Parallelization techniques for scientific and engineering applications and implementation of the boundary element method (BEM)

Jeffrey Scott Clary

Iowa State University
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Parallelization techniques for scientific and engineering applications and implementation of the boundary element method (BEM)

Clary, Jeffrey Scott, Ph.D.

Iowa State University, 1994
Parallelization techniques for scientific and engineering applications
and implementation of the boundary element method (BEM)

by

Jeffrey Scott Clary

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Department: Computer Science
Major: Computer Science

Approved:
Signature was redacted for privacy.
Signature was redacted for privacy.
In Charge of Major Work
Signature was redacted for privacy.
For the Major Department
Signature was redacted for privacy.
For the Graduate College

Iowa State University
Ames, Iowa

1994
TABLE OF CONTENTS

1 INTRODUCTION ..............................................................................................................1

2 MASPAR ARCHITECTURE .................................................................................................4
   2.1 General Architectural Issues ..................................................................................4
   2.2 Specific Issues on the MasPar MP-1 and MP-2 ....................................................4

3 SYSTOLIC ARCHITECTURES AND MESH ALGORITHMS ........................................8
   3.1 Method for Transforming Sequential Algorithms into Systolic Architectures .......9
   3.2 Parallel Systolic Algorithms for Mesh-Connected Computers .........................17

4 MEMORY ACCESS OPTIMIZATIONS ............................................................................24
   4.1 Software Pipelining -- Reducing the Cost of Each Access ....................................24
   4.2 Blocking - Reducing the Number of Accesses ......................................................26
   4.3 Performance Improvements ..................................................................................27

5 THE BOUNDARY ELEMENT METHOD (BEM) ............................................................30
   5.1 BEM in One Dimension ......................................................................................30
       5.1.1 Solving $u(x)$ for a given $f(x)$ and homogeneous b.c.'s ............................32
       5.1.2 Solving $u(x)$ for any $f(x)$ and homogeneous b.c.'s .................................34
       5.1.3 Solving $u(x)$ for any $f(x)$ with fixed-type inhomogeneous b.c.'s ..........37
       5.1.4 Solving $u(x)$ for any $f(x)$ and any b.c.'s ...............................................38
   5.2 Boundary Element Method in Two Dimensions ..................................................43
       5.2.1 Numerical Implementation ......................................................................51
       5.2.2 Time-dependent problems ......................................................................54

6 BEM IMPLEMENTATION ...............................................................................................56
   6.1 BEM Geometry ..................................................................................................57
   6.2 Collocation and Integration ...............................................................................59
   6.3 Solving a System of Linear Equations ...............................................................60
   6.4 The BEM System of Equations .........................................................................65
   6.5 Parallel Implementation .......................................................................................67

7 CORE BEM PARALLEL ALGORITHMS ......................................................................73
   7.1 Matrix Inversion ..................................................................................................73
   7.2 Matrix Multiplication .........................................................................................78
       7.2.1 Logarithmic Sum Algorithm .....................................................................79
       7.2.2 Broadcast Algorithm ...............................................................................80
       7.2.3 Systolic Parallel Algorithm ......................................................................81
       7.2.4 Performance Analysis .............................................................................82
       7.2.5 Source-Level Software Pipelining of Matrix Multiplication in the BEM ...84
   7.3 LU Decomposition .................................................................................................85
       7.3.1 Source-Level Software Pipelining of LU Decomposition in the BEM .......88
ACKNOWLEDGEMENTS

First, I would like to thank those who have financially supported the research reported in this dissertation. IBM corporation supported me with a graduate fellowship. When I left campus and went to work for IBM Rochester before this dissertation was finished, IBM continued its support by permitting me to use company computing facilities and by reimbursing my graduate tuition. I am also grateful to Ames Laboratory for making available the MasPar MP-1 and MP-2 machines necessary for this research. Finally, thanks to the ISU computer science department for supporting me with graduate assistantships throughout my graduate career.

Dr. Suresh Kothari went above and beyond the call of duty in his support of me and my research. To accommodate my working schedule, he met with me on weekends and holidays, spending many hours discussing the work I had done, and helping plan what to do next. His constructive criticism and fresh point of view helped make this dissertation much better than it would otherwise have been. He never lost faith in me when progress was slow in coming. I am proud that Dr. Kothari is my advisor and my friend.

Thanks also goes to Dr. Ambar Mitra of engineering mechanics, my co-major professor. He supplied the expertise on the boundary element method without which this research could not have been done. Besides supplying a sequential Fortran BEM program that formed the basis for the parallel implementation, Dr. Mitra held a summer “seminar” on the BEM just for myself and Dr. Kothari.

Steve and Jodi Jenkins provided shelter for me during many of my weekend trips to Ames to finish this dissertation. I thank them for their hospitality. It made those trips much more pleasant.

I will always be grateful to my parents, Robert and Joyce Clary, for instilling in me at an early age the importance of learning and education, and for supporting me through my under-
graduate years. My earning this doctoral degree is the culmination of a process they set in motion years ago.

My biggest supporter is also the most important person in my life, my wife Jan. We married early in my graduate career, and she financially supported our family quite well for three years, allowing me to pursue my studies. She has been solidly behind me in this research. (In fact, there are those who might think that without that occasional push from behind, I might never have finished.) I can never thank Jan enough for believing in me and just for being in my life. Along the way, she gave me a beautiful daughter and son, Kelsey and Sean.

That brings to mind one last person to thank for being supportive of my research. Jan's mother, Rose Hudson, helped care for Kelsey for several months, freeing up precious time that would have been spent dealing with a child care provider and worrying about whether the care was good enough. Since I took a full-time job in Rochester, Minnesota, my family has spent several two-week visits at her home, giving me the extra time to finish this dissertation.

Finally, I thank God for giving me the ability to perform this work. He and all of those mentioned above have made it possible for me to finally achieve the toughest goal I have set for myself thus far.
ABSTRACT

This dissertation reports the implementation of a boundary element method (BEM) application on the massively parallel MasPar MP-1 and MP-2 computers. That implementation provides a case study to demonstrate several techniques for parallelization of sequential algorithms and for optimization of parallel programs.

An existing formal technique for transforming a sequential algorithm into a systolic architecture is presented. This dissertation then discusses how a parallel systolic algorithm on a mesh-connected computer can be derived from such a systolic architecture. The matrix multiplication algorithm used in the BEM implementation is derived in this way.

As part of the BEM implementation, this dissertation covers a novel method of solving a system of linear equations, using matrix inversion and LU decomposition. This method is shown to be less expensive than LU decomposition alone. Several parallelizations of matrix inversion are considered.

Finally, this dissertation presents techniques for transforming parallel program source code to increase performance. The transformation improves performance by decreasing processor local memory access cost and by increasing processor utilization.
1 INTRODUCTION

This thesis records a study of the implementation of a real-world scientific application on a parallel computer. The material covered ranges from purely theoretical work on how to map sequential algorithms onto parallel architectures, to machine specific programming techniques to squeeze every possible MFLOP from a parallel program. The scientific application implemented was the boundary element method (BEM), a technique for solving a governing differential equation over a two- or three-dimensional problem domain. The BEM was implemented on the MasPar MP-1 and MP-2 machines. The MasPAR has a distributed-memory, mesh-connected, SIMD architecture.

Formal analysis techniques were used to help choose parallel algorithms for various parts of the BEM implementation, in particular for matrix multiplication and matrix inversion.

S.C. Kothari, E. Gannett, and H. Oh provide a formal method to transform a serial algorithm in the form of a nested for loop to a systolic algorithm. The method has been applied to a variety of algorithms, including convolution, matrix multiplication, and LU decomposition [8]. The resulting systolic algorithm can often be converted to an efficient systolic parallel algorithm for a mesh-connected, SIMD computer. The SIMD matrix multiplication algorithm derived from the method has an efficient communication pattern and was used in the implementation of the BEM.

The implementation of matrix inversion is not often considered because its most common use, solving a system of linear equations, can be accomplished with less computation by using LU decomposition. However, special circumstances in the BEM implementation make matrix inversion useful. A formal analysis of execution time was used to help choose an efficient matrix inversion algorithm for the BEM implementation.

Once good parallel algorithms are chosen, by using knowledge of parallel architectures in general, and the MasPar system in particular, an application programmer can make local source
code changes that further decrease program execution time. These kinds of changes are dis-
cussed, using the BEM implementation as a case study.

Finally, on the MasPar computers, achieving peak performance requires careful scheduling
of memory operations. It turns out that execution time can be reduced considerably by applying
a fairly simple source-level software pipelining technique. The effects of this technique are
studied both for matrix multiplication in isolation, and for the BEM implementation as a whole.

Detailed performance measurements were made of the BEM implementation on a range of
problem sizes. The effects of the source code changes mentioned above are presented and
analyzed.

One of the most interesting parts of the implementation effort was the synergy generated
by having engineers and computer scientists cooperate to realize a common goal. The specific
mathematics of the BEM and the serial Fortran BEM implementation are specialized pieces of
engineering knowledge. Parallel programming expertise and an understanding of novel com-
puter architectures comes from a computer science background. Both are needed to efficiently
implement the BEM on a parallel computer. This domain expert/computer scientist model is
very powerful in many application areas, and was crucial to the success of this project.

The intended audience for this thesis includes both computer scientists, interested in parallel
programming techniques, and the computational engineer, interested in using the BEM to solve
engineering problems. In particular, the text and the actual code included here should provide
a starting point for implementing more complex BEM applications, such as 3-dimensional BEM,
on the MasPar.

Here is an outline of the rest of the dissertation. Chapter 2 briefly covers SIMD computers
in general and the MasPar MP-1 and MP-2 in particular. Chapter 3 reviews the systolic method
of Kothari, Gannet, and Oh, and shows how the method can be adapted to derive systolic parallel
algorithms for mesh-connected, SIMD computers. Chapter 4 explains memory access optimi-
zation in detail, using matrix multiplication and LU decomposition as sample problems. Chapter
5 presents the mathematical foundations of the BEM. Chapter 6 details the parallel implementation of the BEM on the MasPar MP-1 and MP-2. Chapter 7 describes the analysis used to choose parallel algorithms for the most computationally intensive parts of the BEM. Finally, Chapter 8 discusses several code changes made to reduce execution time. The rest of the dissertation consists of conclusions, a bibliography, and appendices containing raw performance data, the source code for both serial and parallel BEM implementations, and sample BEM problem data.
2 MASPAR ARCHITECTURE

MasPar computers have a mesh-connected, distributed-memory, single-instruction stream, multiple data stream (SIMD) architecture. Parallel computer architectures in general and the MasPar architecture in particular are covered in detail by other authors, some listed in the bibliography. This chapter will hit the high points of both, as they are related to the BEM implementation and the performance improvement techniques discussed in later chapters. It is intended as a quick summary for readers unfamiliar with SIMD architectures or the MasPar computer.

2.1 General Architectural Issues

Distributed memory means that the expense of interprocessor communication is an issue rather than shared memory bandwidth, as would be the case on a shared-memory machine. A mesh connection topology makes algorithms attractive that have very localized data communication patterns, preferably communicating only with nearest neighbors on the processing element (PE) array. The systolic method described in Chapter 3 leads to algorithms with this kind of communication pattern.

On a SIMD architecture, with every PE executing the same instructions in lock step, the MIMD problems of load-balancing and synchronization are not present. The analogous problem is maximizing processor utilization, that is, minimizing the time each processor spends idle.

2.2 Specific Issues on the MasPar MP-1 and MP-2

The two MasPar models used in this project, the MP-1 and MP-2, are very similar architecturally. The main difference is that the newer MP-2 has more powerful processing elements.
When there is no need to make a distinction, the term "MasPar" will be used to refer to both models.

The MasPar system consists of a front-end, an array control unit (ACU), and data parallel unit (DPU). Figure 1 shows the relationship of the parts. The front-end is Unix box that hosts a file system, compiler, graphical debugger, and other utilities. The ACU controls the processing elements, broadcasting each parallel instruction to the PE array. Each (active) processing element executes parallel instructions in lock step with the rest of the PEs. Processing elements can be "turned off" so that they do not participate in parallel computation. Each PE is directly connected to its eight nearest neighbors via xnet connections. The connections wrap at the edges so the connection topology is toroidal. There is also a router, for implementing arbitrary inter-processor communication patterns. Router communication is generally much slower than xnet communication.

![Figure 1: MasPar System Components](image)

The implementation language for this project was MPL, a parallel dialect of C. MPL extends the C language with the plural keyword. A plural variable is allocated on each processing element. Non-plural, or singular variables are allocated in ACU memory. MPL expressions that
do not access plural data execute entirely on the ACU. Statements that access plural data are broadcast by the ACU to the DPU for parallel execution.

The MasPar PE memory architecture is cacheless and pipelined. That is, after one memory access has been started, instructions can be executed before the memory access finishes, if there are no data dependencies that prevent it. PE memory is accessed only through explicit load/store instructions. Each PE has enough bit-addressable register space for 20 double-precision floats. The MPL language honors the register keyword when possible. Chapter 4 describes methods of modifying source code to take advantage of these features of the memory architecture.

The MP-1 and MP-2 can have from 1K to 16K processors. Both machines have a clock rate of 12.5 MHz, and the same instruction set. However, the MP-1 uses 4-bit processors while the MP-2 uses 32-bit processors. The MP-2 can perform floating point operations four to five times faster than the MP-1. Measured cycle times for several instructions are shown in Table 1.

The architectural features discussed above lead to a number of guidelines an application programmer can use to help produce efficient MasPar programs.

• Keep all PEs busy.
• Keep interprocessor communication to a minimum
• Prefer xnet communication over router communication
• Reduce the time spent accessing PE memory by overlapping memory operations with useful computation.

The extent to which these guidelines are followed will affect how close an implementation gets to the peak performance of the MasPar MP-1 or MP-2.
Table 1: MasPar instruction cycle times

<table>
<thead>
<tr>
<th>Operation</th>
<th>MP-1 Cycles</th>
<th>MP-2 Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit Load</td>
<td>79</td>
<td>38</td>
</tr>
<tr>
<td>32-bit Store</td>
<td>71</td>
<td>33</td>
</tr>
<tr>
<td>64-bit Load</td>
<td>148</td>
<td>71</td>
</tr>
<tr>
<td>64-bit Store</td>
<td>142</td>
<td>65</td>
</tr>
<tr>
<td>Single Precision FP Negate</td>
<td>33</td>
<td>8</td>
</tr>
<tr>
<td>Single Precision FP Add</td>
<td>121</td>
<td>24</td>
</tr>
<tr>
<td>Single Precision FP Multiply</td>
<td>216</td>
<td>40</td>
</tr>
<tr>
<td>Single Precision FP Divide</td>
<td>311</td>
<td>72</td>
</tr>
<tr>
<td>Double Precision FP Negate</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>Double Precision FP Add</td>
<td>178</td>
<td>45</td>
</tr>
<tr>
<td>Double Precision FP Multiply</td>
<td>539</td>
<td>122</td>
</tr>
<tr>
<td>Double Precision FP Divide</td>
<td>957</td>
<td>200</td>
</tr>
<tr>
<td>32-bit Nearest Neighbor Xnet</td>
<td>42</td>
<td>43</td>
</tr>
<tr>
<td>32-bit Pipelined Xnet</td>
<td>75</td>
<td>45</td>
</tr>
<tr>
<td>(start-up time)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(per processor)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>64-bit Nearest Neighbor Xnet</td>
<td>73</td>
<td>76</td>
</tr>
<tr>
<td>64-bit Pipelined Xnet</td>
<td>137</td>
<td>82</td>
</tr>
<tr>
<td>(start-up time)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(per processor)</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
3 SYSTOLIC ARCHITECTURES AND MESH ALGORITHMS

A systolic architecture is an array of processors, each of which performs a simple computation and passes results to one or more neighbors in the array. Generally, all the processors perform identical computations, except for processors on the edge of the array, which may perform input/output functions. An advantage to this kind of processing is that it lends itself to VLSI implementation.

Systolic architectures have been designed for many problems, including matrix multiplication, convolution, and LU decomposition. Kothari, Oh, and Gannett present a formal method for transforming a sequential algorithm into a time-optimal systolic architecture in [8]. The method works for algorithms which can be expressed as nested for loops in which array indices are linear functions of the loop variables. In this thesis, this procedure for transforming a sequential algorithm into a systolic architecture will be called the systolic method.

A systolic architecture implementation of a serial algorithm often suggests an efficient implementation on a mesh-connected parallel computer, such as a MasPar. On such machines, interprocessor communication with nearest neighbors is usually inexpensive. If a suitable initial data distribution can be found, the processors on the mesh computer perform the same computation as the processors of the systolic architecture, and pass values (using nearest neighbor communication) in the same directions as the systolic architecture. Instead of being output at the edge of the array, the result values wrap around the toroidal mesh. Such an implementation will be called a parallel systolic algorithm.

A systolic architecture is usually thought of as having very fine-grained parallelism, with each processing element holding only one data item at a time from each variable stream, and with the number of processing element growing to fit the problem size. A distributed-memory, mesh-connected computer, on the other hand, usually has enough memory per PE to hold many elements from each variable stream, and is expected to be able to handle a variety of problem.
sizes. This gap is bridged by using virtualization of processors. Each PE of the mesh-connected computer does the work of several processing elements of the systolic architecture.

This chapter will review the systolic method for generating systolic architectures, and discuss how a systolic architecture can be used to find an efficient parallel systolic algorithm on a mesh-connected parallel computer. Examples will be used to illustrate the process from start to finish.

3.1 Method for Transforming Sequential Algorithms into Systolic Architectures

This section restates the mechanics of the systolic method, and provides an intuitive view of how and why it works. For a formal proof of its correctness, see the original paper. The systolic method produces what Kothari, et al. call a linear flow systolic architecture (LFSA), characterized by three constraints.

- CONSTRAINT 1: A variable (array element) can participate in at most one computation in any given beat. That is, a variable can only be at one processor at a time.
- CONSTRAINT 2: Each processor performs only one basic computation in any given beat.
- CONSTRAINT 3: The speed and direction of flow for each variable is constant, and communication is between nearest-neighbors only. That is, the flow of a variable in any single direction is in the range [-1, 1].

The systolic method specifies a systolic architecture by defining four functions:

- STEP -- maps a basic computation to a time step
- PLACE -- maps a basic computation to a processing element
- FLOW -- specifies speed and direction of a variable flow
- PATTERN -- specifies the initial locations of a variable
The systolic method handles algorithms that can be expressed as a nested for loop.

for \( i = s_1 \) to \( t_1 \) do
\[
\text{for } i = s_2 \text{ to } t_2 \text{ do}
\]
\[
\text{...}
\]
\[
\text{for } i = s_n \text{ to } t_n \text{ do}
\]
\[
\text{if } \text{COND}_1 (I) \text{ then } \text{COMP}_1 (V^1, V^2, \ldots, V^m)
\]
\[
\text{elsif } \text{COND}_2 (I) \text{ then } \text{COMP}_2 (V^1, V^2, \ldots, V^m)
\]
\[
\text{...}
\]
\[
\text{elsif } \text{COND}_r (I) \text{ then } \text{COMP}_r (V^1, V^2, \ldots, V^m)
\]

Each \( s_k \) and \( t_k \) are constants or linear functions of \( I = (i_1, i_2, \ldots, i_n) \), each for loop has a step of 1 or \(-1\), and each variable stream \( V^k \) is an \( n - 1 \) dimensional array, with each index a linear function of \( I \).

Each variable stream must be indexed on \( n - 1 \) independent functions of \( I \). For example, the array \( A \) in the loop below has two indices, but one is a linear combination of the other.

for \( i = 1 \) to \( N \) do
\[
\text{for } j = 1 \text{ to } N \text{ do}
\]
\[
\text{for } k = 1 \text{ to } N \text{ do}
\]
\[
C[i, k] = A[i, 2^i] + B[j, k]
\]

Only some of the elements of the array \( A \) are being accessed. It is not really being used as a two-dimensional array, and the algorithm does not qualify. If the \( i \)th indexing function \( f_i \) for variable stream \( V (f_1 (I), f_2 (I), \ldots, f_{n-1} (I)) \) is \( f_i = \gamma_{i,1} i_1 + \gamma_{i,2} i_2 + \cdots + \gamma_{i,n} i_n \) then another way of saying the same thing is that the index matrix shown below is of rank \( n - 1 \).

\[
\begin{bmatrix}
\gamma_{1,1} & \gamma_{1,2} & \cdots & \gamma_{1,n} \\
\gamma_{2,1} & \gamma_{2,2} & \cdots & \gamma_{2,n} \\
\vdots & \vdots & \ddots & \vdots \\
\gamma_{n-1,1} & \gamma_{n-1,2} & \cdots & \gamma_{n-1,n}
\end{bmatrix}
\]
The systolic method generates an \( n - 1 \) dimensional systolic array, so it is important that the data arrays have a "real" dimensionality of \( n - 1 \). Less than \( n - 1 \) dimensions would mean broadcasting along the missing dimensions. More than \( n - 1 \) dimensions would imply that more than one element from the array maps to a single processing element.

Each combination of \( I = (i_1, i_2, \ldots, i_n) \) that satisfies one of the conditions of the algorithm is called a valid tuple (or just tuple). It follows from the above constraints for eligible algorithms that each valid tuple corresponds to a unique basic computation.

Since a tuple is determined by an instantiation of \( I \), the STEP function can be expressed in terms of \( I \), as \( \text{STEP} (I) = \alpha_1 i_1 + \alpha_2 i_2 + \ldots + \alpha_n i_n \). There are \( n - 1 \) PLACE functions to determine the processing element to perform each basic computation.

Let \( \text{PLACE}_j (I) = \beta_{j,1} i_1 + \beta_{j,2} i_2 + \ldots + \beta_{j,n} i_n \quad (1 \leq j \leq n - 1) \) determine the \( i^{th} \) index of the processing element that performs the basic computation corresponding to tuple \( I \). Once the step and place function have been defined, flow and pattern functions follow.

The three constraints above can be used as a basis for finding suitable \( \alpha \) and \( \beta \) values. First, consider the system of linear equations below.

\[
\begin{bmatrix}
\alpha_1 & \alpha_2 & \ldots & \alpha_n \\
\gamma_{1,1} & \gamma_{1,2} & \ldots & \gamma_{1,n} \\
\vdots & \vdots & \ddots & \vdots \\
\gamma_{n-1,1} & \gamma_{n-1,2} & \ldots & \gamma_{n-1,n}
\end{bmatrix}
\begin{bmatrix}
i_1 \\
i_2 \\
\vdots \\
i_n
\end{bmatrix}
= 
\begin{bmatrix}
B \\
c_1 \\
\vdots \\
c_{n-1}
\end{bmatrix}
\]

The first equation represents the step function for the tuple \( I \), that is, at what beat that computation occurs. The rest of the equations represent, for some particular variable array, the element of the array used in the computation \( I \). To ensure that a particular array element is used in at most one computation per time step (CONSTRAINT 1), this system of equations must have a unique solution. Thus, the array shown must have a non-zero determinant. Kothari, et al., call this the step constraint. Each variable array introduces a step constraint.
Now consider the following system of equations:

\[
\begin{bmatrix}
\beta_{j,1} & \beta_{j,2} & \ldots & \beta_{j,n} \\
\gamma_{1,1} & \gamma_{1,2} & \ldots & \gamma_{1,n} \\
\vdots & \vdots & \ddots & \vdots \\
\gamma_{n-1,1} & \gamma_{n-1,2} & \ldots & \gamma_{n-1,n}
\end{bmatrix}
\begin{bmatrix}
i'_1 - i_1 \\
i'_2 - i_2 \\
\vdots \\
i'_{n} - i_{n}
\end{bmatrix} =
\begin{bmatrix}
\text{DIFF} \\
0 \\
\vdots \\
0
\end{bmatrix}
\]

This system of equations represents the situation for some array element between two consecutive time steps. The constant DIFF stands for the distance along the \( j \text{th} \) dimension of two processing elements that access a particular array element in consecutive time steps. The rest of the right-hand side is zero because the array element is the same at both time steps. \( I \) and \( I' \) are tuples (basic computations) that consecutively access the array element. There are two cases to consider.

First, if the variable does not flow in the \( j \) direction, then DIFF = 0, and the system is homogenous. Since \( I \) and \( I' \) are different, there is at least one nontrivial solution the system, and the determinant must be zero. On the other hand, if DIFF \( \neq 0 \), because variables flow at a constant rate, \( I' - I \) must be unique. Let \( \Gamma_I \) and \( \Lambda \) be defined as in equations (3) and (4).

(3) \( \Gamma_I =
\begin{bmatrix}
\beta_{j,1} & \beta_{j,2} & \ldots & \beta_{j,I-1} & \text{DIFF} & \beta_{j,I+1} & \ldots & \beta_{j,n} \\
\gamma_{1,1} & \gamma_{1,2} & \ldots & \gamma_{1,I-1} & 0 & \gamma_{1,I+1} & \ldots & \gamma_{1,n} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots \\
\gamma_{n-1,1} & \gamma_{n-1,2} & \ldots & \gamma_{n-1,I-1} & 0 & \gamma_{n-1,I+1} & \ldots & \gamma_{n-1,n}
\end{bmatrix}
\]

(4) \( \Lambda =
\begin{bmatrix}
\alpha_1 & \alpha_2 & \ldots & \alpha_n \\
\beta_{1,1} & \beta_{1,2} & \ldots & \beta_{1,n} \\
\vdots & \vdots & \ddots & \vdots \\
\beta_{n-1,1} & \beta_{n-1,2} & \ldots & \beta_{n-1,n}
\end{bmatrix}
\]
Then by Cramer's Rule, \( i' = i \frac{|\Gamma|}{|\Lambda|} \). To ensure integer variable flow, \(|\Gamma|\) must be a multiple of \( \Lambda \). This can be ensured by requiring that \(|\Lambda| = \pm 1\). (Kothari et al. claim that this strong constraint is justified because in many systolic algorithms, \(|\Gamma| = \pm 1\) for some \( I \).) Thus, from the two cases just discussed, \(|\Lambda|\) must be \( \pm 1 \) or \( 0 \). This is called the place constraint. There is a place constraint for each variable stream in each dimension of the systolic architecture.

Finally, consider the following system of equations:

\[
\begin{bmatrix}
\alpha_1 & \alpha_2 & \cdots & \alpha_n \\
\beta_{1,1} & \beta_{1,2} & \cdots & \beta_{1,n} \\
\vdots & \vdots & \ddots & \vdots \\
\beta_{n-1,1} & \beta_{n-1,2} & \cdots & \beta_{n-1,n}
\end{bmatrix}
\begin{bmatrix}
i_1 \\
i_2 \\
i_3 \\
i_n
\end{bmatrix}
=
\begin{bmatrix}
B \\
p_1 \\
p_2 \\
p_{n-1}
\end{bmatrix}
\]  

(5)

As in the step constraint, the first equation determines at what time step the computation specified by \( I \) occurs. The rest of the equations determine what processing element performs the computation. By CONSTRAINT 2, the time step and processing element determine a unique basic computation, so the system must have a unique solution. As in the step constraint, the determinant of the array must be non-zero. This is called the compatibility constraint.

The step, place, and compatibility constraints can be used to determine all of the \( \alpha \) and \( \beta \) values. The steps of the systolic method will be illustrated using matrix multiplication as an example. The systolic method can be used when the basic computations are not commutative, that is, their order cannot be changed without affecting the result. However, this paper will be restricted to the commutative case.

The matrix multiplication algorithm can be written in nested for loop structure as follows:

```plaintext
for i = 1 to N - 1 do
    for j = 0 to N - 1 do
        for k = 0 to N - 1 do
```

for i = 1 to N - 1 do
for j = 0 to N - 1 do
for k = 0 to N - 1 do
This leads to the following three step constraint arrays, one for each of A, B, and C:

\[
\begin{bmatrix}
\alpha_1 & \alpha_2 & \alpha_3 \\
1 & 0 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\quad
\begin{bmatrix}
\alpha_1 & \alpha_2 & \alpha_3 \\
0 & 0 & 1 \\
0 & 1 & 0 \\
\end{bmatrix}
\quad
\begin{bmatrix}
\alpha_1 & \alpha_2 & \alpha_3 \\
1 & 0 & 0 \\
0 & 1 & 0 \\
\end{bmatrix}
\]

For the three determinants to be non-zero, it is necessary that \( \alpha_1 \neq 0 \), \( \alpha_2 \neq 0 \), and \( \alpha_3 \neq 0 \).

The heuristic of the systolic method calls for \( \alpha \) values to be set to 0 when possible and -1 or 1 otherwise. In this example, let \( \alpha_1 = 1 \), \( \alpha_2 = 1 \), and \( \alpha_3 = 1 \), for \( \text{STEP}(i, j, k) = i + j + k \).

The set of valid tuples must be convex for the systolic method to be applied. To determine the optimal \( \text{STEP} \) function, evaluate it at the corners of this convex set to find the two time steps furthest apart, and so find the elapsed time for the proposed systolic architecture. In this case, at \( (i = 0, j = 0, k = 0) \), \( \text{STEP} = 0 \) and at \( (i = N - 1, j = N - 1, k = N - 1) \), \( \text{STEP} = 3N - 3 \), for an elapsed time of \( 3N - 2 \). This is an optimal step function.

Next, the three variable streams A, B, and C lead to six place constraint arrays, one for each dimension of each variable stream.

\[
\begin{bmatrix}
\beta_{1,1} & \beta_{1,2} & \beta_{1,3} \\
1 & 0 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\quad
\begin{bmatrix}
\beta_{1,1} & \beta_{1,2} & \beta_{1,3} \\
0 & 0 & 1 \\
0 & 1 & 0 \\
\end{bmatrix}
\quad
\begin{bmatrix}
\beta_{1,1} & \beta_{1,2} & \beta_{1,3} \\
1 & 0 & 0 \\
0 & 1 & 0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
\beta_{2,1} & \beta_{2,2} & \beta_{2,3} \\
1 & 0 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\quad
\begin{bmatrix}
\beta_{2,1} & \beta_{2,2} & \beta_{2,3} \\
0 & 0 & 1 \\
0 & 1 & 0 \\
\end{bmatrix}
\quad
\begin{bmatrix}
\beta_{2,1} & \beta_{2,2} & \beta_{2,3} \\
1 & 0 & 0 \\
0 & 1 & 0 \\
\end{bmatrix}
\]

Since the first three arrays are similar to the second three, the constraints on the \( \beta_{1,i} \) terms are the same as those for the \( \beta_{2,i} \) terms. To satisfy the place constraints, the determinant of each of these arrays must be 0, -1, or 1. For \( i = 1 \) and \( i = 2 \), the possible \( \beta \) values implied by the place constraints for A, B, and C are shown below:
PLACE constraint for A: $\beta_{i,2} \in \{-1, 0, 1\}$

PLACE constraint for B: $\beta_{i,1} \in \{-1, 0, 1\}$

PLACE constraint for C: $\beta_{i,3} \in \{-1, 0, 1\}$

To choose $\beta$ values from the possible values above, the systolic method calls for checking combinations against the compatibility constraint, starting with those that produce place functions with minimal span. Combinations with increasing spans are tried until one is found that satisfies compatibility. For matrix multiplication, the compatibility array, with the $\alpha$ values filled in, is shown below:

$$
\begin{bmatrix}
1 & 1 & 1 \\
\beta_{1,1} & \beta_{1,2} & \beta_{1,3} \\
\beta_{2,1} & \beta_{2,2} & \beta_{2,3}
\end{bmatrix}
$$

The determinant is $\beta_{1,2}\beta_{2,3} + \beta_{1,3}\beta_{2,1} + \beta_{1,1}\beta_{2,2} - \beta_{2,1}\beta_{1,2} - \beta_{2,2}\beta_{1,3} - \beta_{2,3}\beta_{1,1}$. Obviously, setting all the $\beta$'s to zero would give the place functions with minimal span, but would not give a nonzero determinant for the compatibility array. Letting all $\beta$ values be zero except for $\beta_{1,1} = 1$ and $\beta_{2,2} = 1$ results in place functions with as small a span as any that satisfy compatibility. So, let $\text{PLACE}(i,j,k) = [i,j]$.

Given the step and place functions derived above, the flow and pattern functions can be determined. The flow function for each variable stream determines the distance it moves in the processing array in a beat. For a variable stream $V(f_f(I), f_2(I), \ldots, f_{n-1}(I))$ the flow function in the $i$th dimension is defined as below:

$$
(6) \quad \text{FLOW}_i(V) = \frac{\text{PLACE}_i}{\text{STEP}} \mod (f_1, f_2, \ldots, f_{n-1})
$$

For the matrix multiplication problem, the flow functions are
(7) FLOW \( A[i, k] \) = \( \frac{[i, j]}{i + j + k} \mod (i, k) = [0, 1] \)

(8) FLOW \( B[k, j] \) = \( \frac{[i, j]}{i + j + k} \mod (k, j) = [1, 0] \)

(9) FLOW \( C[i, j] \) = \( \frac{[i, j]}{i + j + k} \mod (i, j) = [0, 0] \)

Finally, the starting place of each element of each variable stream is determined by the pattern function, derived from step, place, and flow as shown below for the \( ith \) dimension \( (t_0 \) is the value of the first time step):

(10) \( \text{PATTERN}_i (V) = \text{PLACE}_i - (\text{STEP} - t_0) \text{FLOW}_i (V) \)

For matrix multiplication, the three pattern functions are

(11) \( \text{PATTERN} (A[i, k]) = [i, j] - (i + j + k) [0, 1] = [i, -(i + k)] \)

(12) \( \text{PATTERN} (B[k, j]) = [i, j] - (i + j + k) [1, 0] = [- (j + k), j] \)

(13) \( \text{PATTERN} (C[i, j]) = [i, j] - (i + j + k) [0, 0] = [i, j] \)

Figure 2 illustrates the initial data distribution for the derived matrix multiplication systolic architecture, for an example with \( N = 3 \).

The algorithm executed by each processing element follows from the step and flow functions derived above, where \( a, b, \) and \( c \) are the elements of \( A, B, \) and \( C \) currently residing on that processor.

\[
\text{for } s = 0 \text{ to } 3 \times N - 3 \text{ do } \\
\begin{align*}
\text{begin } \\
\quad c &= c + a \times b \\
\text{shift a one PE east} \\
\text{shift b one PE south} \\
\text{end}
\end{align*}
\]
3.2 Parallel Systolic Algorithms for Mesh-Connected Computers

Often, a systolic architecture for a particular problem suggests a parallel systolic algorithm for a mesh-connected computer. The architecture to implement matrix multiplication derived above can easily be transformed into Cannon’s parallel algorithm [2]. In this case, it turns out that the only transformation required is to take the place functions modulo $N$, “wrapping” the arrays onto a toroidal mesh. The flow functions remain the same. The initial data distribution is shown in Figure 3.

In general, a systolic architecture can be transformed into a parallel systolic algorithm that takes no more steps. The place and step functions for the systolic architecture can always be normalized to start with zero; this will be assumed below. The following transformation, where
STEP denotes the step function for the parallel systolic algorithm, and the place, flow and pattern functions are denoted similarly, defines a parallel systolic algorithm with the same number of steps and PEs as its corresponding systolic architecture.

(14) $\text{PLACE}''_i = \text{PLACE}_i$

(15) $\text{FLOW}''_i = \text{FLOW}_i$ (except that off-the-edge flows wrap on toroidal mesh)

(16) $\text{PATTERN}''_i = \text{PATTERN}_i \mod \text{span}(\text{PLACE}_i)$

(17) $\text{STEP}'' = \text{STEP}$

Figure 3: Initial data distribution for matrix multiplication systolic parallel algorithm

The combination of toroidal wrapping and the modulo in the pattern function results in a particular variable element reaching a particular place at the same step as it would have in the systolic architecture. The modulo in the pattern function might result in more variables being mapped to a single PE for the parallel systolic algorithm than for the systolic architecture. This means more required memory per processor, and more communication per step.

This simple transformation does not accomplish anything profound. The parallel systolic algorithm just simulates the systolic architecture, storing values that would have been fed into edge processors at appropriate places in the mesh until they are needed. However, the systolic parallel algorithm shown above for matrix multiplication does not just use this simple transformation, since its execution time is $N$, compared to $3N - 2$ for the corresponding systolic algo-
rithm. This parallel systolic algorithm takes advantage of the fact that the modulo in the PATTERTNM functions happen to preload A, B, and C values in such a way that some basic computations can be performed before they are scheduled by the systolic architecture STEP function. For example, C[0,1] = C[0,1] + A[0,2]*B[2,1] is performed at step 0, instead of at step 3 as in the systolic architecture.

