To the Graduate Council:

I am submitting herewith a thesis written by James S. Payne entitled "Porting an Implicit Method of Lines Solver to Distributed-Memory Parallel Processors." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Dr. Jack J. Dongarra, Major Professor

We have read this thesis
and recommend its acceptance:

Dr. William F. Lawkins

Dr. Michael W. Berry

Accepted for the Council:

Associate Vice Chancellor
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgement of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature James S. Payne

James S. Payne

Date February 10, 1994

Porting an Implicit Method of Lines Solver to Distributed-Memory Parallel Processors

A Thesis
Presented for the
Master of Science
Degree

The University of Tennessee, Knoxville

James S. Payne

May 1994

Abstract

This thesis presents the design, implementation and evaluation of a strategy for the porting of a numerical method for solving partial differential equations from a sequential processor platform to a parallel processor platform while minimizing the rewriting of existing code. Two different architectures are used in the evaluation, the Intel iPSC/860 which is a distributed memory system and the Kendall Square Multiprocessor which is a virtual shared memory system.

This evaluation uses a numerical method called the implicit Method of Lines to approximate a partial differential equation by a system of ordinary differential equations and then uses an ordinary differential equation solver developed for a sequential processor to compute an approximation to the partial differential equation. The ordinary differential equation solver is used without modification, and the speed-up of the calculated solution is obtained by parallelizing the formation of a Jacobian matrix and the solution of the linear system of equations involving that matrix.

The results include the speed-up that is obtained for the solution of the partial differential equation and the performance rate that is obtained for the LU decomposition by the linear algebra routines in the solution of the system of equations. Also presented is the method of porting distributed-memory processor routines to a virtual shared-memory processors.

Table of Contents

List of Figures

Chapter 1

Introduction

In this thesis a strategy for porting a numerical method for solving partial differential equations, the implicit Method of Lines, from a sequential processor platform to a parallel processor platform is devised, implemented and evaluated. A parallel processor platform offers the potential of allowing the solution of much larger problems because of reduced time to solve such a problem. However, to take advantage of this increased ability, the need to rewrite large, well-developed, and time-tested code is generally a requirement. The strategy considered here involves the use of a sequential implicit Method of Lines package in conjunction with selected parallel routines and parallel linear algebra libraries to produce an efficient use of the parallel architecture without having to modify the ordinary differential equations solver used in the implicit Method of Lines package. The effectiveness of this strategy is evaluated on two distinct parallel architectures.

The Method of Lines is a generic name used for a numerical method used to solve partial differential equations (PDEs) by approximating a PDE with a system of ordinary differential equations (ODEs). An approximate solution to the PDE is then calculated using library routines generally referred to as ODE solvers. The test problem used in this thesis is nonlinear and stiff. Because the test problem is stiff, the system of ODEs is solved implicitly. Further, because the test problem is nonlinear, each new solution determined by the implicit ODE solver is the solution to a nonlinear system of algebraic equations. The ODE solver used in this thesis incorporates a form of the Newton-Raphson method to solve that nonlinear system of algebraic equations. To implement Newton-Raphson's method, a linear system of equations is constructed and solved.

This thesis evaluates the implementation of the implicit Method of Lines on a distributed-memory computer system, the Intel iPSC/860 and on a shared virtual memory computer

system, the Kendall Square Research Computer. The Intel iPSC/860 Hypercube Computer will be referred to as the Intel and the Kendall Square Research Computer as the KSR.

The implicit ODE solver used in this implementation of Method of Lines is DDRIVE[1]. This library of routines is similar to the well known LSODE[2] ODE solver. The parallel linear algebra package used is ScaLAPACK[3]. This package uses a 2-dimension grid of processors to distribute the storage requirement for a matrix and the work of performing the LU factorization. The Basic Linear Algebra Communication package, BLACS[4], which is used in conjunction with ScaLAPACK, is designed to optimize communication on the Intel.

The first objective of this thesis is that a strategy can be devised that will efficiently parallelize the implicit Method of Lines without the need to do an extensive rewrite of the applications program. The second objective is to demonstrate that the same strategy can be used on both a virtual shared-memory and a distributed-memory processor platform. The third objective is to show speed-up obtained by this strategy will not be severely restricted by leaving a major portion of the code sequential. The fourth objective is to demonstrate that the performance rate observed in the linear algebra routines should be similar to that observed in benchmark evaluations of the linear algebra package. The strategy for the implementation being presented in this thesis is to leave the ODE solver and subroutines that are problem dependant unmodified and, within the Newton-Raphson step for approximating the solution to a nonlinear system of equations, to parallelize the formation of the linear system of equations and the solution of that system.

This thesis is divided into five chapters. Chapter 2 provides a brief description of the test problem and Method of Lines numerical method. Chapter 3 describes the strategy used to parallelize the Parallel Method of Lines on the Intel and the KSR and describes the means used to port the Intel ScaLAPACK libraries to the KSR. Chapter 4 provides a brief description of the platforms being used. Chapter 5 presents the empirical data and analysis of that data. Chapter 6 contains the summary and conclusions. Appendix A contains a list of acronyms used in this thesis. Appendix B is a list of terms and their definitions. Appendix C contains

source code for the CSEND and CRECV functions being used to port the Intel ScaLAPACK libraries to the KSR as described in Chapter 3.

Chapter 2

The Test Problem

Description of Problem

The reason for choosing the test problem described in this section is that it has been used for similar studies elsewhere and the results of those studies are published in the literature. The selected test problem is a good example of problems solved by implicit Method of Lines solvers. It is also a benchmark used to evaluate simulation languages in *Benchmark Fluid Flow Problems for Continuous Simulation Languages*[5] and the performance of a shared memory parallel computer in *Experiments with Method of Lines Solvers on a Shared Memory Computer*[6].

The test problem characterizes the flow of water through a steam generator while the water is still in a sub-cooled state. The problem is to determine the steady state properties of the water given the defined boundary conditions. Starting from an initial guess of the state, the properties of the water oscillate with time until the steady state condition is reached. Tracking the transient state until the properties stabilize is an interesting test problem for the Method of Lines solvers as the problem demonstrates stiffness characteristics that are typical of fluid problems.

This problem is defined by the one-dimensional Euler's Equation

$$\frac{\partial U}{\partial t} + A \bullet \frac{\partial U}{\partial z} = C, \quad 0 < z < L, \tag{1}$$

where

$$C = \left(0, \quad -KG * \left|\frac{G}{\rho}\right| - \rho g_a \sin\theta, \quad \frac{a^2 \ \Phi \ P_H \ \kappa}{C_p \ A_f}\right)^T , \tag{2}$$

$$U = (\ \rho, \ G, \ T\ )^T , \tag{3}$$

and

$$A = \begin{pmatrix} 0 & 1 & 0 \\ \frac{1}{\rho\kappa} - \frac{G^2}{\rho^2} & 2\frac{G}{\rho} & \frac{\beta}{\kappa} \\ \frac{-a^2\beta\bar{T}G}{\rho^2 C_p} & \frac{a^2\beta\bar{T}}{\rho C_p} & \frac{G}{\rho} \end{pmatrix} . \tag{4}$$

The dependant vector,U, is a composite function of density,ρ, flow rate,G, and temperature,T. The values of α, κ, β, C_p in the above matrix are derived from an equation of state for the fluid. A complete definition of symbols and units is found in Appendix B.

The boundary condition determines the inlet density and temperature and the outlet flow rate. For this problem those values are

$$\begin{aligned} \rho(0,t) &= \rho_0 = 795.521 , \\ T(0,t) &= T_0 = 255.000 , \\ G(L,t) &= G_0 = 270.900 . \end{aligned} \tag{5}$$

The selected test problem Equations 1-5 contains three unknowns that are solved simultaneously. Figure 1 illustrates the solution of one of those unknowns, flow rate, G, to give a perspective of the problem.

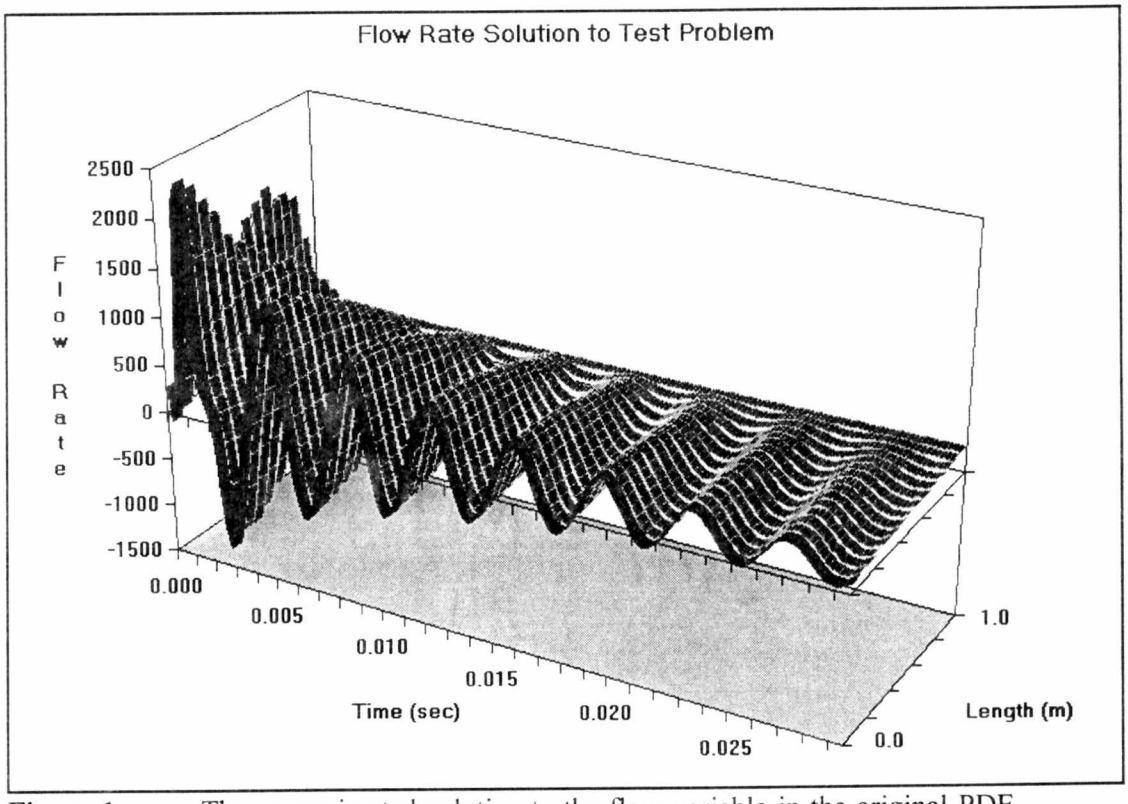

Figure 1 The approximated solution to the flow variable in the original PDE.

Method of Lines Solution of PDEs

The solution of the test problem is approximated numerically using the implicit Method of Lines. The Method of Lines solves Equation 1, first, by discretizing spatial variability and using finite difference approximations for the spatial (z) derivatives and, second, by solving the resulting system of ordinary differential equations using a standard ODE solver software package. To do this, a partition z of the interval [0,L] is defined by $z_i = L*(i-1)/m$, i=1,m+1,

where L is the length of the spatial interval being studied and m is the number of subintervals in the partition. This procedure generates a system of ODEs

$$\frac{dU}{dt} = f(U) \quad , \tag{6}$$

where the i-th member of the system is

$$\frac{dU_i}{dt} = f_i(U) \quad . \tag{7}$$

Define the notation

$$U_{i,j} = U(z_i, t_j) \quad . \tag{8}$$

The implicit ODE solver uses Gear's Method, also known as Backward Differential Formulation (BDF), to approximate dU/dt. To describe Gear's Method, assume the PDE has been approximated from time t_{j-k} through time t_j so that $U_{i,j-k}$ through $U_{i,j}$ are known. A polynomial $P^*(t, U_{i,j+1}, U_{i,j}, U_{i,j-1}, ..., U_{i,j-k-1})$ is constructed using Newton's backward-difference basis functions[7] that is a k-order interpolate of $U = (U_{i,j+1}, U_{i,j}, U_{i,j-1}, ..., U_{i,j-k-1})^T$. The polynomial P^* is then differentiated with respect to time at t_{j+1}, resulting in a system of equations that can be arranged into the form

$$\frac{dP_i^*}{dt} = \alpha_i * U_{i,j+1} + \beta_i \quad , \tag{9}$$

where

$$\alpha_i = \alpha_i (U_{i,j}, U_{i,j-1}, \ldots , U_{i,j-k-1}) \quad , \tag{10}$$

$$\beta_i = \beta_i (U_{i,j}, U_{i,j-1}, \ldots , U_{i,j-k-1}) \quad . \tag{11}$$

Using the approximation

$$\frac{dP^*}{dt} \approx \frac{dU}{dt} \tag{12}$$

and combining Equations 6, 7, and 9 gives

$$\alpha * U + \beta = f(U) \quad , \tag{13}$$

where the i-th component of vector U is $U_i = U_{i,j+1}$.

