i+jLDB];
        for (k=0;k<j;k++) {
          kLDB = k*LDB;
          factor -= A[k+jLDA] * B[i+kLDB];
        }
        B[i+jLDB] = factor;
      }
    }
```

BIBLIOGRAPHY

[1] Parallel Performance Benchmark page. Last accessed August 4, 2004. Available at http://www.cs.man.ac.uk/cnc/projects/ecovism/benchmarks.html

[2] J. Dongarra et al. HPL - A Portable Implementation of the High-Performance Linpack. Last accessed August 4, 2004. This benchmark is made available at http://www.netlib.org/benchmark/hpl/

[3] J. Dongarra et al. The Linpack Benchmark. Latest release July 22, 2004. This benchmark is made available at http:/www.top500.org/lists/linpack.php

[4] J. Dongarra, J. Bunch, C. Moler and G. W. Stewart. "LINPACK Users Guide". SIAM, Philadelphia, PA, 1979.

[5] MPI - The Message Passing Interface Standard. Latest version released July 22, 2004. Available at http://www-unix.mcs.anl.gov/mpi/

[6] M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra, MPI: The Complete Reference, MIT Press, Cambridge, Massachusetts, 1996.

[7] Basic Linear Algebra Subprograms. Accessed July 22, 2004. Available at http://www.netlib.org/blas/

[8] Hans-Werner Meuer. "The Mannheim Supercomputer Statistics 1986-1992". Accessed August 4, 2004. Available at http://www.top500.org/reports/1993/chapter2_4.html

[9] J. Dongarra and P. Luszczek and A. Petitet. "The LINPACK Benchmark: Past, Present, and Future". Concurrency and Computation: Practice and Experience 15, pp. 1-18, 2003

[10] J. Dongarra, I.S. Duff, D.C. Sorensen and H.A. van der Vorst. "Solving Linear Systems on Vector and Shared Memory Computers". Society for Industrial and Applied Mathematics, 1991.

[11] S. Huss-Lederman, E. M. Jacobson, A. Tsao, T. Turnbull and J. R. Johnson, "Implementation of Strassen's algorithm for matrix multiplication". Proceedings of the 1996 ACM/IEEE conference on Supercomputing, p.32-es, November 1996, Pittsburgh, United States.

[12] B. Grayson and R. van de Geijn,"A High Performance Parallel Strassen Implementation," Parallel Processing Letters, Vol 6, No. 1, 1996.

[13] M.S. Thottethodi, S. Chatterjee and A.R. Lebeck. "Tuning Strassen's Matrix Multiplication for Memory Efficiency", Proceedings of Supercomputing '98, November 1998.

[14] S. Toledo "Locality of reference in LU decomposition with partial pivoting". SIAM J. Matrix Anal. Appl. 18, no. 4, 1065-1081, MathSciNet, 1997.

[15] J. W. Demmel and R. S. Schreiber, "Stability of Block LU Factorization", Numerical Linear Algebra with Applications, 2(2):173-190, 1995.

[16] K. Hwang and Z. XU, "Scalable Parallel Computing: Technology, Architecture, Programming". McGraw-Hill, Inc., New York, NY, 1998.

[17] J. May, "Parallel I/O for High Performance Computing".
San Franscisco, CA: Morgan Kaufmann Publishers, 2001.

[18] A. Grama, et al. Introduction to Parallel Computing.
New York: Addison Wesley, 2003.

[19] C. Lawson, R.Hanson, D.Kincaid and F.Krogh. "Basic Linear Algebra Subprograms for Fortran Usage". ACM Transactions on Mathematical Software,(308-325), Sept 1979.

[20] J.J. Dongarra, J. DuCroz, S. Hammarling and R. Hanson. "An Extended Set of Fortran Basic Linear Algebra Subprograms". ACM Transactions on Mathematical Software,(1-32), March 1988.

[21] J.J. Dongarra, J. Du Croz, S. Hammarling and I.S. Duff (1990)."A set of level 3 basic linear algebra subprograms". ACM Transactions on Mathematical Software, (1-17), March 1990.