This is possible because toroidal wrap causes the same set of variable elements to visit the same PEs at regular intervals. For the matrix multiplication, the same set of variable elements returns to a given PE every \( N \) steps. In this thesis, the number of steps taken for all variable elements to return to the same PEs will be called \( F \), the folding factor. The folding factor can be determined for any systolic architecture with all FLOW functions of the form \( \frac{1}{k} \), for positive integer \( k \), as follows:

\[
(18) \quad F = \text{LCM} \left\{ \frac{1}{\text{FLOW}_i(V)} \middle| \begin{array}{c}
i \text{ a dimension}, \\
V \text{ a variable w/ non-zero flow}
\end{array} \right\}
\]

A particular variable \( V \) will return to the same place in the \( i \)th dimension in a number of steps equal to the length of the PE array in that dimension times the inverse of its flow. The least common multiple of this value for all variables and all dimensions gives the number of steps taken for the same distribution of variables to PEs to appear.

If the set of variable elements needed to perform a computation are scheduled by the systolic architecture to come together at place \( P \) on step \( S \), then those variables also come together at place \( P \) on step \( S + kF \), for any integer \( k \). In the matrix multiplication example, \( C[0,1] = C[0,1] + A[0,2]*B[2,1] \), scheduled by the systolic architecture for step 3, can just as well be performed at step \( 3 + (-1)N = 0 \). It also turns out that the PE needed is not being used for any other computation at step 0. By folding the matrix multiplication step function by \( N \), the total execution time is reduced from \( 3N - 2 \) to \( N \). The pattern, flow, and place functions are just those
specified in the systolic architecture to parallel systolic algorithm transformation specified above. Only the step function is different.

For matrix multiplication, one might argue that all that is being saved is the time to load and drain the systolic array, and that cost must be paid anyway to preload the three matrices. However, in real applications, matrix multiplication is part of a larger algorithm, and the matrices might be constructed on the PE array by previous computation. This will be seen to be the case for the boundary element method (BEM), discussed in a later chapter.

At this point, we have a method for transforming a systolic architecture into a systolic parallel algorithm which has at least as good an execution time. One might wonder whether using the systolic method to transform a sequential program into a systolic architecture, and then using the method described here to transform it into a parallel systolic algorithm always produces an optimal algorithm, as it did for matrix multiplication. The answer is no, as can be shown using the systolic architecture for convolution described by Kothari, et al.

Convolution can be described by the following sequential algorithm:

\[
\text{for } i = 0 \text{ to } 2 \times N \text{ do} \\
\quad \text{for } j = 0 \text{ to } i \text{ do} \\
\qquad \text{if } 0 \leq j \text{ and } j \leq N \text{ and } i - j \leq N \text{ then} \\
\quad \quad C[i] = C[i] + A[j] \times B[i - j]
\]

The time-optimal systolic architecture is defined as follows:

(19) \( \text{STEP} = -i + 2j \)

(20) \( \text{PLACE} = j \)

(21) \( \text{FLOW} (A) = 0 \)

(22) \( \text{FLOW} (B) = 1 \)

(23) \( \text{FLOW} (C) = \frac{1}{2} \)

(24) \( \text{PATTERN} (A) = j \)
(25) PATTERN (B) = (i - j) - N

(26) PATTERN (C) = \frac{1}{2} i - \frac{1}{2} N

Execution time is 2N + 1 and \text{span}(\text{PLACE}) is N + 1. Because of the 1/2 flow of variable stream C, F = 2N + 2 for this problem. Since F > \text{span} (\text{STEP}) , it clearly does no good to be able to fold computations forward or backward in time by multiples of F. The execution time of the parallel systolic algorithm derived from this systolic architecture is just 2N + 1.

However, a non-optimal systolic architecture for convolution leads to a better parallel systolic algorithm. Consider the systolic architecture defined by the following functions:

(27) \text{STEP} = i + j

(28) \text{PLACE} = i - j

(29) \text{FLOW} (A) = 1

(30) \text{FLOW} (B) = 0

(31) \text{FLOW} (C) = -1

(32) \text{PATTERN} (A) = -2j

(33) \text{PATTERN} (B) = i - j

(34) \text{PATTERN} (C) = 2i

\text{Span} (\text{STEP}) is 3N + 1, \text{span} (\text{PLACE}) is N + 1, and because all non-zero flows are 1 or -1, F = N + 1. Since F < \text{span} (\text{STEP}) , there might be an opportunity for folding computations to earlier steps, reducing parallel systolic algorithm execution time. Figure 4 shows the positions of A, B, and C elements at the seven times steps of the systolic architecture for N = 2. The integer pairs at some PEs at some time steps indicate the (i, j) computation being performed.
Figure 4: Systolic Architecture for Convolution ($N = 2$)
Applying the parallel systolic algorithm transformation leads to a parallel systolic algorithm that completes in three steps, as shown in Figure 5.

Future work in this area will be toward the goal of modifying the systolic method so that it produces not necessarily the time optimal systolic architecture, but a systolic architecture that, when transformed as described above, produces a time optimal parallel systolic algorithm. If that goal can be achieved, we will have a systematic method of transforming a large class of sequential algorithms into efficient parallel algorithms for mesh-connected computers. Until then, the systolic method can still be used to suggest good parallel systolic algorithms.

<table>
<thead>
<tr>
<th>Time Step 1</th>
<th>Time Step 2</th>
<th>Time Step 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>(2,1)</td>
<td>(4,2)</td>
</tr>
</tbody>
</table>

Figure 5: Parallel Systolic Algorithm for Convolution ($N = 2$)
4 MEMORY ACCESS OPTIMIZATIONS

As discussed in previous chapters, interprocessor communication and processor utilization are important performance factors on mesh-connected, SIMD computers in general, and MasPar computers in particular. However, a program with optimal communication patterns, and 100% processor utilization can fail to achieve optimal performance because of the cost of local memory accesses on a PE. This chapter describes techniques to reduce the cost of processor memory operations in a MasPar program.

Recall that the MasPar system has a (cacheless) load/store pipelined memory architecture. That is, every memory operation is the result of an explicit load or store, and these operations can be pipelined if data dependencies do not prevent it. Also, the MPL compiler provides control over what variables are kept in the PE general registers. These architectural features allow the application programmer to make source code changes to reduce the cost of PE memory accesses.

Two techniques for reducing the cost of memory accesses, normally discussed in compiler literature, are presented here in the context of source code modification. **Software pipelining** overlaps memory accesses with computation and/or communication. **Blocking** reduces the number of loads in a loop. Matrix multiplication pseudocode is used below to illustrate both techniques. Multiplication of submatrices is carried out on each PE as part of an overall parallel matrix multiplication algorithm using 2D scatter or block decomposition.

The performance gained by applying these methods to the most computation intensive core algorithms of the BEM implementation are presented in Chapter 7. The effect on the overall performance of the BEM implementation is discussed in Chapter 8.

4.1 Software Pipelining — Reducing the Cost of Each Access

**Software pipelining** is a well-known technique allowing the overlap of memory accesses with other operations on processors with a pipelined load/store architecture. The cost of memory
accesses can be reduced by “hiding” them behind other computation or communication operations. To achieve such overlaps on the MasPar system, we discuss a programming technique called *source-level software pipelining*. The technique is primarily targeted at for loops where each iteration involves memory accesses and other operations (e.g. floating point computations) that can be overlapped with memory accesses. Often the memory accesses and the floating point computations within the same iteration cannot be overlapped because of data dependencies between the two. However, the operations in one iteration of the loop can be overlapped with the memory accesses in the next iteration. In essence, the data for the floating point computation is prefetched, and the prefetching is overlapped with operations on already fetched data. Similar techniques for overlapping the operations from two or more successive iterations of a loop are considered in [6,16,17] and [10] for VLIW architectures.

The technique is widely applicable to computation intensive for loops. Researchers at ISU have used it in several algorithms, including fast fourier transform, LU decomposition, Gauss-Jordan elimination, and Hopfield neural network updating and learning algorithms. The technique is illustrated in figures Figures 6 and 7 below using matrix multiplication.

```plaintext
register a, b, c;
for i = 0 to M-1
    for j = 0 to M-1
        begin
            c = C[i, j]
            for k = 0 to M-1
                begin
                    a = A[i, k]
                    b = B[k, j]
                    c += a * b
                    C[i, j] = c
                end
        end
```

Figure 6: Basic submatrix multiply
register a0, a1, b0, b1, c;

for i = 0 to M-1
    begin
        for j = 0 to M-1
            begin
                c = C[i, j]
                a0 = A[i, 0]
                b0 = B[0, j]
                for k = 0 to iM-2
                    begin
                        a1 = A[i, k+1]
                        b1 = B[k+1, j]
                        c += a0 * b0
                        a0 = a1
                        b0 = b1
                    end
                c += a0 * b0
                C[i, j] = c
            end
        end
    end

Figure 7: Submatrix multiply with software pipelining

In the plain loop, elements of the A and B arrays are used in floating point operations immediately after they are accessed, so the floating point operations cannot start until the memory accesses are complete. But by suitable reprogramming of the loop, each fetch can be started some time before the data is actually needed.

In the pipelined loop, computation c += a0 * b0 can be started while the immediately preceding accesses of arrays A and B are still in progress. The technique can be combined with loop unrolling for additional improvement. In practice, the loop should be unrolled so that four memory accesses are started at a time, since the MasPar architecture allows that many pending memory operations.
4.2 Blocking - Reducing the Number of Accesses

Blocking of algorithms is a well-known technique for reducing memory access time by increasing the usage of faster levels of memory hierarchies [18]. As we describe it, the technique requires the availability of a fair number of registers to the application programmer, but could also be used to take advantage of cache memory. We will illustrate the technique using the same example of submatrix multiplication $C = AB$. As shown in the basic loop (Figure 6), ordinarily a register is assigned to an element of $C$ which is accessed in all iterations of the inner loop. In this technique, a block slides over the $C$ matrix, keeping the first $W$ elements from the row in registers, then the next set of $W$ elements in registers, and so on.

The advantage of the blocking technique comes from the fact that it allows the use of one load in place of $W$ loads. In the basic matrix multiplication loop, one element of the $C$ matrix is completely calculated at a time requiring accesses to elements of the $A$ matrix across a row and elements of the $B$ matrix down a column. Using the blocking technique, once an element of the $A$ matrix is loaded into a register, it is used $W$ times in computations of the $W$ elements of the $C$ matrix stored in the block. Thus, having $W$ elements of the $C$ matrix in a block reduces the number of loads for elements of the $A$ matrix by a factor of $W$. A rearranged loop using a block of size $W$ is shown in Figure 8. To simplify the illustration, the set of registers used is treated as an array $c[W]$.

4.3 Performance Improvements

The effects of source-level software pipelining and blocking applied to matrix multiplication were measured using an implementation of Cannon's parallel algorithm on the MasPar. The effect of applying source-level software pipelining to LU decomposition has been measured by ISU graduate student Youngtae Kim. He provided unpublished LUD results for this dissertation.
These results are summarized in Tables 2 and 3. For large problems, applying both source-level software pipelining and blocking gives a speedup of 1.3 on the MP-1 and 1.9 on the MP-2. For LU decomposition, the speedups on the MP-1 and MP-2 were 1.4 and 1.8, respectively.

```c
register a, b, c[W];
for i = 0 to M-1
begin
    for j = 0 to M/W-1
    begin
        for p = 0 to W-1
        begin
            c[p] = C[i, j*W+p]
            for k = 0 to M-1
            begin
                a = A[i, k]
                for p = 0 to W-1
                begin
                    b = B[k, j*w + p]
                    c[p] += a*b
                end
            end
            for p = 0 to W-1
            C[i, j*W+p] = c[p]
        end
    end
end
```

Figure 8: Submatrix multiplication with blocking

Keep in mind that these speedups are for simple algorithms, measured in isolation, at nearly the memory capacity of the machine. Real applications contain more overhead. They also often require working with several matrices at once, so that the maximum problem size for a given memory capacity is reduced. In Chapter 7, the performance of matrix multiplication and LU decomposition will be revisited, in the context of an application program, an implementation of the boundary element method (BEM). It will be seen that though the gains to be made are somewhat muted in a real-world program, it is still worth the application programmer’s time to consider memory access optimizations.
### Table 2: Effect of memory optimizations on matrix multiplication (time in seconds)

<table>
<thead>
<tr>
<th></th>
<th>Matrix Size</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1024</td>
<td>2048</td>
<td>3072</td>
<td>4096</td>
</tr>
<tr>
<td>MP-1</td>
<td>Base</td>
<td>3.5</td>
<td>24.8</td>
<td>80.1</td>
<td>185.6</td>
</tr>
<tr>
<td></td>
<td>Pipelined</td>
<td>3.0</td>
<td>20.4</td>
<td>65.3</td>
<td>150.3</td>
</tr>
<tr>
<td></td>
<td>Pipelined &amp; Blocked</td>
<td>2.6</td>
<td>18.8</td>
<td>60.8</td>
<td>141.2</td>
</tr>
<tr>
<td>MP-2</td>
<td>Base</td>
<td>3.7</td>
<td>28.3</td>
<td>94.0</td>
<td>220.9</td>
</tr>
<tr>
<td></td>
<td>Pipelined</td>
<td>2.3</td>
<td>16.6</td>
<td>54.5</td>
<td>127.3</td>
</tr>
<tr>
<td></td>
<td>Pipelined &amp; Blocked</td>
<td>1.9</td>
<td>14.6</td>
<td>48.6</td>
<td>114.2</td>
</tr>
</tbody>
</table>

### Table 3: Effect of memory optimizations on LU decomposition (time in seconds)

<table>
<thead>
<tr>
<th></th>
<th>Matrix Size</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1024</td>
<td>2048</td>
<td>3072</td>
<td>4096</td>
<td>5120</td>
</tr>
<tr>
<td>MP-1</td>
<td>Base</td>
<td>3.9</td>
<td>22.9</td>
<td>69.4</td>
<td>155.9</td>
<td>294.8</td>
</tr>
<tr>
<td></td>
<td>Pipelined</td>
<td>3.0</td>
<td>16.9</td>
<td>50.1</td>
<td>111.2</td>
<td>208.9</td>
</tr>
<tr>
<td></td>
<td>Library</td>
<td>3.5</td>
<td>18.1</td>
<td>52.2</td>
<td>114.2</td>
<td>212.5</td>
</tr>
<tr>
<td>MP-2</td>
<td>Base</td>
<td>4.2</td>
<td>27.1</td>
<td>85.3</td>
<td>195.3</td>
<td>372.9</td>
</tr>
<tr>
<td></td>
<td>Pipelined</td>
<td>2.9</td>
<td>16.0</td>
<td>48.9</td>
<td>110.2</td>
<td>208.8</td>
</tr>
<tr>
<td></td>
<td>Library</td>
<td>3.5</td>
<td>19.2</td>
<td>56.6</td>
<td>124.9</td>
<td>233.4</td>
</tr>
</tbody>
</table>
5 THE BOUNDARY ELEMENT METHOD (BEM)

The boundary element method (BEM), in 2 dimensions, can be used to solve the following class of problems:

Given:

- a region $D$ with boundary $S$, and a function $\Phi (x, y)$ defined on $D$.
- a statement about the 2nd order gradient of $\Phi$, such as $\nabla^2 \Phi = 0$
- the value of $\Phi$ on some portion of the boundary $S_D$ (the Dirichlet boundary)
- the value of the quantity $\frac{\partial \Phi}{\partial n}$ on the other portion of the boundary $S_N$ (the Neuman boundary) where $n$ is the outward normal at the boundary point

Find:

- $\Phi$ at any point inside the boundary

5.1 BEM in One Dimension

The method can be best understood by first working through a one-dimensional problem. Consider a piece of string fastened at two points on the x-axis, one with $x = 0$ and the other with $x = l$, as shown in Figure 9.

The string is acted on by an upward force. The function $f(x)$ specifies the force per unit length at each $x$, and is piecewise continuous. The problem is to find the displacement, $u(x)$ for any $x$ between 0 and $l$. Consider a segment of the string between $x_1$ and $x_2$. Let $S_1$ be the force acting along the string at $x_1$ toward the $x = 0$ end. Let $S_2$ be the force acting along the string at $x_2$ toward the $x = l$ end. Then the $x$ component of $S_1$ is $S_1 \cos \Theta_1$ and the $x$ component of $S_2$ is $S_2 \cos \Theta_2$. We assume horizontal equilibrium, since the string is fastened at both ends, so $S_2 \cos \Theta_2 = S_2 \cos \Theta_2 = T$ (the tension in the string). The vertical component of $S_1$ is
Figure 9: A string displaced by an upward pressure $f(x)$. 

$-S_1 \sin \Theta_1$ and the vertical component of $S_2$ is $S_2 \sin \Theta_2$. The total force exerted on the string between $x_1$ and $x_2$ is

$$\int_{x_1}^{x_2} f(x) \, dx$$

(35)

For the same reason, we assume vertical equilibrium. Thus,

$$-S_1 \sin \Theta_1 + S_2 \sin \Theta_2 + \int_{x_1}^{x_2} f(x) \, dx = 0$$

(36)

and substituting $T$ in the first two terms gives

$$-T \tan \Theta_1 + T \tan \Theta_2 + \int_{x_1}^{x_2} f(x) \, dx = 0$$

(37)

Since $\tan \Theta_1 = \left. \frac{du}{dx} \right|_{x=x_1}$ and $\tan \Theta_2 = \left. \frac{du}{dx} \right|_{x=x_2}$,
Now by the Mean Value Theorem, there exists a \( \zeta \) such that

\[
(39) \quad -T \frac{du}{dx}_{x=x_1} + T \frac{du}{dx}_{x=x_2} = -(x_2 - x_1)f(\zeta)
\]

Rearranging,

\[
(40) \quad \frac{-T \left( \frac{du}{dx}_{x=x_2} - \frac{du}{dx}_{x=x_1} \right)}{x_2 - x_1} = f(\zeta)
\]

which by the definition of differentiation as \( x_2 - x_1 \) approaches 0 gives

\[
(41) \quad -T \frac{d^2u}{dx^2}_{x=\zeta} = f(\zeta)
\]

### 5.1.1 Solving \( u(x) \) for a given \( f(x) \) and homogeneous b.c.'s

The above differential equation, together with the boundary conditions, can be used to find \( u(x) \) given any piecewise continuous \( f(x) \). Consider the following example illustrated in Figure 10 and described by equation (42).

\[
(42) \quad -T \frac{d^2u}{dx^2} = f(x) \quad \begin{cases} 
0 & 0 \leq x < x_0 - \frac{\epsilon}{2} \\
p & x_0 - \frac{\epsilon}{2} \leq x \leq x_0 + \frac{\epsilon}{2}, \ u = 0 \text{ at } x = 0 \text{ and } x = l \\
0 & x_0 + \frac{\epsilon}{2} < x \leq l
\end{cases}
\]
Integrating for the first continuous piece, we get \( u = Ax + B \). Since \( u = 0 \) at \( x = 0 \), we conclude \( B = 0 \) and \( u = Ax \). On the \( f(x) = p \) continuous piece, integration produces

\[
u = Cx + D - \left( \frac{p}{2T} \right) x^2
\]

Finally, on the last continuous piece, \( u = Ex + F \), and since \( u = 0 \) at \( x = l \), \( 0 = El + F \), which implies \( u = E(x - l) \).

The problem is one of finding the unknowns \( A \), \( C \), \( D \), and \( E \). Since \( u \) and \( du/dx \) are both continuous (because \( f \) is piecewise continuous), two equations can be found at each point \( x_0 - \frac{\varepsilon}{2} \) and \( x_0 + \frac{\varepsilon}{2} \), as follows:

\[
A \left( x_0 - \frac{\varepsilon}{2} \right) = C \left( x_0 - \frac{\varepsilon}{2} \right) + D - \frac{p}{2T} \left( x_0 - \frac{\varepsilon}{2} \right)^2
\]

\[
A = C - \frac{p}{T} \left( x_0 - \frac{\varepsilon}{2} \right)
\]

(43)

\[
C \left( x_0 + \frac{\varepsilon}{2} \right) + D - \frac{p}{2T} \left( x_0 + \frac{\varepsilon}{2} \right)^2 = E \left( x_0 + \frac{\varepsilon}{2} - l \right)
\]

\[
C - \frac{p}{T} \left( x_0 + \frac{\varepsilon}{2} \right) = E
\]
Solving this system of equations for the unknowns and substituting them into the expressions for \( u(x) \) above leads to equation (44).

\[
(44) \quad u(x) = \frac{p \varepsilon}{T} \begin{cases} 
\frac{l-x_0}{l}x & 0 \leq x < x_0 - \frac{\varepsilon}{2} \\
\frac{x^2}{2\varepsilon} + \left( \frac{x_0}{\varepsilon} - \frac{x_0}{l} + \frac{1}{2} \right)x - \frac{1}{2\varepsilon} (x_0 - \frac{\varepsilon}{2})^2 & x_0 - \frac{\varepsilon}{2} \leq x \leq x_0 + \frac{\varepsilon}{2} \\
\frac{l-x}{l}x_0 & x_0 + \frac{\varepsilon}{2} < x \leq l
\end{cases}
\]

5.1.2 Solving \( u(x) \) for any \( f(x) \) and homogeneous b.c.'s

By introducing the concept of an "accessory problem" it is possible to find \( u(x) \) for any \( f(x) \) after solving a single system of equations. The accessory problem to solve is to find \( g \),

\[
g(0) = 0, \quad g(l) = 0, \quad \text{and} \quad -\frac{d^2 g}{dx^2} = \delta(x-x_0)
\]

where \( \delta \) is the Dirac delta function, a spike of infinite height and 0 width centered at \( x = x_0 \). The function \( \delta \) can be viewed as the limit of a family of functions as \( \varepsilon \rightarrow 0 \), as illustrated in Figure 11.

This definition of the Dirac delta function implies

\[
(45) \quad \int_a^b \delta(x-x_0) = 0 \quad \text{if} \ x_0 \ \text{is outside} \ (a, b)
\]

and

\[
(46) \quad \int_a^b \delta(x-x_0) = 1 \quad \text{if} \ x_0 \ \text{is inside} \ (a, b)
\]

This leads to an important sifting property of the Dirac delta function, namely
Figure 11: Dirac delta as a family of functions as $\varepsilon \to 0$

\[ (47) \quad \int_{a}^{b} \Phi(x) \delta(x-x_0) \, dx = 0 \]

if $x_0$ is outside $(a, b)$ and

\[ (48) \quad \int_{a}^{b} \Phi(x) \delta(x-x_0) \, dx = \int_{x_0-\frac{\varepsilon}{2}}^{x_0+\frac{\varepsilon}{2}} \Phi(x) \frac{1}{\varepsilon} \, dx = \int_{x_0-\frac{\varepsilon}{2}}^{x_0+\frac{\varepsilon}{2}} \Phi(x) \frac{1}{\varepsilon} \, dx = \Phi(x_0) \]

if $x_0$ is inside $(a, b)$.

The solution function $g(x|x_0)$ is called the Green's function, and can be thought of as the deflection at $x$ due to a unit force at $x_0$. Integrating the differential equation as in the example above, $g(x|x_0) = Ax + B$ when $0 \leq x \leq x_0$, and $g(x|x_0) = Cx + D$ when $x_0 < x \leq l$. Since $g(x|x_0) = 0$ at $x = 0$ and $x = l$, $B = 0$ and $D = -Cl$. Since $g(x|x_0)$ must be continuous at $x_0$, $Ax_0 = C(x_0 - l)$. From this, it follows that
(49) \[ g(x|x_0) = \begin{cases} \frac{Ax}{l-x} & 0 \leq x < x_0 \\ \frac{Ax_0}{l-x_0} & x_0 < x \leq l \end{cases} \]

By the same reasoning used to derive equation (38) but with a unit upward force at \( x_0 \) replacing the integral of \( f \),

\[ \frac{dg}{dx} \bigg|_{x=x_0^+} - \frac{dg}{dx} \bigg|_{x=x_0^-} = -\frac{1}{T} \]

where \( T \) is the horizontal component of tension. This is a special case of the \textit{jump property} of the Dirac delta function. From this can be determined

\[ \frac{Ax_0}{x_0-l} - A = \frac{1}{T} \]

and

\[ A = \frac{l-x_0}{lT} \]

So Green’s function is

\[ (53) \quad g(x|x_0) = \begin{cases} \frac{l-x_0}{lT} x & 0 \leq x < x_0 \\ \frac{l-x}{lT} x_0 & x_0 < x \leq l \end{cases} \]

Notice the similarity between the pressure functions shown in Figures 10 and 11. If \( p = l/\epsilon \) they are the same in the limit as \( \epsilon \to 0 \). Also, substituting \( p = l/\epsilon \) in equation (44) yields equation (53). This suggests a physical interpretation for the mathematical concept of the Dirac delta function.
As explained in [1], the superposition principle makes it possible to use Green's function above to solve the deflection problem for arbitrary piecewise continuous \( f(x) \), with \( f(0) = 0 \) and \( f(l) = 0 \). According to the superposition principle, if \( u_1(x) \) and \( u_2(x) \) are deflections corresponding to \( f_1(x) \) and \( f_2(x) \) respectively, and \( c_1 \) and \( c_2 \) are arbitrary constants, then \( c_1u_1(x) + c_2u_2(x) \) is the deflection corresponding to \( c_1f_1(x) + c_2f_2(x) \).

If the interval \((0, l)\) is divided into \( n \) subintervals \( \xi_1 \) to \( \xi_n \), each of length \( \Delta \xi = l/n \), then the deflection due to a piecewise continuous pressure \( f(x) \) can be approximated by the deflection due to the small concentrated forces \( f(\xi_1) \Delta \xi, \ldots, f(\xi_n) \Delta \xi \) located at \( \xi_1, \ldots, \xi_n \), respectively. Given that \( g(x|\xi_k) \) is the deflection due to the pressure function \( \delta(x - \xi_k) \) and extending the superposition principle to allow concentrated forces, we can conclude that the deflection due to the force \( f(\xi_k) \Delta \xi \) at \( \xi_k \) is equal to \( g(x|\xi_k)f(\xi_k) \Delta \xi \). Then the deflection due to all of the \( f(\xi_k) \Delta \xi \) loads is \( \sum_{k=1}^{n} g(x|\xi_k)f(\xi_k) \Delta \xi \). Letting \( \Delta \xi \) approach zero gives

\[
54) \quad u(x) = \int_{0}^{l} g(x|\xi) f(\xi) d\xi
\]

5.1.3 Solving \( u(x) \) for any \( f(x) \) with fixed-type inhomogeneous b.c.'s

A formula for the inhomogeneous system

\[
55) \quad -T^2 \frac{d^2 u}{dx^2} = f(x) \quad u = a \text{ at } x = 0, \quad u = b \text{ at } x = l
\]

can also be found using the Green's function, by the following derivation. Multiplying by \( g \) the differential equation for \( u \), and multiplying by \( u \) the differential equation for \( g \), subtracting the second from the first, and integrating gives
The term on the left can be integrated by parts, and the second term on the right is just \( u(\xi) \) by the sifting property, so

\[
(57) \quad u(\xi) = \int_0^l f(x) g(x, \xi) \, dx + T u'(\xi)
\]

This is called the boundary integral equation. So far in this derivation no boundary conditions for \( g \) have been stated. If they are set to 0 at \( x = 0 \) and \( x = l \), as above, then \( g \) is as defined in equation (53), and

\[
(58) \quad u(\xi) = \int_0^l f(x) g(x, \xi) \, dx + \frac{l - \xi}{l} a + \frac{\xi}{l} b
\]

By choosing appropriate boundary conditions on the differential equation for \( g \), we can construct accessory problems that will allow solving for \( u(x) \) when \( u(0) \) is known and \( du/dx \) at \( x = l \) is known, or vice versa. For example, if \( u(x) = a \) and \( du/dx \) at \( x = l \) is equal to \( b \), then choose an accessory function \( g \) with boundary conditions \( g(0) = 0 \) and \( dg/dx = 0 \) at \( x = l \). Then

\[
(59) \quad u(x) = \int_0^l f(\xi) g(x, \xi) \, d\xi + T b g(l) + T a \left. \frac{dg}{dx} \right|_{x=0}
\]

Note that the function \( g \) is different for each of the three possible types of boundary conditions. A unique solution cannot be found if only \( du/dx \) is known at both 0 and \( l \).
5.1.4 Solving $u(x)$ for any $f(x)$ and any b.c.'s

It is possible to solve one accessory problem, called the fundamental solution, that will allow solutions for $u(x)$ given any of the three combinations of boundary conditions mentioned above. This time a function $G$ will be defined so that

$$\frac{d^2 G}{dx^2} = \delta(x - \xi)$$

Integrating produces $G = Ax + B$ for $x < \xi$ and $G = Cx + D$ for $x > \xi$. $G$ is continuous at $x = \xi$, so

$$A\xi + B = C\xi + D$$

Using the jump property of the Dirac delta function at $x = \xi$ gives

$$\left. \frac{dG}{dx} \right|_{\xi^+} - \left. \frac{dG}{dx} \right|_{\xi^-} = 1$$

so $A - C = 1$. Substituting $A = C - l$ in equation (61) gives $B = \xi + D$. In terms of $A$ and $D$, 

$$G = \begin{cases} Ax + \xi + D & x < \xi \\ Ax + x + D & x > \xi \end{cases}$$

Assume as boundary conditions that $G(x) = 0$ at $x = \xi - l$ and $x = \xi + l$ for some arbitrary constant $l$, as illustrated in Figure 12.

Plugging the values at these two points into equation (63) and subtracting the first from the second leads to $2Al + l = 0$, and $A = -1/2$. Also, from the first equation, $D = -A\xi + Al - \xi$ and $D = -\xi/2 - l/2$. In terms of the arbitrary $l$, $G$ and $G' = F$ can be written as

$$g(x, \xi) = -\frac{l}{2} + \frac{1}{2}|\xi - x|$$

and
Figure 12: Graph of the fundamental solution, $G$

(65) $F(x, \xi) = \frac{d}{dx} G(x, \xi) = \begin{cases} -\frac{1}{2} & x < \xi \\ \frac{1}{2} & x > \xi \end{cases}$

Now to solve $u(x)$ given $u'' = f(x)$ with boundary conditions at 0 and $L$, use the same trick as in equation (56) to get

(66) $\int_0^L (gu'' - ug'') \, dx = \int_0^L f(x) g(x, \xi) \, dx - \int_0^L \delta(x - \xi) u(x) \, dx$

Letting $q(x) = du/dx$, this gives

(67) $G(x, \xi) q(x) \bigg|_0^L - u(x) F(x, \xi) \bigg|_0^L + u(\xi) = \int_0^L G(x, \xi) f(x) \, dx$

and
\[ u(\xi) = G(L, \xi)q(L) - G(0, \xi)q(0) - F(L, \xi)u(L) \]
\[ + \int_0^L G(x, \xi)f(x) \, dx \]
\[ (68) \]

Since \( f, F, \) and \( G \) are known functions, \( u(\xi) \) can be found if \( u \) and \( q \) are both known at both 0 and \( L \). However, in a boundary value problem, only one of \( u \) and \( q \) is given at each boundary point. To find the two unknowns, the most obvious approach is to form two equations by plugging \( \xi = 0 \) and \( \xi = L \) into equation (68). However, \( F(0, 0) \) and \( F(L, L) \) are undefined. Instead, use values a distance of \( \varepsilon \) from each endpoint to form the system of equations

\[
\begin{bmatrix}
  u(\varepsilon) \\
  u(L - \varepsilon)
\end{bmatrix}
+ \begin{bmatrix}
  -G(0, \varepsilon) & G(L, \varepsilon) \\
  -G(0, L - \varepsilon) & G(L, L - \varepsilon)
\end{bmatrix}
\begin{bmatrix}
  q(0) \\
  q(L)
\end{bmatrix}
- \begin{bmatrix}
  -F(0, \varepsilon) & F(L, \varepsilon) \\
  -F(0, L - \varepsilon) & F(L, L - \varepsilon)
\end{bmatrix}
\begin{bmatrix}
  u(0) \\
  u(L)
\end{bmatrix}
= \begin{bmatrix}
  f_1 \\
  f_2
\end{bmatrix}
\]
\[ (69) \]

where the following abbreviations are defined:

\[ f_1 = \int_0^L G(x, \varepsilon)f(x) \, dx \quad \text{and} \quad f_2 = \int_0^L G(x, L - \varepsilon)f(x) \, dx \]

Using the definitions for \( F \) and \( G \) given in equations (64) and (65),

\[ G(0, \varepsilon) = -\frac{1}{2} + \frac{\varepsilon}{2} \quad G(L, \varepsilon) = -\frac{1}{2} + \frac{1}{2}(L - \varepsilon) \]
\[ G(L, L - \varepsilon) = -\frac{1}{2} + \frac{\varepsilon}{2} \quad G(0, L - \varepsilon) = -\frac{1}{2} + \frac{1}{2}(L - \varepsilon) \]
\[ F(0, \varepsilon) = -\frac{1}{2} \quad F(L, \varepsilon) = \frac{1}{2} \]
\[ F(L, L - \varepsilon) = -\frac{1}{2} \quad F(0, L - \varepsilon) = \frac{1}{2} \]
\[ (71) \]

Taking the limit as \( \varepsilon \to 0 \) results in the system of equations
Since \( l \) was chosen arbitrarily, it is convenient here to set \( l = L \) to get the system of equations

\[
\begin{bmatrix} u(0) \\ u(L) \end{bmatrix} + \begin{bmatrix} \frac{l}{2} & -\frac{l}{2} + \frac{L}{2} \\ \frac{L}{2} & -\frac{l}{2} \end{bmatrix} \begin{bmatrix} q(0) \\ q(L) \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} u(0) \\ u(L) \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}
\]

which simplifies to

\[
\begin{bmatrix} \frac{L}{2} & 0 \\ 0 & -\frac{L}{2} \end{bmatrix} \begin{bmatrix} q(0) \\ q(L) \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} u(0) \\ u(L) \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}
\]

Now if a solution exists given a set of boundary conditions, it can be found by plugging the knowns into the system above, rearranging, and solving for the unknowns. In particular, the system has a unique solution if \( u \) is known at one or both \( 0 \) and \( L \), but results in a singular matrix (no unique solution) if only \( q \) is known at each \( 0 \) and \( L \). This makes sense because two functions of the same shape, differing by a constant, could have the same \( du/dx \) at \( 0 \) and \( L \).

As an example, the following system is obtained when \( u(0) = a \) and \( q(L) = b \):

\[
\begin{bmatrix} \frac{L}{2} & 0 \\ 0 & -\frac{L}{2} \end{bmatrix} \begin{bmatrix} q(0) \\ b \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} a \\ u(L) \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}
\]

which can be rearranged to form
A typical two-dimensional boundary value problem is illustrated in Figure 13. A function \( \Phi(x, y) \) is defined on the region \( D \). \( \Phi \) is known on the portion of the boundary \( S_D \), \( q = \frac{\partial \Phi}{\partial n} \) is known on the portion of the boundary \( S_N \). Laplace’s equation \( \nabla^2 \Phi = 0 \) holds in \( D \) and on the boundary. The object is to find \( \Phi \) at any point in \( D \).

The boundary element method in two dimensions depends on the *Gauss Divergence Theorem*, which says

\[
\int_{\Omega} \nabla \cdot \mathbf{A} \, d\Omega = \int_{\Lambda} (\mathbf{A} \cdot \mathbf{n}) \, d\Lambda
\]
where \( \hat{A} \) is a vector field over the area \( \Omega \), \( \Lambda \) is the boundary of \( \Omega \), and \( \hat{n} \) is the outward-pointing unit normal vector to \( \Lambda \). Informally, the correctness of Gauss's Theorem is illustrated in Figure 14.