If f is a nonlinear function of U, then Equation 13 is a nonlinear system of equations for the vector U. The Newton-Raphson method is used to solve that system of nonlinear equations. Define F(U) using Equation 13 as

$$F(U) = f(U) - \alpha * U - \beta \quad . \tag{14}$$

Equation 13 is equivalent to

$$F(U) = 0 \quad . \tag{15}$$

The Newton-Raphson method uses the Jacobian matrix J(U), which is

$$J(U) = \frac{\partial F}{\partial U}(U) = \frac{\partial f}{\partial U}(U) - \alpha I \quad . \tag{16}$$

Define the perturbed vector U' as

$$U' = U + \epsilon e_j \quad , \tag{17}$$

where e_j is the j-th canonical basis vector. To construct the j-th column of the Jacobian matrix, compute

$$J_j(U) = \frac{f(U) - f(U')}{\epsilon} - \alpha e_j \qquad (18)$$

where J_j is the j-th column of the Jacobian matrix J defined by Equation 16.

The Newton-Raphson method is described informally by Algorithm 1.

Algorithm 1 Newton-Raphson Method

Purpose:	To approximate the solution of a nonlinear system F(U) = 0 given the initial approximation U^0
Given:	The number of equations, n, in the nonlinear system of equations, an initial guess of $U = U^0$, the desired accuracy tolerance, TOL, and the maximum number of iterations, Max.
Step 1	Set k = 0
Step 2	while (k < Max)
Step 3	Compute the Jacobian matrix, J^k, where $J^k = \partial F(U^k)/\partial U$
Step 4	A^k = LU factorization of $J^k(U)$
Step 5	Use forward/backsolve on n x n linear system $A^k x = -F(U^k)$
Step 6	Set $U^{k+1} = U^k + x$
Step 7	If $\|x\| <$ TOL break
Step 8	k =k+1
Step 9	end while
Step 10	return

The ODE solver determines the polynomial P^*, controls the time step size, the order of P^* and all the steps in Algorithm 1 except for Steps 3, 4 and 5. The user supplies the routines for the construction of the Jacobian matrix (Step 3) and for solving the linear system of equations (Steps 4 and 5). Steps 3,4, and 5 can be parallelized.

For the test problem used in this thesis, a sparse non-symmetric matrix is generated. However, the implementation considered here does not make use of the matrix sparsity. The reason this

is not done is so the evaluation of potential speed-up that might be achieved is valid for an arbitrary test problem, and not just one that results in a sparse Jacobian matrix.

Chapter 3

Method of Parallelization

Overview of Sequential Flow

Figure 2 shows the flow of the sequential program in flow chart form. The principal parts of the program are

(1) the calculation of the time derivative dU/dt (FSDERIV),

(2) the implicit ODE solver (DDRIVE),

(3) the Jacobian matrix formation (PREPJ),

(4) the LU factorization (DGETRF), and

(5) the solution of the linear system of equations using forward/backsolve (DGETRS).

The program also uses a subroutine YOWSER as an interface routine between DDRIVE and the three subroutines used to form the matrix and perform the linear algebra computations. The program flow starts with the initialization of the problem by defining the spatial mesh based on discretization of the spatial domain requested by the user. The initial value of the solution is set and the initial time derivatives are computed using FSDERIV. Then, for each desired approximation to the solution at a given time, the implicit ODE solver DDRIVE is called. DDRIVE calls the routine FSDERIV or the flow control routine YOWSER. YOWSER calls either PREPJ and DGETRF routines or DGETRS routine based on a flag set by DDRIVE.

For the test problem, Equations 1-5, the calculation of dU/dt is performed by calling the routine FSDERIV, which in turn calls several other problem dependent routines unique to the selected test problem. These routines were written for other studies and details can be found in the literature[5][6] . The implicit ODE solver is DDRIVE[1]. A general outline of the numerical method implemented in DDRIVE is given in Chapter 2.

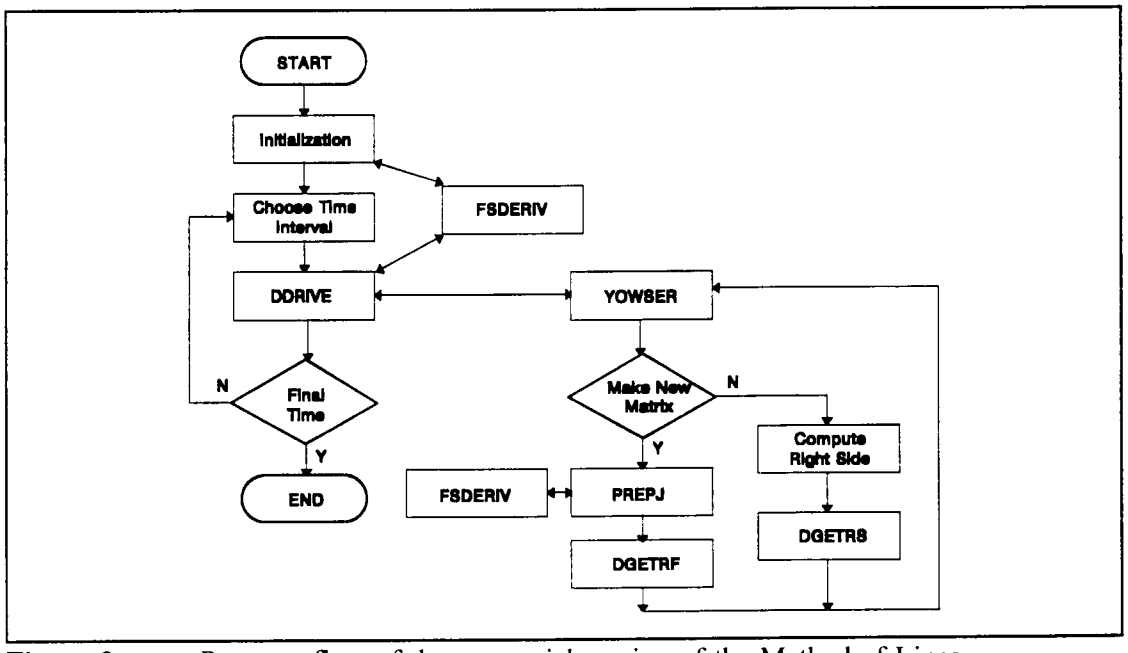

Figure 2 Program flow of the sequential version of the Method of Lines

The routine PREPJ implements Equations 17 and 18 to construct the Jacobian matrix. Equation 18 requires a function evaluation for each column on the Jacobian and this function evaluation must be computed in a single operation. This limits the ability to parallelize the Jacobian matrix formation which is discussed later.

The LU factorization and forward/backsolve steps for solving a linear system of equations use standard linear algebra library routines. The routines used to perform the linear algebra in the sequential version, DGETRF and DGETRS, are from the LAPACK library[8]. The routine YOWSER is used to direct the flow of calls from DDRIVE to the appropriate routines. It is an interface routine and only minor calculations are done in YOWSER.

Parallelization of Program Flow

This thesis examines a strategy to make effective use of parallel computers to solve the test problem without rewriting the complex and time proven implicit ODE solver. Figure 3 shows that the bulk of the time required for a sequential version to execute is the time used for the

formation of the Jacobian matrix, the LU factorization, and the forward/backsolve step. Therefore the parallelization strategy being used in this thesis is to convert these three processes to run in parallel.

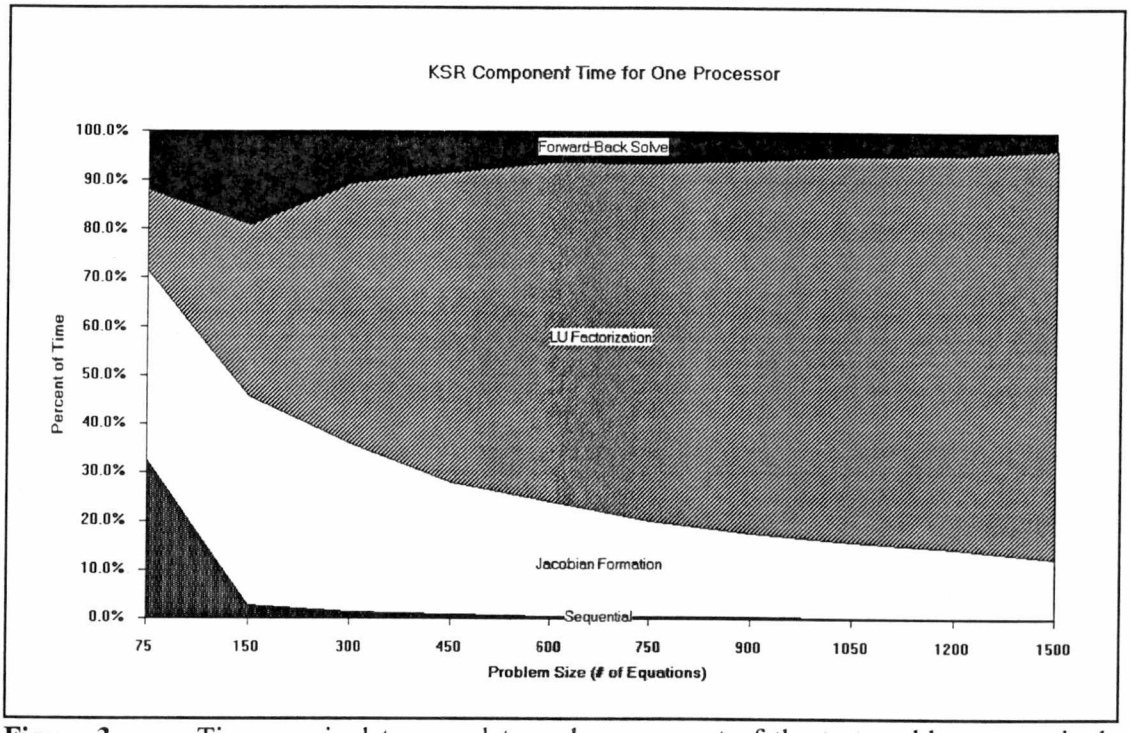

Figure 3 Time required to complete each component of the test problem on a single processor of the KSR

The ScaLAPACK Algorithm

The Method of Lines implementation being used in this thesis uses the ScaLAPACK library to perform the parallel linear algebra. This section briefly describes the method employed in the ScaLAPACK library to perform a parallel LU factorization of a matrix.

The ScaLAPACK algorithm maps an N x N matrix onto a two dimension grid of processors. Given the total number of processors, P_{total}, to be used in the parallel factorization of a matrix, the processors are laid out in a 2D grid such that $P_{total} = P_{row} \times P_{col}$, where P_{row} is the number of processor rows and P_{col} is the number of processor columns. The row number, column

number and processor number are zero indexed. These processors are numbered in row major order such that the $P_n = P_{ij}$ where $n = i * P_{col} + j$.

The linear algebra parallelization method partitions the global matrix into square submatrices of a rank called blocksize or n_b that is set by the user. Each submatrix, A_{ij}, is assigned to processor P_n according to the formula .

$$n = (j-1) \bmod P_{col} + ((i-1) \bmod P_{row}) * P_{col} \quad . \tag{19}$$

This assignment of each submatrix remains constant throughout the computation of the test problem.

In describing the LU factorization, assume the factorization of a NxN matrix in Figure 4 has proceeded through k columns so that all but the labeled portion has been updated. Let m = N-k. Panel B is a submatrix of rank m x n_b contained in a single processor column. Panel C is an n_b x m-n_b submatrix contained in a single processor row and Panel D is a m-n_b x m-n_b submatrix. Let P_{pivot} be the processor to which is assigned the upper n_b x n_b submatrix of B shown as B_f. The submatrix B_f is part of Panel B.

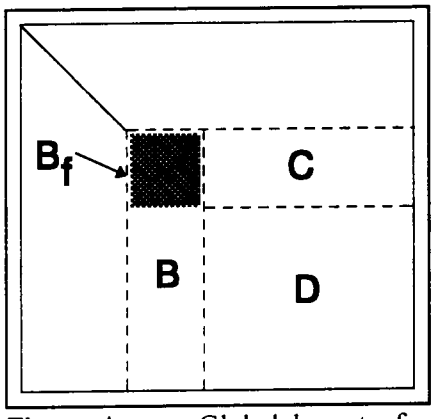

Figure 4 Global layout of a LU factorization of a matrix

During the next step of the factorization, Panel B is factored using row pivoting within Panel B which requires all row processors assigned a portion of Panel B to communicate with processor P_{pivot}. This communication involves each processor passing its assigned Panel B portion of the matrix row with the largest pivot element to P_{pivot}. Processor P_{pivot} determines if pivoting is necessary and if it is, P_{pivot} sends the Panel B portion of the pivot row to the appropriate processor. Once Panel B is completely factored, the pivot information is communicated to all other processor columns to allow pivoting of the rest of the rows on both sides of Panel B.