[22] A. Chtchelkanova, J. Gunnels, G. Morrow, J. Overfelt and R. Van De Geijn, "Parallel Implementation of BLAS: General Techniques for Level 3 BLAS", Tech. Rep. TR95-49, Department of Computer Sciences, UT-Austin, 1995.

[23] M.J. Quinn. "Parallel Programming in C with MPI and OpenMP". Dubuque,Iowa: McGraw-Hill, 2003.

[24] H. Pfneiszl and G. Kotsis. "Benchmarking Parallel Processing Systems: A Survey". Technical Report, University of Vienna, 1996.

[25] T. Guignon. "BLASTH, a BLAS library for dual SMP computer". Laboratoire ASCI, Orsay, France, 1999.

[26] C. Addison, V. Getov, A. Hey, R.W. Hockney and I.C. Wolton. "Benchmarking for Distributed Memory Parallel Systems: Gaining Insight from Numbers". Parallel Computing, 1653-1668,Vol 20, Nov 1994.

[27] J. Dongarra, I. S. Duff, D.C. Sorensen and H. van der Vorst. "Numerical Linear Algebra on High-Performance Computers". Society for Industrial & Applied Mathematics,Philadelphia,PA, 1998.

[28] E. Rosti, G. Serazzi, E. Smirni and M.S. Squillante, "Models of Parallel Applications with Large Computation and I/O Requirements", IEEE Transactions on Software Engineering, Vol. 28, No. 3, pp. 286-307, March 2002.

[29] E. Smirni, R. A. Aydt, A. A. Chen and D. A. Reed, "I/O Requirements of Scientific Applications: An Evolutionary View", Proceedings of the High Performance Distributed Computing (HPDC '96), p.49, August 06-09, 1996.

[30] R. Oldfield and D. Kotz. "Scientific applications using parallel I/O". In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, High Performance Mass Storage and Parallel I/O: Technologies and Applications, chapter 45, pages 655-666. IEEE Computer Society Press and John Wiley & Sons, 2001.

[31] D. G. Feitelson, P. F. Corbett, S. J.Baylor and Y.Hsu, "Parallel I/O Subsystems in Massively Parallel Supercomputers", IEEE Parallel and Distributed Technology: Systems & Technology, v.3 n.3, p.33-47, September 1995.

[32] M.Cohen Austrowiek and P.Grassi, "UNIX I/O Performance Measurement Methodologies Applied To Old And New Storage Technologies", CMG-Italia and EuroCMG 2002.

[33] T.M. Ruwart, "File System Benchmarks, Then, Now, and Tomorrow", Projceedings, 18th IEEE Symposium on Mass Storage Systems and Technologies / 9th NASA Goddard Conference on Mass Storage Systems and Technologies, 2001.

[34] S.Toledo, "A survey of out-of-core algorithms in numerical linear algebra", External memory algorithms, American Mathematical Society, Boston, MA, 1999.

[35] J. S. Vitter, "External Memory Algorithms and Data Structures: Dealing with MASSIVE DATA", ACM Computing Surveys, 33(2), June 2001, 209-271.

[36] J. Nieplocha and I. Foster, "Disk Resident Arrays: An Array-Oriented I/O Library for Out-Of-Core Computations", Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation, p.196, March 27-31, 1996.

[37] S. Asami, N. Talagala and D. Patterson, "Designing a Self Maintaining Storage System",Proceedings of the 1999 IEEE Symposium on Mass Storage Systems, 1999.

[38] D. Kotz, "Disk-directed I/O for an Out-of-Core Computation", Technical Report: PCS-TR95-251, Dartmouth College, Hanover, NH, 1995.

[39] M. Kandemir, J. Ramanujam and A. Choudhary, "Improving the Performance of Out-of-Core Computations", International Conference on Parallel Processing (ICPP '97).

[40] E. Masciari, C. Pizzuti, G. Raimondo and D. Talia, "Using an Out-of-Core Technique for Clustering Large Data Sets", Proceedings DEXA 2001 Workshops, IEEE Computer Society Press, pp. 133-137, Munich, September 2001.

[41] R. Rabenseifner and A. E. Koniges,"Effective Communication and File-I/O Bandwidth Benchmarks", Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, 24-35, 2001.