![Figure 14: Illustration of Gauss's Theorem](image)

Consider the point \((x, y)\), with a square \(PQRS\) around it. For small \( \Delta x \) and \( \Delta y \), and given \( \hat{A}(x, y) \), the value of a vector field at \((x, y)\), the value of \( \hat{A} \) can be approximated on the square as

\[
\begin{align*}
\text{On } PQ: \hat{A} &= \left[ \hat{A}_x - \frac{\partial A_x}{\partial y} \frac{\Delta y}{2} \right] \hat{i} + \left[ \hat{A}_y - \frac{\partial A_y}{\partial y} \frac{\Delta y}{2} \right] \hat{j} \\
\text{On } QR: \hat{A} &= \left[ \hat{A}_x + \frac{\partial A_x}{\partial x} \frac{\Delta x}{2} \right] \hat{i} + \left[ \hat{A}_y + \frac{\partial A_y}{\partial x} \frac{\Delta x}{2} \right] \hat{j} \\
\text{On } RS: \hat{A} &= \left[ \hat{A}_x + \frac{\partial A_x}{\partial y} \frac{\Delta y}{2} \right] \hat{i} + \left[ \hat{A}_y + \frac{\partial A_y}{\partial y} \frac{\Delta y}{2} \right] \hat{j} \\
\text{On } SP: \hat{A} &= \left[ \hat{A}_x - \frac{\partial A_x}{\partial x} \frac{\Delta x}{2} \right] \hat{i} + \left[ \hat{A}_y - \frac{\partial A_y}{\partial x} \frac{\Delta x}{2} \right] \hat{j}
\end{align*}
\]

The unit normal vectors on \( QR, RS, SP \), and \( PQ \) are just \( \hat{i}, \hat{j}, -\hat{i}, \) and \(-\hat{j}\), respectively. Therefore,
\[
\int_A \hat{A} \cdot \hat{n} d\Lambda = -\left[ \hat{A}_y - \frac{\partial A_y}{\partial y} \frac{\Delta y}{2} \right] \Delta x + \left[ \hat{A}_x + \frac{\partial A_x}{\partial x} \frac{\Delta x}{2} \right] \Delta y \\
+ \left[ \hat{A}_y - \frac{\partial A_y}{\partial y} \frac{\Delta y}{2} \right] \Delta x - \left[ \hat{A}_x + \frac{\partial A_x}{\partial x} \frac{\Delta x}{2} \right] \Delta y \\
= \left( \frac{\partial \hat{A}_x}{\partial x} + \frac{\partial \hat{A}_y}{\partial y} \right) \Delta x \Delta y \\
= \int \nabla \cdot \hat{A} d\Omega
\]

(79)

The next step is to use Gauss's Theorem to derive a formula for \( \Phi \) on \( D \) in terms of the variables on the boundary, as well as a method for computing the unknown quantities on the boundary, given the known quantities. First, let \( G \) be a function of \( x \) and \( y \). As the name suggests, \( G \) will be used in a way similar to the Green's function in one dimension.

Without specifying \( G \) further at this time, consider the integral

\[
\int_D (G \nabla^2 \Phi - \Phi \nabla^2 G) \, dD
\]

which is reminiscent of the left hand sides of the one-dimensional equations (56) and (66). Using the definition of \( \nabla \), the following transformations are possible:

\[
\int_D (G \nabla^2 \Phi - \Phi \nabla^2 G) \, dD \\
= \int_D (G \nabla \cdot \nabla \Phi - \Phi \nabla \cdot \nabla G) \, dD \\
= \int_D \{ [ \nabla \cdot (G \nabla \Phi) - \nabla G \cdot \nabla \Phi] - [ \nabla \cdot (\Phi \nabla G) - \nabla \Phi \cdot \nabla G] \} \, dD \\
= \int_D \{ [ \nabla \cdot (G \nabla \Phi) - \nabla \cdot (\Phi \nabla G) ] \, dD \\
= \int_D \nabla \cdot (G \nabla \Phi - \Phi \nabla G) \, dD \\
= \int_S (G \nabla \Phi - \Phi \nabla G) \cdot \hat{n} dS \quad \text{(Using Gauss’s Theorem)}
\]

(81)
Now, since \( \nabla \Phi \cdot \hat{n} = \frac{\partial \Phi}{\partial n} \),

\[
(82) \quad \int_{\partial} (G \nabla^2 \Phi - \Phi \nabla^2 G) \, dD = \int_{S} (G \frac{\partial \Phi}{\partial n} - \Phi \frac{\partial G}{\partial \hat{n}}) \, dS
\]

This equation is known as Green's 2nd identity. As mentioned above, the function \( G \) is to fill a role similar to that of the function \( g \) in the one-dimensional case. Define \( G \) in terms of \( \bar{P} (x_p, y_p) \) and \( \bar{Q} (x, y) \) as

\[
(83) \quad G (\bar{P}, \bar{Q}) = \ln |\bar{P} - \bar{Q}| = \ln \sqrt{(x - x_p)^2 + (y - y_p)^2}
\]

Where \( \bar{P} \) is a fixed point analogous to \( x_0 \) in the definition of the one-dimensional Green's function. Like that function, \( G \) is not differentiable at \( \bar{P} = \bar{Q} \). For \( \bar{P} \neq \bar{Q} \), it is possible to differentiate \( G \) to obtain

\[
\frac{\partial G}{\partial x} = \frac{x - x_p}{(x - x_p)^2 + (y - y_p)^2}
\]

\[
\frac{\partial G}{\partial y} = \frac{y - y_p}{(x - x_p)^2 + (y - y_p)^2}
\]

\[
(84) \quad \frac{\partial^2 G}{\partial x^2} = \frac{1}{(x - x_p)^2 + (y - y_p)^2} - \frac{2 (x - x_p)^2}{[(x - x_p)^2 + (y - y_p)^2]^2}
\]

\[
\frac{\partial^2 G}{\partial y^2} = \frac{1}{(x - x_p)^2 + (y - y_p)^2} - \frac{2 (y - y_p)^2}{[(x - x_p)^2 + (y - y_p)^2]^2}
\]

\[
\nabla^2 G = \frac{\partial^2 G}{\partial y^2} + \frac{\partial^2 G}{\partial x^2} = \frac{2}{(x - x_p)^2 + (y - y_p)^2} - \frac{2}{(x - x_p)^2 + (y - y_p)^2} = 0
\]
Since $\nabla^2 \Phi = 0$ and $\nabla^2 G = 0$,

$$\int_D \left[ \nabla^2 (P, \bar{Q}) \frac{\partial \Phi}{\partial n} - \Phi \left( \frac{\partial \Phi}{\partial n} \right) \nabla^2 G (P, \bar{Q}) \right] dD (\bar{Q}) = 0$$

so by equation (82)

$$\int_S \left[ \nabla^2 (P, \bar{Q}) \frac{\partial \Phi}{\partial n} - \Phi \left( \frac{\partial \Phi}{\partial n} \right) \nabla^2 G (P, \bar{Q}) \right] dS (\bar{Q}) = 0$$

From now on, $G$ will be written as a shorthand for $G (P, \bar{Q})$, and $G'$ will be written as a shorthand for $\frac{\partial G}{\partial n} (P, \bar{Q})$. Also, inside integrals, $\Phi$ and $\Phi'$ will stand for $\Phi (\bar{Q})$ and $\frac{\partial \Phi}{\partial n} (\bar{Q})$ respectively, where $\bar{Q}$ is the variable of integration.

Now equation (86) can be used to derive a formula for $\Phi$ in terms of $G$, $\Phi$, and $\frac{\partial \Phi}{\partial n}$ on the boundary. First, consider a point $\bar{P}$ on the area $D$. Notice that the integral in equation (85) cannot be evaluated, since $G$ is not defined for $P = \bar{Q}$. However, the integral can be evaluated on a very similar region $D - D_\epsilon$, illustrated in Figure 15.

Here, the segments $\overline{AB}$ and $\overline{BA}$ are drawn separately for clarity, but are really the same line segment. As $\epsilon$ approaches 0, $D - D_\epsilon$ becomes more similar to $D$. Since now $\bar{P}$ is not included in the region, the integral over the region is defined, and the related boundary integral can be equated with zero to get

$$\int_{S_D + S_N + \overline{BA} + S_e + \overline{AB}} [G \Phi' - \Phi G'] dS = 0$$

The arrows in the figure show the “direction” of the integration. The unit normal is always to the right. Therefore, the pieces of the integration over $\overline{AB}$ and $\overline{BA}$ cancel, so
Figure 15: A boundary value problem

\[ \int_{S_D + S_N} [G\Phi' - \Phi G'] \, dS = -\int_{S_\epsilon} [G\Phi' - \Phi G'] \, dS \]

(88)

Considering just the right-hand side, and using the definition of $G$, the integral can be simplified as follows:

\[ -\int_{S_\epsilon} [G\Phi' - \Phi G'] \, dS \]

\[ = -\int_{S_\epsilon} \left[ \ln \epsilon \Phi' - \Phi \frac{1}{\epsilon} \right] \, dS \]

(89)

\[ = -\int_0^{2\pi} \left[ \ln \epsilon \Phi' - \Phi \frac{1}{\epsilon} \right] \epsilon \, d\Theta \]

\[ = -\epsilon \ln \epsilon \int_0^{2\pi} \Phi' \, d\Theta - \int_0^{2\pi} \Phi \, d\Theta \]

Taking the limit as $\epsilon \to 0$, the first term becomes 0 and second becomes $-2\pi \Phi (\hat{P})$. Therefore, on the region $D$, 

\[ \int_{S_{\epsilon}} \Phi \, d\Theta \]
\begin{equation}
2\pi\Phi (\hat{P}) = \int_{S_D + S_N} [\Phi G' - G\Phi'] dS
\end{equation}

This is the two-dimensional boundary integral equation.

Deriving the boundary integral equation for a point on a smooth portion of the boundary $S$ can be accomplished using a similar construction. Here the boundary integral is the one which cannot be evaluated when $\hat{P} = \hat{Q}$, so a similar region, not containing that point is constructed (Figure 16).

Using Gauss's theorem and the fact that the domain integral equals zero as before,

\begin{equation}
\int_{S_D + S_N - \beta + S_\epsilon} [G\Phi' - \Phi G'] dS = 0
\end{equation}

The details of the simplification of the $S_\epsilon$ term are the same, except the integral as $\epsilon$ approaches 0 is over a half circle, so

\begin{equation}
\pi\Phi (\hat{P}) = \int_{S_D + S_N} [\Phi G' - G\Phi'] dS
\end{equation}

Finally, at a point $\hat{P}$ on $S$ where there is a singularity, $D - D_\epsilon$ is constructed as illustrated in Figure 17, where $\omega$ is the angle made by the tangents to the boundary on either side of $\hat{P}$. 

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Figure16.png}
\caption{A boundary value problem}
\end{figure}
Figure 17: A boundary value problem

Again, the simplification is similar, and for this kind of point

\begin{equation}
\omega \Phi (\hat{P}) = \int_{S_D + S_N} [\Phi G' - G \Phi'] \, dS
\end{equation}

The three versions of the boundary integral equation can be combined into a compact form as in equation (94), where \( \omega \) is the interior angle between two tangents drawn on either side of \( \hat{P} \):

\begin{equation}
C (\hat{P}) \Phi (\hat{P}) = \int_{S_D + S_N} [\Phi G' - G \Phi'] \, dS
\end{equation}

\begin{equation}
C (\hat{P}) = \begin{cases}
2\pi & \text{if } \hat{P} \text{ on } D \\
\omega & \text{if } \hat{P} \text{ on } S_D + S_N
\end{cases}
\end{equation}

The value of \( \omega \) for any given \( \hat{P} \) can be found quite easily by considering the case where the function \( \Phi \) is a constant 1. This means

\begin{equation}
\omega = \int_{S_D + S_N} G' \, dS
\end{equation}

Since the value of \( \Phi \) in no way affects the geometry of the problem, this formula for \( \omega \) holds in general.
With equations (94) and (95) in hand, it is possible to use the boundary element method to solve the Laplace problem in two dimensions. The boundary $S$ is assumed to be discretized into $N$ geometric points, with either $\Phi$ or $\frac{\partial \Phi}{\partial n}$ known at each point. To find the unknowns on the boundary, $\hat{P}$ is held fixed at each point, and numerical integration is used to produce an equation. By collocating at each point, a system of $N$ equations in $N$ unknowns is generated. This system can be solved in whatever way is convenient. Once all the values on the boundary are known, calculating $\Phi$ at any point in $D$ is a simple integration.

5.2.1 Numerical Implementation

The implementation of the boundary element method considered in this research assumes the boundary is broken up into elements, each associated with a number of nodes, which for now can be considered geometric points. The number of nodes per element determines the order of the symbolic integration performed over each element: two nodes for linear, three nodes for quadratic, etc. Figure 18 shows a schematic of the discretization of a boundary value problem.

![Figure 18: Discretization of a boundary value problem](image-url)
The "real" geometry is shown in dashed lines. The elements of the discretization are shown as dark lines between dark hashes. A node is defined at least every place two elements meet. More nodes are defined within each element if the problem is quadratic or higher order. (There are some complicating details involving double nodes at geometric singularity; these will be ignored for now.) If $NP$ is the number of elements, and $S_j$ is the $jth$ element, then the discretized version of the boundary integral equation is

$$\sum_{i=1}^{NP} \left[ \int_{S_j} (\Phi G' - \Phi' G) dS_j \right]$$

It is important to remember here that discretization means approximation of the real function $\Phi$. The integration over each element uses the concept of a shape function to approximate $\Phi$ and $\Phi'$. A local set of coordinates for an element is adopted, as illustrated in Figure 19. In this linear case, a function $f$ on the element is approximated as $f(s) = f_1 H_1(s) + f_2 H_2(s)$ where $f_i$ is the discrete value of the function at local node $i$, and $H_i$ is the $ith$ component of the shape function. In this case $H_1(s) = 1 - s/h$ and $H_2(s) = s/h$, for a linear approximation. For elements with more nodes, the method generalizes to higher orders.

Figure 19: Element local coordinates

Using this approximation with shape functions, it is possible to write the boundary integral equation as
where \( NK \) is the number of nodes per element and \( \Phi_{ij} \) is the value of the function at the \( j \)th node of the \( i \)th element. Rearranging,

\[
\sum_{i=1}^{NP} \int_{S_i} G' dS_i \Phi (\hat{P}) = \sum_{i=1}^{NP} \int_{S_i} G' \sum_{j=1}^{NK} \Phi_{ij} H_j (s) dS_i \\
- \sum_{i=1}^{NP} \int_{S_i} G' \sum_{j=1}^{NK} \Phi'_{ij} H_j (s) dS_i
\]

Now if the total number of nodes is \( N \), collocating at each point \( \hat{P} \) gives \( N \) equations in \( N \) unknowns, since only one of \( \Phi \) and \( \Phi' \) are unknown at any point. Furthermore, the integration can be performed knowing only the geometry. The resulting matrix can be LU decomposed, and used to solve multiple problems with different values for \( \Phi \) and \( \Phi' \).

### 5.2.2 Time-dependent problems

Since the focus of this research is the parallelization of the boundary element method, it is desirable to solve problems for which conventional computers take a large amount of execution time. The simple Laplace problems described above can be solved in a relatively short time. Time-dependent problems, such as heat transfer, are more computationally challenging. The heat equation can be written as

\[
(99) \quad \nabla^2 \Phi = f
\]
where \( f = \frac{\partial \Phi}{\partial t} \). Then the boundary integral equation is

\[
(100) C \left( \tilde{P} \right) \Phi_i \left( \tilde{P} \right) = \int_{S_B + S_N} \left[ \Phi^i G' - G \Phi'^i \right] dS + \int_D f G dD
\]

where \( \Phi_i \) is the temperature at time \( t \). Since \( a \frac{\partial \Phi}{\partial t} = a \left( \frac{\Phi_t - \Phi_{t-1}}{\Delta t} \right) \), the boundary integral equation can be restated as

\[
(101) C \left( \tilde{P} \right) \Phi_i \left( \tilde{P} \right) = \int_{S_B + S_N} \left[ \Phi^i G' - G \Phi'^i \right] dS + \frac{a}{\Delta t} \int_D G \Psi_i dD - \frac{a}{\Delta t} \int_D G \Psi_i dD
\]

where \( \Psi_i \) is the temperature on the domain at time \( t \). Because of the integral over \( D \), the domain must be discretized as well as the boundary. As in the time-independent case, the integration to form the system of equations is done only once, but collocation and integration is over both boundary and domain nodes. Finally, the solution must be “stepped” from the initial conditions to the desired time in small increments. Equation (102) shows the discrete form (corresponding to the time-independent version from equation (98)).

\[
(102) C \left( \tilde{P} \right) \Phi_i^{(t)} \left( \tilde{P} \right) = \sum_{i=1}^{NP} \sum_{j=1}^{NK} \Phi_{ij}^{(t)} \int_{S_i} G_i H_j \left( s \right) dS_i - \sum_{i=1}^{NP} \sum_{j=1}^{NK} \Phi_{ij}^{(t)} \int_{S_i} G_i H_j \left( s \right) dS_i
\]

\[
+ \frac{a}{\Delta t} \sum_{i=1}^{NELX} \Phi_{i}^{(t)} \int_{S_i} G_i \left( s \right) dD_i - \frac{a}{\Delta t} \sum_{i=1}^{NELX} \Phi_{i}^{(t-1)} \int_{S_i} G_i \left( s \right) dD_i
\]

Let \( N \) be the sum of discrete boundary nodes and domain elements. By choosing a \( \tilde{P} \) at each boundary node and at each domain element centroid, a system of \( N \) unknowns in \( N \) equations is formed. This provides the basis for the BEM implementation discussed in the next chapter.
This chapter describes an implementation on the MasPar MP-1 and MP-2 computers of the time-dependent BEM for solving heat transfer problems, described in the last chapter. The parallel implementation was adapted from a sequential program, written in Fortran, provided by Dr. Ambar Mitra. The Fortran code can be found in Appendix 3. The language MPL (a parallel dialect of C) was chosen for this implementation because it provides flexibility for achieving better performance by giving the programmer control of data distribution to the PE array, and of the instructions executed by the processing elements. Since MPL is a parallel version of C, the first step in implementing the BEM code was to convert it from Fortran to normal serial C code. The C code can be found in Appendix 4. Finally, the sequential implementation in C was translated to MPL. The MPL code is in Appendix 5.

In Chapter 5, the underlying mathematics of the boundary element method (BEM) was discussed. Recall that the discrete integral equation for the time-dependent BEM (102) provides the basis for the numerical implementation. Figure 20 shows the high-level numerical tasks to be performed by the BEM application. First, the discrete geometry which describes the problem boundary and domain is read as input. Then, collocation and integration are performed to construct a set of linear equations that can be used to solve the heat equation over the problem domain. Several matrix algebraic manipulations are performed so that this system of equations can be solved efficiently at each time step. Finally, iteration is performed until the desired time step is reached. An iteration step involves inputting time-dependent boundary conditions and solving the system of equations. If intermediate results are desired for the time step, then some extra work is performed to compute them, and they are output. These tasks are discussed in further detail in the sections below. Finally, a strategy for distributing the BEM data elements to MasPar processing elements, and for parallelizing the numerical computations of the BEM, is presented.
6.1 BEM Geometry

To solve the integrals in equation (102), the boundary and domain of a problem geometry are discretized, as shown in Figure 21, for a simple oblong geometry. The boundary is divided into boundary elements, which contain two kinds of boundary nodes, end nodes and interior nodes. In the figure, end nodes are shown as large dots, and interior nodes as small dots. The nodes are numbered to show that they are ordered in a counter-clockwise direction around the boundary. For a given problem, the boundary elements are all either linear, quadratic, or cubic. A linear element has no interior nodes, a quadratic element one interior node, and a cubic element
two interior nodes. Each boundary element shares an end node with each of its neighboring elements. The domain (area) of the geometry is discretized into triangular domain elements. Notice that the domain discretization is independent of the boundary discretization. The following quantities are defined to refer to the discretization.

\[ NP: \] number of boundary elements
\[ NK: \] nodes per boundary element
\[ NODES: \] number of boundary nodes
\[ NELX: \] number of domain elements

![Figure 21: Boundary and Domain Discretization](image)

The C declarations in Figure 22 show explicitly the data items that make up boundary and domain nodes and elements.

Notice that there is no explicit data structure for boundary elements. Boundary element \( i \) contains boundary nodes \( i(NK-1) \) to \( (i+1)(NK-1) \).

---

1. There is an exception to this rule which will be discussed later in this chapter.
2. In the C and MPL code, the data structures are organized somewhat differently, but the same data is represented. They are presented here in this form for ease of explanation.
6.2 Collocation and Integration

By fixing a point \( \bar{P} \) at a boundary node or domain element centroid and solving the integrals from equation (102), an equation with \( NODES + NELX \) unknowns is derived. This is called collocation at \( \bar{P} \). There is one unknown, either \( \Phi_{ij}^{(t)} \) or \( \Phi'_{ij}^{(t)} \), at each boundary node in the discretization. There is another unknown \( \Psi_i^{(t)} \) for each domain element. By collocating at each boundary node and at each domain element centroid, and solving the integrals, a system of \( NODES + NELX \) equations in as many unknowns is formed.

The mathematical details of the symbolic integration scheme used in the BEM implementation were provided by Dr. Ambar Mitra. However, to understand the rest of the BEM process it is necessary to know what are the inputs and outputs of the integration process. Figure 23 shows the inputs and outputs for the boundary and domain integration processes in terms of the coefficient matrices, where \( Node (i) \) denotes the \( i \)th boundary node, \( Elem (Node (i)) \) denote the element(s) containing \( Node (i) \), \( Centroid (i) \) denotes the centroid of the \( i \)th domain ele-
Figure 23: Inputs and outputs of the integration processes

ment, and Corners \((j)\) denote the three corner nodes of the \(i\)th domain element. The outputs are matrix elements that form the BEM system of linear equations, to be explained later in this chapter.

6.3 Solving a System of Linear Equations

A system of equations \(Ax = b\) can be solved in more than one way. Inverting \(A\) allows the system to be solved for any \(b\) by performing a matrix-vector multiplication. However, it takes fewer computations to LU decompose \(A\) than to invert \(A\). Once \(A\) is LU decomposed, a solution for any \(b\) can be obtained by performing an LU solve, and LU solve does not take many more computations than a matrix-vector multiplication.

In practice, LU decomposition is almost always preferred over matrix inversion to solve a system of equations. For this reason, studying the implementation of matrix inversion is not
usually considered interesting. Special circumstances in the BEM make a hybrid approach attractive. This approach is described in general terms and its execution time analyzed below.

The analysis will be done independently of any architecture, sequential or parallel. The execution time will be estimated by counting the number of floating point operations performed. To further simplify the analysis, it will be assumed that floating point addition, subtraction, and comparison take time $T_a$ and that floating point multiplication and division take time $T_m$. The following execution time estimates for simple matrix algorithms (Table 4) will be used. The parameters are the lengths of the matrices involved.

Table 4: Execution times for matrix algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>LU decomposition:</td>
<td>$\frac{1}{3} n^3 (T_m + T_a) + n^2 T_a + \frac{5}{3} n (T_a + T_m)$</td>
</tr>
<tr>
<td>LU solve:</td>
<td>$n^2 (T_m + T_a) + nT_m$</td>
</tr>
<tr>
<td>Matrix multiply:</td>
<td>$n_1 n_2 n_3 (T_a + T_m)$</td>
</tr>
<tr>
<td>Matrix Add:</td>
<td>$n_1 n_2 T_a$</td>
</tr>
<tr>
<td>Matrix-vector multiply:</td>
<td>$n_1 n_2 (T_m + T_a)$</td>
</tr>
<tr>
<td>Vector add:</td>
<td>$nT_a$</td>
</tr>
<tr>
<td>Matrix invert:</td>
<td>$\text{LUD} (n) + n \text{LUS} (n)$</td>
</tr>
</tbody>
</table>

All of the above estimates follow from common implementations of the matrix algorithms. The inversion algorithm is based on LU decomposing the matrix, and then solving the system against the identity matrix.
Consider a system \( Ax = b \), where \( A \) is an \( N \times N \) array. The usual way to solve that system of equations for \( k \) vectors would be to LU decompose \( A \) and LU solve for each of the \( k \) vectors, for a total execution time of

\[
\frac{1}{3} N^3 (T_m + T_d) + \frac{5}{3} N (T_m + T_d) + k (N^2 (T_m + T_d) + NT_m)
\]

Now consider the same system of equations, partitioned as follows:

\[
\begin{bmatrix}
A_1 & A_2 \\
A_3 & A_4
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2
\end{bmatrix}
= 
\begin{bmatrix}
b_1 \\
b_2
\end{bmatrix}
\]

where \( A_1 \) is \( m_1 \times m_1 \), \( A_2 \) is \( m_1 \times m_2 \), \( A_3 \) is \( m_2 \times m_1 \), \( A_4 \) is \( m_2 \times m_2 \), \( x_1 \) and \( b_1 \) are \( m_1 \times 1 \), and \( x_2 \) and \( b_2 \) are \( m_2 \times 1 \), and \( n = m_1 + m_2 \). Given this partition, the original matrix equation can be written in two parts, \( A_1 x_1 + A_2 x_2 = b_1 \) and \( A_3 x_1 + A_4 x_2 = b_2 \). If \( A_1 \) is invertible, then from the first, \( x_1 \) can be written as

\[
(105) x_1 = A_1^{-1} b_1 - A_1^{-1} A_2 x_2
\]

Substituting into the second matrix equation gives

\[
(106) (-A_3 A_1^{-1} A_2 + A_4) x_2 = b_2 - A_3 A_1^{-1} b_1
\]

The setup for solving the system takes the computations shown in Table 5. Each iteration requires the computations shown in Table 6. The total cost for solving the system for \( k \) vectors is

\[
(107) 
\left( \frac{4m_1^3}{3} + \frac{m_2^3}{3} + 2m_1^2 m_2 + m_1 m_2^2 + 2m_1^2 + m_1 m_2 + \frac{5m_1}{3} + \frac{5m_2}{3} \right) (T_m + T_d)
\]

\[
+ (2m_2^2 + m_2) T_d + k \left[ (m_2^2 + m_1 m_2 + m_2) (T_m + T_d) \right]
\]
Table 5: Setup steps for solving linear system

<table>
<thead>
<tr>
<th>Equation</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1^{-1}$</td>
<td>INV ($m_1$)</td>
</tr>
<tr>
<td>$-A_3A_1^{-1}$</td>
<td>MM ($m_2, m_1, m_1$)</td>
</tr>
<tr>
<td>$(A_3A_1^{-1})A_2$</td>
<td>MM ($m_2, m_1, m_2$)</td>
</tr>
<tr>
<td>$(A_3A_1^{-1})A_2 + A_4$</td>
<td>MA ($m_2, m_2$)</td>
</tr>
<tr>
<td>LU decompose $-A_3A_1^{-1}A_2 + A_4$</td>
<td>LUD ($m_2$)</td>
</tr>
<tr>
<td>$(A_3A_1^{-1})b_1$</td>
<td>MVM ($m_2, m_1$)</td>
</tr>
<tr>
<td>$b_2 - (A_3A_1^{-1})b_1$</td>
<td>VA ($m_2$)</td>
</tr>
<tr>
<td>$A_1^{-1}b_1$</td>
<td>MVM ($m_1, m_1$)</td>
</tr>
<tr>
<td>$A_1^{-1}A_2$</td>
<td>MM ($m_1, m_1, m_2$)</td>
</tr>
</tbody>
</table>

Table 6: Per-iteration steps for solving linear system

<table>
<thead>
<tr>
<th>Equation</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LU Solve for $x_2$</td>
<td>LUS ($m_2$)</td>
</tr>
<tr>
<td>$(A_1^{-1}A_2)x_2$</td>
<td>MVM ($m_1, m_2$)</td>
</tr>
<tr>
<td>$(A_1^{-1}b_1) - (A_1^{-1}A_2x_2)$</td>
<td>VA ($m_2$)</td>
</tr>
</tbody>
</table>
If \( m_1 = m_2 = \frac{N}{2} \), then the cost for this hybrid method becomes

\[
(108) \quad \left( \frac{7}{12} N^3 + \frac{3}{4} N^2 + \frac{5}{3} N \right) (T_m + T_a) + \frac{1}{2} (N^2 + N) T_a + \frac{1}{2} k \left[ N^2 (T_m + T_a) + NT_a \right]
\]

The coefficient of the cubed term for the regular LU decomposition and solve is smaller than that for the hybrid method, so for solving the system of equations for a single vector the regular method is clearly better. However, when \( k \) becomes large the situation is quite different. When \( k = N \), the cost for solving for \( k \) vectors under the hybrid method becomes

\[
(109) \quad \left( \frac{13}{12} N^3 + \frac{3}{4} N^2 + \frac{5}{3} N \right) (T_m + T_a) + (N^2 + \frac{1}{2} N) T_a
\]

versus

\[
(110) \quad \left( \frac{4}{3} N^3 + N^2 + \frac{5}{3} N \right) (T_m + T_a)
\]

for the regular method.

Even at \( k = N \), the coefficient of the \( N^3 \) term is smaller by a factor of 1.2 under the hybrid system. Inspection of the term multiplied by \( k \) in both methods shows that, as \( k \) grows, the cost of the hybrid method will approach one-half that of the normal method.

If a fast algorithm (such as Strassen's) were used to compute the necessary matrix products, then for even smaller values of \( k \) the hybrid method to be better.

In summary, we have a method for solving a linear system of equations that is better than normal LU decomposition under the following conditions. First, there must be a way to partition the matrix into four submatrices, roughly equal in size, such that the upper left submatrix is invertible. Second, the problem must call for solving the system of equations for many \( b \) vectors, say \( N \) or more. The BEM application considered in this project satisfies both of these conditions.
6.4 The BEM System of Equations

From Section 6.2, a system of NODES + NELX equations in NODES + NELX unknowns can be formed by collocating at each boundary node and at each domain element and solving the integrals of equation (102). The BEM implementation exploits the hybrid method of solving a system of equations discussed in the last section. The large system of equations is partitioned quite naturally based on boundary nodes and domain elements, as shown in equation (111).

\[
\begin{bmatrix}
B_{nj} & B_d \\
D_{nj} & D_d
\end{bmatrix}
\begin{bmatrix}
\Phi^{(i)}_n \\
\Psi^{(i)}
\end{bmatrix} =
\begin{bmatrix}
B_k \Phi^{(i)}_k - B_d \Psi^{(i-1)} \\
D_k \Phi^{(i)}_k - D_d \Psi^{(i-1)}
\end{bmatrix}
\]

The coefficient matrices and variable vectors are described below.

- \(B_{nj}, B_k \) (NODES \( \times \) NODES) Coefficients for boundary values produced by collocating at a boundary node and integrating over the boundary. These two matrices contain the values of \(\int G(\tilde{P}, \tilde{Q}) H_j(s) \, dS_i \) and \(\int G'(\tilde{P}, \tilde{Q}) H_j(s) \, dS_i \) in equation (102), rearranged to separate the coefficients of known values from the coefficients of unknown values. The \(C(\tilde{P})\) term from the equation is also absorbed by these matrices.

- \(B_d\) (NODES \( \times \) NELX) Coefficients for domain values produced by collocating at a boundary node and integrating over the domain. This matrix contains the values of \(\frac{a}{\Delta t} \int G(s) \, dD_i\) in equation (102).
Coefficients for boundary values produced by collocating at a domain element centroid and integrating over the boundary. These two matrices contain the values of
\[ \int_{S_i} G(\tilde{P}, \tilde{Q}) H_j(s) \, dS_i \] and \[ \int_{S_i} G'(\tilde{P}, \tilde{Q}) H_j(s) \, dS_i \] in equation (102), rearranged to separate the coefficients of known values from the coefficients of unknown values.

Coefficients for domain values produced by collocating at a domain element centroid and integrating over the domain. This matrix contains the values of \[ \frac{\Delta t}{\alpha_i} \int_{S_i} G(s) \, dD_i \] in equation (102).

Constant coefficient matrix \( 2\Pi I \) (where \( I \) is the identity matrix). This is \( C(\tilde{P}) \) on the domain.

Boundary unknown values at time step \( t \)

Boundary known values at time step \( t \)

Domain values at time step \( t \)

Once the coefficient values are computed using the symbolic integration process, the hybrid method for solving a linear system is applied. A complication is that getting the \( b \) vector for each time step requires computing two matrix-vector products. However, this would also be the case for normal LU decomposition and solve. Also, as can be seen from equation (111), the new \( b \) vector depends only on the domain values from the previous time step and on the known boundary values from the present time step. Unless the unknown boundary values for a time step are needed for output, they need not be computed at each time step.
6.5 Parallel Implementation

The BEM process consists mostly of operations on two-dimensional matrices and one-dimensional vectors. To work with these matrices and vectors on a mesh-connected, distributed memory parallel computer like the MasPar, the data elements must be distributed to the memory of individual processors of the PE array. There are two commonly used schemes for distributing data elements to processors on a parallel computer.

A one-dimensional scatter decomposition places the $i$th component of a vector on processor $(i \mod p)$, where $p$ is the number of processors. Given a two-dimensional data matrix $A$ and a two-dimensional processor array $P$, a two-dimensional scatter decomposition places data element $A(i,j)$ on processor $P(i \mod p_1, i \mod p_2)$, where $p_1$ and $p_2$ are, respectively, the number of rows and columns in the processor array.

A one-dimensional block decomposition groups consecutive data elements into blocks of $b = \lceil n/p \rceil$, where $n$ is the number of data items, and $p$ is the number of processors. Data element $i$ is placed on processor $(i \div b)$. Given an $(n_1 \times n_2)$ two-dimensional data matrix $A$ and a $(p_1 \times p_2)$ two-dimensional processor array $P$, a two-dimensional block decomposition places data element $A(i,j)$ on processor $P(i \div b_1, j \div b_2)$, where $b_1 = n_1/p_1$ and $b_2 = n_2/p_2$.

Scatter decomposition was chosen for the BEM implementation. For nodal data, the technique was modified. Since nodes belonging to the same boundary element participate in the same computations during integration, putting them close together on the PE array reduces the need for interprocessor communication. Therefore, the nodes are broken into groups based on the boundary elements they belong to, and then the groups are scattered to the PE array. Boundary node and domain element data is in one-dimensional arrays, but must be distributed to a two-dimensional PE array. The one-dimensional arrays are scatter-decomposed onto each row and each column of the PE array. Why this is useful will become clear when the data distribution
of the coefficient matrices is discussed below. Table 7 gives an exact definition for the data distribution of boundary nodes and domain elements.

Notice that the nodes are broken into groups of \( NK - 1 \), not \( NK \). An end node is shared by two neighboring boundary elements, and must be stored with one or the other. The convention chosen was to group the first \( NK - 1 \) nodes of a boundary element together. The last node of that element is stored with the next group, exactly one processor away.

Table 7: Boundary node and domain element data distribution

<table>
<thead>
<tr>
<th>Data Element</th>
<th>PE Row Placement</th>
<th>PE Column Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node ((i))</td>
<td>((i \text{ div } (NK - 1)) \mod P)</td>
<td>ALL</td>
</tr>
<tr>
<td>Node ((i))</td>
<td>ALL</td>
<td>((i \text{ div } (NK - 1)) \mod P)</td>
</tr>
<tr>
<td>Centroid ((i))</td>
<td>(i \mod P)</td>
<td>ALL</td>
</tr>
<tr>
<td>Corners ((i))</td>
<td>ALL</td>
<td>(i \mod P)</td>
</tr>
</tbody>
</table>

Once the data distribution for the boundary nodes and domain elements is settled, the data distribution for coefficient matrices comes from the inputs and outputs of the integration processes (recall Figure 23). Computation of each element of row \( i \) of \( B_u, B_v, \text{ or } B_d \) requires \( \text{Node}(i) \) as one input, and \( \text{Node}(i) \) is duplicated on each PE in row \((i \text{ div } (NK - 1)) \mod P\) of the PE array. Therefore, row \( i \) of each of these coefficient matrix matrices is placed on PE row \((i \text{ div } (NK - 1)) \mod P\). The rest of the data distributions follow similarly as shown in Table 8. Once this data layout is set up, the only interprocessor communication needed for integration is a nearest neighbor fetch during boundary integration to get the last node of each boundary element from the next processor to the east.

The discussion above assumes that each pair of neighboring boundary elements share a common endpoint. However, the BEM allows neighboring elements not to share an endpoint,
Table 8: Coefficient matrix data distribution.

<table>
<thead>
<tr>
<th>Matrix Element</th>
<th>PE Row Placement</th>
<th>PE Column Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_n(i,j)$</td>
<td>$(i \text{ div } (NK - 1)) \text{ mod } P$</td>
<td>$(j \text{ div } (NK - 1)) \text{ mod } P$</td>
</tr>
<tr>
<td>$B_k(i,j)$</td>
<td>$(i \text{ div } (NK - 1)) \text{ mod } P$</td>
<td>$(j \text{ div } (NK - 1)) \text{ mod } P$</td>
</tr>
<tr>
<td>$B_d(i,j)$</td>
<td>$(i \text{ div } (NK - 1)) \text{ mod } P$</td>
<td>$j \text{ mod } P$</td>
</tr>
<tr>
<td>$D_n(i,j)$</td>
<td>$i \text{ mod } P$</td>
<td>$(j \text{ div } (NK - 1)) \text{ mod } P$</td>
</tr>
<tr>
<td>$D_k(i,j)$</td>
<td>$i \text{ mod } P$</td>
<td>$(j \text{ div } (NK - 1)) \text{ mod } P$</td>
</tr>
<tr>
<td>$D_d(i,j)$</td>
<td>$i \text{ mod } P$</td>
<td>$j \text{ mod } P$</td>
</tr>
</tbody>
</table>

as shown in Figure 24. The end nodes are drawn separately for clarity, but they actually have the same geometric coordinates. This situation occurs at points of singularity.