The submatrices of the factored Panel B are distributed among the other processors to which a portion of Panel C or D is assigned . The row processors assigned a portion Panel C update Panel C by

$$C' = -B_{LI} * C \quad , \tag{20}$$

where B_{LI} is the lower unity triangular matrix of B_f. The updated Panel C is then communicate to all other processors to which a portion of Panel D is assigned. Panel D is then updated in parallel by

$$D' = D - B * C' \quad . \tag{21}$$

Each processor stores only its assigned A_{ij} submatrices, so that, the maximum storage required for a any single processor is $(N/P_{row}+n_b)$ x $(N/P_{col}+n_b)$. The mapping of the submatrices on to the processor grid remains constant throughout the computational process. The x and b vectors of the linear system of equations, Ax=b, are stored only on processors in the first processor column. In cases where $N > P_{row}$ x n_b, the b-vector blocks associated with a given processor row are packed into the top of the processor's b-vector before the forward/backsolve subroutine is called. After the forward/backsolve step is completed, the x-vector is collected from each processor in the first column row and unpacked before the x-vector is returned to the ODE solver.

Parallelization on Intel

The parallelization on the Intel is greatly simplified because LAPACK has parallel versions of DGETRF and DGETRS in the library known as ScaLAPACK[3]. These library routines, PDGETRF and PDGETRS, expect the matrix and the right-hand side to be distributed among the processors as described in the previous section.

Since the ODE solver only runs on a single node which is refer to as the primary node, the program that runs on that node is considerably different than the program on the remaining

nodes. But, during the linear algebra process, all nodes look the same. To achieve this, the interface routine YOWSER is modified to be the main program on all nodes except the primary node. On the primary node, YOWSER and its subroutines are modified to be part of the parallel code. Figure 5 shows the section of code which runs on the primary node and Figure 6 shows the program that runs on the non-primary nodes. Also shown in Figure 5 are the clock locations used to perform the timings.

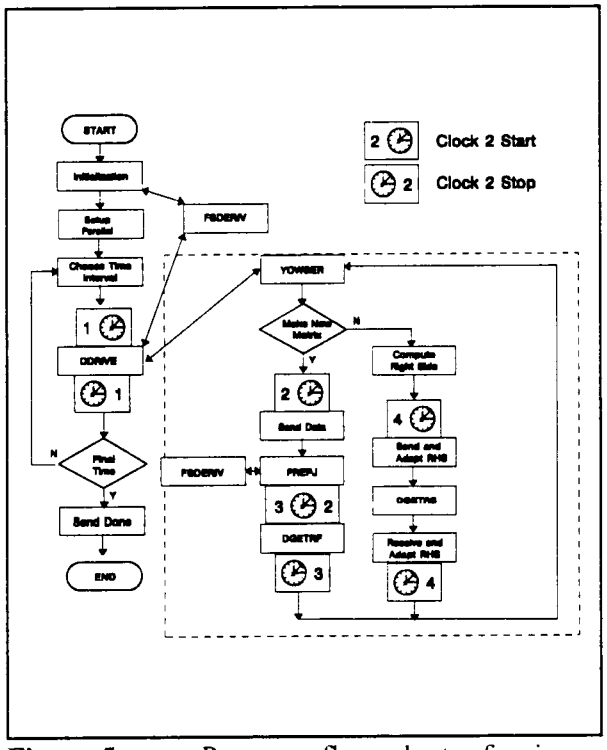

Figure 5 Program flow chart of primary node

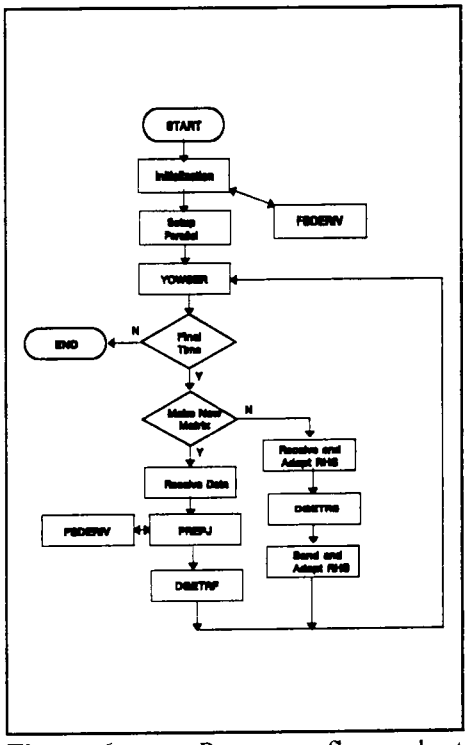

Figure 6 Program flow chart for auxiliary nodes

After modifying YOWSER to synchronize the nodes through communications, the only part of the sequential program that requires further modification is the PREPJ routine. Since PREPJ forms the matrix to be factored, the ideal approach is for each processor to construct and store only the portion of the global matrix mapped to that processor. However, this is not possible if the matrix is divided among a 2D grid of processors that contains more than one processor row because of the need to form an entire matrix column at once. The next best approach, given this constraint, is to have each processor in a processor row form the entire matrix column and then store only the portion assigned to that processor. This constraint

means the parallel performance of the PREPJ routine will be based only on the number of processor columns. The later approach is the method used in this research.

Parallelization on the KSR

The parallelization on the KSR is a greater challenge than that on the Intel. No parallel linear algebra routines are readily available for the KSR. Also, since the KSR is a virtual shared memory computer, it is not obvious whether converting a sequential linear algebra package or a distributed-memory parallel package offers the most potential for a successful implementation.

In considering how to implement parallelization on the KSR, the LAPACK library routines being used in the sequential version of the test problem and the parallel version of the Intel ScaLAPACK routines are examined. Both packages provide a blocked algorithm that uses the block saxpy operation shown in Equation 21 for updating of Panel D in Figure 4. In addition the parallel version provides a packing operation that reduces the total number of bytes being sent in a given communication. Also, the parallel version is designed use a distributed matrix where each processor is assigned a discrete portion of the matrix. If the sequential version is used and the matrix is addressed using global indices, multiple incidents may occur where two processors will probably share the same 128-byte subpage of memory. This sharing will probably cause multiple communications as first one processor then the other updates the same subpage of memory especially in the updating of Panel D. In addition, in the sequential version, a row pivot will require $2n_b$ times as many communications each of which is 128 bytes long as the parallel version which minimizes communication by providing packing operations. It also contains the algorithm for distributing the work among multiple processors. It is for these reasons the distributed-memory parallel version of ScaLAPACK is selected for use on the KSR.

In order to port the ScaLAPACK library to the KSR, some changes to the ScaLAPACK routines are needed. While variables that exist in common blocks on the Intel are shared only

on a single processor unless they are explicitly communicated to the other processors, on the KSR the default is any variable in a common block is considered to be common to all processors. Fortunately, there exists a means on the KSR to override the default so that common blocks may be shared by routines on a single processor but not among other processors. However, this requires minor modification the ScaLAPACK routines and its associated communication routine package, BLACS.

In the porting of the BLACS routine to the KSR a problem has been identified. The BLACS routines packs messages as much as possible to reduce communication time. The term packing means to transfer discrete sections of memory into one contiguous vector. In one of these packaging operations the BLACS routine packs a vector of double precision variables (8 bytes) followed by a single integer variable (4 bytes). Following the single integer variable a second vector of double precision variables and a single integer is added to the contiguous packing vector. A pointer is assigned to the starting location of the second vector so the first variable in the second vector can be used in a comparison operation. This approach is satisfactory on the Intel but generates an error on the KSR when the system attempts to access a double precision value that does not start on an 8-byte boundary. To correct this problem, a dummy integer value is added to the contiguous vector following the first integer value thereby forcing the second array to start on an 8-byte boundary.

On the Intel communication needed among processors is done by the BLACS routines calling the Intel primitives CSEND and CRECV. These primitive routines are part of the system library on the Intel but, of course, they do not exist on the KSR. In order to allow the BLACS routines to be used on the KSR, two routines are written on the KSR to emulate the Intel CSEND and CRECV.

The CSEND and CRECV routines are designed to ensure no unexpected communications occur. To do this, each processor is assigned a partition of global memory for store messages being communicated to another processor. For these partitions of global memory only the processor assigned to a given partition is allow to write into it, but all processors may read from it. When a processor is sending a message, the processor copies the message to be

communicated into a specified location of its assigned portion of global memory. The messages are padded to ensure all messages start and end on the 128-byte subpage boundary. A second partition of global memory is used to store message headers. For this partition, any processor can write to it but only the processor to which it has been assigned will read from it. This allows a processor waiting for data to enter a spin loop and continually access a partition of message headers without communication except when a message header has been updated by another processor. Once a message header is found with the requested message id number, a copy of the message in the first section of global memory is sent to the receiving processor and there copied into local memory. Using this design, the only need for processor locks is during the writing of the message header by the sending processor.

The CSEND and CRECV routines are designed specifically for this application and are not meant to be generic or global communication routines. Because of this, error checking that exists in normal routines has not been included. In addition, the method of locking processors to ensure that no two processors attempt to write to the same memory location simultaneously is not fully optimized. The routines are also designed to allow up to 64 messages to be pending, that is sent but not received, at one time. Algorithm 2 informally describes the CSEND routine and Algorithm 3 outlines the CRECV routine. The actual C language routines are found in Appendix C.

Testing has shown the CSEND and CRECV routines are about 50% slower than the tcgmsg communication package that has been designed for message passing on shared virtual memory multiprocessors. The results of a communication test program using CSEND and CRECV routines as used in the Method of Lines software are shown in Figure 7 as the Normal CSEND curve. Timings for a communication test program using tcgmsg are shown by the dotted line in Figure 7. Since the timing results for CSEND and CRECV are done using a special communications test program, the processor lock can be removed and the number of pending messages reduced from 64 to 4. By removing the processor lock and reducing the maximum number of pending messages to four, the CSEND and CRECV routines perform comparably with the tcgmsg package. This indicates that although these functions are not fully optimized

they do not significantly effect the overall performance of the Method of Lines solver on the KSR.

Algorithm 2 The CSEND routine

Purpose: A KSR emulation of the Intel version of a communication primitive to communicate between processors.

Given: The message identification number, msgid, the buffer containing the message, buff, the length of the message and the destination processor, dest.

Step 1: me = processor_id
Step 2: global_data(index) = local_data
(Note: Each processor owns its own segment of the global data area and it is the only processor that will ever write into that area.)
Step 3: index = index + message_length; if (index > global_data_area) index = 0;
Step 4: Lock processors.
Step 5: header(dest).msgid =msgid; header(dest).index = index; header(dest).len=len; header(dest).from=me
(Note: If destination < 0, the items in step 5 are sent to all processors except the processor sending to message.)
Step 6: Unlock processors.
Step 7: Return

Algorithm 3 The CRECV routine

Purpose: A KSR emulation of the Intel version of a communication primitive to communicate between processors.

Given: The message identification number, msgid, the buffer to receive the message, buff, the length of the message.

Step 1: counter=0; me=processor_id
Step 2: while (header(me).msgid ≠ msgid)
Step 3: counter=counter+1
Step 4: if(counter > maximum_count) print error message and return
end while
Step 5: local_data = global_data(header(me).index)
Step 6: header(me).msgid = 0 (Note: Setting msgid to 0 indicated message received)
Step 7: Return

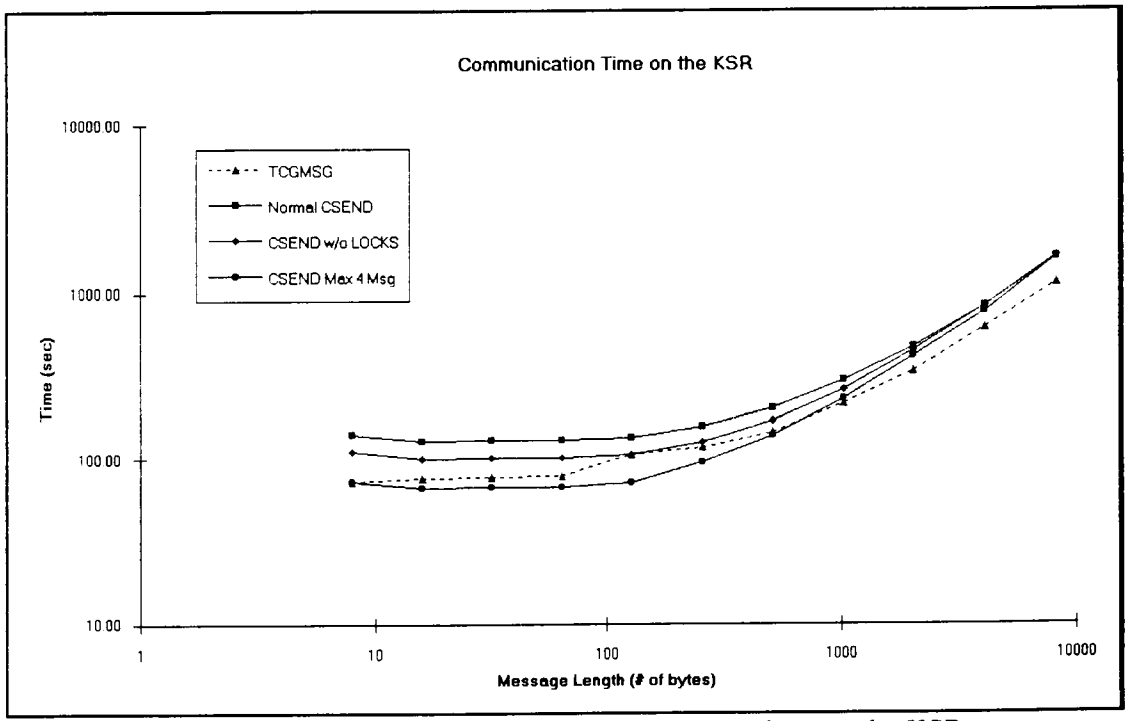

Figure 7 Communication time for CSEND/CRECV routines on the KSR

Chapter 4

Description of Processors Used

The Intel iPSC/860 Computer

The Intel iPSC/860 is a distributed memory hypercube that uses 128 - 40 MHz i860 processors. Each i860 has a 8 Kbytes of data cache and 8 Mbytes of memory. The processor has independent integer and floating point units. The floating point unit has an independent pipelined adder and multiplier with an advertised peak rate of 64 Mflops.