[42] Z. Li, J.H. Reif and S.K.S. Gupta, " Synthesizing Efficient Out-of-Core Programs for Block Recursive Algorithms Using Block-Cyclic Data Distributions", IEEE Transactions on Parallel and Distributed Systems, March 1999.

[43] S. Chatterjee, V.V. Jain, A.R. Lebeck, S. Mundhra and M. Thottethodi, "Nonlinear array layouts for hierarchical memory systems", Proceedings of the 13th international conference on Supercomputing, p.444-453, June 20-25, 1999, Rhodes, Greece.

[44] C.E. Leiserson, S. Rao and S. Toledo, "Efficient out-of-core algorithms for linear relaxation using blocking covers", Journal of Computer and System Sciences, v.54 n.2, p.332-344, April 1997.

[45] S. Toledo and F.G. Gustavson, "The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations", Proceedings of the fourth workshop on I/O in parallel and distributed systems: part of the federated computing research conference, p.28-40, May 27-27, 1996.

[46] W.C. Reiley and R.A. van de Geijn, "POOCLAPACK: Parallel Out-of-Core Linear Algebra Package", University of Texas at Austin, Austin, TX, 1999.

[47] J. Dongarra and E.F. D'Azevedo, "The Design and Implementation of the Parallel Out-of-core ScaLAPACK LU, QR and Cholesky Factorization Routines", ORNL/TM-13372, April 1997.

[48] J.J. Dongarra, S. Hammarling and D. W. Walker, "Key Concepts for Parallel Out-Of-Core LU Factorization", Parallel Computing, Vol. 23, pages 49-70, 1997.

[49] ATLAS (Automatically Tuned Linear Algebra Software) project. Latest version released December 2003. Available at http://math-atlas.sourceforge.net/

[50] R. Thakur, W. Gropp and E. Lusk. "A case for using MPI's derived datatypes to improve I/O performance". In Proceedings of SC98: High Performance Networking and Computing, November 1998.

[51] R. Rew and G. Davis. "NetCDF: An interface for scientific data access". IEEE Computer Graphics and Applications, 10(4):76-82, July 1990.

[52] HDF5-A New Generation of HDF. Accessed July 22, 2004. Online HDF5 documentation at http://hdf.ncsa.uiuc.edu/HDF5/

[53] J. Barkes, M.R. Barrios, F. Cougard, P. Crumley, D. Marin, H. Reddy and T. Thitayanun, "GPFS: A Parallel File System", IBM International Technical Support Organization, April 1998. Available at www.redbooks.ibm.com

[54] S. Soltis, G. Erickson, K. Preslan, M. O'Keefe and T. Ruwart,"The Global File System: A File System for Shared Disk Storage", IEEE Transactions on Parallel and Distributed Systems, October 1997.

[55] D. Womble, D. Greenberg, R. Riesen and S. Wheat, "Out of core, out of mind: Practical parallel I/O". In Proceedings of the Scalable Parallel Libraries Conference, pages 10-16, Mississippi State University, October 1993.

[56] P.M. Chen and D.A. Patterson, "A new approach to I/O performance evaluation: self-scaling I/O benchmarks, predicted I/O performance". Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems, May 1993.

[57] MS. Wu, S. Aluru and R. Kendall, "Mixed Mode Matrix Multiplication". IEEE International Conference on Cluster Computing (CLUSTER'02), Sept 2002.

ACKNOWLEDGMENTS

This work was performed under auspices of the U. S. Department of Energy under contract W-7405-Eng-82 at Ames Laboratory operated by the Iowa State University of Science and Technology. The United States government has assigned the DOE Report number IS-T 2196. Funding was provided by the Mathematical, Information and Computational Science division of the Office or Advanced Scientific Computing Research. This research was performed in part using computational resources of the National Energy Research Scientific Computing Center.

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Ricky Kendall for his guidance, patience and support throughout this research and the writing of this thesis. His insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education. I would also like to thank my committee members for their efforts and contributions to this work: Dr. Simanta Mitra and Dr. Mark Gordon

I am indebted to my parents for providing me with all affection and support at all times of needs. My graduate study would not have been possible without their moral support. I would like to thank all my friends who have been a part of my graduate study. Special thanks to Melanie Eckhart for making me feel at home at school ever since the first day I joined Iowa State University.