The presence of non-shared end nodes would seem to rule out the data decomposition scheme above, since it would mean no set formula to determine what boundary nodes belonged to what boundary element. However, the scheme can be saved by adding a dummy element to the boundary, as shown in Figure 24. In this example, because the elements are quadratic, the dummy element introduces one dummy node, Node (7).

![Figure 24: Neighboring boundary elements that do not share an end node.](image-url)
Dummy nodes and elements are not processed normally during integration. When collocating at a dummy node, the corresponding coefficient matrix row is filled with zeroes, except for a one on the diagonal. The coefficient matrix column corresponding to a dummy node is also zeroed out, except for that diagonal element.

![Diagram](image)

Figure 25: Adding a dummy element.

The coefficient matrices represent a system of equations \( Ax = b \). The addition of dummy node \( \text{Node}(i) \) adds one extra unknown \( x_i \) and one extra equation \( x_i = b_i \) to the system. In all rows except row \( i \), the multiplier for \( x_i \) is zero. Thus, the rest of the system has the same solution it would have if it did not contain \( x_i \).

This insertion of dummy elements is a regularizing transformation. It trades off a larger problem size for a more regular data set. In the BEM implementation, this transformation allows a data distribution that results in very little interprocessor communication, and good load balancing.

The rest of the parallel implementation is a set of matrix algorithms: matrix multiplication, LU decomposition, and matrix inversion. LU decomposition on the MasPar computers has been
studied in detail by another graduate student in the ISU Computer Science Department, Youngtae Kim. His work was used to guide the implementation of LU decomposition in this project. Parallel algorithms for matrix multiplication and matrix inversion are analyzed in Chapter 7.

LU decomposition and Gauss-Jordan matrix inversion both achieve better load balancing if 2D scatter decomposition is used. However, the matrix inverted in the BEM, namely $B_u$, is distributed to the PE array using the modified scatter scheme based on boundary element grouping. It turns out that the inversion process (Gauss-Jordan or LUD) can treat the matrix as if it were distributed using unmodified 2D scatter decomposition, and the BEM process will still yield correct results.

The modification to the 2D scatter decomposition for the BEM program amounts to permuting the rows and columns of the matrix in the same way. This permutation of rows and columns can be written in terms of pre- and post-multiplying by a permutation matrix $P^T$. Instead of inverting $B_u$, the inversion process inverts $PB_uP^T$. However,

\begin{equation}
(112) \quad (PB_uP^T)^{-1} = (P^T)^{-1}B_u^{-1}P^{-1}
\end{equation}

and since the transpose of a permutation matrix is its inverse,

\begin{equation}
(113) \quad (P^T)^{-1}B_u^{-1}P^{-1} = PB_u^{-1}P^T
\end{equation}

But $PB_u^{-1}P^T$ is just the inversion of the desired matrix, with rows and columns permuted as specified by the modified 2D scatter decomposition scheme. So the result is the same whether the inversion algorithm treats the matrix as being distributed using unmodified or modified 2D scatter decomposition.

Finally, the issue of PE array input/output was largely ignored in this project. Since the focus of this thesis is not fast parallel I/O, the implementation merely performs I/O on the front-end.

---

1. A permutation matrix is a matrix with exactly one element equal to 1 per row and per column.
machine and passes values to the PE array one value at a time. Better performance could be achieved by using block transfers between the front-end and the PE, or perhaps by using a disk array. However, as will be seen in the next chapter, I/O time is not a big issue for the BEM. The I/O time is obviously linear with the size of the input and output data. On the other hand, many of the computations of the BEM, such as matrix multiplication and LU decomposition, take time that is cubic with the size of the input.
7 CORE BEM PARALLEL ALGORITHMS

From Chapter 6, there are three parts of the BEM that are most computationally expensive, at least asymptotically. Matrix multiplication, LU decomposition, and matrix inversion are all cubic algorithms. The execution times of the other parts of the BEM implementation are quadratic, at most, in the size of the input. Therefore, to achieve good overall performance, it is likely to be most important to find efficient parallel algorithms for these three matrix operations. The actual execution timings bear this out.

7.1 Matrix Inversion

As discussed in Chapter 6, the BEM requires the repeated solving of a system of linear equations. Using some known properties of the linear system in the BEM, the problem of repeatedly solving the system is transformed into a problem of repeatedly solving a smaller linear system, along with some extra matrix operations. The transformed problem performs fewer total computations. Matrix multiplication and matrix inversion are the two significant extra operations performed, in terms of execution time. LU decomposition and LU solve are used to solve the smaller system of linear equations.

Implementing matrix inversion efficiently is not often considered, because the most common use of matrix inversion is to solve a system of linear equations, and it is less expensive to use LU decomposition to solve a system of a given size. However, because it is used in the BEM, it is worth the effort here to study matrix inversion in detail. Matrix inversion accounts for a significant fraction of the total execution time. Four different matrix inversion algorithms were implemented and their performance measured. The analysis to follow refers to those timings, shown in Table 9.

In the Fortran code, matrix inversion is performed by LU decomposing the matrix, and then by solving for each column of the identity matrix to generate each column of the inverse. Since
parallel LU decomposition and LU solve are used elsewhere in the BEM implementation, it was simple to combine them to implement matrix inversion.

Table 9: Matrix Inversion Execution Time (seconds)

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUD and LUS</td>
<td>48.9</td>
<td>284.2</td>
<td>838.8</td>
<td>1845.6</td>
<td>3439.5</td>
</tr>
<tr>
<td>LUD and Pipelined LUS</td>
<td>2.6</td>
<td>11.4</td>
<td>30.1</td>
<td>57.8</td>
<td>100.8</td>
</tr>
<tr>
<td>Gauss-Jordan Elimination</td>
<td>0.5</td>
<td>2.7</td>
<td>7.7</td>
<td>16.6</td>
<td>30.9</td>
</tr>
<tr>
<td>Simultaneous LUS</td>
<td>0.7</td>
<td>2.8</td>
<td>7.5</td>
<td>15.3</td>
<td>27.8</td>
</tr>
<tr>
<td>MP-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUD and LUS</td>
<td>30.9</td>
<td>200.6</td>
<td>625.1</td>
<td>1420.8</td>
<td>2703.7</td>
</tr>
<tr>
<td>LUD and Pipelined LUS</td>
<td>2.7</td>
<td>13.1</td>
<td>35.4</td>
<td>73.3</td>
<td>130.7</td>
</tr>
<tr>
<td>Gauss-Jordan Elimination</td>
<td>0.5</td>
<td>3.0</td>
<td>8.9</td>
<td>20.0</td>
<td>37.6</td>
</tr>
<tr>
<td>Simultaneous LUS</td>
<td>0.6</td>
<td>3.0</td>
<td>8.8</td>
<td>18.8</td>
<td>34.7</td>
</tr>
</tbody>
</table>

As can be seen in the first and fifth rows of the table, this naive implementation of matrix inversion performs terribly on a mesh computer. LU solve is an inherently serial operation. The forward and backward solve operations iterate over the columns of the LU matrix, with each iteration using values computed in the previous step. From chapter 6, the execution time of sequential LU solve can be estimated by

\[
(114) \text{LUS} (n) = n^2 (T_a + T_m) + nT_m
\]

The parallel LU solve in the BEM implementation assumes a 2D scatter decomposition of the LU matrix for better processor utilization during LU decomposition. The high-level structure of the forward substitution part of the parallel LU solve is described by the following pseudocode fragment, where \(n\) is the length of the LU matrix and \(p\) is the length of the square processor mesh.
For $i = 1$ to $n/p$ do
  For $j = 1$ to $p$ do
    begin
      On PE column $j$ do
        begin
          Swap current element with LU pivot element
          Broadcast current element down PE column
          For $k = j$ to $n/p$ do
            Update solution vector element with the $(k, i)$ element of LU submatrix
            (one add and one multiply)
          end
        end
      Shift solution vector east
    end
end

A straightforward estimation of parallel execution time is

$$\sum_{i=1}^{n/p} \left[ \sum_{j=1}^{p} \left( T_s + T_b + \frac{n}{p} T_c + \sum_{k=i}^{n/p} (T_a + T_m) \right) \right]$$

given the following execution time constants:

- $T_s$  time for pivot swap
- $T_b$  time for PE broadcast
- $T_c$  time for nearest-neighbor communication
- $T_a$  time for floating point add
- $T_m$  time for floating point multiply

Backward substitution takes an identical amount of time, except that no pivot swap is required. Putting these two times together and simplifying results in this estimate for total parallel execution time of the LU solve:

$$\frac{n^2}{p} \left( 2T_c + T_a + T_m \right) + n \left( T_s + 2T_b + T_a + T_m \right)$$
The inefficiency of the parallel LU solve comes from the fact that only one PE column is active at a time. This shows up in the execution time formula in that the $n^2$ term is divided by $p$, the length of the square PE array, rather than by $p^2$, the number of PEs.

There are several parts of the BEM that take cubic execution time, so this inefficiency in a quadratic algorithm does not at first seem disastrous. However, to invert an $n \times n$ matrix requires performing $n$ of these LU solves, making the inversion itself an inefficient cubic operation. The matrix inversion make execution time so high that it would not even be worth the effort to implement a parallel BEM. For comparison, see Tables 32 through 41 in Appendix 1 for sample execution times of sequential BEM implementations.

The matrix inversion execution time can be reduced by recognizing that there are $N$ independent LU solves to be performed. During forward substitution, once the first solve has progressed to the second PE column, another solve can be started in the first PE column. The LU solves can be pipelined so that by the time the first solve reaches the rightmost PE column, each PE column is performing forward substitution. A similar pipelined procedure is performed for back substitution. The second and sixth rows of the table show the execution times using this method.

Even with the pipelined LU solve, matrix inversion continues to account for about a third of execution time. Another matrix inversion method is Gauss-Jordan elimination. It is similar to LU decomposition, but requires no solve steps to complete the inversion. The third and seventh rows of the table show the execution times using Gauss-Jordan elimination.

Gauss-Jordan elimination outperforms the pipelined LU solve approach, but there is a way to get a similar performance advantage without changing the underlying mathematical technique. Matrix inversion using LU decomposition/solve requires an LU solve for each of the $N$ columns of the identity matrix. In this last inversion implementation, each PE array column simultaneously performs the computations for solving a column of the identity matrix, with
columns of the LU matrix broadcast across the PE array as needed. This simultaneous LU solve approach uses some non-nearest-neighbor communication to broadcast columns of LU, but unlike the pipelined LU solve, keeps every PE column busy most the time. The following pseudocode fragment shows the high-level structure of the forward substitution phase of this algorithm:

\[
\begin{align*}
\text{For } i = 1 \text{ to } n/p \text{ do} \\
&\text{For } j = 1 \text{ to } p \text{ do} \\
&\quad \text{begin} \\
&\quad \text{For } k = 1 \text{ to } n/p \text{ do} \\
&\quad\quad \text{Each PE column perform pivot swap for one solution vector} \\
&\quad\quad \text{Broadcast column } i \times p + j \text{ of LU matrix across PE array} \\
&\quad \text{For } k = 1 \text{ to } n/p \text{ do} \\
&\quad\quad \text{begin} \\
&\quad\quad\quad \text{Broadcast current solution vector element down each PE column} \\
&\quad\quad\quad \text{For } m = i \text{ to } n/p \text{ do} \\
&\quad\quad\quad\quad \text{Update a solution vector element (one add and one multiply)} \\
&\quad\quad \text{end} \\
&\quad \text{end} \\
&\text{end}
\end{align*}
\]

The execution time estimate for the forward substitution is

\[
(117) \quad \frac{n}{p} \sum_{i=1}^{n/p} \sum_{j=1}^{p} \left[ \sum_{k=1}^{n/p} T_z + T_b + \sum_{k=1}^{n/p} \left( T_b + \sum_{m=1}^{n/p} (T_a + T_m) \right) \right]
\]

Again, backward substitution takes the same execution time, except that no pivot swap is required, so the total execution time for the \( n \) LU solves necessary to complete the inversion is

\[
(118) \quad \frac{n^3}{p^2} (T_a + T_m) + \frac{n^2}{p} (T_z + 2T_b + T_a + T_m) + n2T_b
\]

This simultaneous LU solve method is clearly better than the pipeline LU solve method. The cubic term of the execution time is divided by \( p^2 \), the number of PEs, rather than by \( p \), the length of the PE array. Thus, the method has much better processor utilization. The fourth and eighth rows of the table show the execution time using this method.
The performance improvements from choosing a more efficient parallel matrix inversion algorithm were dramatic. For both the MP-1 and MP-2, the execution time of the simultaneous LU solve method compared with the first naive algorithm discussed was better by a factor roughly equal to the length of the PE array, just as would be suggested by the formal analysis. In general, when a parallel algorithm shows such dismal processor utilization, a different approach to parallelization is probably in order.

7.2 Matrix Multiplication

Matrix multiplication serves as an integral part of many larger scientific algorithms. For example, implementation of level 3 BLAS routines can be based on matrix multiplication. As described in Chapter 6, several matrix products are calculated by the BEM program. The matrices multiplied have lengths proportional to the size of the input to the BEM, so the execution time of the matrix multiplication part of the BEM is cubic in the size of the input.

Three parallel algorithms are described to calculate the product $C$ of matrices $A$ and $B$, each of size $N \times N$. The first is quite inefficient, and included mostly to illustrate how not to implement a computationally intensive operation. The second and third are well-known and quite efficient, and have been studied on a variety of architectures. A formal analysis of the three algorithms will lead to a choice of one of them for inclusion in the BEM implementation. Actual timings on the MasPar computers will be used to verify that decision.

The processor array is assumed to be of size $P \times P$. For simplicity, the algorithms are illustrated using a hypothetical $4 \times 4$ PE array, and for $4 \times 4$ $A$, $B$, and $C$ matrices. Multiplication of matrices with length a multiple of the length of the PE array is performed using block decomposition. The algorithms remain the same except that on each processor, instead of scalar addition and multiplication, matrix addition and multiplication is performed on submatrices. $A$, $B$, and $C$ are assumed to be loaded in normal order. If an algorithm requires some shifting
of array elements, then that shifting is counted as part of the execution time. The algorithms may shift elements of \( A \) and \( B \) around during execution, but must put them back in their original positions before terminating.

7.2.1 Logarithmic Sum Algorithm

This algorithm requires loading the \( A \) matrix in normal order and the \( B \) matrix transposed. It produces the \( C \) matrix in normal order as shown in Figure 26. In each iteration, the algorithm computes \( N \) values of \( C \). Notice that the parallel prefix sum requires \( \log N \) communication steps and as many addition steps. Communication hops of up to \( N/2 \) processors are required.

The algorithm is shown below:

**Transpose \( B \)**

For \( i = 0 \) to \( P - 1 \)

begin

STEP 1 (Multiplication): \( c_{\text{temp}} = a \times b \)

STEP 2 (Parallel Prefix Sum): Sum \( c_{\text{temp}} \) in row \( j \) into \( c \) in column \( (i + j) \) mod \( P \)

STEP 3 (Communication): Shift each \( b \) one processor north

end

**Un-Transpose \( B \)**
In each iteration, the algorithm calculates $P$ values of $C$. On the 0th iteration the diagonal elements of $C$ are calculated, on first iteration the elements one to the right of the diagonal are calculated, and so on.

### 7.2.2 Broadcast Algorithm

This algorithm, reported in [3], begins with matrices $A$, $B$, and $C$ all stored in normal order. As in algorithms 1 and 2, $c_{ij}$ is computed on processor $P_i$. For example, $c_{00}$ is computed by calculating the products $a_{00}b_{00}$, $a_{01}b_{10}$, and $a_{02}b_{20}$, and $a_{03}b_{30}$ and accumulating the sum in 4 successive iterations, the first of which is illustrated in Figure 27.

![Figure 27: Matrix multiplication using broadcast](image)

The algorithm is shown below:

```
For $i = 0$ to $P-1$
    begin
      STEP 1 (Broadcast): Broadcast $a$ from column $(i + j) \mod P$ into atemp across each row $j$ of processors
      STEP 2 (Multiplication): $ctemp = atemp \times b$
      STEP 3 (Addition): $c = c + ctemp$
      STEP 4 (Communication): Shift each $b$ north one row
    end
```
7.2.3 Systolic Parallel Algorithm

This algorithm appears in [2] and was derived independently in [8]. It requires that the matrices \( A \) and \( B \) be initially loaded in a shifted order. Each row of \( A \) is shifted east until each diagonal element \( a_{ij} \) is on the eastmost edge of the processor array. Similarly, each column of \( B \) is shifted south until each diagonal element is on the southmost edge, as shown in Figure 28.

![Initial data layout](a) Initial data layout ![A matrix shifted west](b) A matrix shifted west ![B matrix shifted north](c) B matrix shifted north

Starting from the initial layout, successive layouts are shown after each communication step in the first iteration of the loop. The value \( c_{ij} \) is computed on processor \( P_{ij} \). For example, \( c_{00} \) is computed by calculating the products \( a_{i0}b_{00} \), \( a_{01}b_{10} \), and \( a_{02}b_{20} \), and \( a_{03}b_{30} \) and accumulating the sum in four successive iterations of the loop on processor \( P_{00} \). This and other such systolic algorithms can be designed using the method described in [8].

The algorithm is shown below:

\[
\begin{align*}
\text{Shift A and B as required} \\
\text{For } i = 0 \text{ to } P-1 \\
\hspace{1em} \text{begin} \\
\hspace{2em} \text{STEP 1 (Multiplication): } & \text{ ctemp } = a \times b \\
\hspace{2em} \text{STEP 2 (Addition): } & \text{ c } = c + \text{ctemp} \\
\hspace{2em} \text{STEP 3 (Communication): Shift each a one processor west} \\
\hspace{2em} \text{STEP 4 (Communication): Shift each b one processor north} \\
\hspace{1em} \text{end} \\
\text{Un-Shift A and B}
\end{align*}
\]
7.2.4 Performance Analysis

The following formulas for the execution time of the three matrix multiplication algorithms are useful in analyzing the performance of the algorithms and the effectiveness of optimizations.

\[(119) T_{\text{log}} = P \left[ M^3 (T_m + T_a + 2T_s) + M^2 (\log P (T_a + T_c) + T_x + 3T_s) \right] + 2M^2 (T_r + 2T_s) \]

\[(120) T_{\text{broadcast}} = P \left[ M^3 (T_m + T_a + 2T_s) + M^2 (T_c + T_x + 6T_s) \right] \]

\[(121) T_{\text{systolic}} = P \left[ M^3 (T_m + T_a + 2T_s) + M^2 (4T_x + 10T_s) \right] \]

where

\( N \) length of the matrices \( A, B, \) and \( C \)

\( P \) length of the PE array

\( M \) length of a submatrix on a PE \( (N/P) \)

\( T_m \) time for floating point multiply

\( T_a \) time for floating point add

\( T_s \) time for memory load or store

\( T_x \) time for nearest-neighbor communication

\( T_c \) time for non-nearest-neighbor communication

\( T_r \) time for transpose-pattern communication (router)

These approximate formulas follow from the algorithm descriptions above, but a few comments will make them clearer. The \( M^3 \) terms in each formula come from the cubic operation of submatrix multiplication. The \( 2T_s \) associated with \( M^3 \) in each formula comes from loading elements of the \( A \) and \( B \) submatrix. The \( T_s \) associated with \( M^2 \) comes from loading and storing elements of the \( C \) submatrix, and the fact that communicating a matrix element from one processor to another requires a load at the source and a store at the destination. The quantity \( T_c \)
is not really a constant, because it varies with the communication pattern. However, since for any physical machine it has a constant upper bound, it is convenient to treat it as a constant.

The last term of $T_{\log}$ represents the time to transpose the $B$ matrix. The $M^2$ term of $T_{systolic}$ includes time for performing the necessary shifts of $A$ and $B$.

Notice that the logarithm sum operation in the first algorithm takes logarithmic time in the length of the PE array, not the length of the matrix multiplied. For any real machine, the log term becomes a constant (6 or 7 in this study). On a fixed-size machine, the behavior of the three algorithms is asymptotically identical as problem size grows. Indeed, any reasonable parallel algorithm based on normal serial matrix multiplication (i.e. not based on methods such as those proposed by Strassen, Winograd, etc. [9]) will have an execution time of order $N^3$. Reducing execution time becomes a problem of reducing constants.

Even though it reduces to a constant for any physical machine, that log term should make it impossible for the first algorithm to compete with the other two, especially when source-level software pipelining is used to make memory accesses less important. Another way of looking at it is that during the logarithmic sum operation, some PE columns lay idle while others calculate the sum. This reduced PE utilization takes a toll on performance.

The expected execution times of the broadcast and systolic algorithms are quite similar. The former must perform a non-nearest neighbor communication during the broadcast step, but does not have to shift the $A$ matrix to the west as in the systolic algorithm. The broadcast algorithm also does not need to perform pre- and post-shifting on matrices $A$ and $B$. Which algorithm actually performs better depends on the exact values of $T_c$ and $T_x$, and on how well memory accesses can be pipelined. Indeed, as shown in Table 10, the broadcast algorithm performs best on the MP-1, while the systolic algorithm wins out on the MP-2. (These matrix multiplications were performed in single precision, with source-level software pipelining.)
Table 10: Performance of Matrix Multiplication Algorithms (time in seconds)

<table>
<thead>
<tr>
<th></th>
<th>Matrix Size</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1024</td>
<td>2048</td>
<td>3072</td>
</tr>
<tr>
<td>MP-1</td>
<td>Logarithmic Sum</td>
<td>4.3</td>
<td>25.6</td>
<td>76.9</td>
</tr>
<tr>
<td></td>
<td>Broadcast</td>
<td>2.7</td>
<td>19.1</td>
<td>62.1</td>
</tr>
<tr>
<td></td>
<td>Systolic</td>
<td>3.0</td>
<td>20.4</td>
<td>65.3</td>
</tr>
<tr>
<td>MP-2</td>
<td>Logarithmic Sum</td>
<td>4.0</td>
<td>23.6</td>
<td>70.4</td>
</tr>
<tr>
<td></td>
<td>Broadcast</td>
<td>2.3</td>
<td>16.9</td>
<td>55.0</td>
</tr>
<tr>
<td></td>
<td>Systolic</td>
<td>2.3</td>
<td>16.6</td>
<td>54.5</td>
</tr>
</tbody>
</table>

For the BEM implementation, the systolic algorithm was chosen over the broadcast algorithm for two reasons. First, the broadcast algorithm requires extra memory for a temporary matrix. In fact, that is why there is no broadcast execution time for a 4096 x 4096 matrix on the MP-2 in the table above; there was not enough memory. Second, in the BEM implementation it would be possible to avoid some of the overhead from pre- and post-shifting the A and B matrices, and so make the systolic algorithm a clear winner. In the BEM, several matrix products are calculated, and in two cases the products have the same A factor. The pre-shifting of A could be performed once for both products. Also, the A and B matrices are usually not used in the BEM after the matrix product is calculated, so it would not always be necessary to post-shift A and B back to their original positions. These optimizations have not yet been made, but using the systolic algorithm for matrix multiplication allows for that improvement when time permits.

7.2.5 Source-Level Software Pipelining of Matrix Multiplication in the BEM

In Chapter 4, memory access optimizations were shown to produce speedups of 1.3 and 1.9 on the MP-1 and MP-2, respectively, for large matrix multiplication problems. Table 11 shows
that for large BEM problems, the speedup in the matrix multiplication part of the process is 1.1 and 1.4 on the MP-1 and MP-2, respectively, for the largest problem size.

The gains are not as dramatic for three reasons. First, in the BEM implementation, to keep the code as readable as possible, only source-level software pipelining was applied. The blocking technique could be applied to get further gains. Second, the maximum length of the matrices multiplied in the BEM was smaller than in the raw matrix multiplication experiments. If PE memory were available to run larger problems, one would expect to see somewhat larger speed-ups due to pipelining. Finally, the portion of the BEM implementation being called “matrix multiplication” for simplicity also contains some extra minor matrix operations (three matrix additions) necessary to set up the system of equations to solve.

| Table 11: Effect of Software Pipelining on Matrix Multiplication in the BEM |
|----------------------------------|--------|---------|--------|--------|--------|
|                                | 256    | 512     | 768    | 1024   | 1280   |
| **MP-1**                        |        |         |        |        |        |
| Without Pipelining              | 0.9    | 4.8     | 14.0   | 30.7   | 57.0   |
| With Pipelining                 | 0.8    | 4.4     | 12.7   | 27.6   | 51.0   |
| **MP-2**                        |        |         |        |        |        |
| Without Pipelining              | 1.0    | 6.1     | 18.4   | 40.8   | 76.5   |
| With Pipelining                 | 0.8    | 4.6     | 13.5   | 29.7   | 55.5   |

7.3 LU Decomposition

Younhtae Kim, another student in the Iowa State University Computer Science Department, has studied LU decomposition in detail on the MasPar computers. His algorithm was adapted with permission for the BEM implementation, and is described here.

LU decomposition without partial pivoting of an $N \times N$ matrix takes $N$ iterations. Each iteration consists of three steps, as described below. Partial pivoting will be covered later.
For \( k = 1 \) to \( N \)

STEP 1 (Coefficient Inversion): \( a_{kk} = 1.0 / a_{kk} \)

\[
\begin{bmatrix}
  a_{k+1,k} \\
  a_{k+2,k} \\
  \vdots \\
  a_{k+n,k}
\end{bmatrix} = a_{kk} \times 
\begin{bmatrix}
  a_{k+1,k} \\
  a_{k+2,k} \\
  \vdots \\
  a_{k+n,k}
\end{bmatrix}
\]

STEP 2 (Multiplier Calculation):

\[
\begin{bmatrix}
  a_{k+1,k} \\
  a_{k+2,k} \\
  \vdots \\
  a_{k+n,k}
\end{bmatrix}
\]

STEP 3 (Submatrix Update):

\[
\hat{A} = \hat{A} - a_{kk+1} \cdot a_{k,k+1} \cdots a_{k,k+1}
\]

where \( \hat{A} = 
\begin{bmatrix}
  a_{k+1,k+1} \cdots a_{k+1,k+1} \\
  \vdots \\
  a_{k+1,k+1} \cdots a_{k+1,k+1}
\end{bmatrix}
\]

In step 1, a diagonal element \( a_{kk} \) is inverted on the PE containing it. In stage 2, the multipliers in the subcolumn vector \( C_k = (a_{k+1,k}, a_{k+2,k}, \ldots, a_{nk})^T \) are calculated using the diagonal element from step 1. The multipliers are stored in the lower triangular part of the LU decomposition. Finally, in step 3, \( C_k \) is broadcast across the PE columns and the subrow vector \( R_k = (a_{k,k+1}, a_{k,k+2}, \ldots, a_{kn}) \) is broadcast down the PE rows, using pipelined xnet communication. The submatrix \( \hat{A} \) is updated with the broadcast elements using the formula \( a = a - rc \) where \( a \) is the element of \( \hat{A} \) on the PE, \( r \) is the element from \( R_k \), and \( c \) is the element from
Figure 29 illustrates the $k$th loop of the algorithm. The shaded area represents the part of the matrix that has already been decomposed and requires no further calculation. The size of the matrix to be updated gets smaller with each iteration.

Partial pivoting in LU decomposition results in more stable matrix calculation. Pivoting requires finding the matrix element of column $k$ with the greatest absolute value, and then exchanging the row with that element with row $k$. To find the maximum element, each PE with elements from column $k$ performs a sequential search of its elements. Then a divide and conquer strategy is used to find the PE with the maximum element. Finally, pipelined xnet operations are used to exchange the pivot row with row $k$.

Because the LU decomposition algorithm computes on a smaller square submatrix each iteration, 2D scatter decomposition was chosen as the method of laying out matrix elements on the PE array, to reduce the number of idle processors. A scattered layout is a load-balanced layout for the parallel LU decomposition algorithm [4].
7.3.1 Source-Level Software Pipelining of LU Decomposition in the BEM

Table 11 shows the effect of source-level software pipelining on LU decomposition time in the BEM implementation. The speedups were not as high as those measured for LU decomposition alone. This difference could have been because of a number of factors. In particular, the execution time of the LU decomposition is very sensitive to the number of pivots that have to be performed. Because the pivoting algorithm is not particularly well-suited to software pipelining, a LU decomposing a matrix that required a lot of pivoting would see less gain from software pipelining.

Table 12: Effect of Software Pipelining on LU Decomposition in the BEM

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Without Pipelining</td>
<td>0.3</td>
<td>1.0</td>
<td>2.4</td>
<td>4.5</td>
<td>7.5</td>
</tr>
<tr>
<td>With Pipelining</td>
<td>0.3</td>
<td>0.9</td>
<td>2.1</td>
<td>3.9</td>
<td>6.6</td>
</tr>
<tr>
<td>MP-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Without Pipelining</td>
<td>0.2</td>
<td>0.9</td>
<td>2.2</td>
<td>4.6</td>
<td>9.0</td>
</tr>
<tr>
<td>With Pipelining</td>
<td>0.2</td>
<td>0.7</td>
<td>1.8</td>
<td>3.5</td>
<td>7.1</td>
</tr>
</tbody>
</table>
8 SOURCE CODE TRANSFORMATIONS FOR INCREASED PERFORMANCE

As discussed earlier, this thesis is concerned with achieving high performance for scientific applications on mesh-connected, distributed-memory, SIMD parallel computers. This chapter covers the techniques used to improve the performance of the parallel BEM implementation on the MasPar computer. The first technique is applicable to any machine with a SIMD architecture, while the rest of them are more "code-tuning" techniques that depend on specific architectural features of the MasPar system. Their applicability to other parallel machines would depend on the presence of those architectural features.

The execution times presented in this chapter were excerpted from the complete set of measurements for this project, found in Appendix 1.

8.1 Transforming Algorithms for SIMD Architectures

On a SIMD computer, every active processing element executes the same instructions at each time step. If, for example, half of the PEs need to execute function A and the other half need to execute function B, then each half must be idle while the other half works. Processor utilization is the fraction of PEs doing useful work. To achieve high performance, it is necessary to keep processor utilization as high as possible as much of the time as possible.

There is a sequence of code in the sequential Fortran BEM implementation that translates very naturally into a parallel switch statement in MPL. The MPL statement is analogous to the switch statement in C. In a C switch statement, one code branch is selected from many based on the value of a switch variable. In MPL, each PE selects a code branch to execute based on the value of a plural variable on that PE. However, since all active PEs on a SIMD machine must execute the same instructions at each time step, the different branches of the parallel switch statement must be serialized. For each branch of the switch, only a fraction of the PEs are active,
and processor utilization goes down. The more computationally expensive the code in the different branches of the switch, the more this reduced processor utilization degrades overall performance.

The particular switch statement in the BEM implementation has 19 branches, each calling a subroutine called \textit{aint} with different arguments. This subroutine is computationally expensive, with many calls to trigonometric functions. This inefficiency, if not addressed, causes integration to dominate the execution time of the BEM program.

The solution to the problem is to reorganize the code so that calls to \textit{aint} with different arguments on different processors can be executed at the same time. This is done by copying the different arguments to the functions on the various PEs into temporary variables. This copying must still be done serially for each different branch, but is an inexpensive operation, relative to the call to \textit{aint}. Once the temporary variables are loaded, the call to \textit{aint} can proceed with all or most of the PEs active. Since the code section changed in the actual application is rather long, the principle is illustrated below using a similar, sample code fragment. The original MPL code had a structure similar to the following:

```plaintext
plural int icase;
plural double x, y, z;

...

switch (icase) {
    case 1: return aint(x, y, z);
    case 2: return aint(y, z, x);
    case 3: return aint(z, x, y)
        + aint(x, z, y);
    case 4: return aint(z, y, x);
        + aint(y, x, z);
}
```
The improved code is shown here:

```c
plural int icase;
plural double x, y, z;
plural double arg1, arg2, arg3, result;
...
switch (icase) {
    case 1: arg1 = x; arg2 = y; arg3 = z; break;
    case 2: arg1 = y; arg2 = z; arg3 = x; break;
    case 3: arg1 = z; arg2 = x; arg3 = y; break;
    case 4: arg1 = y; arg2 = x; arg3 = z; break;
}
result = aint(arg1, arg2, arg3);
if (icase > 2) {
    switch (icase)
    {
        case 3: arg1 = x; arg2 = z; arg3 = y; break;
        case 4: arg1 = y; arg2 = x; arg3 = z; break;
    }
    result += aint(arg1, arg2, arg3);
}
```

In the example, the number of calls to `aint` was reduced from six to two. The general principle here is to try to reduce the amount of time spent with less than full processor utilization. In this case, the program performs some quick setup with low processor utilization to allow the expensive calls to `aint` to proceed with high processor utilization. The improved performance from this change can be seen in Table 15.

### Table 13: Effect of Parallel Switch Transformation on Integration Time

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration Execution Time (seconds)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MP-1 (before)</td>
<td>4.8</td>
<td>15.5</td>
<td>35.3</td>
<td>83.4</td>
<td>126.3</td>
</tr>
<tr>
<td>MP-1 (after)</td>
<td>1.2</td>
<td>4.5</td>
<td>10.0</td>
<td>17.9</td>
<td>27.9</td>
</tr>
<tr>
<td>MP-2 (before)</td>
<td>4.9</td>
<td>18.4</td>
<td>41.3</td>
<td>94.5</td>
<td>145.3</td>
</tr>
<tr>
<td>MP-2 (after)</td>
<td>1.5</td>
<td>5.8</td>
<td>12.9</td>
<td>23.3</td>
<td>36.2</td>
</tr>
</tbody>
</table>
8.2 Tuning Code for a Specific Machine

When the best parallel algorithms have been chosen, and the code has been rearranged to avoid "SIMD" pitfalls like the one discussed above, still more performance can be squeezed out by tuning the source code based on knowledge of the target machine. The two changes discussed in this section are specific to the MasPar hardware and compiler. Each specific kind of machine undoubtedly has characteristics that determine what kind of tuning can be used to good advantage.

The parallel BEM implementation for this project was a direct adaptation of a Fortran program. No effort was made to avoid redundant expression in the source code. For example, trigonometric functions appear multiple times in the integration code with the same invariant arguments. With a Fortran optimizing compiler, this is probably not a problem. However, in MPL, this results in inefficient code.

The MPL source code for the BEM implementation was modified to store common subexpressions in register variables, as much as possible. These code changes can be found in integrate.m and domainint.m in Appendix 5. The difference this made to integration execution time can be seen in Table 15. Notice that this improvement is cumulative with the parallel switch transformation discussed above. For some problem sizes, using this technique led to slightly different BEM results, not easily explained. One might blame floating point precision problems, except that a test program that made an extra call to a MasPar trignometric library function and then threw away the result obtained different results than one which did not call the function at all. In the size 1280 test case, this phenomenon actually caused a difference in LU decomposition time, because the slightly different matrix values led to different pivoting characteristics.

As was emphasized in Chapters 4 and 7, getting maximum performance from the MasPar computers also requires hand-tuning memory accesses so that as much as possible, they are overlapped with other useful computation. It was shown that for a simple algorithm like matrix
multiplication, on a 4096 x 4096 matrix, speedup from memory access optimizations could approach 1.3 and 2 on the MP-1 and MP-2, respectively. A more interesting question is how much speedup can be obtained from memory access optimizations on a real-world application.

Table 14: Effect of Common Subexpression Elimination on Integration Time

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>1.2</td>
<td>4.5</td>
<td>10.0</td>
<td>17.9</td>
<td>27.9</td>
</tr>
<tr>
<td>Execution</td>
<td>0.9</td>
<td>3.2</td>
<td>7.1</td>
<td>12.7</td>
<td>19.9</td>
</tr>
</tbody>
</table>

Source-level software pipelining was applied where the structure of the BEM implementation allowed. Software pipelining can most easily be applied to fairly small loops that access successive elements of an array, and perform some computation or PE communication using those elements. In the BEM implementation, opportunities for this kind of improvement can be found in the inversion, matrix multiplication, LU decomposition, and LU solve code. To avoid harming the readability of an already complex program more than was necessary, no loop unrolling was used, even though it would allow longer software pipelines. Pipelining was also only applied to inner loops. Table 15 shows the effect of software pipelining on overall execution time of the BEM. These timings were made with all of the optimizations discussed previously in place.