The program compilations are done with -04 optimization and run on the Nx3.3.2 operating systems[9]. Computation preform by the LAPACK and ScaLAPACK routines uses the assembly language version of the Basic Linear Algebra Subroutines (BLAS)[10] library.

The Kendall Square Research Computer

The Kendall Square Research multiprocessor used in this research is a virtual shared memory computer which consists of 64 - 20 MHz dual path processors. The processors, a KSR proprietary design, provide two floating point operations per clock cycle making the processors comparable to single path processors running a 40 MHz clock. Each processor has 256 Kbytes of data cache, 32 Mbytes of memory, and independent integer and floating point units. The floating point unit supports a pipelined adder and multiplier for an advertised peak rate of 40 Mflops per node.

Program compilations are done with -02 -r8 -i4 flags and run on a modified version of OSF/1. Parallelization within the program is achieved by using parallel regions and a constant teamid[11] to ensure the same processors are used throughout computations. Computation

preform by the LAPACK and ScaLAPACK routines uses the FORTRAN language version of the Basic Linear Algebra Subroutines (BLAS) library.

Table 1 provides a summary of parameters related to the Intel iPSC/860 and the KSR. Also included for reference are similar parameters for the Intel Delta and the Intel Paragon.

Table 1 System Parameters of Selected Multiprocessors[12]

Parameter	KSR	iPSC/860	Delta	Paragon
Clock rate (MHz)	20	40	40	50
Data cache (KB)	256	8	8	16
Memory size (MB/CPU)	32	8	16	16
Peak memory to CPU bandwidth (MB/s)	160	160	160	400
Peak proc. to proc. bandwidth (MB/s)	34	2.8	17	200
Remote mem latency (μs)	6.7[1]	150	150	25
Peak Mflops (64-bit)	40	64[2]	64[2]	75
Linpack Mflops (100x100)	15	6.5	6.5	?
Dhrystone mips (v1.0)	12.9	29.4	29.4	?
Max processors	1088	128	512	2048

[1] Latency time for hardware-controlled communications. Does not include time required for memory locking associated with message passing routines.

[2] Advertised peak rate. Observed peak rate is approximately 40 Mflops.

Chapter 5

Results of Parallelization

Measurement of Results

To evaluate the effectiveness of porting the implicit Method of Lines to a parallel platform several parameters are examined. These include total execution time to generate a solution, the amount of speed-up achieved, and the performance rate in terms of millions of floating point operations per second (Mflops). The execution time used is a global or wall clock time.

Speed-up can be measured in several ways. For this research speed-up, S_p, is defined by

$$S_p = \frac{T_s}{T_p} \quad , \tag{22}$$

where T_s is the time required for the sequential version to execute on a single processor and T_p is the time for the parallel version to execute on p processors. Ideal speed-up is achieved if $S_p = p$.

The performance rate is a measure of the floating point operations per second. In order to determine the performance rate the number of floating point operations must be known. For the LU factorization of a dense matrix of sufficient size, the performance rate is

$$Mflops = \frac{2n^3}{3T_{LU} * 10^6} \quad , \tag{23}$$

where n is the rank of the matrix being solved, and T_{LU} is the time to perform the LU factorization (See Clock 3 in Figure 5.) The performance rate shown in this research is the average rate of all LU factorization required to approximate the solution to the PDE problem. Because the determination of floating point operations on the formation of the Jacobian and the ODE solver is not readily available, the empirical results reflect only the performance rate of the LU factorization.

Selection of Optimal Blocksize

One factor that affect the performance of the linear algebra routines is the selection of blocksize, n_b, described early in the ScaLAPACK Algorithm. In order to ensure optimal performance of the Method of Lines, several different blocksizes are tried and these results are presented in Figure 8. This figure shows that the selection of blocksize for the test problem does not have a major effect on the total solution time. However, blocksize does effect the amount of work space that must be allocated for ScaLAPACK communications. Therefore, to simplify the analysis of the results and minimize work space requirements, a blocksize $n_b = 5$ is used for the remaining computations on the Intel.

Similar results are achieved on the KSR except in a range of 8-20 instead of the 3-12 seen on the Intel. Referring to Table 1, this is due to the larger cache size on the KSR. On the KSR, computations are done using a blocksize $n_b = 12$ because that is the optimal value for most of the problem sizes used in this research.

Presentation of Empirical Results

Figure 9 shows the time required to execute the test problem on the Intel for various numbers of processors and problem sizes. This figure shows a significant reduction in the total execution time as the number of processors increases for all sufficiently large problem sizes. For example, in the case of a problem size of 900 equations, the execution time is reduced from 750 seconds using one processor to 127 seconds using 32 processors and to less than 100

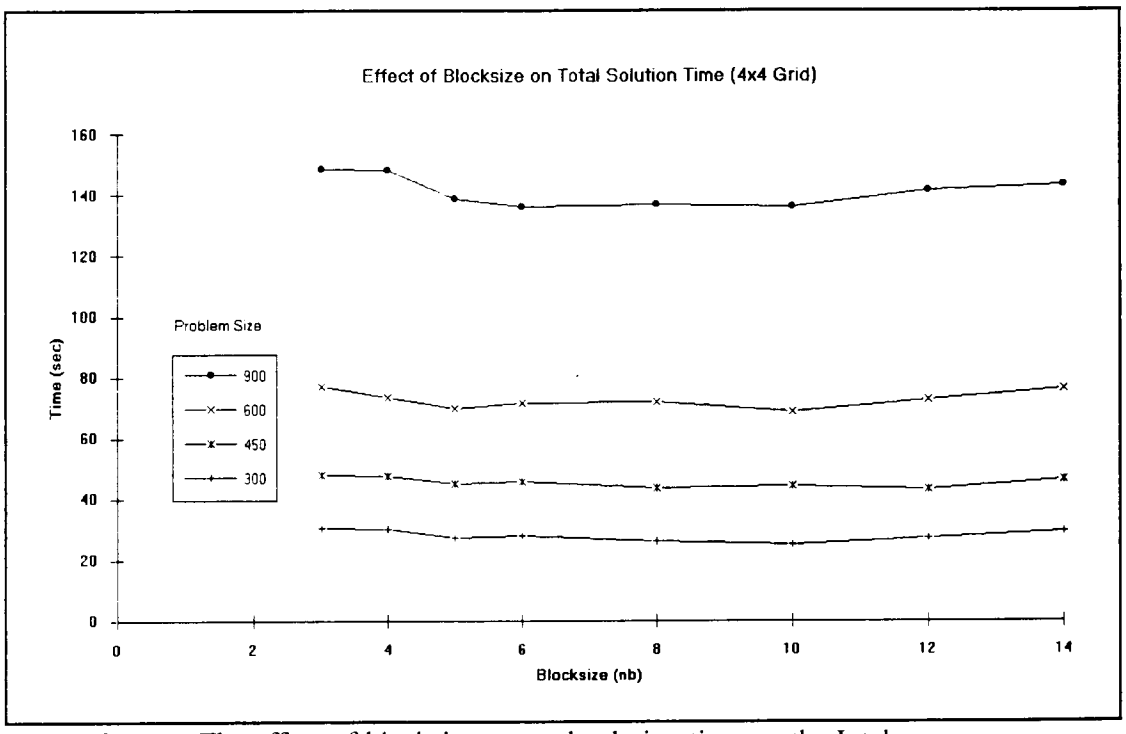

Figure 8 The effect of blocksize on total solution time on the Intel

seconds using 64 processors. Of particular interest is the change from the 2x4 to the 4x4 processor grid, which is discussed later.

Figure 10 shows the execution time for the KSR. Figure 10 is similar to Figure 9 for the Intel except the maximum number of processors in Figure 10 is 32 processors and Figure 9 has 64 processors. On the KSR the total execution time for a problem size of 900 equations is reduced from 650 seconds for one processor to 107 seconds for 32 processors. Note the minimal change in execution time for processor grid sizes larger than a 2x4 grid. On a single KSR processor, the execution time for problem sizes of 1200 and 1500 equations is not shown. This is because the ordinate of Figures 9,10 have the same scale. The execution time for a problem size of 1200 equations is 1608 seconds and for 1500 equations is 3111 seconds.

One advantage of using parallel architectures is the ability to distribute data over several processors, thereby being able to operate on larger problem sizes. This advantage is demonstrated in Figure 9. On a single Intel processor, the maximum size problem is 900 equations while a problem size of 1800 equations is accommodated using 32 processors and

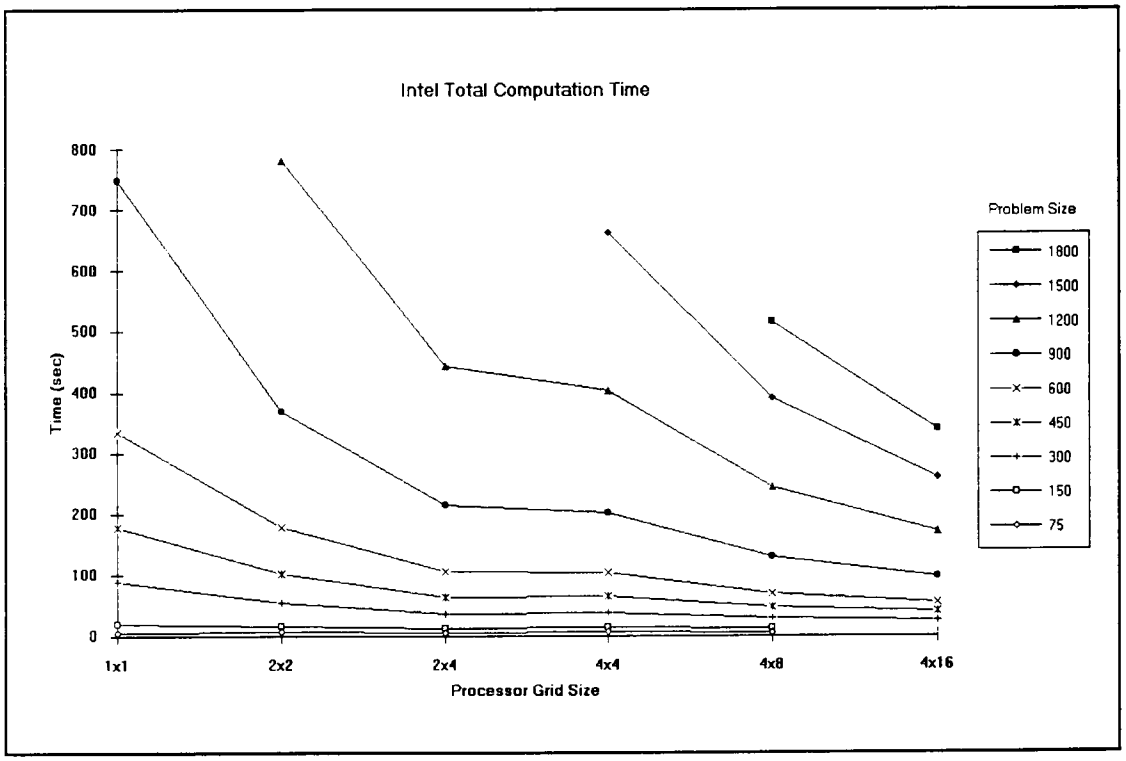

Figure 9 Total execution time on the Intel

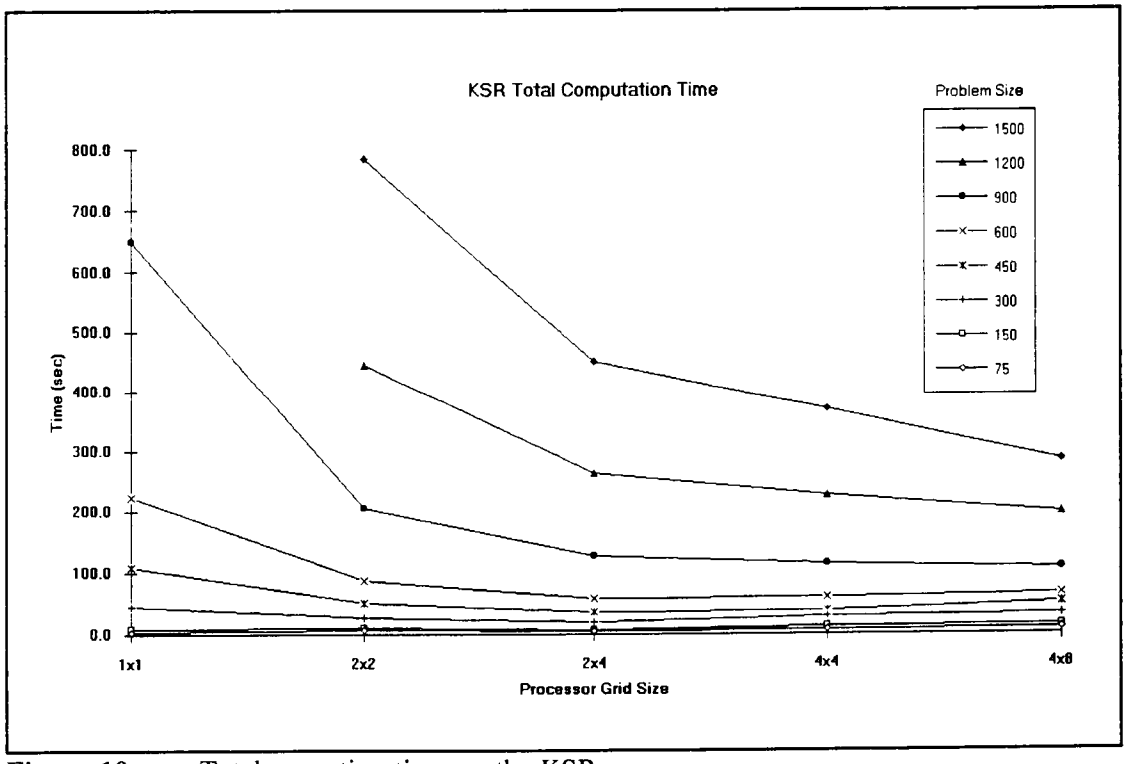

Figure 10 Total execution time on the KSR

a problem of 2400 equations is accommodated using 64 processors.

Another measure of effectiveness is the total amount of speed-up that is achieved. Figure 11 displays speed-up on the Intel and Figure 12 displays speed-up on the KSR. Again using the example of 900 equations, the Intel using a 4x16 grid achieves a speed-up $S_p = 7.8$ and the KSR achieves a speed-up $S_p = 7.1$ on a 2x16 grid. The 2x16 grid is used on the KSR instead of a 4x16 grid due to hardware problems that prevented the use of more than 32 processors

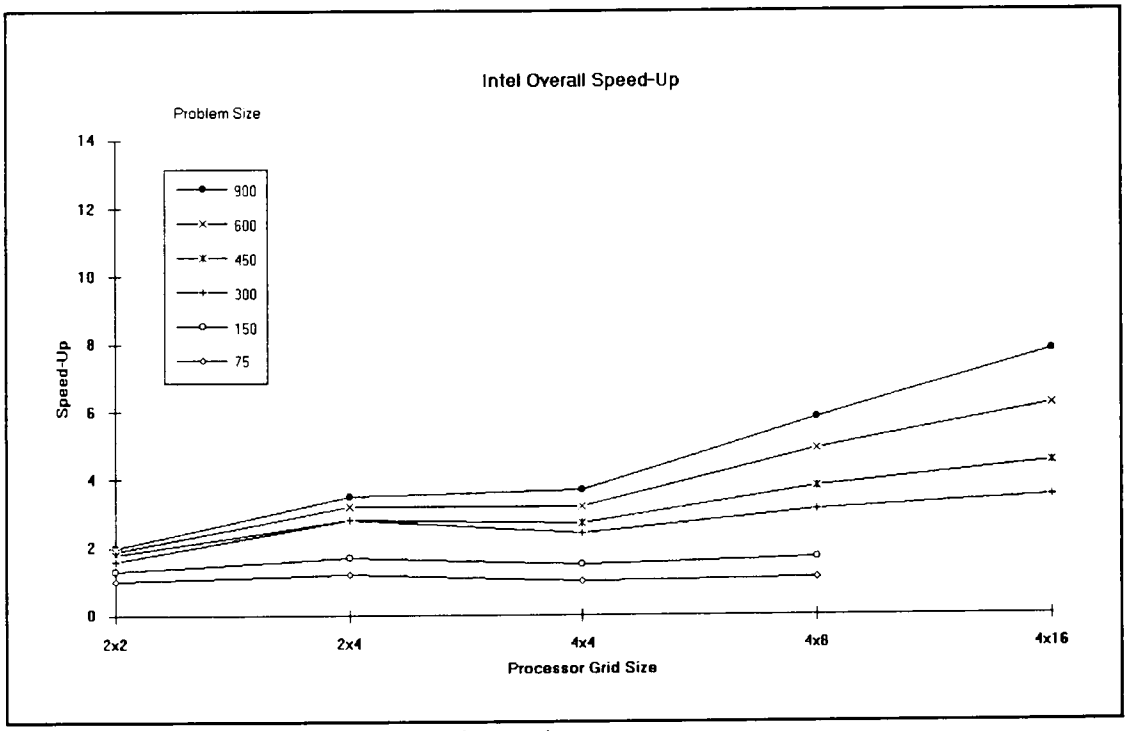

Figure 11 Overall speed-up on the Intel

during the course of this study. Note the abscissa in Figures 9-12 is equally spaced on grid size and a doubling of the number of processors occurs for each step except for the last step of Figure 12. This effectively creates a log scale on the abscissa, a point that needs to be considered in the analysis of the data presented in this charts.

Figure 13 shows performance in Mflops for the LU factorization portion of the test problem on the Intel. The dotted line in Figure 13 is the performance rate obtained on the Intel running

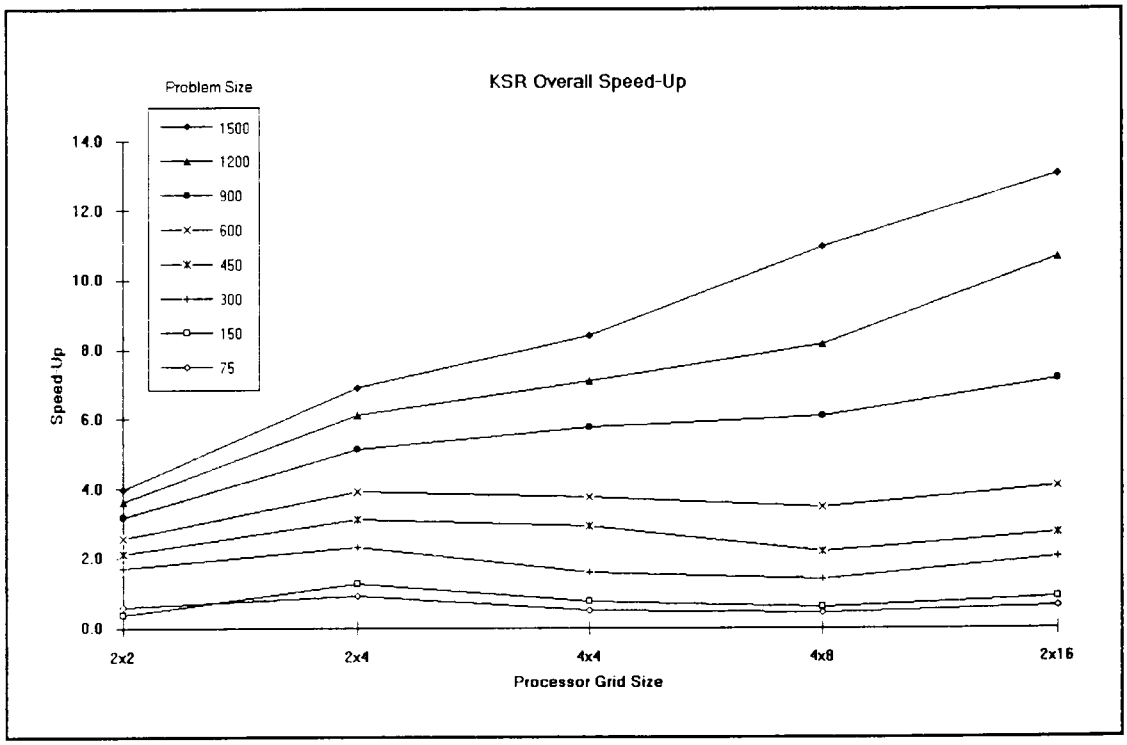

Figure 12 Overall speed-up rate on the KSR

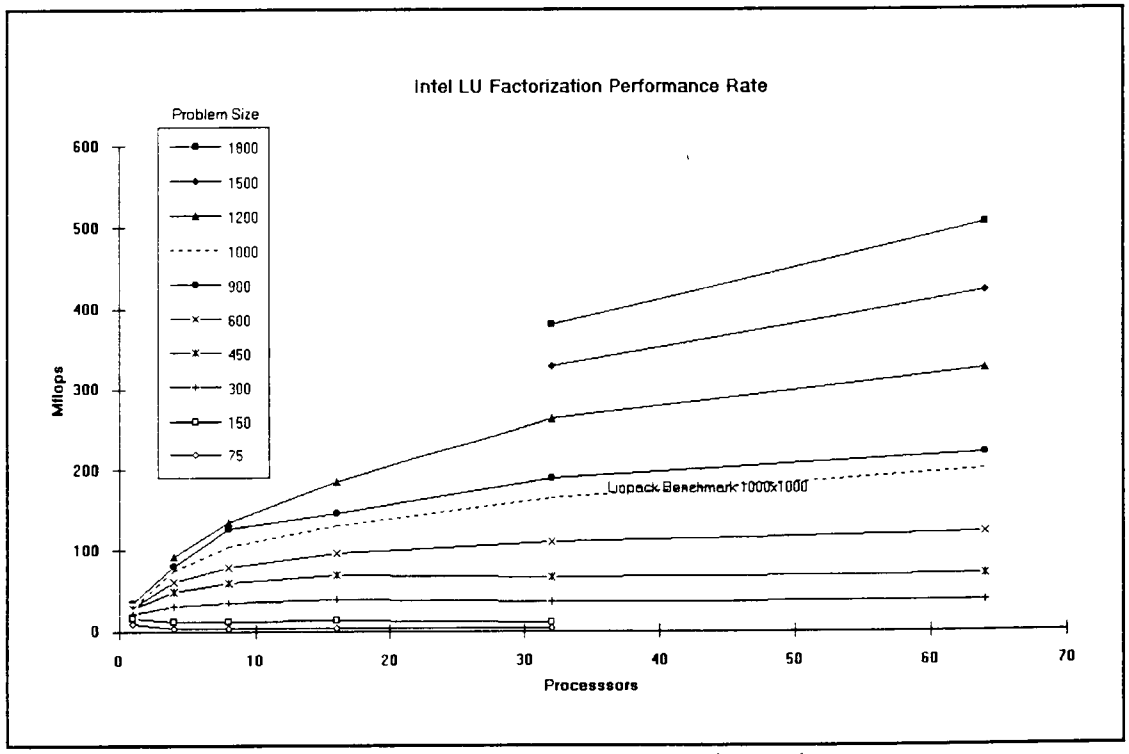

Figure 13 Performance rate for LU factorization on the Intel

a LINPACK benchmark program[13]. In addition to the data shown in Figure 13, the Intel achieved a performance rate 670 Mflops on a problem size of 2400 equations using 64 processors.

Figure 14 is a chart similar to Figure 13 showing the performance rate for the KSR. The dotted line again shows the performance rate reported by Kendall Square Research Corporation for the LINPACK benchmark[13]. For comparison, the KSR achieved a performance rate of 229 Mflops using 32 processors on a problem size of 1500 equations and the Intel achieved 329 Mflops for the same number of processors and equations. Qualitatively, Figures 13,14 are similar and display the same asymptotic characteristics as the number of processors is increased. Note the scale for both the abscissa and ordinate are the same in both figures to highlight this feature.