The speedups of from 1.08 to 1.21 for large BEM problems are not as dramatic as those for matrix multiplication in isolation for two main reasons. First, there is a part of the BEM code to which source-level software pipelining was not effectively applied. This part includes frontend I/O and integration. This project did not focus on parallel I/O issues, and loaded the PE
array sequentially, in a way that did not lend itself to software pipelining. Integration is dominated by floating point mathematical operations, so software pipelining would have little effect.

Second, because the BEM implementation requires several arrays to be in memory simultaneously, the maximum problem size (determined by available PE memory) was much smaller than for matrix multiplication.

In summary, regardless of the source code transformations applied, the overall execution time remained cubic in the input size, due to the \( N^3 \) terms of the matrix multiplication, LU decomposition, and matrix inversion algorithms. Without changing the underlying algorithms being parallelized (e.g. using Strassen's matrix multiplication algorithm), any reasonable parallel implementation would have an execution time of order \( N^3 \), on a real, fixed-sized computer.

The improvements to the matrix inversion algorithm, although dramatic, were constant for each MasPar machine, roughly equal to the length of the PE array. Transforming the parallel switch code speeded up integration only by a constant of about 4. However, notice that without that improvement, the quadratic integration operation took more time than the cubic matrix multiplication operations, even for the largest problem size that fit on the machine.
9 PARALLEL PERFORMANCE RELATIVE TO SEQUENTIAL MACHINES

The performance of the sequential Fortran and C versions of the BEM applications was measured on several fairly powerful sequential workstations. This chapter compares the execution time of the parallel BEM application with these sequential implementations and provides a very rough price/performance comparison of the MasPar machines with the workstations.

The fastest version of the MasPar BEM application (with all optimizations including software pipelining) was used in this comparison. For the sequential programs, one well-known optimization was implemented. Choosing a particular ordering for the three nested loops of matrix multiplication causes the algorithm to access memory in a way that better utilizes cache memory. The times shown in this chapter are for the cache-friendly versions of BEM. See Appendix 1 for the times for matrix multiplication with a suboptimal loop ordering.

Memory and CPU quota limits made it impossible to run some of the larger cases on some of the sequential machines. Also, the timings were made while the systems were in normal use by other users. User CPU time was measured (rather than elapsed time) to minimize the effects of other jobs on the timings, but there was still quite a bit of variability.

The HP9000 workstation had 256 megabytes of main memory, and the RS6000 had 64 megabytes. The DEC Alpha timings were made for the author by a third party on a machine of unknown price and configuration. They should be used only to get a general idea of the performance that can be achieved on another powerful workstation.

The machine prices shown below were obtained verbally from vendors, and should be considered approximate. In particular, the MP-1 is no longer sold with 256 megabytes of PE memory, so that price is a very rough estimate of what such a machine would cost at this time.

Table 16 shows the raw execution times of the MasPar and sequential computers. Notice that for the largest problem sizes, MasPar execution time was at most about two minutes, while the sequential execution time was an hour or more. Using the parallel implementation might be
important to an engineer using the BEM application, by providing a quick enough turnaround
time to allow him to work in interactive mode, rather than in batch mode.

Table 17 shows a price/performance measure for the BEM. Problem size divided by execution
time is used as a unit of speed. Note that since the BEM is dominated by order $N^3$ algorithms,
this "speed" measurement naturally decreases with problem size. However, this is true for all
machines, so the comparison is still fair. The measure of speed is divided by the price of the
machine to get a measure for price/performance.

Table 16: Execution times on MP-1, MP-2 and several sequential machines

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP-1</td>
<td>6.8</td>
<td>19.7</td>
<td>41.4</td>
<td>74.5</td>
<td>123.1</td>
</tr>
<tr>
<td>MP-2</td>
<td>6.5</td>
<td>19.4</td>
<td>42.5</td>
<td>79.2</td>
<td>133.4</td>
</tr>
<tr>
<td>HP 9000/755 (C)</td>
<td>64.0</td>
<td>433.9</td>
<td>1214.2</td>
<td>2741.0</td>
<td>--</td>
</tr>
<tr>
<td>RS6000 7012/340 (C)</td>
<td>85.4</td>
<td>492.6</td>
<td>1285.5</td>
<td>2763.0</td>
<td>5402.5</td>
</tr>
<tr>
<td>RS6000 7012/340 (Fortran)</td>
<td>62.5</td>
<td>317.6</td>
<td>787.7</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>DEC Alpha (C)</td>
<td>40.9</td>
<td>260.1</td>
<td>743.3</td>
<td>1676.2</td>
<td>3167.2</td>
</tr>
<tr>
<td>DEC Alpha (Fortran)</td>
<td>27.7</td>
<td>165.0</td>
<td>475.2</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

Table 17: \(\frac{\text{NODES/SEC}}{\$100000}\) on MP-1, MP-2 and several sequential machines

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Price</th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP-1</td>
<td>$225,000</td>
<td>16.7</td>
<td>11.6</td>
<td>8.2</td>
<td>6.1</td>
<td>4.6</td>
</tr>
<tr>
<td>MP-2</td>
<td>$430,000</td>
<td>9.2</td>
<td>6.1</td>
<td>4.2</td>
<td>3.0</td>
<td>2.2</td>
</tr>
<tr>
<td>HP 9000/755 (C)</td>
<td>$60,000</td>
<td>6.7</td>
<td>2.0</td>
<td>1.1</td>
<td>0.6</td>
<td>--</td>
</tr>
<tr>
<td>RS6000 7012/340 (C)</td>
<td>$15,000</td>
<td>20.0</td>
<td>6.9</td>
<td>4.0</td>
<td>2.5</td>
<td>1.6</td>
</tr>
<tr>
<td>RS6000 7012/340 (Fortran)</td>
<td>$15,000</td>
<td>27.3</td>
<td>10.7</td>
<td>6.5</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>
The sequential workstations compete well with the MasPar computers only for small problems. Programs on the MasPar often are more efficient at larger problem sizes. Also, as the larger problem sizes drive up working set size on the sequential machines, it is possible that caching characteristics degrade.

In summary, the MasPar implementation of the BEM compares well with the sequential implementation on a variety of workstations, if quick turnaround is needed or if the problem sizes are large. An interesting future direction would be to compare the MasPar implementation to a BEM implementation on a vector supercomputer, such as a Cray.
A boundary element method (BEM) application was implemented on the MasPar MP-1 and MP-2 computers. Different parallel algorithms were investigated for computationally intensive parts of the program. The performance of the implementation was measured for a variety of problem sizes. For comparison, the performance of a sequential implementation was measured on several sequential computers. Code changes were made to the parallel implementation to improve performance, and the effects were measured and analyzed.

The work is important in two ways. First, it illustrates techniques for producing efficient parallel programs on the MasPar machines, in enough detail to be useful to real-world application programmers. Second, the implementation provides a base on which further parallel BEM work can be built.

A systolic architecture for matrix multiplication derived from the systolic method of Chapter 3 was adapted for use in the BEM implementation. Analysis of the parallel complexity of various matrix inversion algorithms led showed the way to an efficient one. Knowledge of the MasPar memory architecture and of software pipelining (from the compiler literature) provided another method to increase performance. This thesis shows how to put "theory" into practice to solve real scientific problems.

Often, students of parallel algorithms concentrate on asymptotic behavior, assuming an arbitrarily large number of processors and memory. However, programs are always implemented on physical machines, with a fixed number of processors and amount of memory. It was shown that (barring parallelization of sub-cubic sequential algorithms for matrix multiplication, LU decomposition, and inversion) any reasonable implementation of the BEM would have an execution time of order \( N^3 \) in the size of the input. Every performance improvement discussed affected execution time by a constant factor only. Yet these improvements made the difference
between a successful implementation, and one which hardly outperforms the sequential code on a workstation.

The importance of constant factors established, it would not be wise to completely ignore the effects of increasing the number processors arbitrarily. The MP-2 in this study had 16K PEs. It is interesting to ask whether the BEM implementation would be efficient on a hypothetical MasPar with, say 1M PEs. For example, the length of such a PE array (1024 x 1024) might make row and column broadcasting too expensive to use. This would require rethinking some of the algorithms in the BEM implementation.

It would be interesting to consider other parallel LU decomposition algorithms for the BEM. A systolic LU decomposition algorithm can be derived using the systolic method, but other priorities kept it from being considered here.

The input/output in this BEM implementation is simple element-by-element loading to and fetching from the PE array. There are block oriented DPU I/O routines available that might speed up this process. Even more interesting would be investigating using a parallel disk array.

From the engineering perspective, there are a couple of directions to go for richer BEM implementations. A three-dimensional BEM implementation would be of more practical use, and might offer even more opportunity for efficient parallelism. The same is true for a BEM implementation for a non-linear differential equation. This implementation will provide a solid foundation for such future development efforts.
BIBLIOGRAPHY


APPENDIX 1: PERFORMANCE DATA

Tables 18 through 31 show the MasPar MP-1 and MP-2 execution times for the base BEM implementation, and for the BEM implementation with each performance improvement discussed above applied cumulatively. Execution time is broken down for the various phases of the program, and shown for a variety of problem sizes. Tables 32 through 41 show the same timings on a variety of sequential machines for comparison. For both the parallel and sequential versions, the source code was instrumented with system calls to capture processor usage for each phase.

The geometry used in the timings is a square flat plate, as shown in figure 30. The boundary elements are linear. The nodes at the four corners are not shared, though here they are pictured one on top of the other. A Fortran program supplied by Dr. Ambar Mitra generates geometries similar to this one, varying the number of boundary elements, boundary nodes and domain...
elements. The actual test cases were constructed with the number of boundary nodes and the number of domain elements as close to multiples of 128 as possible without going over. 128 is the length of the PE array of the MasPar MP-1 used in this study.

Tables 18 and 19 show the execution times for the base MPL version of the BEM implementation, before any effort was made to increase its performance. Tables 20 and 21 show the BEM phase execution times using the pipelined LU solve scheme for matrix inversion. Notice that before this improvement, inversion time dominates the total execution time. Afterward, inversion time is less than half of total execution time. Tables 22 and 23 show the BEM phase execution times using Gauss-Jordan elimination for matrix inversion. Tables 24 and 25 show the BEM phase execution times using the simultaneous LU solve scheme for matrix inversion. The rest of the results for parallel execution time shown in this appendix are for runs using this superior inversion method. Tables 26 and 27 show the BEM phase execution times with the “parallel switch transformation” applied. The rest of the results for parallel execution time shown in this appendix are for runs using this transformation. Tables 28 and 29 show the BEM phase execution times with “hand-coded common subexpression elimination” applied. The rest of the results for parallel execution time shown in this appendix are for runs using this optimization. Tables 30 and 31 show the BEM phase execution times with source-level software pipelining applied. The rest of the tables show the phase timings for the C and Fortran BEM implementations on several sequential machines.
### Table 18: Base MPL - 16K PE MP-1

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>4.8</td>
<td>15.5</td>
<td>35.3</td>
<td>83.4</td>
<td>126.3</td>
</tr>
<tr>
<td>Inversion</td>
<td>47.4</td>
<td>273.6</td>
<td>805.0</td>
<td>1767.7</td>
<td>3287.7</td>
</tr>
<tr>
<td>MatMult</td>
<td>0.8</td>
<td>4.8</td>
<td>14.0</td>
<td>30.7</td>
<td>57.0</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>0.3</td>
<td>1.0</td>
<td>2.4</td>
<td>4.5</td>
<td>8.9</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.2</td>
<td>0.5</td>
<td>1.1</td>
<td>1.8</td>
<td>2.6</td>
</tr>
<tr>
<td>I/O</td>
<td>4.1</td>
<td>8.3</td>
<td>12.0</td>
<td>16.0</td>
<td>20.2</td>
</tr>
<tr>
<td>Total</td>
<td>57.6</td>
<td>303.6</td>
<td>869.7</td>
<td>1904.0</td>
<td>3502.8</td>
</tr>
</tbody>
</table>

### Table 19: Base MPL - 4K PE MP-2

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>4.9</td>
<td>18.4</td>
<td>41.3</td>
<td>94.5</td>
<td>145.2</td>
</tr>
<tr>
<td>Inversion</td>
<td>29.0</td>
<td>186.9</td>
<td>580.4</td>
<td>1314.7</td>
<td>2498.0</td>
</tr>
<tr>
<td>MatMult</td>
<td>1.0</td>
<td>6.1</td>
<td>18.4</td>
<td>40.8</td>
<td>76.6</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>0.2</td>
<td>0.9</td>
<td>2.2</td>
<td>4.6</td>
<td>12.0</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.1</td>
<td>0.4</td>
<td>0.8</td>
<td>1.3</td>
<td>2.0</td>
</tr>
<tr>
<td>I/O</td>
<td>3.6</td>
<td>7.4</td>
<td>10.7</td>
<td>14.1</td>
<td>18.2</td>
</tr>
<tr>
<td>Total</td>
<td>38.9</td>
<td>220.1</td>
<td>653.8</td>
<td>1470.0</td>
<td>2752.1</td>
</tr>
</tbody>
</table>
### Table 20: Pipelined LU Solve for Inversion - 16K PE MP-1

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>4.8</td>
<td>15.5</td>
<td>35.4</td>
<td>83.4</td>
<td>126.3</td>
</tr>
<tr>
<td>Inversion</td>
<td>2.6</td>
<td>11.2</td>
<td>29.5</td>
<td>56.2</td>
<td>97.7</td>
</tr>
<tr>
<td>MatMult</td>
<td>0.9</td>
<td>4.8</td>
<td>14.0</td>
<td>30.7</td>
<td>57.0</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>0.3</td>
<td>1.0</td>
<td>2.4</td>
<td>4.5</td>
<td>8.9</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.2</td>
<td>0.5</td>
<td>1.1</td>
<td>1.8</td>
<td>2.6</td>
</tr>
<tr>
<td>I/O</td>
<td>4.2</td>
<td>8.3</td>
<td>12.0</td>
<td>15.9</td>
<td>20.2</td>
</tr>
<tr>
<td>Total</td>
<td>13.0</td>
<td>41.3</td>
<td>94.3</td>
<td>192.3</td>
<td>312.7</td>
</tr>
</tbody>
</table>

### Table 21: Pipelined LU Solve for Matrix Inversion - 4K PE MP-2

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>5.0</td>
<td>18.4</td>
<td>41.3</td>
<td>94.5</td>
<td>145.2</td>
</tr>
<tr>
<td>Inversion</td>
<td>2.6</td>
<td>12.5</td>
<td>35.5</td>
<td>68.9</td>
<td>122.2</td>
</tr>
<tr>
<td>MatMult</td>
<td>1.0</td>
<td>6.1</td>
<td>18.4</td>
<td>40.8</td>
<td>76.5</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>0.2</td>
<td>0.9</td>
<td>2.2</td>
<td>4.6</td>
<td>12.0</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.1</td>
<td>0.4</td>
<td>0.8</td>
<td>1.3</td>
<td>2.0</td>
</tr>
<tr>
<td>I/O</td>
<td>3.6</td>
<td>7.4</td>
<td>10.7</td>
<td>14.4</td>
<td>18.2</td>
</tr>
<tr>
<td>Total</td>
<td>12.6</td>
<td>45.6</td>
<td>106.9</td>
<td>224.5</td>
<td>376.2</td>
</tr>
</tbody>
</table>
Table 22: Gauss-Jordan Matrix Inversion - 16K PE MP-1

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>4.8</td>
<td>15.5</td>
<td>35.3</td>
<td>83.4</td>
<td>126.3</td>
</tr>
<tr>
<td>Inversion</td>
<td>0.6</td>
<td>2.7</td>
<td>7.7</td>
<td>16.6</td>
<td>30.9</td>
</tr>
<tr>
<td>MatMult</td>
<td>0.9</td>
<td>4.8</td>
<td>14.0</td>
<td>30.7</td>
<td>57.0</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>0.3</td>
<td>1.0</td>
<td>2.4</td>
<td>4.5</td>
<td>8.9</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.2</td>
<td>0.5</td>
<td>1.1</td>
<td>1.8</td>
<td>2.6</td>
</tr>
<tr>
<td>I/O</td>
<td>4.7</td>
<td>8.9</td>
<td>11.9</td>
<td>16.0</td>
<td>20.3</td>
</tr>
<tr>
<td>Total</td>
<td>11.5</td>
<td>33.4</td>
<td>72.5</td>
<td>152.9</td>
<td>246.0</td>
</tr>
</tbody>
</table>

Table 23: Gauss-Jordan Matrix Inversion - 4K PE MP-2

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>5.0</td>
<td>18.4</td>
<td>41.3</td>
<td>94.5</td>
<td>145.2</td>
</tr>
<tr>
<td>Inversion</td>
<td>0.5</td>
<td>3.0</td>
<td>8.9</td>
<td>19.9</td>
<td>37.6</td>
</tr>
<tr>
<td>MatMult</td>
<td>1.0</td>
<td>6.1</td>
<td>18.4</td>
<td>40.8</td>
<td>76.6</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>0.2</td>
<td>0.9</td>
<td>2.2</td>
<td>4.6</td>
<td>12.1</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.1</td>
<td>0.4</td>
<td>0.8</td>
<td>1.3</td>
<td>2.0</td>
</tr>
<tr>
<td>I/O</td>
<td>3.7</td>
<td>7.4</td>
<td>10.7</td>
<td>14.3</td>
<td>18.2</td>
</tr>
<tr>
<td>Total</td>
<td>10.6</td>
<td>36.1</td>
<td>82.3</td>
<td>175.5</td>
<td>291.8</td>
</tr>
</tbody>
</table>
### Table 24: Simultaneous LU Solve for Matrix Inversion - 16K PE MP-1

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>4.8</td>
<td>15.5</td>
<td>35.3</td>
<td>83.4</td>
<td>126.3</td>
</tr>
<tr>
<td>Inversion</td>
<td>0.7</td>
<td>2.8</td>
<td>7.0</td>
<td>14.1</td>
<td>25.4</td>
</tr>
<tr>
<td>MatMult</td>
<td>0.9</td>
<td>4.8</td>
<td>14.0</td>
<td>30.7</td>
<td>57.0</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>0.3</td>
<td>1.0</td>
<td>2.4</td>
<td>4.5</td>
<td>8.9</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.2</td>
<td>0.5</td>
<td>1.1</td>
<td>1.8</td>
<td>2.6</td>
</tr>
<tr>
<td>I/O</td>
<td>4.0</td>
<td>8.3</td>
<td>12.0</td>
<td>15.9</td>
<td>22.0</td>
</tr>
<tr>
<td>Total</td>
<td>10.9</td>
<td>32.8</td>
<td>71.8</td>
<td>150.3</td>
<td>242.2</td>
</tr>
</tbody>
</table>

### Table 25: Simultaneous LU Solve for Matrix Inversion - 4K PE MP-2

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>4.9</td>
<td>18.4</td>
<td>41.3</td>
<td>94.5</td>
<td>145.3</td>
</tr>
<tr>
<td>Inversion</td>
<td>0.5</td>
<td>2.6</td>
<td>7.4</td>
<td>15.8</td>
<td>28.8</td>
</tr>
<tr>
<td>MatMult</td>
<td>1.0</td>
<td>6.1</td>
<td>18.4</td>
<td>40.9</td>
<td>76.6</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>0.2</td>
<td>0.9</td>
<td>2.2</td>
<td>4.6</td>
<td>12.0</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.1</td>
<td>0.4</td>
<td>0.8</td>
<td>1.3</td>
<td>2.0</td>
</tr>
<tr>
<td>I/O</td>
<td>3.7</td>
<td>7.4</td>
<td>10.7</td>
<td>16.0</td>
<td>19.4</td>
</tr>
<tr>
<td>Total</td>
<td>10.5</td>
<td>35.8</td>
<td>80.9</td>
<td>173.1</td>
<td>284.2</td>
</tr>
</tbody>
</table>
Table 26: Transformation of Parallel Switch - 16K PE MP-1

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>1.2</td>
<td>4.5</td>
<td>10.0</td>
<td>17.9</td>
<td>27.9</td>
</tr>
<tr>
<td>Inversion</td>
<td>0.7</td>
<td>2.6</td>
<td>7.0</td>
<td>14.1</td>
<td>25.4</td>
</tr>
<tr>
<td>MatMult</td>
<td>0.9</td>
<td>4.8</td>
<td>14.0</td>
<td>30.7</td>
<td>57.0</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>0.3</td>
<td>1.0</td>
<td>2.4</td>
<td>4.5</td>
<td>8.9</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.2</td>
<td>0.5</td>
<td>1.1</td>
<td>1.8</td>
<td>2.6</td>
</tr>
<tr>
<td>I/O</td>
<td>4.1</td>
<td>8.3</td>
<td>11.9</td>
<td>16.0</td>
<td>20.3</td>
</tr>
<tr>
<td>Total</td>
<td>7.3</td>
<td>21.8</td>
<td>46.4</td>
<td>84.9</td>
<td>142.2</td>
</tr>
</tbody>
</table>

Table 27: Transformation of Parallel Switch - 4K PE MP-2

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>1.5</td>
<td>5.8</td>
<td>12.9</td>
<td>23.3</td>
<td>36.2</td>
</tr>
<tr>
<td>Inversion</td>
<td>0.6</td>
<td>2.6</td>
<td>7.4</td>
<td>15.8</td>
<td>28.8</td>
</tr>
<tr>
<td>MatMult</td>
<td>1.0</td>
<td>6.1</td>
<td>18.4</td>
<td>40.8</td>
<td>76.6</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>0.2</td>
<td>0.9</td>
<td>2.2</td>
<td>4.6</td>
<td>12.0</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.1</td>
<td>0.4</td>
<td>0.8</td>
<td>1.3</td>
<td>2.0</td>
</tr>
<tr>
<td>I/O</td>
<td>3.6</td>
<td>7.7</td>
<td>10.8</td>
<td>14.4</td>
<td>18.1</td>
</tr>
<tr>
<td>Total</td>
<td>7.1</td>
<td>23.5</td>
<td>52.5</td>
<td>100.2</td>
<td>173.7</td>
</tr>
</tbody>
</table>
Table 28: Hand-coded Common Subexpression Elimination - 16K PE MP-1

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>0.9</td>
<td>3.2</td>
<td>7.1</td>
<td>12.7</td>
<td>19.9</td>
</tr>
<tr>
<td>Inversion</td>
<td>0.7</td>
<td>2.7</td>
<td>7.0</td>
<td>14.1</td>
<td>25.4</td>
</tr>
<tr>
<td>MatMult</td>
<td>0.9</td>
<td>4.8</td>
<td>14.0</td>
<td>30.7</td>
<td>57.0</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>0.3</td>
<td>1.0</td>
<td>2.4</td>
<td>4.5</td>
<td>7.5</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.2</td>
<td>0.5</td>
<td>1.1</td>
<td>1.8</td>
<td>2.6</td>
</tr>
<tr>
<td>I/O</td>
<td>4.0</td>
<td>8.3</td>
<td>11.9</td>
<td>15.9</td>
<td>20.4</td>
</tr>
<tr>
<td>Total</td>
<td>6.9</td>
<td>20.4</td>
<td>43.5</td>
<td>79.6</td>
<td>132.9</td>
</tr>
</tbody>
</table>

Table 29: Hand-coded Common Subexpression Elimination - 4K PE MP-2

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>1.1</td>
<td>4.2</td>
<td>9.4</td>
<td>17.0</td>
<td>26.6</td>
</tr>
<tr>
<td>Inversion</td>
<td>0.6</td>
<td>2.6</td>
<td>7.4</td>
<td>15.8</td>
<td>28.8</td>
</tr>
<tr>
<td>MatMult</td>
<td>1.0</td>
<td>6.1</td>
<td>18.4</td>
<td>40.8</td>
<td>76.5</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>0.2</td>
<td>0.9</td>
<td>2.2</td>
<td>4.6</td>
<td>9.0</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.1</td>
<td>0.4</td>
<td>0.8</td>
<td>1.3</td>
<td>2.0</td>
</tr>
<tr>
<td>I/O</td>
<td>3.6</td>
<td>7.4</td>
<td>10.7</td>
<td>14.2</td>
<td>18.1</td>
</tr>
<tr>
<td>Total</td>
<td>6.7</td>
<td>21.6</td>
<td>49.0</td>
<td>93.7</td>
<td>161.0</td>
</tr>
</tbody>
</table>
Table 30: Source-level Software Pipelining - 16K PE MP-1

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>0.9</td>
<td>3.2</td>
<td>7.1</td>
<td>12.7</td>
<td>19.9</td>
</tr>
<tr>
<td>Inversion</td>
<td>0.6</td>
<td>2.7</td>
<td>6.3</td>
<td>12.7</td>
<td>22.9</td>
</tr>
<tr>
<td>MatMult</td>
<td>0.8</td>
<td>4.4</td>
<td>12.7</td>
<td>27.6</td>
<td>51.0</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>0.3</td>
<td>0.9</td>
<td>2.1</td>
<td>3.9</td>
<td>6.6</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.2</td>
<td>0.5</td>
<td>1.0</td>
<td>1.7</td>
<td>2.4</td>
</tr>
<tr>
<td>I/O</td>
<td>4.0</td>
<td>8.2</td>
<td>12.1</td>
<td>15.9</td>
<td>20.3</td>
</tr>
<tr>
<td>Total</td>
<td>6.8</td>
<td>19.7</td>
<td>41.4</td>
<td>74.5</td>
<td>123.1</td>
</tr>
</tbody>
</table>

Table 31: Source-level Software Pipelining - 4K PE MP-2

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>1.2</td>
<td>4.2</td>
<td>9.5</td>
<td>17.1</td>
<td>26.6</td>
</tr>
<tr>
<td>Inversion</td>
<td>0.5</td>
<td>2.3</td>
<td>6.4</td>
<td>13.4</td>
<td>24.3</td>
</tr>
<tr>
<td>MatMult</td>
<td>0.8</td>
<td>4.6</td>
<td>13.5</td>
<td>29.7</td>
<td>55.5</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>0.2</td>
<td>0.7</td>
<td>1.8</td>
<td>3.5</td>
<td>7.1</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.1</td>
<td>0.3</td>
<td>0.7</td>
<td>1.1</td>
<td>1.7</td>
</tr>
<tr>
<td>I/O</td>
<td>3.7</td>
<td>7.3</td>
<td>10.7</td>
<td>14.3</td>
<td>18.2</td>
</tr>
<tr>
<td>Total</td>
<td>6.5</td>
<td>19.4</td>
<td>42.5</td>
<td>79.2</td>
<td>133.4</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>512</td>
<td>768</td>
<td>1024</td>
<td>1280</td>
</tr>
<tr>
<td>----------------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>Integration</td>
<td>28.1</td>
<td>122.5</td>
<td>250.0</td>
<td>444.7</td>
<td>--</td>
</tr>
<tr>
<td>Inversion</td>
<td>11.8</td>
<td>96.5</td>
<td>327.1</td>
<td>782.5</td>
<td>--</td>
</tr>
<tr>
<td>MatMult</td>
<td>42.2</td>
<td>365.4</td>
<td>1198.2</td>
<td>2933.5</td>
<td>--</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>5.1</td>
<td>48.4</td>
<td>140.9</td>
<td>340.0</td>
<td>--</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.1</td>
<td>0.4</td>
<td>0.9</td>
<td>1.6</td>
<td>--</td>
</tr>
<tr>
<td>I/O</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.4</td>
<td>--</td>
</tr>
<tr>
<td>Total</td>
<td>87.3</td>
<td>633.5</td>
<td>1917.4</td>
<td>4502.8</td>
<td>--</td>
</tr>
</tbody>
</table>

Table 33: C Program - HP9000 (IKJ MATMULT)

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>26.9</td>
<td>119.7</td>
<td>251.2</td>
<td>438.6</td>
<td>--</td>
</tr>
<tr>
<td>Inversion</td>
<td>12.3</td>
<td>104.2</td>
<td>326.8</td>
<td>770.6</td>
<td>--</td>
</tr>
<tr>
<td>MatMult</td>
<td>18.9</td>
<td>159.9</td>
<td>486.5</td>
<td>1195.3</td>
<td>--</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>5.7</td>
<td>49.5</td>
<td>148.3</td>
<td>334.6</td>
<td>--</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.1</td>
<td>0.4</td>
<td>1.0</td>
<td>1.6</td>
<td>--</td>
</tr>
<tr>
<td>I/O</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.4</td>
<td>--</td>
</tr>
<tr>
<td>Total</td>
<td>64.0</td>
<td>433.9</td>
<td>1214.2</td>
<td>2741.0</td>
<td>--</td>
</tr>
</tbody>
</table>
Table 34: Fortran Program- RS6000

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>47.9</td>
<td>203.5</td>
<td>428.6</td>
<td>762.0</td>
<td>--</td>
</tr>
<tr>
<td>Inversion</td>
<td>3.6</td>
<td>26.5</td>
<td>87.6</td>
<td>204.4</td>
<td>--</td>
</tr>
<tr>
<td>MatMult</td>
<td>90.6</td>
<td>762.9</td>
<td>2425.2</td>
<td>5755.9</td>
<td>--</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>0.8</td>
<td>6.7</td>
<td>19.2</td>
<td>43.4</td>
<td>--</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.4</td>
<td>1.5</td>
<td>5.1</td>
<td>9.0</td>
<td>--</td>
</tr>
<tr>
<td>I/O</td>
<td>0.3</td>
<td>0.7</td>
<td>1.0</td>
<td>1.5</td>
<td>--</td>
</tr>
<tr>
<td>Total</td>
<td>143.1</td>
<td>1000.4</td>
<td>2961.7</td>
<td>6767.1</td>
<td>--</td>
</tr>
</tbody>
</table>

Table 35: Fortran Program- RS6000 (JKI MATMULT)

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>48.0</td>
<td>203.5</td>
<td>430.5</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Inversion</td>
<td>3.7</td>
<td>26.8</td>
<td>87.9</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>MatMult</td>
<td>9.7</td>
<td>79.8</td>
<td>249.5</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>0.8</td>
<td>6.8</td>
<td>18.7</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.3</td>
<td>1.5</td>
<td>4.8</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>I/O</td>
<td>0.3</td>
<td>0.7</td>
<td>1.1</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Total</td>
<td>62.5</td>
<td>317.6</td>
<td>787.7</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>
Table 36: C Program- RS6000

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>57.4</td>
<td>243.4</td>
<td>513.9</td>
<td>915.3</td>
<td>1430.3</td>
</tr>
<tr>
<td>Inversion</td>
<td>5.7</td>
<td>45.0</td>
<td>150.3</td>
<td>357.7</td>
<td>801.7</td>
</tr>
<tr>
<td>MatMult</td>
<td>110.6</td>
<td>870.0</td>
<td>2645.0</td>
<td>6303.0</td>
<td>12561.3</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>9.8</td>
<td>106.0</td>
<td>319.5</td>
<td>770.2</td>
<td>1625.4</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.1</td>
<td>0.5</td>
<td>1.1</td>
<td>4.3</td>
<td>5.1</td>
</tr>
<tr>
<td>I/O</td>
<td>0.5</td>
<td>1.0</td>
<td>1.4</td>
<td>1.9</td>
<td>2.3</td>
</tr>
<tr>
<td>Total</td>
<td>184.1</td>
<td>1265.8</td>
<td>3631.3</td>
<td>8352.3</td>
<td>16426.2</td>
</tr>
</tbody>
</table>

Table 37: C Program- RS6000 (IKJ MATMULT)

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>57.5</td>
<td>244.0</td>
<td>515.4</td>
<td>918.8</td>
<td>1431.4</td>
</tr>
<tr>
<td>Inversion</td>
<td>5.8</td>
<td>45.3</td>
<td>151.9</td>
<td>358.6</td>
<td>802.5</td>
</tr>
<tr>
<td>MatMult</td>
<td>11.8</td>
<td>95.6</td>
<td>296.2</td>
<td>704.4</td>
<td>1519.5</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>9.8</td>
<td>106.3</td>
<td>320.5</td>
<td>775.1</td>
<td>1641.4</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.1</td>
<td>0.5</td>
<td>1.1</td>
<td>4.2</td>
<td>5.5</td>
</tr>
<tr>
<td>I/O</td>
<td>0.5</td>
<td>0.9</td>
<td>1.3</td>
<td>1.8</td>
<td>2.3</td>
</tr>
<tr>
<td>Total</td>
<td>85.4</td>
<td>492.6</td>
<td>1285.5</td>
<td>2763.0</td>
<td>5402.5</td>
</tr>
</tbody>
</table>
Table 38: Fortran Program- DEC Alpha

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>15.0</td>
<td>63.4</td>
<td>133.4</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Inversion</td>
<td>3.2</td>
<td>24.0</td>
<td>85.3</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>MatMult</td>
<td>34.6</td>
<td>289.5</td>
<td>989.3</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>0.6</td>
<td>6.6</td>
<td>19.7</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.1</td>
<td>0.6</td>
<td>1.3</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>I/O</td>
<td>0.3</td>
<td>0.5</td>
<td>0.7</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Total</td>
<td>53.7</td>
<td>384.1</td>
<td>1228.4</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

Table 39: Fortran Program- DEC Alpha (JKI MATMULT)

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>15.1</td>
<td>63.5</td>
<td>133.6</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Inversion</td>
<td>3.3</td>
<td>24.5</td>
<td>87.5</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>MatMult</td>
<td>8.5</td>
<td>69.9</td>
<td>233.5</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>0.6</td>
<td>6.6</td>
<td>19.9</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.1</td>
<td>0.6</td>
<td>1.2</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>I/O</td>
<td>0.3</td>
<td>0.5</td>
<td>0.7</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Total</td>
<td>27.7</td>
<td>165.0</td>
<td>475.2</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>
Table 40: C Program- DEC Alpha

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>26.8</td>
<td>113.4</td>
<td>239.9</td>
<td>427.8</td>
<td>643.0</td>
</tr>
<tr>
<td>Inversion</td>
<td>4.5</td>
<td>45.7</td>
<td>159.4</td>
<td>378.3</td>
<td>742.2</td>
</tr>
<tr>
<td>MatMult</td>
<td>20.8</td>
<td>194.6</td>
<td>707.6</td>
<td>1691.3</td>
<td>3767.1</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>1.8</td>
<td>18.8</td>
<td>55.4</td>
<td>139.8</td>
<td>455.8</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.1</td>
<td>0.3</td>
<td>0.6</td>
<td>1.1</td>
<td>1.9</td>
</tr>
<tr>
<td>I/O</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.4</td>
<td>0.5</td>
</tr>
<tr>
<td>Total</td>
<td>54.1</td>
<td>372.9</td>
<td>1163.2</td>
<td>2638.7</td>
<td>5610.6</td>
</tr>
</tbody>
</table>

Table 41: C Program- DEC Alpha (IKJ MATMULT)

<table>
<thead>
<tr>
<th></th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>27.0</td>
<td>114.0</td>
<td>241.4</td>
<td>427.4</td>
<td>648.0</td>
</tr>
<tr>
<td>Inversion</td>
<td>4.6</td>
<td>45.9</td>
<td>159.3</td>
<td>377.8</td>
<td>739.1</td>
</tr>
<tr>
<td>MatMult</td>
<td>7.3</td>
<td>80.5</td>
<td>285.5</td>
<td>730.3</td>
<td>1482.2</td>
</tr>
<tr>
<td>LU Decomp</td>
<td>1.8</td>
<td>19.2</td>
<td>56.3</td>
<td>139.1</td>
<td>295.6</td>
</tr>
<tr>
<td>Iteration</td>
<td>0.1</td>
<td>0.3</td>
<td>0.6</td>
<td>1.1</td>
<td>1.8</td>
</tr>
<tr>
<td>I/O</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.4</td>
<td>0.5</td>
</tr>
<tr>
<td>Total</td>
<td>40.9</td>
<td>260.1</td>
<td>743.3</td>
<td>1676.2</td>
<td>3167.2</td>
</tr>
</tbody>
</table>
APPENDIX 2: BEM PROGRAM MAKEFILE

# For HP9000, set CC=gcc.
# For Dec Alpha, set FFLAGS="-O -assume nounderscore"
#
.SUFFIXES: .o .m .c

CFLAGS = -O
FFLAGS = -O
MPFLAGS = -Zq -Zn -nohprofile -Omax

COBJ = cargs.o cbem.o cdomainint.o cerr.o cexact.o cintegrate.o cinvert.o\
      cio.o cmatvec.o cmmult.o cshape.o csolver.o ctime.o cutil.o

FOBJ = fbem.o fmmult.o fcond.o fexact.o ctime.o
FOBJSLOW = fbem.o fmmultslow.o fcond.o fexact.o ctime.o