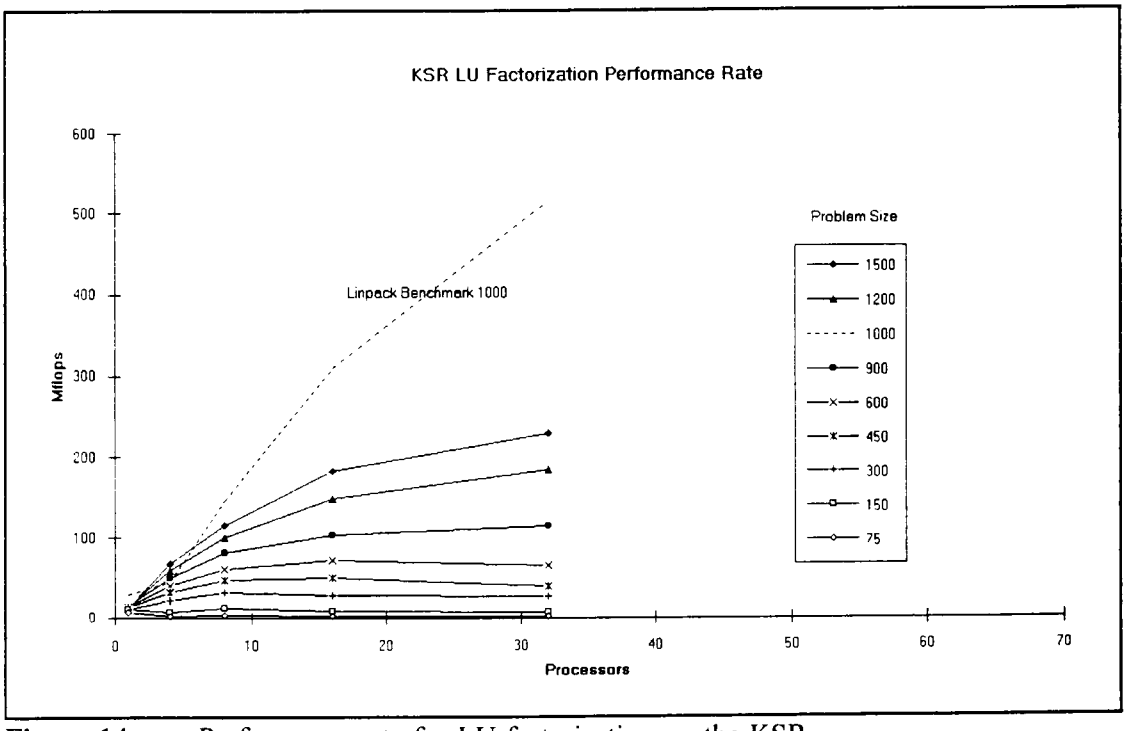

Figure 14 Performance rate for LU factorization on the KSR

Analysis of Results

The first objective of this thesis is that a strategy can be devised that will efficiently parallelize the implicit Method of Lines without the need to do an extensive rewrite of the applications program. The second objective is to demonstrate that the same strategy can be used on both a virtual shared-memory and a distributed-memory processor platform. The third objective is to show speed-up obtained by this strategy will not be severely restricted by leaving a major portion of the code sequential. The fourth objective is to demonstrate that the performance rate observed in the linear algebra routines should be similar to that observed in benchmark evaluations of the linear algebra package.

To determine if the objectives of this research has been achieved, a closer examination of the empirical results is needed. For both the Intel and the KSR the total execution time required to solve a problem of given size is decreased and a measurable amount of speed-up is obtained. Both show a speed-up rate $S_p \approx 6$ for 900 equations using 32 processors on a 4x8 grid. However, the path from a single processor to that point varies for the two platforms. In order to understand the difference between the two paths, a closer look at the computational process is required.

By examining the relative execution time for the principal computational components of the sequential version of the Method of Lines on both platforms a major difference is seen. Figure 3 shows the relative time required to perform those computational components on the KSR and Figure 15 shows same for the Intel. As can be seen from Figures 3,15, the formation of the Jacobian matrix requires a considerably larger fraction of the execution time on the Intel than on the KSR. The only difference in the software between the two programs is the BLAS library used in the linear algebra computations which does not include the Jacobian formation. The Intel uses an assembly language version of the BLAS verses the FORTRAN version used on the KSR. To evaluate how the different versions of the BLAS affect computational time, the sequential Intel program is re-compiled using the FORTRAN version and a comparison of the time required to perform the LU factorization is made. The results are illustrated in Figure 16. As indicated by Figure 16, the assembly language version of the BLAS does account for

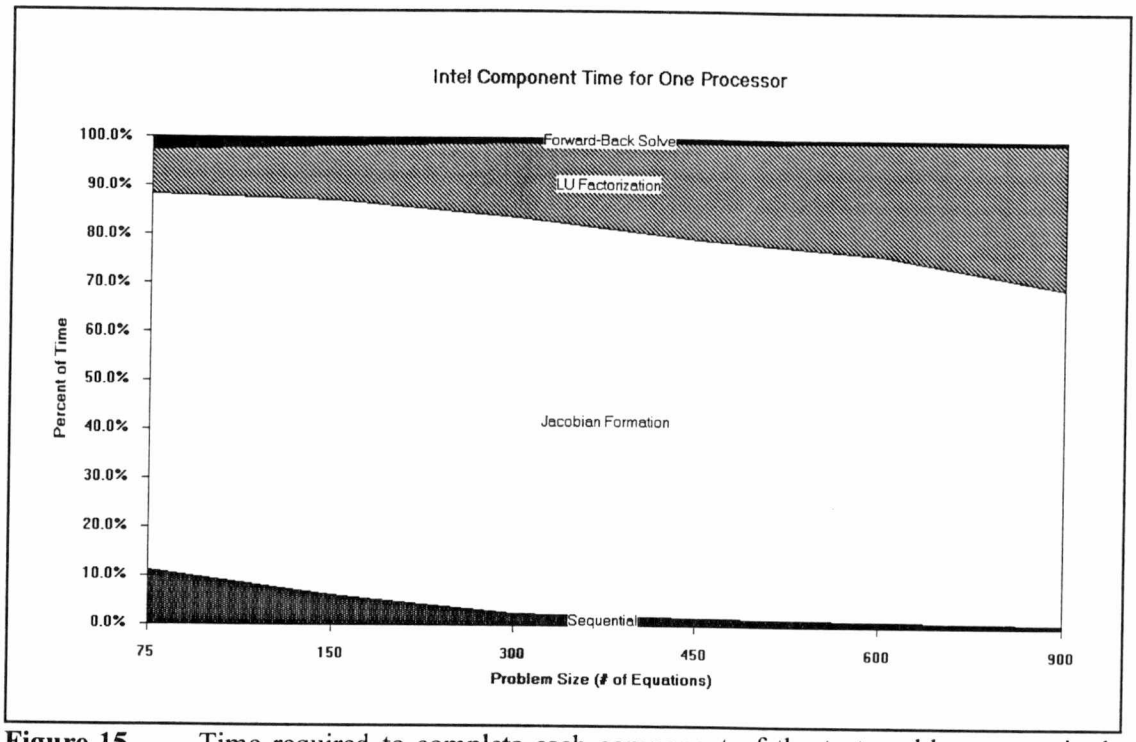

Figure 15 Time required to complete each component of the test problem on a single processor of the Intel

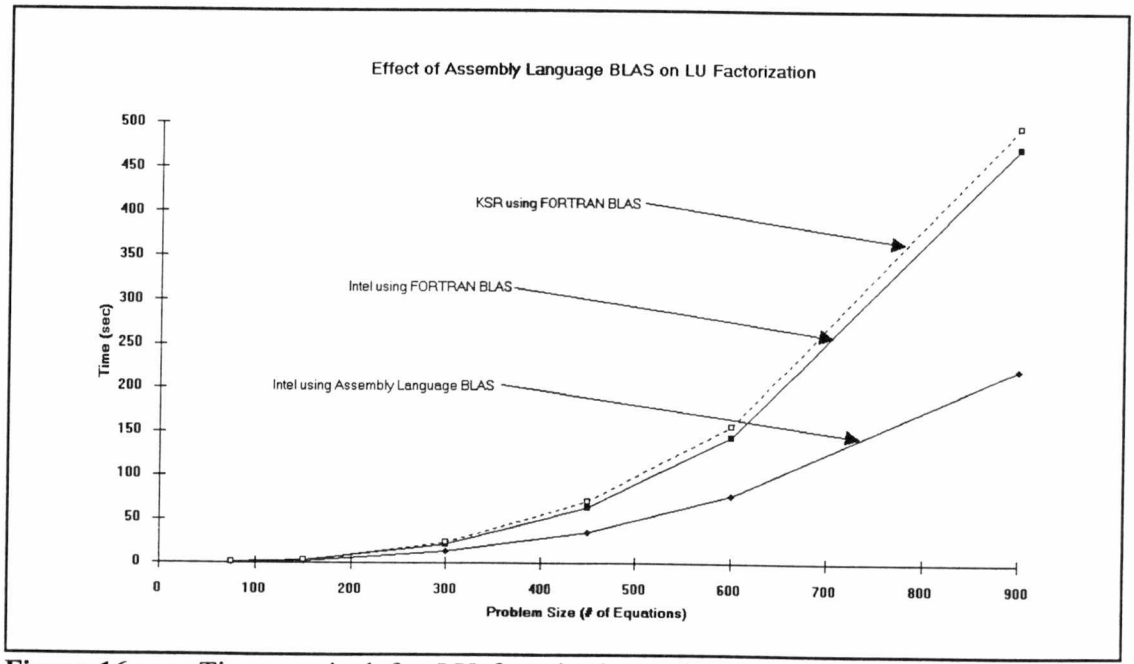

Figure 16 Time required for LU factorization using Assembly Language BLAS and FORTRAN BLAS on a single node of the KSR and Intel processors

a significant reduction in the execution time for the LU factorization. However, the use of the different version of the BLAS does not account for the entire difference between Figures 3 and 15. Using FORTRAN BLAS on the Intel approximately doubles the computational time for the LU factorization, but this would mean that the Jacobian formation will still require 50% of the total time for 900 equations compared to 20% for the KSR. Referring to the FORTRAN BLAS curves in Figure 16, the arithmetic operations of the KSR and the Intel are virtually identical. Referring to Table 1, the most probable reason that can account for the difference in relative time is the cache size for the KSR is considerably larger than that for the Intel. This larger cache allows the routines involved in the Jacobian calculation and the associated data to reside in faster cache memory, thereby producing a more efficient Jacobian formation that on the Intel. Except for those shown in Figure 16, all calculations on the Intel are done using the assembly language BLAS.

When the program is parallelized, the differences in the relative time to compute the Jacobian matrix and to perform the LU factorization between the Intel and KSR continue to be a factor in the overall performance characteristics. Figures 17,18 show relative execution time for the principal computational components on the Intel and the KSR using 16 processors in a 4x4 grid. The dominance of the Jacobian formation on the Intel versus the dominance of the LU factorization of the KSR explains the difference between the execution time reduction curves displayed by Figures 9,10. On the Intel most of the reduction in computational time comes from the formation the Jacobian matrix. To illustrate this, suppose only the Jacobian formation is parallelized. From Figure 19 one can conclude the Jacobian formation scales directly with the number of column processors and is independent of the number of row processors. As the number of processors is increased from 1 to 4 processors arranged in a 2x2 grid, the execution time will be reduced by 2, the number of column processors. As the number of processors is increased to a 2x4 grid the time will again be halved. However, when the grid is increase to 16 processors in a 4x4 grid the execution time will remain constant as there is no increase in the number of column processors. By examining the curve for a problem size of 900 equations in Figure 9, which displays total execution time, it is seen that the behavior described above very close matches what happens on the Intel up to the use of 16 processors in a 4x4 processor grid. Figure 17 shows that for 900 equations on a 4x4 grid the Jacobian

formation is 50% of the total time. So, doubling the number of column processors from a 4x4 processor grid to a 4x8 grid, we conclude the execution time will be reduced by 25% instead of 50%. Referring to the curve in Figure 9 note that again this reflects the reduction in execution time for the Intel. Therefore virtually all the reduction in execution time is accounted for in the Jacobian formation.