MSRC = args.m bem.m cond.m domainint.m dpumap.m err.m exact.m global.m \
        integrate.m invert.m io.m matvec.m mmmult.m shape.m solver.m \
        timer.m \util.m

MOBJ = args.o bem.o cond.o domainint.o dpumap.o err.o exact.o global.o \
       integrate.o invert.o io.o matvec.o mmmult.o shape.o solver.o \
       timer.o \util.o

mbem.mp1: $(MOBJ)
    mpl_cc $(MPFLAGS) $(MOBJ) -lm -o mbem.mp1 ; mplimit mbem.mp1
    pmem 16k

mbem.mp2: $(MSRC) bem.h
    mpl_cc -D_MP2 $(MPFLAGS) $(MSRC) -lm -o mbem.mp2 ; rm *.o ;
    mplimit mbem.mp2 pmem 64k

cbem: $(COBJ)
    $(CC) $(COBJ) -O -lm -o cbem

fbem: $(FOBJ)
    $(FC) -O $(FOBJ) -lm $(FORTLIBS) -o fbem

fbemenslow: $(FOBJSLOW)
    $(FC) -O $(FOBJSLOW) -lm $(FORTLIBS) -o fbemenslow

$(MOBJ): bem.h
$(COBJ): bemc.h

.m.o:
    mpl_cc $(MPFLAGS) -c $*.m
APPENDIX 3: SERIAL FORTRAN BEM SOURCE CODE

C***************************************************************
C* FILE: fbem.f
C*
C* Boundary Element Method (BEM) Application
C*
C* Jeffrey S. Clary
C*
C* (Author: Dr. Ambar Mitra)
C*
C* Iowa State University
C*
C* This is the Fortran BEM application provided by Dr. Ambar Mitra.
C* It has been instrumented to time its phases, and the matrix
C* multiplication routine has been removed to another source file
C* so that two versions of MMMULT can be tested, and the matrix
C* dimensions have been increased. Otherwise, it is practically
C* unchanged.
C***************************************************************

IMPLICIT DOUBLE PRECISION(A-H,O-Z)
COMMON /VARS/ PI
DIMENSION X(1280),Y(1280),IJK(6,1280)
DIMENSION NBDY(1280),IFLAG(1280)
DIMENSION X2(1280),Y2(1280),IJK2(3,1280)
DIMENSION TIN(1280),TINM1(1280),BCOND(1280),AH2(2560)
DIMENSION BCU(1280,1280),BCK(1280,1280)
DIMENSION DCU(1280,1280),DCK(1280,1280)
DIMENSION BC(1280,1280),DC(1280,1280)
DIMENSION IPVT(1280),WORK(2560)
DIMENSION XMAT(1280,1280),YMAT(1280,1280),ZMAT(1280,1280)
DIMENSION AMAT(1280,1280),BVEC(1280),BVAL(1280)
DIMENSION XALT(1280),YALT(1280)
NDIM1=1280
NDIM2=6
NDIM3=3
PI=4.0D0*ATAN(1.0D0)

C READING BOUNDARY INFORMATION

d_io_elapsed = 0.0D0
d_iter_elapsed = 0.0D0
total_elapsed = 0.0D0
elapsed = stopwatch(1)

elapsed = stopwatch(1)
READ(12,*) NP,NODES
C WRITE(13,200) NP,NODES
200 FORMAT(2X,I3,2X,I3)
DO 5 IR=1,NODES
   READ(12,*) X(IR),Y(IR),NBDY(IR),IFLAG(IR)
   C WRITE(13,250) X(IR),Y(IR),NBDY(IR),IFLAG(IR)
250 FORMAT(2X,2F9.4,2X,I3,2X,I3)
5 CONTINUE
READ(12,*) NK
C WRITE(13,200) NK
DO 10 IR=1,NK
   READ(12,*) (IJK(I,IR),I=1,NK)
   C WRITE(13,300) (IJK(I,IR),I=1,NK)
300 FORMAT(2X,6I4)
10 CONTINUE
C
C NFLAG IS THE NO. OF POINTS WITH IFLAG=1
C
READ(12,*) NFLAG
C WRITE(13,200) NFLAG
DO 15 IR=1,NFLAG
   C (XALT,YALT) ARE ALTERNATE COLLOCATION POINTS
   C
   READ(12,*) XALT(IR),YALT(IR)
   C WRITE(13,250) XALT(IR),YALT(IR)
15 CONTINUE
C
C READING DOMAIN INFORMATION
C
READ(12,*) NELX,NX
C WRITE(13,200) NELX,NX
DO 20 IR=1,NX
   READ(12,*) X2(IR),Y2(IR)
   C WRITE(13,250) X2(IR),Y2(IR)
20 CONTINUE
DO 25 IR=1,NELX
   READ(12,*) (IJK2(I,IR),I=1,3)
   C WRITE(13,300) (IJK2(I,IR),I=1,3)
25 CONTINUE
C
C NINT IS THE NUMBER OF TIME MARCHING STEPS
C DT IS THE TIME STEP
C IORDER IS THE ORDER OF TIME FINITE-DIFFERENCING
C
READ(12,*) NINT
READ(12,*) DT
READ(12,*) DIFFK
C
WRITE(13,500) NODES,NELX,NINT,DT,DIFFK
500 FORMAT('NODES= ',I3,2X,'NELX= ',I3,2X,'NINT= ',I2,2X,
& 'DT= ',F6.4,2X,'DIFFUSIVITY= ',F6.4,/) READ(12,*) NPR

elapsed = stopwatch(0)
d_io_elapsed = d_io_elapsed + elapsed
total_elapsed = total_elapsed + elapsed

900 format('Problem size = ', I7)
write(6,900) nodes

ICOUNT=0
FACTOR=1.0/(DT*DIFFK)

elapsed = stopwatch(1)
CALL MATVEC(NP,NODES,NELX,NK,NDIM1,NDIM2,NDIM3,UK,rJK2,
& NBDY,IFLAG,X,Y,X2,Y2,AH2,BCU,BCK,DCU,DCK,BC,DC,
& FACTOR,ICOUNT,XALT,YALT)
elapsed = stopwatch(0)
total_elapsed = total_elapsed + elapsed

901 format(F6.1)
write(6,901) elapsed

elapsed = stopwatch(1)
CALL INVRT(NDIM1, NODES, BCU, IPVT, WORK, AH2, XMAT)
elapsed = stopwatch(0)
total_elapsed = total_elapsed + elapsed
write(6,901) elapsed

C PRINT *, 'inversion done'
elapsed = stopwatch(1)
CALL MMMULT(NDIM1,NODES,NODES,NODES,BCU,BCK,YMAT)
CALL MMMULT(NDIM1,NODES,NODES,NELX,BCU,BC,ZMAT)
CALL MMMULT(NDIM1,NODEX,NODES,DCU,ZMAT,BCU)
CALL MMMULT(NDIM1,NELX,NODES,DCU,YMAT,BCK)
DO 50 I=1,NELX
   DO 45 J=1,NELX
      BC(I,J)=BCU(I,J)-DC(I,J)
   45 CONTINUE
50 CONTINUE
DO 60 I=1,NELX
   DO 55 J=1,NODES
      DC(I,J)=BCK(I,J)-DCK(I,J)
   55 CONTINUE
60 CONTINUE
DO 70 I=1,NELX
   DO 65 J=1,NELX
      IF (I .EQ. J) THEN
         AMAT(I,J)=2*PI+BC(I,J)
      END IF
   65 CONTINUE
70 CONTINUE
ELSE
  AMAT(I,J)=BC(I,J)
ENDIF

CONTINUE

elapsed = stopwatch(0)
total_elapsed = total_elapsed + elapsed
write(6,901) elapsed

C
C LU DECOMPOSITION OF AMAT
C THIS NEEDS TO BE DONE ONLY ONCE
C
elapsed = stopwatch(1)
CALL DECOMP(NDIM1,NELX,COND,IPVT,WORK,AMAT)
C
elapsed = stopwatch(0)
total_elapsed = total_elapsed + elapsed
write(6,901) elapsed

C
C INIT INSERTS THE INITIAL CONDITION
C
elapsed = stopwatch(1)
CALL INIT(NELX,X2,Y2,TINM1,IJK2,NDIM3)
C
elapsed = stopwatch(0)
total_elapsed = total_elapsed + elapsed
C
nint_double = 0.0D0
DO 130 NSTEP=1,NINT
  nint_double = nint_double + 1.0D0
  PRINT *, 'time marching step ',nstep
C
C BOUND INSERTS THE TIME DEPENDENT BOUNDARY CONDITIONS
C
elapsed = stopwatch(1)
CALL BOUND(NODES,X,Y,BCOND,IFLAG,NBDY)
C
elapsed = stopwatch(0)
total_elapsed = total_elapsed + elapsed
C
elapsed = stopwatch(1)
DO 85 I=1,NELX
  BVEC(I)=0.0
  DO 75 J=1,NODES
    BVEC(I)=BVEC(I)+DC(I,J)*BCOND(J)
  75 CONTINUE
  DO 80 J=1,NELX
    BVEC(I)=BVEC(I)+BC(I,J)*TINM1(J)
  80 CONTINUE
  PRINT *, 'cond= ',cond
  elapsed = stopwatch(0)
total_elapsed = total_elapsed + elapsed
write(6,901) elapsed

CONTINUE
CALL SOLVE(NDIML, NELX, BVEC, IPVT, AMAT)
C PRINT *, 'solution done'
DO 90 I=1, NELX
   TIN(I) = BVEC(I)
90 CONTINUE
DO 105 I=1, NODES
   BVAL(I) = 0.0
   DO 95 J=1, NODES
      BVAL(I) = BVAL(I) - YMAT(I, J) * BCOND(J)
   95 CONTINUE
   DO 100 J=1, NELX
      BVAL(I) = BVAL(I) + ZMAT(I, J) * (TIN(J) - TINM1(J))
   100 CONTINUE
105 CONTINUE

elapsed = stopwatch(0)
d_iter_elapsed = d_iter_elapsed + elapsed

NDIV = NSTEP/NPR
NOLD = NDIV*NPR
IF (NOLD .EQ. NSTEP) THEN
   elapsed = stopwatch(1)
   WRITE(13, 550)
550 FORMAT(/2X, 'DOMAIN SOLUTION'/)
   DO 110 II=1, NELX
      XX1 = X2(IJK2(1, II))
      XX2 = X2(IJK2(2, II))
      XX3 = X2(IJK2(3, II))
      YY1 = Y2(IJK2(1, II))
      YY2 = Y2(IJK2(2, II))
      YY3 = Y2(IJK2(3, II))
      XP = (XX1 + XX2 + XX3)/3.0
      YP = (YY1 + YY2 + YY3)/3.0
      TEMP = TIN(II)
      TIME = NSTEP*DT
      CALL EXACT(XP, YP, TIME, TMPEXCT)
      WRITE(13, 600) XP, YP, TEMP, TMPEXCT
600 FORMAT(2X, 'X= ', F8.4, 2X, 'Y= ', F8.4, 2X, 'T= ', F12.4,
         ' T EXACT= ', F12.4)
   110 CONTINUE
ELSE
   elapsed = stopwatch(0)
   WRITE(13, 650) I, NBDY(I), BVAL(I)
650 FORMAT(2X, 'NODE= ', I3, 2X, 'NBDY= ', I1, 3X, F8.4)
ENDIF
elapsed = stopwatch(1)
DO 125 I=1,NELX
   TINM1(I)=TIN(I)
125   CONTINUE

130 CONTINUE
elapsed = stopwatch(0)
total_elapsed = total_elapsed + elapsed
d_iter_elapsed = d_iter_elapsed + elapsed

d_iter_elapsed = d_iter_elapsed / nint_double
write(6,901) d_iter_elapsed
write(6,901) d_io_elapsed
write(6,901) total_elapsed

STOP
END

C SUBROUTINE MATVEC DRIVES ALL INTEGRATION ROUTINES
C
SUBROUTINE MATVEC(NP,NODES,NELX,NK,NDIM1,NDIM2,NDIM3,UK,IJK2,
                  NBDY,IFLAG,X,Y,X2,Y2,AH2,BCU,BCK,DCU,DCK,BC,DC,
                  FACTOR,ICOUNT,XALT,YALT)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
COMMON /VARS/ PI
DIMENSION UK(NDIM2,1),IJK2{NDIM3,1),NBDY(1),IFLAG(1)
DIMENSION X(1),Y(1),X2(1),Y2(1),AH2(1)
DIMENSION BCU(NDIM1,1),BCK(NDIM1,1)
DIMENSION DCU(NDIM1,1),DCK(NDIM1,1)
DIMENSION BC(NDIM1,1),DC(NDIM1,1)
DIMENSION XALT(1),YALT(1)

C COLLOCATING AT THE BOUNDARY AND DOMAIN NODES
C
NTOT=NODES+NELX
DO 100 I=1,NTOT
   IF (I .LE. NODES) THEN
      C (I .LE. NODES) CORRESPOND TO BOUNDARY COLLOCATION
      IF (IFLAG(I) .EQ. 0) THEN
         XP=X(I)
         YP=Y(I)
      ELSE
         ICOUNT=ICOUNT+1
      END IF
   ELSE
      XALT(I)=X(I)
      YALT(I)=Y(I)
      C DOUBLE-NODE COLLOCATION AT DIRICHLET-DIRICHLET CORNERS
      PRINT *, i, iflag(i)
      PRINT *, 'ADDITIONAL COLLOCATION POINT NEEDED'
      PRINT *, 'THE COORDINATE OF COLLOCATION POINT'
PRINT *, 'AND THE DIRICHLET CONDITION MUST'
PRINT *, 'INCLUDED IN THE DATA FILE'
XP=XALT(ICOUNT)
YP=YALT(ICOUNT)
ENDIF
C
C (I .GT. NODES) CORRESPOND TO DOMAIN COLLOCATION
C
ELSE
II=I-NODES
XX1=X2(IJK2(1,II))
XX2=X2(IJK2(2,II))
XX3=X2(IJK2(3,II))
YY1=Y2(IJK2(1,II))
YY2=Y2(IJK2(2,II))
YY3=Y2(IJK2(3,II))
XP=(XX1+XX2+XX3)/3.0
YP=(YY1+YY2+YY3)/3.0
ENDIF
C
C INTEGRATION ON THE BOUNDARY
C
CALL INT4(XP,YP,NK,AH2,NP,NODES,IJK,X,Y,NDIM2)
C
C (I .LE. NODES) MEANS BOUNDARY COLLOCATION
C
IF (I .LE. NODES) THEN
C
C CALCULATION OF RIGID BODY TERM
C
CC=0.0
DO 10 ICC=1,NODES
   CC=CC+AH2(ICC)
10 CONTINUE
C
PRINT *, i,cc
AH2(I)=-CC
C
THE FOLLOWING IS NECESSARY FOR EXTERNAL PROBLEMS
C
IF(AH2(I) .LT. 0.0) AH2(I)=AH2(I)+2*PI
C
BDRY INSERTS THE APPROPRIATE BOUNDARY CONDITIONS
C
BCU IS THE COEFFICIENT MATRIX MULTIPLYING THE UNKNOWNS ON
C THE BOUNDARY. SIMILARLY BCK IS THE MATRIX MULTIPLYING THE
C KNOWNS ON THE BOUNDARY. BCU AND BCK ARE THE MATRICES FOR
C BOUNDARY COLLOCATION.
C
CALL BDRY(BCU,BCK,AH2,NBDY,I,NODES,NDIM1)
ELSE
C (I .GT. NODES) MEANS DOMAIN COLLOCATION
C
II=I-NODES
C
DCU IS THE COEFFICIENT MATRIX MULTIPLYING THE UNKNOWNS ON
C THE BOUNDARY. SIMILARLY DCK IS THE MATRIX MULTIPLYING THE
C KNOWNS ON THE BOUNDARY. DCU AND DCK ARE THE MATRICES FOR
C DOMAIN COLLOCATION.
C
CALL BDRY(DCU,DCK,AH2,NBDY,II,NODES,NDIM1)
ENDIF
C
INTEGRATION OVER THE DOMAIN. INTD DRIVES DOMAIN
C INTEGRATION ROUTINES
C
CALL INTD(XP,YP,BC,DC,NELX,IJK2,X2,Y2,NDIM1,NDIM3,
& I,NODES,FACTOR)
C PRINT *, 'intd done ',i
100 CONTINUE
RETURN
END

SUBROUTINE INTD(XP,YP,BC,DC,NELX,IJK2,X2,Y2,NDIM1,NDIM3,
& I,NODES,FACTOR)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
DIMENSION BC(NDIM1,1),DC(NDIM1,1),IJK2(NDIM3,1),X2(1),Y2(1)
DO 10 J=1,NELX
   XX1=X2(IJK2(1,J))
   XX2=X2(IJK2(2,J))
   XX3=X2(IJK2(3,J))
   YY1=Y2(IJK2(1,J))
   YY2=Y2(IJK2(2,J))
   YY3=Y2(IJK2(3,J))

C CHKCASE DETERMINES THE LOCATION OF (XP,YP)
C WITH RESPECT TO THE TRIANGULAR ELEMENT
C
CALL CHKCASE(XP,YP,XX1,YY1,XX2,YY2,XX3,YY3,ICASE)
C
DOMINT Prepares data for analytic integration over
C constant triangular elements.
C THE OUTPUT VAL CONTAINS VALUE OF INTEGRAL
C
CALL DOMINT(XP,YP,XX1,YY1,XX2,YY2,XX3,YY3,ICASE,XINT)
C
(I .LE. NODES) MEANS BOUNDARY COLLOCATION
C
IF (I .LE. NODES) THEN
   BC(I,J)=XINT*FACTOR
END"
C(IN .GT. NODES) MEANS DOMAIN COLLOCATION
CELSE
   II=I-NODES
   DC(II,J)=XINT*FACTOR
ENDIF
10 CONTINUE
RETURN
END
C
C THIS SUBROUTINE DETERMINES THE CASE FOR INTEGRATION
C
SUBROUTINE CHKCASE(XP,YP,X1,Y1,X2,Y2,X3,Y3,ICASE)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
COMMON /VARS/ PI
SMALL=0.0001
VX1=X1-XP
VY1=Y1-YP
VX2=X2-XP
VY2=Y2-YP
VX3=X3-XP
VY3=Y3-YP
V1=SQRT(VX1*VX1+VY1*VY1)
IF (V1 .LT. SMALL) THEN
   ICASE=1
   RETURN
ELSE
ENDIF
V2=SQRT(VX2*VX2+VY2*VY2)
IF (V2 .LT. SMALL) THEN
   ICASE=2
   RETURN
ELSE
ENDIF
V3=SQRT(VX3*VX3+VY3*VY3)
IF (V3 .LT. SMALL) THEN
   ICASE=3
   RETURN
ELSE
ENDIF
ARG1=(VX1*VX2+VY1*VY2)/(V1*V2)
ARG2=(VX2*VX3+VY2*VY3)/(V2*V3)
ARG3=(VX1*VX3+VY1*VY3)/(V1*V3)
IF (ARG1 .LT. -1.0) THEN
   AL12=PI
ELSE
   IF (ARG1 .GT. 1.0) THEN
   AL12=0.0
"
ELSE
  AL12=ACOS(ARG1)
ENDIF
ENDIF
IF (ARG2 .LT. -1.0) THEN
  AL23=PI
ELSE
  IF (ARG2 .GT. 1.0) THEN
    AL23=0.0
  ELSE
    AL23=ACOS(ARG2)
  ENDIF
ENDIF
IF (ARG3 .LT. -1.0) THEN
  AL31=PI
ELSE
  IF (ARG3 .GT. 1.0) THEN
    AL31=0.0
  ELSE
    AL31=ACOS(ARG3)
  ENDIF
ENDIF
D1=ABS(AL12-PI)
IF (D1 .LT. SMALL) THEN
  ICASE=4
  RETURN
ELSE
ENDIF
D2=ABS(AL23-PI)
IF (D2 .LT. SMALL) THEN
  ICASE=5
  RETURN
ELSE
ENDIF
D3=ABS(AL31-PI)
IF (D3 .LT. SMALL) THEN
  ICASE=6
  RETURN
ELSE
ENDIF
D1=AL12
D2=AL23
D3=AL31
IF (D1 .LT. SMALL) THEN
  ICASE=7
  IF (V1 .GT. V2) ICASE=8
  RETURN
ELSE
ENDIF
IF (D2 .LT. SMALL) THEN
  ICASE=9
IF (V3 .GT. V2) ICASE=10
RETURN
ELSE
ENDIF
IF (D3 .LT. SMALL) THEN
   ICASE=11
   IF (V1 .GT. V3) ICASE=12
   RETURN
ELSE
ENDIF
DIFF=ABS(AL12+AL23+AL31-2*PI)
IF (DIFF .LT. SMALL) THEN
   ICASE=13
   RETURN
ELSE
ENDIF
DIFF=ABS(AL12-AL23-AL31)
IF (DIFF .LT. SMALL) THEN
   ICASE=14
   CR12=VX1*VY2-VY1*VX2
   IF (CR12 .GT. 0.0) ICASE=15
   RETURN
ELSE
ENDIF
DIFF=ABS(AL23-AL12-AL31)
IF (DIFF .LT. SMALL) THEN
   ICASE=16
   CR23=VX2*VY3-VX3*VY2
   IF (CR23 .GT. 0.0) ICASE=17
   RETURN
ELSE
ENDIF
DIFF=ABS(AL31-AL23-AL12)
IF (DIFF .LT. SMALL) THEN
   ICASE=18
   CR31=VX3*VY1-VX1*VY3
   IF (CR31 .GT. 0.0) ICASE=19
   RETURN
ELSE
ENDIF
END

C
C DOMINT PREPARES DATA FOR DOMAIN INTEGRATION
C
SUBROUTINE DOMINT(XP,YP,X1,Y1,X2,Y2,X3,Y3,ICASE,XINT)
C
C THE SUBROUTINE AINT CALCULATES DOMAIN INTEGRALS
C
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
GO TO (10,20,30,40,50,60,70,80,90,100,
110,120,130,140,150,160,170,180,190) ICASE
CONTINUE
CALL AINT(XP, YP, X2, Y2, X3, Y3, VAL)
XINT=VAL
GO TO 200

CONTINUE
CALL AINT(XP, YP, X3, Y3, X1, Y1, VAL)
XINT=VAL
GO TO 200

CONTINUE
CALL AINT(XP, YP, X1, Y1, X2, Y2, VAL)
XINT=VAL
GO TO 200

CONTINUE
CALL AINT(XP, YP, X2, Y2, X3, Y3, VAL)
XINT=VAL
CALL AINT(XP, YP, X3, Y3, X1, Y1, VAL)
XINT=XINT+VAL
GO TO 200

CONTINUE
CALL AINT(XP, YP, X1, Y1, X2, Y2, VAL)
XINT=VAL
CALL AINT(XP, YP, X3, Y3, X1, Y1, VAL)
XINT=XINT+VAL
GO TO 200

CONTINUE
CALL AINT(XP, YP, X1, Y1, X2, Y2, VAL)
XINT=VAL
CALL AINT(XP, YP, X2, Y2, X3, Y3, VAL)
XINT=XINT+VAL
GO TO 200

CONTINUE
CALL AINT(XP, YP, X2, Y2, X3, Y3, VAL)
XINT=VAL
CALL AINT(XP, YP, X1, Y1, X3, Y3, VAL)
XINT=XINT-VAL
GO TO 200

CONTINUE
CALL AINT(XP, YP, X1, Y1, X2, Y2, VAL)
XINT=VAL
CALL AINT(XP, YP, X3, Y3, X2, Y2, VAL)
XINT=XINT-VAL
GO TO 200

CONTINUE
CALL AINT(XP, YP, X3, Y3, X1, Y1, VAL)
XINT=VAL
CALL AINT(XP, YP, X3, Y3, X2, Y2, VAL)
XINT=XINT-VAL
GO TO 200

CONTINUE
CALL AINT(XP, YP, X1, Y1, X2, Y2, VAL)
XINT=VAL
CALL AINT(XP, YP, X1, Y1, X3, Y3, VAL)
XINT=XINT-VAL
GO TO 200

CONTINUE
CALL AINT(XP, YP, X3, Y3, X1, Y1, VAL)
XINT=VAL
CALL AINT(XP, YP, X2, Y2, X1, Y1, VAL)
XINT=XINT-VAL
GO TO 200
110 CONTINUE
CALL AINT(XP, YP, X2, Y2, X3, Y3, VAL)
XINT=VAL
CALL AINT(XP, YP, X2, Y2, X1, Y1, VAL)
XINT=XINT-VAL
GO TO 200
120 CONTINUE
CALL AINT(XP, YP, X1, Y1, X2, Y2, VAL)
XINT=VAL
CALL AINT(XP, YP, X3, Y3, X2, Y2, VAL)
XINT=XINT-VAL
GO TO 200
130 CONTINUE
CALL AINT(XP, YP, X1, Y1, X2, Y2, VAL)
XINT=VAL
CALL AINT(XP, YP, X2, Y2, X3, Y3, VAL)
XINT=XINT+VAL
CALL AINT(XP, YP, X3, Y3, X1, Y1, VAL)
XINT=XINT+VAL
GO TO 200
140 CONTINUE
CALL AINT(XP, YP, X1, Y1, X2, Y2, VAL)
XINT=VAL
CALL AINT(XP, YP, X3, Y3, X1, Y1, VAL)
XINT=XINT+VAL
CALL AINT(XP, YP, X2, Y2, X1, Y1, VAL)
XINT=XINT-VAL
GO TO 200
150 CONTINUE
CALL AINT(XP, YP, X1, Y1, X2, Y2, VAL)
XINT=VAL
CALL AINT(XP, YP, X3, Y3, X2, Y2, VAL)
XINT=XINT-VAL
CALL AINT(XP, YP, X1, Y1, X3, Y3, VAL)
XINT=XINT-VAL
GO TO 200
160 CONTINUE
CALL AINT(XP, YP, X1, Y1, X2, Y2, VAL)
XINT=VAL
CALL AINT(XP, YP, X3, Y3, X1, Y1, VAL)
XINT=XINT+VAL
CALL AINT(XP, YP, X3, Y3, X2, Y2, VAL)
XINT=XINT-VAL
GO TO 200
170 CONTINUE
CALL AINT(XP, YP, X2, Y2, X3, Y3, VAL)
XINT=VAL
CALL AINT(XP, YP, X2, Y2, X1, Y1, VAL)
XINT=XINT-VAL
CALL AINT(XP,YP,X1,Y1,X3,Y3,VAL)
XINT=XINT-VAL
GO TO 200

180 CONTINUE
CALL AINT(XP,YP,X1,Y1,X2,Y2,VAL)
XINT=VAL
CALL AINT(XP,YP,X2,Y2,X3,Y3,VAL)
XINT=XINT+VAL
CALL AINT(XP,YP,X1,Y1,X3,Y3,VAL)
XINT=XINT-VAL
GO TO 200

190 CONTINUE
CALL AINT(XP,YP,X3,Y3,X1,Y1,VAL)
XINT=VAL
CALL AINT(XP,YP,X3,Y3,X2,Y2,VAL)
XINT=XINT-VAL
CALL AINT(XP,YP,X2,Y2,X1,Y1,VAL)
XINT=XINT-VAL

200 CONTINUE
RETURN
END

C
C THIS SUBROUTINE DOES ANALYTIC DOMAIN INTEGRATIONS
C OVER CONSTANT TRIANGULAR ELEMENTS
C
SUBROUTINE AINT(XP,YP,XU,YU,XV,YV,VAL)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
COMMON /VARS/ PI
D1=XU-XP
D2=XV-XP
D3=YU-YP
D4=YV-YP
D5=XV-XU
D6=YV-YU
H=SQRT(D5*D5+D6*D6)
A=SQRT(D1*D1+D3*D3)
B=SQRT(D2*D2+D4*D4)
PHI=DATAN2(06,05)
ALPHA=PHI-.5*PI
IF ( A .EQ. 0.0) GO TO 5
IF ( B .EQ. 0.0) GO TO 8
ARG1=D3*COS(ALPHA)-D1*SIN(ALPHA)
ARG2=D1*COS(ALPHA)+D3*SIN(ALPHA)
THU=ATAN2(ARG1,ARG2)
ARG1=D4*COS(ALPHA)-D2*SIN(ALPHA)
ARG2=D2*COS(ALPHA)+D4*SIN(ALPHA)
THV=ATAN2(ARG1,ARG2)
ALOCA=LOG(A)
ALOGB=LOG(B)
GO TO 9
CONTINUE
ARG1=D4*C0S(ALPHA)-D2*SIN(ALPHA)
ARG2=D2*C0S(ALPHA)+D4*SIN(ALPHA)
THV=ATAN2(ARG1,ARG2)
THU=THV
ALOGA=0.0
ALOGB=LOG(B)
GO TO 9
CONTINUE
ARG1=D3*C0S(ALPHA)-D1*SIN(ALPHA)
ARG2=D1*C0S(ALPHA)+D3*SIN(ALPHA)
THU=ATAN2(ARG1,ARG2)
THV=THU
ALOGA=LOG(A)
ALOGB=0.0
CONTINUE
U=A*SIN(THU)
V=B*SIN(THU)
W=A*COS(THU)
VAL=TAN(THV)*(ALOGB-.5)-TAN(THU)*(ALOGA-.5)
Q2=(THV-THU)-(TAN(THV)-TAN(THU))
VAL=0.5*W*W*(VAL+Q2)
RETURN
END

C BDRY INSERTS BOUNDARY CONDITIONS
C
SUBROUTINE BDRY(A,B,AH2,NBDY,I,NODES,NDIM1)
IMPLICIT DOUBLE PRECISION(A-H,0-Z)
DIMENSION A(NDIM1,1),B(NDIM1,1)
DIMENSION AH2(1),NBDY(1)
DO 40 J=1,NODES
C NBDY=1 => PSI UNKNOWN
C NBDY=2 => PSIP UNKNOWN
GO TO (20,30) NBDY(J)
C
C INSERTING THE BOUNDARY CONDITIONS
C
20 CONTINUE
A(I,J)=AH2(J)
B(I,J)=AH2(J+NODES)
GO TO 40
30 CONTINUE
A(I,J)=AH2(J+NODES)
B(I,J)=AH2(J)
40 CONTINUE
RETURN
END
C
C INT4 DRIVES THE BOUNDARY INTEGRATION ROUTINES
C
SUBROUTINE INT4(XP, YP, NF, AH2, NP, NODES, IJK, X, Y, NIJK)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
COMMON /VARS/ PI
COMMON/PAN/ALOGA, ALOGB, H, U, V, THU, THV, W
DIMENSION AH2(1), G1(6), G1P(6)
DIMENSION IJK(NIJK,1), X(1), Y(1)
DO 5 I=1,2*NODES
   AH2(I)=0.0
5 CONTINUE
DO 20 1=1,NP
   CALL DRIVR(XP, YP, NK, G1, G1P, I, X, Y, IJK, NIJK)
   CALL SHAPE(NK, G1, H)
   CALL SHAPE(NK, G1P, H)
   DO 10 J=1,NK
      AH2(IJK(J,I))=AH2(IJK(J,I))-G1P(J)
      AH2(NODES+IJK(J,I))=AH2(NODES+IJK(J,I))+G1(J)
10 CONTINUE
20 CONTINUE
RETURN
END

C
C DRIVR PREPARES DATA FOR BOUNDARY INTEGRATION
C
SUBROUTINE DRIVR(XP, YP, NK, G1, G1P, I, X, Y, IJK, NIJK)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
COMMON /PAN/ALOGA, ALOGB, H, U, V, THU, THV, W
COMMON/VARS/ PI
DIMENSION G1(1), G1P(1)
DIMENSION IJK(NIJK,1), X(1), Y(1)
D1=X(IJK(1,I))-XP
D2=X(IJK(NK,I))-XP
D3=Y(IJK(1,I))-YP
D4=Y(IJK(NK,I))-YP
D5=X(IJK(NK,I))-X(IJK(1,I))
D6=Y(IJK(NK,I))-Y(IJK(1,I))
H=SQRT(D5*D5+D6*D6)
A=SQRT(D1*D1+D3*D3)
B=SQRT(D2*D2+D4*D4)
PHI=ATAN2(D3*COS(ALPHA)-D1*SIN(ALPHA), D1*COS(ALPHA)+D3*SIN(ALPHA))
IF ( A .EQ. 0.0 ) GO TO 5
IF ( B .EQ. 0.0 ) GO TO 8
ARG1=D3*COS(ALPHA)-D1*SIN(ALPHA)
ARG2=D1*COS(ALPHA)+D3*SIN(ALPHA)
THU=ATAN2(ARG1, ARG2)
ARG1=D4*COS(ALPHA)-D2*SIN(ALPHA)
ARG2=D2*COS(ALPHA)+D4*SIN(ALPHA)
THV=ATAN2(ARG1, ARG2)
ALOGA=LOG(A)
ALOGB=LOG(B)
GO TO 9
CONTINUE
ARG1=D4*COS(ALPHA)-D2*SIN(ALPHA)
ARG2=D2*COS(ALPHA)+D4*SIN(ALPHA)
THV=ATAN2(ARG1,ARG2)
THU=THV
ALOGA=0.0
ALOGB=LOG(B)
GO TO 9

CONTINUE
ARG1=D3*COS(ALPHA)-D1*SIN(ALPHA)
ARG2=D1*COS(ALPHA)+D3*SIN(ALPHA)
THU=ATAN2(ARG1,ARG2)
THV=THU
ALOGA=LOG(A)
ALOGB=0.0

CONTINUE
U=A*SIN(THU)
V=B*SIN(THV)
W=A*COS(THU)
DO 10 INK=1,NK
   CALL L1M(XINT,INK)
   G1(INK)=XINT
   CALL L3M(XINT,INK)
   G1P(INK)=XINT
10 CONTINUE
RETURN
END

SUBROUTINE L1M(XINT,INK)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
COMMON/PAN/ALOGA,ALOGB,H,U,V,THU,THV,W
XINT=0.0
M=INK-1
DO 20 NS1=1,INK
   NS=NS1-1
   MPS=M+NS
   XINT=XINT+NSGN(MPS)*BINOM(M,NS)*G1INT(NS)
20 CONTINUE
RETURN
END

SUBROUTINE L3M(XINT,INK)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
COMMON/PAN/ALOGA,ALOGB,H,U,V,THU,THV,W
XINT=0.0
M=INK-1
DO 20 NS1=1,INK
NS=NS1-1
MPS=M+NS
XINT=XINT+NSGN(MPS)*BINOM(M,NS)*G1PINT(NS)
20 CONTINUE
RETURN
END

C CALCULATES BINOMIAL COEFFICIENTS THAT APPEAR IN BOUNDARY INTEGRALS
C
FUNCTION BINOM(M,I)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
COMMON/PAN/ALOGA,ALOGB,H,U,V,THU,THV,W
COMB=1.0
DO 10 IM=1,M
    COMB=COMB*FLOAT(IM)
10 CONTINUE
DO 20 II=1,I
    COMB=COMB/FLOAT(II)
20 CONTINUE
MMI=M-I
DO 30 IMMI=1,MMI
    COMB=COMB/FLOAT(IMMI)
30 CONTINUE
COMB=COMB*PWR(U,MMI)
BINOM=COMB
RETURN
END

C CALCULATES (X)**N
C
FUNCTION PWR(X,N)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
PWR=1.0
IF ( N .NE. 0 ) PWR=X**N
RETURN
END

C CALCULATES (-1)**N
C
FUNCTION NSGN(I)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
ID2=I/2
II=ID2*2
NSGN=-1
IF(I.EQ.II)NSGN=1
RETURN
END

C CALCULATES A FUNCTION THAT APPEARS IN BOUNDARY INTEGRALS
FUNCTION GIINT(NS)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
COMMON/PAN/ALOGA,ALOGB,H,U,V,THU,THV,W
NSP1=NS+1
NSP2=NS+2
GIINT=(PWR(V,NSP1)*ALOGB-PWR(U,NSP1)*ALOGA
&-SFUNC(NSP2))/NSP1
RETURN
END