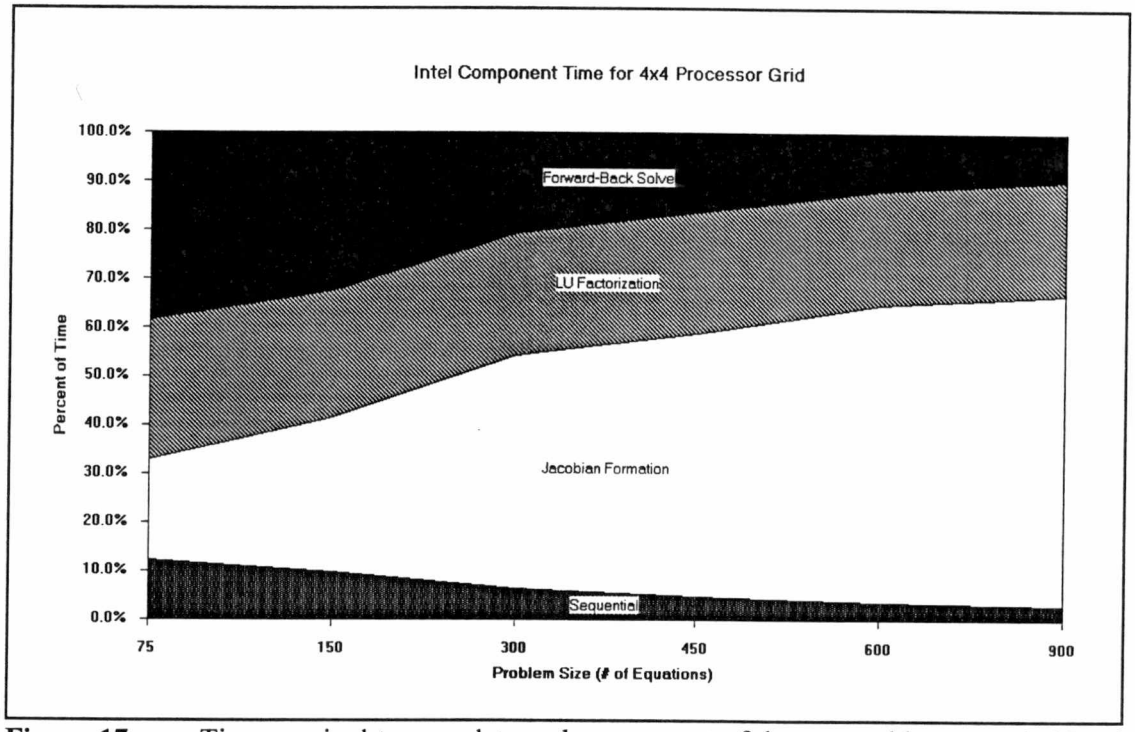

Figure 17 Time required to complete each component of the test problem on a 4x4 Intel grid

For sufficiently large problems, the dominant computational component is the LU factorization on the KSR. On the KSR, like on the Intel, the Jacobian formation is dependant on the number of column processors. However, as displayed in Figures 3,18, the Jacobian formation is only 20% of the total execution time, so increasing the number of column processors will have only a minimal effect on total execution time. Therefore, the amount of reduction in execution time show in Figure 10 can be attributed primarily to reduction in execution time for the linear algebra routines. On the KSR most of the reduction in execution time occurs as the processor grid is increased from 1 processor to 16 processors in a 4x4 grid. Also note the amount of reduction in execution time decreases with each increment step in the processor

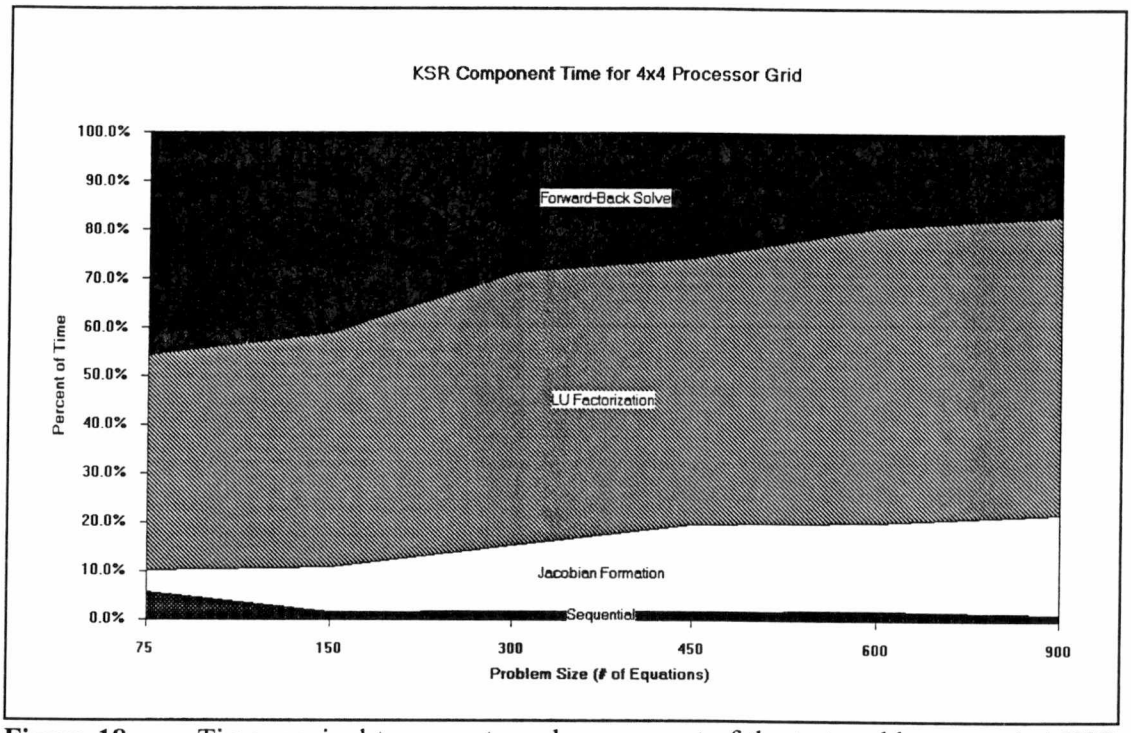

Figure 18 Time required to compete each component of the test problem on a 4x4 KSR grid

grid. This is especially noted remembering the time required for a single process to execute a test problem with a size of 1500 equations is 3100 seconds.

Comparing Figures 11,12, speed-up on the KSR is greater than that observed on the Intel for a relatively small number of processors. For example, Figures 11,12 show the speed-up for 900 equations on a 2x2 KSR grid is $S_p = 3.2$ versus a value $S_p = 2.0$ for the Intel. On the KSR the speed-up $S_p = 3.2$ reflects the speed-up obtained primarily due to the LU factorization as the computational work is distributed among the processors. Continuing to look at the 900 equations curve in Figure 12, increasing the grid size beyond a 2x2 grid have a rapidly decreasing effect on speed-up due, first, to the increased communications cost that offsets the gains made by the distributed calculations and, second, to the reduced potential for speed-up from the Jacobian formation. As opposed to the KSR, the Intel is dominated by the Jacobian formation. This means the speed-up potential on the Intel is limited by the number of column processors for the size problems examined in this study.

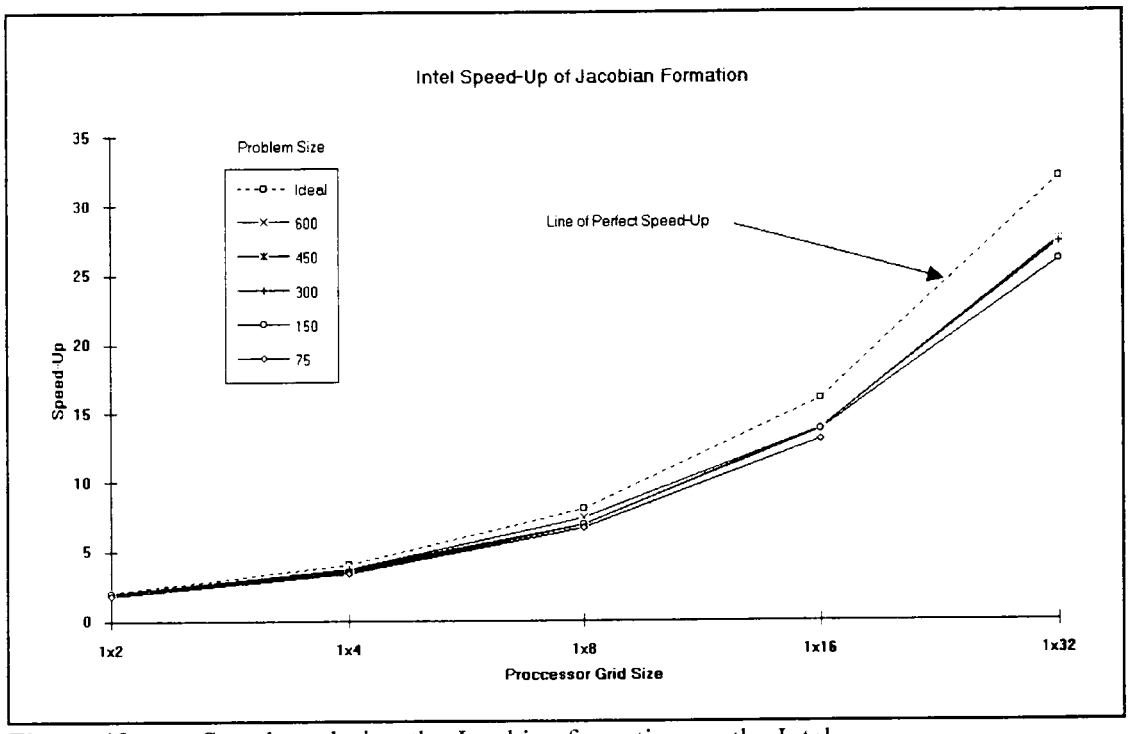

Figure 19 Speed-up during the Jacobian formation on the Intel

The time for the forward/backsolve process shown in Figures 17,18 appears to be disproportional to the time for the LU factorization, even considering that this process is communication intense. This is because, by design of the ODE solver, a new matrix is not constructed and factored for every step in the Newton-Raphson problem. For test problem sizes ranging from 75 to 1800 equations, the matrix formation and factorization, represented by Steps 3 and 4 of Algorithm 1, occur between 14 and 17 times and the forward/backsolve, Step 5 of Algorithm 1, occurs between 65 and 95 times. Both the KSR and Intel use the same number of iterations for a given size problem as the number of iterations is controlled by the ODE solver.

An examination of the performance rate for the LU factorization on the Intel, Figure 13, shows a very close match between the LINPACK benchmark program and the performance seen on the test problem. This indicates the strategy for parallelization has not had a negative impact on the performance rates. However, as shown in Figure 14 the same is not true for the performance rate on the KSR. The reason for this is because the routines used to perform the LINPACK benchmark were designed specifically to make optimal use of the KSR architecture

for the LINPACK benchmark and the routines are not available for use by the general public at this time. However, the performance curves on the KSR are similar to those on the Intel in form and display the same asymptotic behavior. The difference in magnitude shown in Figures 13,14 between the platforms can be attributed to the effect of the assembly language BLAS on the Intel which is shown in Figure 16. This indicates the parallelization strategy on the KSR does not adversely affect the linear algebra routines. It also indicates the CSEND and CRECV routines used in conjunction with the ScaLAPACK libraries provides a performance level comparable to the Intel.

Another method to examine the effectiveness of the parallelization strategy is to examine which improvements are possible if the effort is made to convert all the code to parallel form. By examining the time spent to perform the sequential part of the test problem in Figure 20, it is seen that for sufficiently large problem sizes less than five percent of the time is spent in the sequential part of the test problem. This means that even if the sequential code is converted to run in parallel, the effect on the total time to compute a solution will decrease only slightly. However, some improvement is still possible. The speed of the Jacobian formation is dependant on the number of column processors. If the FSDERIV routine in Figures 5,6 is rewritten to run in parallel, some improvement in speed-up is possible.

This research presented in this thesis has shown a strategy for porting an implicit Method of Lines numerical method to parallel computer architecture can be implemented without the need to rewrite the ODE solver and that this strategy can achieve equivalent performance either a distributed-memory or a virtual shared-memory platform. This ability to port an application without doing a major rewrite of existing code reduces the time required for the porting and improves the reliability of porting to parallel environments. The speed-up rate obtainable in the implicit Method of Lines application is not dependant on the sequential code, but on the effectiveness of the parallel formation of the Jacobian and the linear algebra libraries. The performance rate for the LU decomposition routines on the Intel indicates the implicit Method of Lines application does achieve benchmark results for the Intel linear algebra routines.