C CALCULATES A FUNCTION THAT APPEARS IN BOUNDARY INTEGRALS
C
FUNCTION SFUNC(K)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
COMMON/PAN/ALOGA,ALOGB,H,U,V,THU,THV,W
COMMON/VARS/PI
PINEG=-PI
KB2=K/2
KK=KB2*2
IF ( K .EQ. KK ) THEN
   L=K/2
   L2M1=2*L-1
   SFUNC=THV-THU
   IF ( ABS(W) .LT. 1.0E-6 ) SFUNC=0.0
   IF ( SFUNC .LT. PINEG ) SFUNC=SFUNC+2.0*PI
   IF ( SFUNC .GT. PI ) SFUNC=SFUNC-2.0*PI
   SFUNC=SFUNC*PWR(W,L2M1)
   SUM=0.0
   IF ( L .EQ. 0 ) GO TO 10
   DO 5 I=1,L
      I2M1=2*I-1
      L2MI2=2*(L-I)
      T=NSGN(I)*(PWR(V,I2M1)-PWR(U,I2M1))*PWR(W,L2MI2)/I2M1
      SUM=SUM+T
   5 CONTINUE
   10 SFUNC=NSGN(L)*(SFUNC+SUM)
ELSE
   L=(K-1)/2
   L2=2*L
   SFUNC=(ALOGB-ALOGA)*PWR(W,L2)
   SUM=0.0
   IF ( L .EQ. 0 ) GO TO 20
   DO 15 I=1,L
      I2=2*I
      L2MI2=2*(L-I)
      T=NSGN(I)*(PWR(V,I2)-PWR(U,I2))*PWR(W,L2MI2)/I2
      SUM=SUM+T
   15 CONTINUE
   20 SFUNC=NSGN(L)*(SFUNC+SUM)
ENDIF
RETURN
FUNCTION GIPINT(NS)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
COMMON/PAN/ALOGA,ALOGB,H,U,V,THU,THV,W
GIPINT=VFUNC(NS)
RETURN
END

FUNCTION VFUNC(K)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
COMMON/PAN/ALOGA,ALOGB,H,U,V,THU,THV,W
COMMON/VARS/ PI
PINEG=-PI
KB2=K/2
KK=2*KB2
IF ( K .EQ. KK ) THEN
  L=K/2
  L2=2*L
  VFUNC=THV-THU
  IF ( ABS(W) .LT. 1.0E-6 ) VFUNC=0.0
  IF ( VFUNC .LT. PINEG ) VFUNC=VFUNC+2.0*PI
  IF ( VFUNC .GT. PI ) VFUNC=VFUNC-2.0*PI
  VFUNC=VFUNC*PWR(W,L2)
  SUM=0.0
  IF ( L .EQ. 0 ) GO TO 10
  DO 5 I=1,L
    T=NSGN(I)*(PWR(V,I2M1)-PWR(U,I2M1))*PWR(W,L2MI21)/I2M1
    SUM=SUM+T
  5 CONTINUE
  VFUNC=NSGN(L)*(VFUNC+SUM)
ELSE
  L=(K-1)/2
  L2P1=2*L+1
  VFUNC=(ALOGB-ALOGA)*PWR(W,L2P1)
  SUM=0.0
  IF ( L .EQ. 0 ) GO TO 20
  DO 15 I=1,L
    T=NSGN(I)*(PWR(V,I2)-PWR(U,I2))*PWR(W,L2MI21)/I2
    SUM=SUM+T
  15 CONTINUE
  VFUNC=NSGN(L)*(VFUNC+SUM)
ENDIF
C INTRODUCES THE LAGRANGE POLYNOMIALS

SUBROUTINE SHAPE(NK, GG, H)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
DIMENSION GG(1), SH(8,8), GGD(8)
CALL SHFUNC(SH,H,NK)
DO 25 I=1,NK
   GGD(I)=0.0
   DO 20 J=1,NK
      GGD(I)=GGD(I)+GG(J)*SH(I,J)
   20 CONTINUE
25 CONTINUE
DO 30 I=1,NK
   GG(I)=GGD(I)
30 CONTINUE
RETURN
END

C EXPRESSIONS FOR LAGRANGE POLYNOMIALS

SUBROUTINE SHFUNC(SH,H,NK)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
DIMENSION SH(8,1)
NKM1=NK-1
GO TO (20,30,40) NKM1
20 SH(1,1)=1.0
   SH(1,2)=-1.0/H
   SH(2,1)=0.0
   SH(2,2)=-SH(1,2)
   RETURN
30 HSQ=H*H
   SH(1,1)=1.0
   SH(1,2)=-3.0/H
   SH(1,3)=2.0/HSQ
   SH(2,1)=0.0
   SH(2,2)=4.0/H
   SH(2,3)=-4.0/HSQ
   SH(3,1)=0.0
   SH(3,2)=-1.0/H
   SH(3,3)=2.0/HSQ
   RETURN
40 HSQ=H*H
   HCU=HSQ*H
   SH(1,1)=1.0
   SH(1,2)=-5.5/H
   SH(1,3)=9.0/HSQ
   SH(1,4)=-4.5/HCU
   SH(2,1)=0.0
   RETURN
SH(2,2) = 9.0/H
SH(2,3) = -22.5/HSQ
SH(2,4) = 13.5/HCU
SH(3,1) = 0.0
SH(3,2) = -4.5/H
SH(3,3) = 18.0/HSQ
SH(3,4) = -13.5/HCU
SH(4,1) = 0.0
SH(4,2) = 1.0/H
SH(4,3) = -4.5/HSQ
SH(4,4) = 4.5/HCU
RETURN
END

C
C SUBROUTINE FOR LU-DECOMPOSITION
C
SUBROUTINE DECOMP(NDIM1, N, COND, IPVT, WORK, A)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
DIMENSION IPVT(N), WORK(1), A(NDIM1, 1)
IPVT(N) = 1
IF(N .EQ. 1) GO TO 80
NM1 = N - 1
ANORM = 0.0
DO 10 J = 1, N
   T = 0.0
   DO 5 I = 1, N
      T = T + ABS(A(I, J))
   5 CONTINUE
   IF(T .GT. ANORM) ANORM = T
10 CONTINUE
DO 35 K = 1, NM1
   KP1 = K + 1
   M = K
   DO 15 I = KP1, N
      IF(ABS(A(I, K)) .GT. ABS(A(M, K))) M = I
15 CONTINUE
   IPVT(K) = M
   IF(M .NE. K) IPVT(N) = -IPVT(N)
   T = A(M, K)
   A(M, K) = A(K, K)
   A(K, K) = T
   IF(T .EQ. 0.0) GO TO 35
   DO 20 I = KP1, N
      A(I, K) = -A(I, K) / T
20 CONTINUE
DO 30 J = KP1, N
   T = A(M, J)
   A(M, J) = A(K, J)
   A(K, J) = T
   IF(T .EQ. 0.0) GO TO 30
30 CONTINUE
DO 25 I = KP1, N
A(I,J) = A(I,J) + A(I,K) * T

CONTINUE

CONTINUE

DO 50 K=1,N
   T=0.0
   IF(K .EQ. 1) GO TO 45
   KM1=K-1
   DO 40 I=1,KM1
      T=T+A(I,K)*WORK(I)
   40 CONTINUE
   45 EK=1.0
   IF(T .LT. 0.0) EK=-1.0
   IF(A(K,K) .EQ. 0.0) GO TO 90
   WORK(K)=-(EK+T)/A(K,K)
50 CONTINUE

DO 60 KB=1,NM1
   K=N-KB
   T=0.0
   KP1=K+1
   DO 55 I=KP1,N
      T=T+A(I,K)*WORK(K)
55 CONTINUE
   WORK(K)=T
   M=IPVT(K)
   IF(M .EQ. K) GO TO 60
   T=WORK(M)
   WORK(M)=WORK(K)
   WORK(K)=T
60 CONTINUE

YNORM=0.0
DO 65 I=1,N
   YNORM=YNORM+ABS(WORK(I))
65 CONTINUE

CALL SOLVE(NDIM1,N,WORK,IPVT,A)
ZNORM=0.0
DO 70 I=1,N
   ZNORM=ZNORM+ABS(WORK(I))
70 CONTINUE

COND=ANORM*ZNORM/YNORM
IF(COND .LT. 1.0) COND=1.0
RETURN
80 COND=1.0
IF(A(1,1) .NE. 0.0) RETURN
90 COND=1.0E+32
RETURN
END

C FOR BACK-SUBSTITUTION IN LINEAR SOLVER
C
C SUBROUTINE SOLVE(NDIM1,N,B,IPVT,A)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
DIMENSION B(1),IPVT(1),A(NDIM1,1)
IF( N .EQ. 1) GO TO 50
NM1=N-1
DO 20 K=1,NM1
   KP1=K+1
   M=IPVT(K)
   T=B(M)
   B(M)=B(K)
   B(K)=T
   DO 10 I=KP1,N
      B(I)=B(I)+A(I,K)*T
10 CONTINUE
20 CONTINUE
DO 40 KB=1,NM1
   KM1=N-KB
   K=KM1+1
   B(K)=B(K)/A(K,K)
   T=-B(K)
   DO 30 I=1,KM1
      B(I)=B(I)+A(I,K)*T
30 CONTINUE
40 CONTINUE
B(1)=B(1)/A(1,1)
RETURN
END

C
C INVRT INVERTS A MATRIX
C IN THIS SUBROUTINE AH2 AND XMAT ARE
C BEING USED AS INTERMEDIATE STORAGE
C
SUBROUTINE INVRT(NDIM1,N,A,IPVT,WORK,AH2,XMAT)
C
C THE ORIGINAL MATRIX IS OVERWRITTEN BY ITS INVERSE
C
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
DIMENSION A(NDIM1,1),IPVT(1),WORK(1)
DIMENSION AH2(1),XMAT(NDIM1,1)
CALL DECOMP(NDIM1,N,COND,IPVT,WORK,A)
C
PRINT *, 'cond in invert ',cond
DO 50 I=1,N
   DO 10 J=1,N
      AH2(J)=0.0
      IF (J .EQ. I) AH2(J)=1.0
10 CONTINUE
CALL SOLVE(NDIM1,N,AH2,IPVT,A)
DO 20 J=1,N
   XMAT(J,I)=AH2(J)
20 CONTINUE
RETURN
END
DO 100 J=1,N
   A(I,J) = XMAT(I,J)
  100 CONTINUE
  110 CONTINUE
RETURN
END
C* FILE: fmmult.f
C*
C* Boundary Element Method (BEM) Application
C* Jeffrey S. Clary
C* Iowa State University
C*
C* This file contains a matrix multiplication routine. This is the
C* cache-friendly version using the JKI loop order. Link with this
C* for fast performance.
C*******************************************************************************

C MMMULT MULTIPLIES A(N1 X N2) WITH B(N2 X N3) MATRIX
C AND STORES IT IN AB(N1 X N3)
C
SUBROUTINE MMMULT(NDIM1,N1,N2,N3,A,B,AB)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
DIMENSION A(NDIM1,1),B(NDIM1,1),AB(NDIM1,1)

DO 40 J=1,N3
  DO 50 I=1,N1
    AB(I,J)=0.0
  50 CONTINUE
40 CONTINUE

DO 30 J=1,N3
  DO 20 K=1,N2
    DO 10 I=1,N1
      AB(I,J)=AB(I,J)+A(I,K)*B(K,J)
    10 CONTINUE
  20 CONTINUE
30 CONTINUE
RETURN
END
C**********************************************************************
C* FILE: fmmultslow.f                                              *
C*                                                            *
C* Boundary Element Method (BEM) Application                    *
C* Jeffrey S. Clary                                           *
C* Iowa State University                                      *
C*                                                            *
C* This file contains a matrix multiplication routine. This is  *
C* cache-unfriendly version using the IJK loop order. Link with *
C* for slower performance.                                    *
C**********************************************************************

C
C MMMULT MULTIPLIES A(N1 X N2) WITH B(N2 X N3) MATRIX
C AND STORES IT IN AB(N1 X N3)
C
SUBROUTINE MMMULT(NDIM1,N1,N2,N3,A,B,AB)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
DIMENSION A(NDIM1,1),B(NDIM1,1),AB(NDIM1,1)
DO 30 I=1,N1
   DO 20 J=1,N3
      AB(I,J)=0.0
      DO 10 K=1,N2
         AB(I,J)=AB(I,J)+A(I,K)*B(K,J)
   10 CONTINUE
20 CONTINUE
30 CONTINUE
RETURN
END
C*******************************************************************************/
C* FILE: fcond.f */
C* */
C* Boundary Element Method (BEM) Application */
C* Jeffrey S. Clary */
C* Iowa State University */
C* */
C* This file contains functions to set the initial domain */
C* and the time-dependent boundary conditions. Note that this is */
C* problem-dependent information, and for other problem sets, these */
C* functions would have to be rewritten. For a production code, */
C* these functions could be replaced with ones that read values as */
C* input. */
C* */
C*******************************************************************************/

c

c subroutine for inserting initial condition
c
subroutine init(nelx,x2,y2,tinml,ijk2,ndim3)
IMPLICIT DOUBLE PRECISION(A-H,0-Z)
dimension x2(1),y2(1),ijk2(ndim3,1),tinml(l),x(3),y(3)
pi=4.0*atan(1.0)
do 5 i=l,nelx
   xc=0.0
   yc=0.0
   do 2 j=l,3
      xc=xc+x2(ijk2(j,i))
      yc=yc+y2(ijk2(j,i))
  2 continue
   xc=xc/3.0
   yc=yc/3.0
   tinml(i)=sin(pi*xc)
5 continue
return
end

c

c subroutine for inserting boundary conditions
c
subroutine bound(nodes,x,y,bcond,i flag,nbdy)
IMPLICIT DOUBLE PRECISION(A-H,0-Z)
dimension x(1),y(1),bcond(1),i flag(1),nbdy(1)
do 5 i=l,nodes
   bcond(i)=0.0
5 continue
return
end
C*******************************************************************************/
C* FILE: fexact.f */
C* */
C* Boundary Element Method (BEM) Application */
C* Jeffrey S. Clary */
C* Iowa State University */
C* */
C* This file contains code for finding what the "exact" domain psi */
C* values should be. */
C*******************************************************************************/

subroutine EXACT(XP,YP,TIME,TMPEXCT)
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
COMMON /VARS/ PI
TMPEXCT = SIN(PI*XP)
return
end
APPENDIX 4: SERIAL C BEM SOURCE CODE

/* FILE: bemc.h */
/* */
/* This file contains global constant, variable and function defs. */
/* */
#include <stdio.h>
#include <math.h>

#define PAN 4 /* Maximum number nodes per element */
#define LEN 1280 /* Maximum number nodes or elements */
#define LEN2 2560 /* Double the max. number nodes */
#define ALTLEN 10 /* Max. number alternate collocation nodes */

extern double pi; /* 3.14... */
extern int debug; /* Debug output flag */
extern int echo; /* Input echo flag */
extern int mm_slow; /* Slow matmult flag */
extern int logging; /* Another debug flag */
extern FILE *logfile; /* File to write debug output to */
extern char errbuf[]; /* Buffer for error messages */

/* */
/* Global function defs. See the source. files for function */
/* explanations. */
/* */
void bound(int nodes, double x[], double y[], double bcond[],
    int iflag[], int nbdy[]);
void decomp(int na, int n, double *cond, int ipvt[], double work[],
    double a[LEN][LEN]);
void exact(double xp, double yp, double time, double *tmpexct);
void errexit(char *string);
void initdnt nelx, double x2[], double y2[], double tinml[],
    int ijk2[3][LEN], int ndim3);
void initread(int *np, int *nodes, int *nk, int *nflag,
    int *nx, int *nelx, int *nint, double *dt, double
    *diffk, int *npr, double x[], double y[], double x2[],
    double y2[], int ijk[PAN][LEN], int ijk2[3][LEN],
    int nbdy[], int iflag[], double tin[], double bcond[],
    double ah2[], double bcu[LEN][LEN],
    double bck[LEN][LEN], double dcu[LEN][LEN],
    double dck[LEN][LEN], int ipvt[], double work[],
    double xmat[LEN][LEN], double ymat[LEN][LEN],
    double zmat[LEN][LEN], double amat[LEN][LEN],
    double bvec[LEN], double bval(LEN],
    double xalt[], double yalt[]);
void initwrite(int np, int nodes, int nk, int nflag,
    int nx, int nelx, int nint, double dt, double diffk,
int npr, double x[], double y[], double x2[], double y2[], int ijk[LEN], int ijk2[3][LEN], int nbdy[], int iflag[], double tin[], double bcond[], double ah2[], double bcu[LEN], double bck[LEN][LEN], double dcu[LEN][LEN], double dck[LEN][LEN], double dc[LEN][LEN], int ipvt[], double work[], double xmat[LEN][LEN], double ymat[LEN][LEN], double zmat[LEN][LEN], double amat[LEN][LEN], double bvec[LEN], double bval[LEN], double xalt[], double yalt[];
void invrt(int ndiml, int n, double a[LEN][LEN], int ipvt[], double work[], double ah2[], double xmat[LEN][LEN]);
void invrt2(int n, double a[LEN][LEN], double xmat[LEN][LEN]);
void matvec(int np, int nodes, int nelx, int nk, int ndiml, int ndim2, int ndim3, int ijk[LEN], int ijk2[3][LEN], int nbdy[], int iflag[], double x[], double y[], double x2[], double y2[], double ah2[], double bcu[LEN][LEN], double bck[LEN][LEN], double dcu[LEN][LEN], double dc[LEN][LEN], double dck[LEN][LEN], double dc[LEN][LEN], double factor, int icount, double xalt[], double yalt[]);
void mmmult(int ndiml, int nl, int n2, int n3, double a[LEN][LEN], double b[LEN][LEN], double ab[LEN][LEN]);
double pwr(double, int);
void proc_argc(int argc, char **argv);
void solve(int na, int n, double b[], int ipvt[], double a[LEN][LEN]);
double stopwatch(int resetflag);
/* FILE: cbem.c */
/* */
/* Boundary Element Method (BEM) Application */
/* Jeffrey S. Clary */
/* Iowa State University */
/* */
/* This file contains the main program for the application. */
/**************************************************************/

#include "bemc.h"

double pi;
int debug = 0;
int echo = 0;
in mm_slow =0;
int logging = 0;
FILE *logfile;
char errbuf[80];
double x[LEN], y[LEN]; /* Boundary coordinates */
double x2[LEN], y2[LEN]; /* Domain element corner coords. */
int  ijk[PAN][LEN]; /* Boundary connectivity */
int  ijk2[3][LEN]; /* Domain connectivity */
int  nbdy[LEN]; /* Boundary node type */
    /* 1->phi unknown; 2->phip unknown */
int  iflag[LEN]; /* Double node flag */
double tin[LEN], tin[LEN]; /* Domain value now and previous */
double bcond[LEN]; /* Boundary value */
double ah2[LEN]; /* Scratch array */

/***************************************************************/
/* Coefficient matrices: */
/* */
/* bcu -- boundary collocation/boundary integration unknowns */
/* bck -- boundary collocation/boundary integration knowns */
/* dcu -- domain collocation/boundary integration unknowns */
/* dck -- domain collocation/boundary integration knowns */
/* bc -- boundary collocation/domain integration */
/* dc -- domain collocation/domain integration */
/***************************************************************/

double bcu[LEN][LEN];
double bck[LEN][LEN];
double dcu[LEN][LEN];
double dck[LEN][LEN];
double bc[LEN][LEN];
double dc[LEN][LEN];

/* Scratch arrays for LU decomposition and solving */
int ipvt[LEN];
double work[LEN2];

/* Scratch arrays for matrix manipulations. */
double xmat[LEN][LEN];
double ymat[LEN][LEN];
double zmat[LEN][LEN];
double amat[LEN][LEN];

double bvec[LEN]; /* Domain value (Psi) temporary array */
double bval[LEN]; /* Boundary value (Phi or PhiP) at nodes */
double xalt[ALTLEN]; /* Alternate collocation coords */
double yalt[ALTLEN];

/* The main program. This function corresponds roughly to the main */
/* program in the original Fortran BEM application. */
main (argc, argv)
{
    int ndim1 = LEN; /* Leading dimensions of arrays */
    int ndim2 = PAN; /* Not used in C program */
    int ndim3 = 3;

    int np; /* # boundary elements */
    int nodes; /* # boundary nodes */
    int nk; /* nodes per element */
    int nflag; /* # alt. collocation pts */
    int nelx; /* # domain elements */
    int nx; /* # domain nodes */
    int nint; /* # times steps */
    int npr; /* print interval */

    int nstep; /* Current time step */

double dt; /* delta of time step */
double diffk; /* Diffusivity */
double factor; /* 1.0 / (dt * diffk) */

double cond; /* Condition number */
double temp; /* Local temp for Psi */

double xx1, xx2, xx3; /* Temps for calculating centroid */
double yy1, yy2, yy3;
double xp, yp; /* Domain element centroids */

double time; /* Current time */
double tmpexct; /* "Exact" value of Psi */
int icount; /* Loop counters */
int i, j, ii;

double elapsed; /* Timing variables */
double io_elapsed = 0.0;
double iter_elapsed = 0.0;
double total_elapsed = 0.0;

proc_args(argc, argv);
pi = (double) 4.0 * atan ((double) 1.0);
/* GET RID OF ANY TIMER STARTUP TIME */
stopwatch(1);
stopwatch(1);

/* GEOMETRY I/O */
initread(&np, &nodes, &nk, &nflag, &nx, &nelx, &nint,
    &dt, &diffk, &npr, x, y, x2, y2, ijk, ijk2, nbdy, iflag,
    tin, bcond, ah2, bcu, bck, dcu, dck, bc, dc, ipvt, work,
    xmat, ymat, zmat, amat, bvec, bval, xalt, yalt);
if (echo)
    initwrite(np, nodes, nk, nflag, nx, nelx, nint, dt, diffk,
        npr, x, y, x2, y2, ijk, ijk2, nbdy, iflag, tin, bcond,
        ah2, bcu, bck, dcu, dck, bc, dc, ipvt, work, xmat, ymat,
        zmat, amat, bvec, bval, xalt, yalt);
elapsed = stopwatch(0);
total_elapsed += elapsed;
io_elapsed += elapsed;

fprintf(stderr, "Problem size = %d\n", nodes);
icount = 0;
factor = 1.0 / (dt * diffk);

stopwatch(1);

/* COLLOCATION AND INTEGRATION */
matvec(np, nodes, nelx, nk, ndim1, ndim2, ndim3, ijk, ijk2,
    nbdy, iflag, x, y, x2, y2, ah2, bcu, bck, dcu, dck,
    bc, dc, factor, icount, xalt, yalt);
elapsed = stopwatch(0);
total_elapsed += elapsed;
fprintf(stderr, "%10.11f\n", elapsed);

stopwatch(1);

/* MATRIX MANIPULATION */
/* BETTER INVERT USED BELOW
invrt(ndim1, nodes, bcu, ipvt, work, ah2, xmat);

/*
 invrt2(nodes, bcu, xmat);
 elapsed = stopwatch(0);
 total_elapsed += elapsed;
 fprintf(stderr, "%10.11f\n", elapsed);

 stopwatch(1);
 mmult(ndim1, nodes, nodes, nodes, bcu, bck, ymat);
 mmult(ndim1, nodes, nodes, nelx, bcu, bc, zmat);
 mmult(ndim1, nelx, nodes, nelx, dcu, zmat, bcu);
 mmult(ndim1, nelx, nodes, nodes, dcu, ymat, bck);

 for (i=0; i<nelx; i++)
 for (j=0; j<nelx; j++)
 bc[i][j] = bcu[i][j] - dc[i][j];

 for (i=0; i<nelx; i++)
 for (j=0; j<nodes; j++)
 dc[i][j] = bck[i][j] - dck[i][j];

 for (i=0; i<nelx; i++)
 for (j=0; j<nelx; j++)
 if (i == j)
 amat[i][j] = 2*pi + bc[i][j];
 else
 amat[i][j] = be[i][j];
 elapsed = stopwatch(0);
 total_elapsed += elapsed;
 fprintf(stderr, "%10.11f\n", elapsed);

 /* LU DECOMPOSITION OF AMAT -- DONE ONLY ONCE */
 stopwatch(1);
 decomp(ndim1, nelx, amat, &cond, ipvt, work, amat);
 elapsed = stopwatch(0);
 total_elapsed += elapsed;
 fprintf(stderr, "%10.11f\n", elapsed);

 /* INIT INSERTS THE INITIAL CONDITION */
 stopwatch(1);
 init(nelx, x2, y2, tinml, ijk2, ndim3);
 elapsed = stopwatch(0);
 io_elapsed += elapsed;
 total_elapsed += elapsed;

 for (nstep=1; nstep<=nint; nstep++)
 {
 stopwatch(1);
 /* BOUND INSERTS THE TIME-DEPENDENT BOUNDARY CONDITIONS */
 bound(nodes, x, y, bcond, iflag, nbdy);
elapsed = stopwatch(0);
io_elapsed += elapsed;
total_elapsed += elapsed;

stopwatch(1);
for (i=0; i<nelx; i++)
{
    bvec[i] = 0.0;
    for (j=0; j<nodes; j++)
        bvec[i] += dc[i][j] * bcond[j];
    for (j=0; j<nelx; j++)
        bvec[i] += bc[i][j] * tinml[j];
}
solve(ndiml, nelx, bvec, ipvt, amat);

for (i=0; i<nelx; i++)
    tin[i] = bvec[i];

for (i=0; i<nodes; i++)
{
    bval[i] = 0.0;
    for (j=0; j<nodes; j++)
        bval[i] -= ymat[i][j] * bcond[j];
    for (j=0; j<nelx; j++)
        bval[i] += zmat[i][j] * (tin[j] - tinml[j]);
}

iter_elapsed += stopwatch(0);

/* OUTPUT INTERMEDIATE RESULTS IF DESIRED */
if (nstep % npr == 0)
{
    stopwatch(1);

    /* PRINT THE DOMAIN VALUES */
    printf("Domain Solution\n");
    printf("Exact Solution\n");
    printf("X Y Temp Tmpexct\n");
    for (ii=0; ii<nelx; ii++)
    {
        xx1 = x2[ijk2[0][ii]];
        xx2 = x2[ijk2[1][ii]];
        xx3 = x2[ijk2[2][ii]];
        yy1 = y2[ijk2[0][ii]];
        yy2 = y2[ijk2[1][ii]];
        yy3 = y2[ijk2[2][ii]];
        xp = (xx1+xx2+xx3)/3.0;
        yp = (yy1+yy2+yy3)/3.0;
        temp = tin[ii];
        time = nstep * dt;
exact(xp, yp, time, &tmpexc);
    printf("%10.4lf %10.4lf %10.4lf %10.4lf\n", xp, yp, temp, tmpexc);
}

/* PRINT THE NODAL VALUES */
for (i=0; i<nodes; i++)
    printf("%d %d %0.4lf\n", i, nbdy[i], bval[i]);
elapsed = stopwatch(0);
total_elapsed += elapsed;
io_elapsed += elapsed;
}

stopwatch(1);
for (i=0; i<nelx; i++)
    tinml[i] = tin[i];
elapsed = stopwatch(0);
total_elapsed += elapsed;
it_iter_elapsed += elapsed;
}
it_iter_elapsed /= (double)nint;
fprintf(stderr, "%10.11f\n", it_iter_elapsed);

fprintf(stderr, "%10.11f\n", io_elapsed);
total_elapsed += it_iter_elapsed;
fprintf(stderr, "%10.11f\n", total_elapsed);
}
/* FILE: cargs.c */
/* */
/* Boundary Element Method (BEM) Application */
/* Jeffrey S. Clary */
/* Iowa State University */
/* */
/* This file contains code to process the command line arguments */
/* **************************************************************/
#include <strings.h>
#include "bemc.h"

/*-----------------------------------------------*/
/* This function inspects the arguments one at a time, and turns on */
/* the appropriate option flags. */
/*-----------------------------------------------*/

void proc_args(int argc, char **argv)
{
    while ((--argc) > 0)
    {
        argv++;
        if (!strcmp(*argv, "-debug"))
            debug = 1;
        else if (!strcmp(*argv, "-echo"))
            echo = 1;
        else if (!strcmp(*argv, "-mmslow"))
        {  
            mm_slow = 1;
            fprintf(stderr, "Slow matrix multiplication\n");
        }
        else if (!strcmp(*argv, "-log"))
        {  
            logging = 1;
            argv++;
            argc--;
            logfile = fopen(*argv, "w");
        }
        else
        {  
            sprintf(errbuf, "in proc_args unknown option \%s", *argv);
            errexit(errbuf);
        }
    }
}
/** File: cdomainint.c */
/** */
/** Boundary Element Method (BEM) Application */
/** Jeffrey S. Clary */
/** Iowa State University */
/** */
/** This file contains routines for domain integration. */
/** */
/** This code is adapted directly from Dr. Ambar Mitra's Fortran */
/** integration code. As such, it is not heavily documented. */
*******************************************************************************/
#include "bemc.h"
#include <math.h>

static double aloga, alogb;
static double h, u, v,
thu, thv, w;

*******************************************************************************/
double calc_alxx(double arg)
{
    if (arg < -1.0)
        return pi;
    if (arg > 1.0)
        return 0.0;
    return acos(arg);
}

/******************************************************************************/
/* This function determines the case for integration. */
/******************************************************************************/
int chkcase(double xp, double yp, double xl, double yl, double x2, double y2, double x3, double y3)
{
    double small = 0.0001;

    double vx1 = x1 - xp;
    double vy1 = y1 - yp;
    double vx2 = x2 - xp;
    double vy2 = y2 - yp;
    double vx3 = x3 - xp;
    double vy3 = y3 - yp;

    double v1, v2, v3;

    double arg1, arg2, arg3;
    double a112, a123, a131;

    v1 = sqrt(vx1*vx1 + vy1*vy1);
    if (v1 < small)
return 1;

v2 = sqrt(vx2*vx2 + vy2*vy2);
if (v2 < small)
    return 2;

v3 = sqrt(vx3*vx3 + vy3*vy3);
if (v3 < small)
    return 3;

all2 = calc_alxx((vx1*vx2 + vy1*vy2) / (v1*v2));
al23 = calc_alxx((vx2*vx3 + vy2*vy3) / (v2*v3));
al31 = calc_alxx((vx1*vx3 + vy1*vy3) / (v1*v3));

if (fabs(all2 - pi) < small)
    return 4;
if (fabs(all2 - pi) < small)
    return 5;
if (fabs(all2 - pi) < small)
    return 6;

if (all2 < small)
    if (v1 <= v2)
        return 7;
    else
        return 8;

if (al23 < small)
    if (v3 <= v2)
        return 9;
    else
        return 10;

if (all2 + al23 + al31 - 2*pi < small)
    return 13;

if (fabs(all2-al23-al31) < small)
    if ((vx1*vy2 - vy1*vx2) <= 0.0)
        return 14;
    else
        return 15;

if (fabs(al23-al12-al31) < small)
    if ((vx2*vy3 - vx3*vy2) <= 0.0)
        return 16;
else
    return 17;

if (fabs(al31-al23-all2) < small)
    if ((vx3*vyl - vxl*vy3) <= 0.0)
        return 18;
    else
        return 19;
}

/* */
/* This function does analytic domain integration over constant */
/* triangular elements. */
/* */
#define aint(xp, yp, xu, yu, xv, yv)
{
    double dl = xu - xp;
    double d2 = xv - xp;
    double d3 = yu - yp;
    double d4 = yv - yp;
    double d5 = xv - xu;
    double d6 = yv - yu;

    double a = sqrt(dl*dl + d3*d3);
    double b = sqrt(d2*d2 + d4*d4);

    double phi = atan2(d6, d5);
    double alpha = phi - 0.5*pi;
    double argl, arg2;
    double val, q2;
    h = sqrt(d5*d5 + d6*d6);

    if (a == 0.0) /* FORTRAN LABEL 5 */
    {
        argl = d4*cos(alpha) - d2*sin(alpha);
        arg2 = d2*cos(alpha) + d4*sin(alpha);
        thu = thv = atan2(argl, arg2);
        aloga = 0.0;
        alogb = log(b);
    }
    else if (b == 0.0) /* FORTRAN LABEL 8 */
    {
        arg1 = d3*cos(alpha) - d1*sin(alpha);
        arg2 = d1*cos(alpha) + d3*sin(alpha);
        thv = thu = atan2(arg1, arg2);
        aloga = log(a);
        alogb = 0.0;
    }
else
{
    arg1 = d3*cos(alpha) - d1*sin(alpha);
    arg2 = d1*cos(alpha) + d3*sin(alpha);
    thu = atan2(arg1, arg2);
    arg1 = d4*cos(alpha) - d2*sin(alpha);
    arg2 = d2*cos(alpha) + d4*sin(alpha);
    thv = atan2(arg1, arg2);
    aloga = log(a);
    alogb = log(b);
}

u = a*sin(thu);
V = b*sin(thu);
w = a*cos(thu);
val = tan(thv)*(alogb-0.5) - tan(thu)*(aloga-0.5);
q2 = (thv-thu) - (tan(thv) -tan(thu));

return 0.5 * w*w * (val+q2);
}

/*----------------------------------------------*/
/* This function prepares data for domain integration...aint() */
/* calculates domain integrals. */
/*----------------------------------------------*/
double domint(double xp, double yp, double x1, double y1, double x2,
               double y2, double x3, double y3, int icase)
{
    switch (icase)
    {
    case  1: return aint(xp, yp, x2, y2, x3, y3);
    case  2: return aint(xp, yp, x3, y3, x1, y1);
    case  3: return aint(xp, yp, x1, y1, x2, y2);
    case  4: return aint(xp, yp, x2, y2, x3, y3) +
              aint(xp, yp, x3, y3, x1, y1);
    case  5: return aint(xp, yp, x1, y1, x2, y2) +
              aint(xp, yp, x3, y3, x1, y1);
    case  6: return aint(xp, yp, x1, y1, x2, y2) +
              aint(xp, yp, x2, y2, x3, y3);
    case  7: return aint(xp, yp, x2, y2, x3, y3) -
              aint(xp, yp, x1, y1, x3, y3);
    case  8: return aint(xp, yp, x3, y3, x1, y1) -
              aint(xp, yp, x3, y3, x2, y2);
    case  9: return aint(xp, yp, x1, y1, x2, y2) -
              aint(xp, yp, x1, y1, x3, y3);
    case 10: return aint(xp, yp, x3, y3, x1, y1) -
              aint(xp, yp, x2, y2, x1, y1);
    case 11: return aint(xp, yp, x2, y2, x3, y3) -
              aint(xp, yp, x2, y2, x1, y1);
    case 12: return aint(xp, yp, x1, y1, x2, y2) -
              aint(xp, yp, x3, y3, x2, y2);
    }
case 13: return aint(xp, yp, xl, yl, x2, y2) +
aint(xp, yp, x2, y2, x3, y3) +
aint(xp, yp, x3, y3, xl, yl);

case 14: return aint(xp, yp, x2, y2, x3, y3) +
aint(xp, yp, x3, y3, xl, yl) -
aint(xp, yp, x2, y2, xl, yl);

case 15: return aint(xp, yp, xl, yl, x2, y2) -
aint(xp, yp, x3, y3, x2, y2) -
aint(xp, yp, xl, yl, x3, y3);

case 16: return aint(xp, yp, xl, yl, x2, y2) +
aint(xp, yp, x3, y3, x1, yl) -
aint(xp, yp, x3, y3, x2, y2);

case 17: return aint(xp, yp, x2, y2, x3, y3) -
aint(xp, yp, x2, y2, x1, yl) -
aint(xp, yp, x3, x1, yl, y3);

case 18: return aint(xp, yp, x1, yl, x2, y2) +
aint(xp, yp, x2, x3, y3) -
aint(xp, yp, x1, yl, x3, y3);

case 19: return aint(xp, yp, x3, y3, xl, yl) -
aint(xp, yp, x3, x2, y2) -
aint(xp, yp, x2, y2, xl, yl);

} 

void intd(double xp, double yp, double bc[LEN][LEN],
double dc[LEN][LEN], int nelx, int ijk2[3][LEN], double x2[],
double y2[], int ndim1, int ndim3, int i, int nodes, double factor) 
{
  int ii, j;
  int icase;
  double xx1, xx2, xx3;
  double yy1, yy2, yy3;
  double xint;

  for (j=0; j<nelx; j++)
  
  }
/* CONSTANT TRIANGULAR ELEMENTS. OUTPUT XINT CONTAINS */
/* VALUE OF INTEGRAL */
xint = domint(xp, yp, xx1, yy1, xx2, yy2, xx3, yy3, icase);

if (i<nodes)
    /* BOUNDARY COLLOCATION */
    {
    bc[i][j] = xint * factor;
    }
else
    /* DOMAIN COLLOCATION */
    {
    ii = i - nodes;
    dc[ii][j] = xint * factor;
    }
#include "bemc.h"

void errexit(char *string)
{
    fprintf(stderr, "Error: %s\n", string);
    exit(-1);
}
/* FILE: cexact.c */
/* */
/* Boundary Element Method (BEM) Application */
/* Jeffrey S. Clary */
/* Iowa State University */
/* */
/* This file contains code for finding what the "exact" domain psi */
/* values should be. */
/*****************************************************************************/