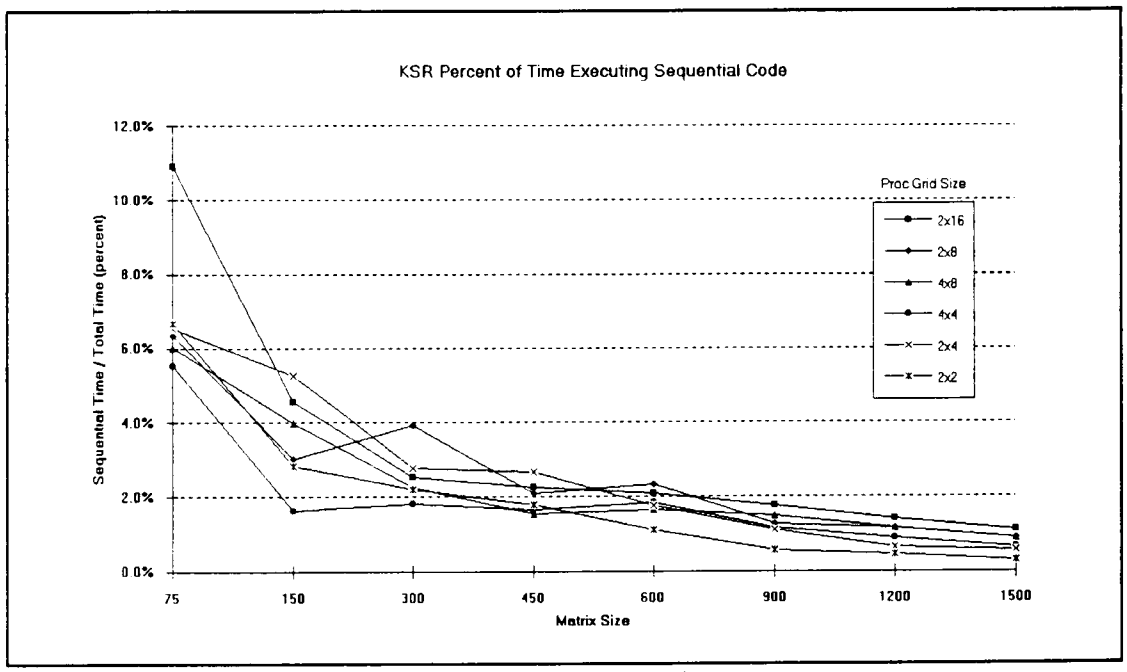

Figure 20 Percent of time executing sequential code

Chapter 6

Summary and Conclusions

This thesis has presented a strategy for porting an application, the implicit Method of Lines, from a sequential computer platform to a parallel computer platform. In porting the application four objectives have been identified. Each of those objectives are achieved during the course of this research.

First, using the devised strategy, the implementation of the implicit Method of Lines provides a reasonable speed-up on parallel computer architecture without the need to rewrite the ODE solver. This ability reduces the time required and improves the reliability of porting the Method of Lines numerical method to parallel environments.

Second, the maximum problem size capable of being solved is increased due to the distributed data storage. This is shown by the Intel having a maximum problem size of 900 equations if the test problem is executed on a single node but using 64 processors, the maximum problem size is increased to 2400 equations .

Third, the speed-up for the application is not dependant on the sequential code, but on the rate that can be achieved by the parallel routines. Even for large number of processors, the amount of total time spend in the sequential code is less than 5% of the total executing time.

Fourth, on the Intel the performance rate or the speed of calculations for the LU decomposition routines compare favorably with results from the LINPACK benchmark. This indicates the strategy used achieved a level of performance from the linear algebra routines consistent with the routines capabilities. On the KSR the performance rate is significantly less that test results for the LINPACK benchmark as reported by Kendall Square Research, but does compare favorably with the results on the Intel. Difference between the performance rate on the Intel

and the performance rate on the KSR can be attributed to the use of the FORTRAN version of the BLAS routines.

In addition to meeting the objectives of this thesis, two addition points of interest are achieved. The first is the porting of parallel libraries, ScaLAPACK and BLACS, from a distributed-memory platform to a virtual shared-memory platform. The porting operation requires only minimal changes to parallel library code and once ported to the virtual shared-memory platform the libraries perform at a comparable rate to the original distributed-memory version.

The second point of interest is the overall performance rate for the Method of Lines application is nearly the same on both platforms. Differences in the performance can be attributed to the use of assembly language BLAS on the Intel and the larger cache size available on the KSR.

References

References

1. D. K. Kahaner, C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, Englewood Cliffs, New Jersery, 1989

2. A.C. Hindmarsh, "Toward a Systemized Collection of ODE Solvers," Proceedings, Vol 1, 10th IMACS Congress on System Simulation and Scientific Computation, Montreal, Canada, August 1982, pp. 427-432

3. J. Choi, J.Dongarra, R. Pozo, D. Walker, ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers, Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation, McLean, Virginia, October 1992, IEEE Press, pp120-127.

4. J. J. Dongarra, R. A. van de Geijn, and R. C. Whaley. *A User's Guide to the BLACS*, LAPACK Working Note 57, Technical Report CS-93-187, University of Tennessee, 1993.

5. S. Thompson and P. G. Tuttle, Benchmark Fluid Flow Problems for Continuous Simulation Languages, *Comp. & Math. with Appls. 12A*, 345-351 (1986)

6. E. Ng, S. Thompson and P. G. Tuttle, "Experiments with Method of Lines Solvers on a Shared Memory Parallel Computer," *Advances in Computer Methods for Partial Differential Equations VII, Proceedings of the Seventh IMACS International Symposium on Computer Methods for Partial Differential Equations,* Bethlehem, Pennsylvania, 1987, pp. 161-166.

7. C.W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall, Inc, Englewood, N.J. 1971, pp. 105-107

8. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, D. Sorensen, *LAPACK Users' Guide*, SIAM Publications, Philadelphia, PA, 1992.

9. Intel, *iPSC Programmer's Reference Manual*, Intel 311708-2, Portland, Oregon, 1989.

10. J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Math. Soft.*, 16, 1:1-17, March 1990, Argonne National Laboratory, ANL-MCS-P88-1, August 1988.

11. KSR Parallel Programming, Kendall Square Research Corporation, Walthan, NJ, August 1991

12. W. F. Lawkins and J. S. Payne, *An Implementation of a Parallel MOL Solver on the Intel Gamma Parallel Computer*, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1992.

13. T. H. Dunigan, *Multi-Ring Performance of the Kendall Square Multiprocessor*, Technical Report ORNL/TM-12331, Oak Ridge National Laboratory, Oak Ridge, Tennessee, September 1993

Appendices

Appendix A

Definition of Acronyms

Intel	Intel iPSC/860 Computer System
KSR	Kendall Square Research Computer System
MOL	Method of Lines
PDE	Partial Differential Equation
ODE	Ordinary Differential Equation
BDF	Backward Differential Formulation
BLACS	Basic Linear Algebra Communication Subprograms
BLAS	Basic Linear Algebra Subprograms

Appendix B

Definition of Terms

ρ	Density (kg/m^3).
G	Flow Rate (kg/m^2s).
T	Temperature (o C).
K	Frictional pressure drop coefficient = 10.0.
g_a	Gravitational acceleration = 9.80665 (m/s^2).
θ	90°.
Φ	Heat flux = 1.1e+05 (w/m^2).
P_H	Heated perimeter = 7.97318e+02 (m).
A_f	Flow area = 3.82760 (m^2).
L	Distance evaluated = 1.0 (m).
a	Sound speed (m/s).
C_p	Constant pressure specific heat (kJ/kg-K).
κ	Isothermal compressibility (1/MPa).
β	Coefficient of volume expansion (1/K).
$\bar{T}$	Absolute temperature = T + 273.15 (°K).

Appendix C

Source Code for CSEND and CRECV Functions

```
/*************************************************************************
* File:  RX.C
* Written:      JSPayne 06/10/93
*
* Purpose:      To provide function to emulate the Intel iPSC/860 hypercube on the Kendall
*                       Square Research Computer.
*************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#include <errno.h>

/*************************************************************************
* Limit static and extern variables to a single processor
*************************************************************************/
#define static static __private
#define extern extern __private

/*************************************************************************
* Set the maximum number of pending messages and processors
*************************************************************************/
#define MAXMSG 64
#define MAX_PROC 64

/*************************************************************************
```

```
* Define the storage structure for message headers
*****************************************************************************/
struct messages {
  int msgid[MAXMSG];
  int len[MAXMSG];
  int index[MAXMSG];
  int from[MAXMSG];
} procid[MAX_PROC];

/*****************************************************************************
* Define global storage area for messages
*****************************************************************************/
#define MAXBUFF 512000
char messbuff[MAX_PROC][MAXBUFF];

/*****************************************************************************
* Define locking variable
*****************************************************************************/
pthread_mutex_t m_lock1;

/*****************************************************************************
* Function: crecv
*
* Purpose:      Copies messages from global data area to local data area as requested by local
*               processor.  Forces local processor is wait until requested message is
*                received.
*
* Input:        msgid - mesaage id of request message
*               a - pointer to local data area
*               len - length of message to be received
*
```

```
* Returns:      None
*****************************************************************************/

void crecv (int msgid, void *a, long len)
{
        int i, j, me, first, index, msg = 0, stat = 0;

/*****************************************************************************
*       Determine processor id
*****************************************************************************/
        me = ipr_mid_();

/*****************************************************************************
*       Loop until a message header is found with request id
*****************************************************************************/
        while (procid[me].msgid[msg] != msgid) msg = ++msg % MAXMSG;

/*****************************************************************************
*       Copy message from global memory area into local memory
*****************************************************************************/
        index = procid[me].index[msg];
        memcpy (a, messbuff[procid[me].from[msg]]+index, len);

/*****************************************************************************
*       Set message header id to 0 to indicate message has been received
*****************************************************************************/
        procid[me].msgid[msg] = 0;

}               /* End crecv */

/*****************************************************************************
```

```
* Function: csend
*
* Purpose:      Copies messages from local memory to global memory and set message
*               header of destination processor to show a message is pending
*
* Input:        msgid - message id of message to be sent
*               a - pointer to local data area
*               len - length of message to be sent
*               dest - processor id of processor to receive message
*                       Note: if dest < 0 send message to all processor except self
*               mypid - not used, provided for compatiblilty purpose
*
* Returns:      None
*******************************************************************************/

void csend (int msgid, void *a, long len, int dest, int mypid)
{
        int i, first, msg, me, index, llen = len, stat, nodes;
        static int lstmsg = 0, nxtmsg = 0;

/*******************************************************************************
*       Determine processor id
*******************************************************************************/
        me = ipr_mid_();

/*******************************************************************************
*       Copy local memory to global memory on 128-byte boundary
*******************************************************************************/
        if (nxtmsg+len > MAXBUFF) nxtmsg = 0;
        index = nxtmsg;
        memcpy (messbuff[me]+nxtmsg, a, len);
```

```
        nxtmsg += len + 128 - len % 128;

/*****************************************************************************
*       Lock process so only one processor will write to dest header at a time
*****************************************************************************/
        stat = pthread_mutex_lock(&m_lock1);

/*****************************************************************************
* Determine if message is going to everyone or just one processor
*****************************************************************************/
        if (dest < 0) {

/*****************************************************************************
*               For every processor in allocated team
*****************************************************************************/
                for (dest = 0; dest < ipr_tsize_(); ++dest) {
                        if (dest != me) {

/*****************************************************************************
*                               Search message headers of dest processor for open header
*****************************************************************************/
                                msg = (lstmsg + 1) % MAXMSG;
                                while (1 == 1) {
                                        if ( procid[dest].msgid[msg] == 0) break;
                                        msg = ++msg % MAXMSG;
                                }

/*****************************************************************************
*                               Copy message info to dest message header
*****************************************************************************/
                                lstmsg = msg;
```

```
                        procid[dest].from[msg] = me;
                        procid[dest].len[msg] = len;
                        procid[dest].index[msg] = index;
                        procid[dest].msgid[msg] = msgid;
                }               /* end if */
        }               /* end for */

/*************************************************************************
*       Else message just for processor dest
*************************************************************************/
        } else {

/*************************************************************************
*               Search message headers of destination processor for open header
*************************************************************************/
                msg = (lstmsg + 1) % MAXMSG;
                while (1 == 1) {
                        if ( procid[dest].msgid[msg] == 0) break;
                        msg = ++msg % MAXMSG;
                }

/*************************************************************************
*               Copy message info to dest message header
*************************************************************************/
                lstmsg = msg;
                procid[dest].from[msg] = me;
                procid[dest].len[msg] = len;
                procid[dest].index[msg] = index;
                procid[dest].msgid[msg] = msgid;
        }               /* endelse */
```

```
/*****************************************************************************
*       Unlock process to allow all processor to run in parallel
*****************************************************************************/
        stat = pthread_mutex_unlock(&m_lock1);

}               /* end csend */
```

VITA

James Stephen Payne was born in Robinson, Illinois on October 18, 1950. He attended elementary school at South Terrace Elementary School in Wadesville, Indiana and graduated from North Posey High School in 1968. Before entering the Air Force in 1970 he attended college at Lee College in Cleveland, Tennessee and the University of Evansville in Evansville, Indiana. After serving almost four years in the Air Force as an electronic technician, he returned to college and graduated in 1977 with a Bachelor of Science Degree in Mechanical Engineering from the University of Arizona. In 1987 he entered the University of Tennessee and in 1993 received a Master of Science Degree in Computer Science.

In 1977 he worked for the DuPont Corporation in Jonesboro, Arkansas as a process engineer. This was followed by two years of employment with the Lawrence Livermore National Laboratory in Livermore, California starting in 1979. Since then he has been employed at the Department of Energy complex in Oak Ridge, Tennessee currently under management by Martin-Marietta Corporation. He is currently working as a computer specialist and is a lead analyst on a major computer project at the Oak Ridge facilities.