#include "bemc.h"

/*****************************************************************************/
/* This function calculates the "exact" domain PSI value for some */
/* (x,y) point at some time. It is intended as a check against the */
/* values calculated by the BEM process. For different problems */
/* the routine would have to be rewritten, or modified to read its */
/* values from input. */
/*****************************************************************************/

void exact(double xp, double yp, double time, double *tmpexct)
{
    *tmpexct = sin(pi*xp);
}
/* File: cintegrate.c */
/* */
/* Boundary Element Method (BEM) Application */
/* Jeffrey S. Clary */
/* Iowa State University */
/* */
/* This file contains functions for boundary integration. The */
/* functions were adapted directly from Dr. Ambar Mitra's code for */
/* symbolic integration, so they are not heavily documented. */
/* Numerical functions. */

#include "bemc.h"
#include <math.h>

static double aloga, alogb;
static double h, u, v, thu, thv, w;

double binom(int m, int i)
{
    double comb = 1.0;
    int im, ii, mmi, immi;

    for (im=1; im<=m; im++)
        comb *= (double) im;
    for (ii=1; ii<=i; ii++)
        comb /= (double) ii;
    mmi = m - i;
    for (immi=1; immi<=mmi; immi++)
        comb /= (double) immi;

    tmp = pwr(u,mmi);
    comb *= tmp;
    /*
    comb *= pwr(u,mmi);
    */

    return comb;
}
double sfunc(int k)
{
    double result;
    double sum = 0.0;
    double pineg = -pi;
    int l;
    int i, i2, i2ml;

    if (k == (k/2)*2)
    {
        l = k/2;
        result = thv-thu;
        if (fabs(w) < 1.0e-12) result = 0.0;
        if (result < pineg) result += 2.0*pi;
        if (result > pi) result -= 2.0*pi;

        result *= pwr(w,2*l-1);
        for (i=l; i<=l; i++)
        {
            i2ml = i*2-l;
            sum += nsgn(i)*(pwr(V,i2ml)-pwr(u,i2ml))*pwr(w,2*(1-i))/i2ml;
        }
    }
    else
    {
        l = (k-l)/2;
        result = (alogb-aloga)*pwr(w,2*l);
        for (i=l; i<=l; i++)
        {
            i2 = 2*i;
            sum += nsgn(i)*(pwr(V,i2)-pwr(u,i2))*pwr(w,2*(1-i))/i2;
        }
    }
    return nsgn(l) * (result + sum);
}

/*-----------------------------------------------*/
/*-----------------------------------------------*/

double vfuncdnt k)
{
    double result;
    double sum = 0.0;
    double pineg = -pi;
    int l;
    int i, i2, i2ml;

    if (k == (k/2)*2)
    {
        l = k/2;
        result = thv-thu;
        if (fabs(w) < 1.0e-12) result = 0.0;
        if (result < pineg) result += 2.0*pi;
        if (result > pi) result -= 2.0*pi;

        result *= pwr(w,2*l-1);
        for (i=l; i<=l; i++)
        {
            i2ml = i*2-l;
            sum += nsgn(i)*(pwr(V,i2ml)-pwr(u,i2ml))*pwr(w,2*(1-i))/i2ml;
        }
    }
    else
    {
        l = (k-l)/2;
        result = (alogb-aloga)*pwr(w,2*l);
        for (i=l; i<=l; i++)
        {
            i2 = 2*i;
            sum += nsgn(i)*(pwr(V,i2)-pwr(u,i2))*pwr(w,2*(1-i))/i2;
        }
    }
    return nsgn(l) * (result + sum);
}
result = thv-thu;
if (fabs(w) < 1.0e-12) result = 0.0;
if (result < pineg) result += 2.0*pi;
if (result > pi) result -= 2.0*pi;
result *= pwr(w,2*1);
for (i=1; i<1; i++)
{
    i2ml = 2*i-1;
    sum += nsgn(i)*(pwr(v,i2ml)-pwr(u,i2ml))*pwr(w,2*(1-i)+1)/i2ml;
}
else
{
    l = (k-1)/2;
    result = (alogb-aloga)*pwr(w,2*l+1);
    for (i=1;i<=1; i++)
    {
        i2 = 2*i;
        sum += nsgn(i)*(pwr(v,i2)-pwr(u,i2))*pwr(w,2*(1-i)+1)/i2;
    }
    return nsgn(l) * (result + sum);
}

double glint(int ns)
{
    return (pwr(v,ns+1)*alogb-pwr(u,ns+1)*aloga-sfunc(ns+2))/(ns+1);
}

double glpint(int ns)
{
    return vfunc(ns);
}

double llmdnt(int ink)
{
    double xint = 0.0;
    int m = ink-1;
    int mps;
    int ns;

    for (ns=0; ns<ink; ns++)
double 13m(int ink)
{
    double xint = 0.0;
    int m = ink - 1;
    int mps;
    int ns;

    for (ns=0; ns<ink; ns++)
    {
        mps = m + ns;
        xint += nsgn(mps) * binom(m,ns) * glpint(ns);
    }
    return xint;
}

/* */
/* Calculates integral of 1/r. */
/* */

#include "stdlib.h"
#include "math.h"
#include "stdio.h"
#endif

void drivr(double xp, double yp, int nk, double gl[], double glp[], int i, double x[], double y[], int ijk[NAN][LEN], int nijk)
{
    double d1, d2, d3, d4, d5, d6;
    double a, b;
    double alpha, phi;
    double argl, arg2;
    int ink;

    d1 = x[ijk[0][i]] - xp;
    d2 = x[ijk[nk-1][i]] - xp;
    d3 = y[ijk[0][i]] - yp;
    d4 = y[ijk[nk-1][i]] - yp;
    d5 = x[ijk[nk-1][i]] - x[ijk[0][i]];
    d6 = y[ijk[nk-1][i]] - y[ijk[0][i]];

    h = sqrt(d5*d5+d6*d6);
    a = sqrt(d1*d1+d3*d3);
    b = sqrt(d2*d2+d4*d4);
    phi = atan2(d6,d5);
    alpha = phi - (double) 0.5 * pi;

    if (a == 0.0)
{ /* LABEL 5 IN FORTRAN */
    arg1 = d4*cos(alpha)-d2*sin(alpha);
    arg2 = d2*cos(alpha)+d4*sin(alpha);
    thv = atan2(arg1, arg2);
    thu = thv;
    aloga = 0.0;
    alogb = log(b);
}
else if (b == 0.0)
{
    /* LABEL 8 IN FORTRAN */
    arg1 = d3*cos(alpha) - d1*sin(alpha);
    arg2 = d1*cos(alpha) + d3*sin(alpha);
    thu = atan2(arg1, arg2);
    thv = thv;
    aloga = log(a);
    alogb = log(b);
}
else
{
    arg1 = d3*cos(alpha) - d1*sin(alpha);
    arg2 = d1*cos(alpha) + d3*sin(alpha);
    thu = atan2(arg1, arg2);
    arg1 = d4*cos(alpha) - d2*sin(alpha);
    arg2 = d2*cos(alpha) + d4*sin(alpha);
    thv = atan2(arg1, arg2);
    aloga = log(a);
    alogb = log(b);
}
/* LABEL 9 FORTRAN */
u = a*sin(thu);
v = b*sin(thv);
w = a*cos(thu);

for (ink=0; ink<nk; ink++)
{
    gl[ink] = llm(ink+l);
    glp[ink] = 13m(ink+l);
}

/* This is the driver function for boundary integration. */
/* This is the driver function for boundary integration. */
void int4(double xp, double yp, int nk, double ah2[], int np,
    int nodes, int ijk[PAN][LEN], double x[], double y[],
    int nijk)
{
    double gl[6], glp[6];
int i, j;

for (i=0; i<2*nodes; i++)
    ah2[i] = 0.0;

for (i=0; i<np; i++)
    {
        drivr(xp, yp, nk, gl, glp, i, x, y, ijk, nijk);
        shape(nk, gl, h);
        shape(nk, glp, h);
        for (j=0; j<nk; j++)
            {
                ah2[ ijk[j][i] ] -= glp[j];
                ah2[ nodes+ijk[j][i] ] += g1[j];
            }
    }
FILE: cinvert.c

Boundary Element Method (BEM) Application
Jeffrey S. Clary
Iowa State University

This file contains various implementations of matrix inversion.

#include "bemc.h"

This function inverts a matrix by LU decomposing it and solving for the columns of the identity matrix.

NOTE the original matrix "a" is overwritten by its inverse.

void invrtdnt ndiml, int n, double a[LEN][LEN], int ipvt[],
double work[], double ah2[], double xmat[LEN][LEN])
{
    int i,j;
    double cond;

decomp(ndiml, n, &cond, ipvt, work, a);
    if (debug) fprintf(stderr, "Condition number = %lf in invrt\n");

    for (i=0; i<n; i++)
    {
        for (j=0; j<n; j++)
        {
            ah2[j] = 0.0;
            if (j == i)
                ah2[j] = 1.0;
        }
        solve(ndiml, n, ah2, ipvt, a);
        for (j=0; j<n; j++)
            xmat[j][i] = ah2[j];
    }

    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            a[i][j] = xmat[i][j];
}

This function implements inversion as Gauss-Jordan elimination.

void invrt2(int n, double a[LEN][LEN], double b[LEN][LEN])
{
    int ipvt[LEN];
    int i,j,k,m;
    double t;
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    if (i==j)
      b[i][j] = 1.0;
    else
      b[i][j] = 0.0;
for (i=0; i<n; i++)
{ /* FIND PIVOT */
  m = i;
  for (j=i+1; j<n; j++)
  {
    if (fabs(a[j][i]) > fabs(a[m][i]))
      m = j;
  }
  ipvt[i] = m;
  if (m != i)
  {
    /* SWAP PIVOT ROW OF A and B */
    for (j=0; j<n; j++)
    {
      t = a[i][j];
      a[i][j] = a[m][j];
      a[m][j] = t;
      t = b[i][j];
      b[i][j] = b[m][j];
      b[m][j] = t;
    }
  }
  /* Make pivot element 1 */
  t = a[i][i];
  for (j=0; j<n; j++)
  {
    a[i][j] /= t;
    b[i][j] /= t;
  }
  for (j=0; j<n; j++)
  {
    if (j != i)
    {
      t = a[j][i];
      for (k=0; k<n; k++)
      {
        a[j][k] -= t*a[i][k];
        b[j][k] -= t*b[i][k];
      }
    }
  }
\[ q = f(T)^+ \]
#include "bemc.h"

/* Read the problem geometry into scalar arrays. */
/* *---------------------------------------------------------------------*/
void initread(int *np, int *nodes, int *nk, int *nflag,
   int *nx, int *nelx, int *nint, double *dt, double *diffk,
   int *npr, double x[], double y[], double x2[], double y2[],
   int ijk[PAN][LEN], int ijk2[3][LEN], int nbdy[], int iflag[],
   double tin[], double bcond[], double ah2[], double bcu[LEN][LEN],
   double bck[LEN][LEN], double dcu[LEN][LEN], double dck[LEN][LEN],
   double bc[LEN][LEN], double dc[LEN][LEN], int ipvt[], double work[],
   double xmat[LEN][LEN], double ymat[LEN][LEN], double zmat[LEN][LEN],
   double amat[LEN][LEN], double bvec[LEN], double bval[LEN],
   double xalt[], double yalt[])
{
   int i, ir;

   /* READ BOUNDARY INFORMATION */
   /* NUMBER OF PANELS, NUMBER OF NODES */
   if (scanf("%d %d", np, nodes) != 2)
      errexit("in initread getting number of panels and nodes");
   /* FOR EACH NODE COORDS, BOUND TYPE, AND DOUBLE PT. FLAG */
   for (ir=0; ir<*nodes; ir++)
      if (scanf("%lf %lf %d %d", &x[ir], &y[ir], &nbdy[ir], &iflag[ir]) != 4)
         {
            sprintf(errbuf,
                "in initread getting (x,y,nbdy,iflag) (ir=%d)", ir);
            errexit(errbuf);
         }

   /* NUMBER OF NODES PER PANEL */
   if (scanf("%d", nk) != 1)
errexit("in initread getting nk");

/* CONNECTIVITY MATRIX */

for (ir=0; ir<*np; ir++)
for (i=0; i<*nk; i++)
  if (scanf("%d", &ijk[i][ir]) != 1)
    {  
      sprintf(errbuf, "in initread getting ijk[%d][%d]", i, ir);
      errexit(errbuf);
    }

/* NFLAG IS # OF PTS. WITH IFLAG SET */
if (scanf("%d", nflag) != 1)
  errexit("in initread getting nflag");

/* ALTERNATE COLLOCATION PTS. */
for (ir=0; ir<*nflag; ir++)
  if (scanf("%lf %lf", &xalt[ir], &yalt[ir]) != 2)
    {  
      sprintf(errbuf, "initread getting (xalt[%d], yalt[%d])", ir, ir);
      errexit(errbuf);
    }

/* READ DOMAIN INFORMATION */

/* */
if (scanf("%d %d", nelx, nx) != 2)
  errexit("initread getting nelx and nx");
for (ir=0; ir<*nx; ir++)
  if (scanf("%lf %lf", &x2[ir], &y2[ir]) != 2)
    {  
      sprintf(errbuf, "initread getting (x2[%d], y2[%d])", ir, ir);
      errexit(errbuf);
    }
for (ir=0; ir<*nelx; ir++)
  if (scanf("%d", &ijk2[i][ir]) != 1)
    {  
      sprintf(errbuf, "initread getting ijk[%d][%d]", i, ir);
      errexit(errbuf);
    }

/* NINT = NUMBER OF TIME-MARCHING STEPS */
/* DT = TIME STEP */
/* IORDER = ORDER OF TIME FINITE-DIFFERENCING */
if (scanf("%d %lf %lf", nint, dt, diffk) != 3)
  errexit("in initread getting nint, dt, and diffk");
if (scanf("%d", npr) != 1)
  errexit("in initread getting npr");
/* Write out the geometry from the scalar arrays. */

void initwrite(int np, int nodes, int nk, int nflag,
    int nx, int nelx, int nint, double dt, double diffk,
    int npr, double x[], double y[], double x2[], double y2[],
    int ijk[Pan][LEN], int ijk2[3][LEN], int nbdy[], int iflag[],
    double tin[], double bcond[], double ah2[], double bcu[LEN][LEN],
    double bck[LEN][LEN], double dcu[LEN][LEN], double dck[LEN][LEN],
    double bc[LEN][LEN], double dc[LEN][LEN], int ipvt[], double work[],
    double xmat[LEN][LEN], double ymat[LEN][LEN], double zmat[LEN][LEN],
    double bcu[LEN][LEN], double bvec[LEN], double bval[LEN],
    double xalt[], double yalt[])
{
    int i, ir;

    printf("Number panels = %8d\n", np);
    printf("Nodes/Panel = %8d\n", nodes);
    printf("# Iflags set = %8d\n", nflag);
    printf("Nx = %8d\n", nx);
    printf("Nelx = %8d\n", nelx);
    printf("Nint = %8d\n", nint);
    printf("Dt = %11.2f\n", dt);
    printf("Diffusivity = %11.2f\n", diffk);

    printf("\nBoundary Information\n\n");
    printf(" Node X Y Nbdy Iflag\n");
    for (ir=0; ir<nodes; ir++)
        printf("%4d %10.21f %11.21f %6d %8d\n", ir, x[ir], y[ir], nbdy[ir], iflag[ir]);

    printf("\nConnectivity Information\n\n");
    printf(" Panel Local Node # Global Node #\n");
    for (ir=0; ir<np; ir++)
        for (i=0; i<nk; i++)
            printf("%6d %10d %18d\n", ir, i, ijk[i][ir]);

    printf("\nAlternate collocation points\n\n");
    printf(" Point # X Y\n");
    for (ir=0; ir<nflag; ir++)
        printf("%5d %10.2lf %10.2lf\n", ir, xalt[ir], yalt[ir]);

    printf("\nDomain Information\n\n");
    printf(" Point2 # X2 Y2\n");
    for (ir=0; ir<nx; ir++)
        printf("%6d %13.2lf %10.2lf\n", ir, x2[ir], y2[ir]);

    printf("\nConnectivity Information\n\n");
    printf(" Panel Local Node # Global Node #\n");
    for (ir=0; ir<nelx; ir++)
        for (i=0; i<nk; i++)
            printf("%6d %10d %18d\n", ir, i, ijk[i][ir]);
for (i=0; i<3; i++)
    printf("%6d %10d %16d\n", ir, i, ijk2[i][ir]);
printf("\n");
}

/* This function inserts the time-dependent boundary conditions. */
void bound(int nodes, double x[], double y[], double bcond[],
           int iflag[], int nbdy[])
{
    int i;

    for (i=0; i<nodes; i++)
        bcond[i] = 0.0;
}

/* This function inserts the initial condition. */
void init(int nelx, double x2[], double y2[], double tinml[],
          int ijk2[3][LEN], int ndim3)
{
    int i;
    double x;

    for (i=0; i<nelx; i++)
    {
        x = x2[ijk2[0][i]] + x2[ijk2[1][i]] + x2[ijk2[2][i]];
        x /= 3.0;
        tinml[i] = sin(pi*x);
    }
}
/* FILE: cmatvec.c */
/* */
/* Boundary Element Method (BEM) Application */
/* Jeffrey S. Clary */
/* Iowa State University */
/* */
/* This file contains functions that drive the collocation process. */
/* **************************************************************************/ 

#include "bemc.h"

/* */
/* This function inserts boundary conditions. */
/* */
/* nbdy == 1 ==> psi unknown */
/* nbdy == 2 ==> psip unknown */
/* */

void bdry(double a[LEN][LEN], double b[LEN][LEN], double ah2[], int nbdy[], int i, int nodes, int ndiml)
{
    int j;

    for (j=0; j<nodes; j++)
        if (nbdy[j] == 1)
            { 
                a[i][j] = ah2[j];
                b[i][j] = ah2[j+nodes];
            }
        else
            { 
                a[i][j] = ah2[j+nodes];
                b[i][j] = ah2[j];
            }
}

/* */
/* This is the driver function for collocation. */
/* */

void matvec(int np, int nodes, int nelx, int nk, int ndiml, int ndim2, int ndim3, int ijk[LEN][LEN], int ijk2[3][LEN], int nbdy[], int iflag[], double x[], double y[], double x2[], double y2[], double ah2[], double bcu[LEN][LEN], double bck[LEN][LEN], double dcu[LEN][LEN], double dck[LEN][LEN], double bc[LEN][LEN], double dc[LEN][LEN], double factor, int icount, double xalt[], double yalt[])
{
    int i, ii, icc;
    int ntot;
    int
double cc;
double xx1, xx2, xx3;
double yy1, yy2, yy3;
double xp, yp;

/* COLLOCATING AT BOUNDARY AND DOMAIN NODES */
ntot = nodes + nelx;
for (i=0; i<ntot; i++)
{
    if (i<nodes)
        /* BOUNDARY COLLOCATION */
        {
            if (iflag[i] == 0)
                {
                    xp = X[i];
                    yp = y[i];
                }
            else
                /* DOUBLE-NODE COLLOC. AT DIRECHLET-DIRECHLET CORNER */
                {
                    printf("Additional collocation point needed.\n");
                    printf("The coordinate of the collocation point\n");
                    printf("and the dirichlet condition must be\n");
                    printf("included in the data file\n");
                    xp = xalt[icount];
                    yp = yalt[icount];
                }
        }
    else
        /* DOMAIN COLLOCATION */
        {
            ii = i - nodes;
            xx1 = x2[ijk2[0][ii]];
            xx2 = x2[ijk2[1][ii]];
            xx3 = x2[ijk2[2][ii]];
            yy1 = y2[ijk2[0][ii]];
            yy2 = y2[ijk2[1][ii]];
            yy3 = y2[ijk2[2][ii]];
            xp = (xx1+xx2+xx3) / 3.0;
            yp = (yy1+yy2+yy3) / 3.0;
        }
}
/* INTEGRATION ON THE BOUNDARY */
int4(xp, yp, nk, ah2, np, nodes, ijk, x, y, ndim2);

if (i<nodes)
    /* CALCULATION OF RIGID BODY TERM */
    {
        cc = 0.0;
        for (icc=0; icc<nodes; icc++)
            cc += ah2[icc];
if (debug)
    fprintf(stderr, "Node %5d: cc = %10.21f\n", i, cc);

ah2[i] = (-cc);

/* THE FOLLOWING IS NECESSARY FOR EXTERNAL PROBLEMS */
if (ah2[i] < 0.0)
    ah2[i] += 2*pi;

/* BDY INSERTS THE APPROPRIATE BOUNDARY CONDITIONS */
/* BCU IS THE COEFF. MATRIX MULTIPLYING THE UNKNOWNS ON */
/* THE BOUNDARY. SIMILARLY BCK IS THE MATRIX MULTIPLYING */
/* THE UNKNOWNS ON THE BOUNDARY. BCU AND BCK ARE THE */
/* MATRICES FOR BOUNDARY COLLOCATION. */
bdry(bcu, bck, ah2, nbdy, i, nodes, ndim1);
}
else
/* DOMAIN COLLOCATION */
{
    ii = i - nodes;
    /* DCU IS THE COEFF. MATRIX MULTIPLYING THE UNKNOWNS ON */
    /* THE BOUNDARY. SIMILARLY DCK IS THE MATRIX MULTIPLYING */
    /* THE UNKNOWNS ON THE BOUNDARY. DCU AND DCK ARE THE */
    /* MATRICES FOR DOMAIN COLLOCATION. */
bdry(dcu, dck, ah2, nbdy, ii, nodes, ndim1);
}

/* INTEGRATION OVER THE DOMAIN. (INTD DRIVES DOMAIN */
/* INTEGRATION ROUTINES */
intd(xp, yp, be, dc, nelx, ijk2, x2, y2, ndim1, ndim3,
       i, nodes, factor);
if (debug) printf("intd done\n");
}
/* FILE: cmmult.c */
/* */
/* Boundary Element Method (BEM) Application */
/* Jeffrey S. Clary */
/* Iowa State University */
/* */
/* This file contains matrix multiplication code. */
/******************************************************************************/

#include "bemc.h"

/******************************************************************************/
/* Multiplies matrix a(nl x n2) by matrix b(n2 x n3) and stores the */
/* result in matrix ab(nl x n3). */
/* */
/* Uses the non-cache-friendly IJK loop order. */
/******************************************************************************/

void mmmultslow(int ndiml, int nl, int n2, int n3, double a[LEN][LEN],
                 double b[LEN][LEN], double ab[LEN][LEN])
{
    int i, j, k;
    for (i=0; i<nl; i++)
        for (j=0; j<n3; j++)
            ab[i][j] = 0.0;
    for (i=0; i<nl; i++)
        for (j=0; j<n3; j++)
            for (k=0; k<n2; k++)
                ab[i][j] += a[i][k] * b[k][j];
}

/******************************************************************************/
/* Multiplies matrix a(nl x n2) by matrix b(n2 x n3) and stores the */
/* result in matrix ab(nl x n3). */
/* */
/* If slow matrix multiplication is being used, then it calls */
/* mmmultslow to use non-cache-friendly loop order. Otherwise, it */
/* uses the cache-friendly IKJ order. */
/******************************************************************************/

void mmmult(int ndiml, int nl, int n2, int n3, double a[LEN][LEN],
            double b[LEN][LEN], double ab[LEN][LEN])
{
    int i, j, k;
    if (mm_slow)
    {
        mmmultslow(ndiml, n1, n2, n3, a, b, ab);
        return;
    }
for (i=0; i<n1; i++)
    for (j=0; j<n3; j++)
        ab[i][j] = 0.0;

for (i=0; i<n1; i++)
    for (k=0; k<n2; k++)
        for (j=0; j<n3; j++)
            ab[i][j] += a[i][k] * b[k][j];
#include "bemc.h"

void shfunc(double sh[8][8], double h, int nk)
{
    double hsq = h * h;
    double hcu = hsq * h;

    switch (nk)
    {
    case 2 :
        sh[0][0] = 1.0;
        sh[0][1] = -1.0 / h;
        sh[1][0] = 0.0;
        sh[1][1] = -sh[0][1];
        break;
    case 3 :
        sh[0][0] = 1.0;
        sh[0][1] = -3.0 / h;
        sh[0][2] = 2.0 / hsq;
        sh[1][0] = 0.0;
        sh[1][1] = 4.0 / h;
        sh[1][2] = -4.0 / hsq;
        sh[2][0] = 0.0;
        sh[2][1] = -1.0 / h;
        sh[2][2] = 2.0 / hsq;
        break;
    case 4 :
        sh[0][0] = 1.0;
        sh[0][1] = -5.5 / h;
        sh[0][2] = 9.0 / hsq;
        sh[0][3] = -4.5 / hcu;
        sh[1][0] = 0.0;
        sh[1][1] = 9.0 / h;
        sh[1][2] = -22.5 / hsq;
        sh[1][3] = 13.5 / hcu;
        sh[2][0] = 0.0;
```c
/* This is the shape function driver function. */
void shape(int nk, double gg[], double h)
{
    double sh[8][8], ggd[8];
    int i, j;

    shfunc(sh, h, nk);

    for (i=0; i<nk; i++)
    {
        ggd[i] = 0.0;
        for (j=0; j<nk; j++)
            ggd[i] += gg[j] * sh[i][j];
    }

    for (i=0; i<nk; i++)
        gg[i] = ggd[i];
}
```
#include "bemc.h"
#include <math.h>

/* This function implements LU decomposition. */
void decomp(int na, int n, double *cond, int ipvt[], double work[],
            double a[LEN][LEN])
{
    double anorm, ynorm, znorm, t;
    int nml, i, j, k;
    int m, kpl, kml;
    int ek, kb;

    ipvt[n-1] = 1; /* HMMM ??? */

    if (n!=1) {
        nml = n-1;
        anorm = 0.0;
        for (j=0; j<n; j++)
            { 
                t = 0.0;
                for (i=0; i<n; i++)
                    t += fabs(a[i][j]);
                if (t > anorm)
                    anorm = t;
            }

        /* PIVOT ON ALL BUT LAST ROW */
        for (k=0; k<nml; k++)
            {
                /* SEARCH OTHER ROWS FOR BETTER PIVOT */
                kpl = k+1;
                m = k;
                for (i=kpl; i<n; i++)
                    { 
                        if (fabs(a[i][k]) > fabs(a[m][k]))
                            m = i;
            
                        t = 0.0;
                        for (i=0; i<n; i++)
                            t += fabs(a[i][j]);
                        if (t > anorm)
                            anorm = t;
                    }
    }
ipvt[k] = m;

if (m != k)
    ipvt[n-1] = -ipvt[n-1];

t = a[m][k];
a[m][k] = a[k][k];
a[k][k] = t;
if (t != 0.0)
{
    for (i=kpl; i<n; i++)
        a[i][k] = -a[i][k]/t;
    for (j=kpl; j<n; j++)
    {
        t = a[m][j];
a[m][j] = a[k][j];
a[k][j] = t;
        if (t != 0.0)
            for (i=kpl; i<n; i++)
                a[i][j] += a[i][k]*t;
    }
}

for (k=0; k<n; k++)
{
    t = 0.0;
    if (k != 0)
    {
        km1 = k - 1;
        /* PROCESS UP TO BUT NOT INCLUDING K */
        for (i=0; i<=km1; i++)
            t += a[i][k] * work[i];
    }
    ek = 1.0;
    if (t < 0.0)
        ek = -1.0;
    if (a[k][k] == 0.0)
    {
        *cond = 9999.99;
        return;
    }
    work[k] = -(ek+t)/a[k][k];
}

for (kb=0; kb<nm1; kb++)
{
    /* FIXUP FOR C ARRAY */
k = n - 2 - kb;
t = 0.0;
kpl = k + 1;
for (i=kpl; i<n; i++)
    t += a[i][k] * work[k];
work[k] = t;
m = ipvt[k];
if (m != k)
{
    t = work[m];
    work[m] = work[k];
    work[k] = t;
}
ynorm = 0.0;
for (i=0; i<n; i++)
    ynorm += fabs(work[i]);
solve(na, n, work, ipvt, a);
znorm = 0.0;
for (i=0; i<n; i++)
    znorm += fabs(work[i]);
*cond = anorm*znorm/ynorm;
if (*cond < 1.0)
    *cond = 1.0;
return;
else
    *cond = (a[0][0] != 0.0) ? 1.0 : 8888.88;
return;
}

/* This function implements LU solve. */
void solve(int na, int n, double b[], int ipvt[], double a[LEN][LEN])
{
    if (n != 1)
    {
        int nml = n - 1;
        int kpl, kb, kml;
        int i,k,m;
        double t;

        /* FORWARD SUBSTITUTION */
        for (k=0; k<nml; k++)
        {
            kpl = k + 1;
            /* SWAP PIVOT */
            m = ipvt[k];
            t = b[m];
            b[m] = b[k];
            b[k] = t;
            for (i=kpl; i<n; i++)
b[i] += a[i][k] * t;
}

/* BACK SUBSTITUTION */
for (kb=0; kb<nml; kb++)
{
    /* FIXUP FROM FORTRAN ARRAY */
    kml = n - 2 - kb;
    k = kml + 1;
    b[k] /= a[k][k];
    t = -b[k];
    /* FIXUP FROM FORTRAN ARRAY */
    for (i = 0; i<=kml; i++)
    {
        b[i] += a[i][k] * t;
    }
}

b[0] /= a[0][0];
}
#ifndef hpux
#define getrusage(a, b) syscall(SYS_GETRUSAGE, a, b)
#endif /* hpux */

#include <sys/time.h>
#include <sys/resource.h>

struct timeval t;
struct rusage r;

double setpoint = 0.0;

double stopwatch(int resetflag)
{
    double newtime;
    double rval;

    getrusage(RUSAGE_SELF, &r);
    t.tv_sec = r.ru_utime.tv_sec + r.ru_stime.tv_sec;
    t.tv_usec = r.ru_utime.tv_usec + r.ru_stime.tv_usec;

    newtime = (double)t.tv_sec + ((double)t.tv_usec)/1000000.0;
    rval = newtime - setpoint;

    if (resetflag)
        setpoint = newtime;

    return rval;
}
#include "bemc.h"
#include <math.h>

int nsgn(int i)
{
    return (i & 0x01) ? -1 : 1;
}

double pwr(double x, int n)
{
    return (n==0) ? 1.0 : pow(x, (double) n);
}
APPENDIX 5: MPL BEM SOURCE CODE

/****** FILE: bem.h ******/
/* Boundary Element Method (BEM) Application */
/* Jeffrey S. Clary */
/* Iowa State University */
/*
This file contains global constant, variable and function defs. */
/*********************************************************************************/
#include <mpl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#define SLEN 1280 /* Length for singular input arrays */
#define PAN 4 /* Maximum number of nodes per element */
#define MAXALT 10 /* Maximum number of alternate collocation points */

/* This project used an MP-1 with 16 Kbytes memory per PE and an */
/* MP-2 with 64 Kbytes memory per PE. Hence the different consts */
ifdef _MP2.
define LEN 20 /* Length of local PE submatrices. */
define NXPROC_C 64 /* Compile time value for nxproc */
else
define LEN 10 /* Length of local PE submatrices. */
define NXPROC_C 128 /* Compile time value for nxproc */
endif

/****** Global variable defs. ******/
/*********************************************************************************/
extern double pi; /* 3.14... */
extern char errbuf[]; /* Buffer for error messages */
extern int debug; /* Flag for debugging */
extern int echo; /* Flag for echoing input */
extern int opt_cse; /* Flag common subexpression elim. opt */
extern int opt_switch; /* Flag plural switch opt in aint() */
extern int opt_invalp; /* Flag pipelined LU inversion opt */
extern int opt_invcol; /* Flag column-oriented LU invert. opt */
extern int opt_invj; /* Flag Gauss-Jordan inversion opt */
extern int opt_spl; /* Flag software pipelining */
extern int logging; /* Another debug flag */
extern FILE *logfile; /* File descriptor for debugging */
extern int ipvtpe[NXPROC_C][LEN]; /* temp for LU pivot array */
extern int ipvtmem[NXPROC_C][LEN]; /* temp for LU pivot array */

extern int ijk[FAN][SLEN];  /* Boundary connectivity */
extern int ijk2[3][SLEN];   /* Domain connectivity */
extern double s_x[], s_y[]; /* Temps for bnd node input */
extern double s_x2[], s_y2[];/* Temps for domain node input */
extern double s_xalt[], s_yalt[];/* Temps for alt. node input */
extern int s_nbdy[], s_iflag[]; /* More tems */

extern int np; /* # boundary elements */
extern int nodes; /* # boundary nodes */
extern int nk; /* # nodes per element */
extern int nflag; /* # alt. collocation pts */
extern int nelx; /* # domain elements */
extern int nx; /* # domain nodes */
extern int nint; /* # times steps */
extern int npr; /* print interval */

extern int nelxblk; /* domain elem submatrix len */
extern int npblk; /* bndry elem submatrix len */
extern int nodesblk; /* bndry node submatrix len */
extern int npadd; /* Number of dummy elems */
extern int nodesadd; /* Number of dummy nodes */

extern plural double ah1[LEN]; /* Tmp array for collocation */
extern plural double ah2[LEN]; /* Tmp array for collocation */
extern plural double r_x[LEN], r_y[LEN]; /* Boundary coordinates */
extern plural double c_x[LEN], c_y[LEN]; /* Boundary coordinates */
extern plural double bcond[LEN]; /* Boundary value (Phi) */
extern plural double bval[LEN]; /* Computed Phi at nodes */
extern plural double xalt[LEN]; /* Alt. collocation coords. */
extern plural double yalt[LEN]; /* Alt. collocation coords. */

extern plural int r_nbdy[LEN]; /* Boundary node type */
extern plural int c_nbdy[LEN]; /* Boundary node type */
extern plural int c_iflag[LEN]; /* Double node flag */

extern plural double tin[LEN]; /* Current Psi value */
extern plural double tinml[LEN]; /* Previous Psi value */
extern plural double bvec[LEN]; /* Psi temporary array */
extern plural int r_realelem[LEN]; /* 1 => not a dummy node */
extern plural int c_realelem[LEN]; /* 1 => not a dummy node */
extern plural double c_cx[LEN]; /* X coord. dom elem centroid */
extern plural double c_cy[LEN]; /* Y coord. dom elem centroid */
extern plural double r_cx[LEN]; /* X coord. dom elem centroid */
extern plural double r_cy[LEN]; /* Y coord. dom elem centroid */
extern plural double r_cx1[LEN]; /* X coord. dom elem pt 1 */
extern plural double r_cx2[LEN]; /* X coord. dom elem pt 2 */
extern plural double r_cx3[LEN]; /* X coord. dom elem pt 3 */
extern plural double r_cy1[LEN]; /* Y coord. dom elem pt 1 */
extern plural double r_cy2[LEN]; /* Y coord. dom elem pt 2 */
extern plural double r_cy3[LEN]; /* Y coord. dom elem pt 3 */
extern plural int realdelem[LEN]; /* !zero -> not unused elem */

/* ------------------------------------------ */
/* Coefficient matrices: */
/* ------------------------------------------ */
/* bcu -- boundary collocation/boundary integration unknowns */
/* bck -- boundary collocation/boundary integration knowns */
/* dcu -- domain collocation/boundary integration unknowns */
/* dck -- domain collocation/boundary integration knowns */
/* bc -- boundary collocation/domain integration */
/* dc -- domain collocation/domain integration */
/* ------------------------------------------ */

extern plural double bcu[LEN][LEN];
extern plural double bck[LEN][LEN];
extern plural double dcu[LEN][LEN];
extern plural double dck[LEN][LEN];
extern plural double be[LEN][LEN];
extern plural double dc[LEN][LEN];

/* Temporary matrices to hold the results of matrix algebraic */
/* manipulations. */
/* ------------------------------------------ */

extern plural double xmat[LEN][LEN];
extern plural double ymat[LEN][LEN];
extern plural double zmat[LEN][LEN];
extern plural double amat[LEN][LEN];

extern double dt; /* delta of time step */
extern double diffk; /* Diffusivity */
extern double factor; /* 1.0 / (dt * diffk) */

/* ------------------------------------------ */
/* Global function defs. See the function headers in the source */
/* files for explanations. */
/* ------------------------------------------ */
plural double binom(int m, int i);
void bound();
void col_invrt(int nblk, plural double a[LEN][LEN],
               int ipvtpe[NXPROC_C][LEN], int
               ipvtmem[NXPROC_C][LEN],
               plural double xmat[LEN][LEN]);
void col_solve(int nblk, int ipvtpe[NXPROC_C][LEN],
int ipvtmem[NXPROC_C][LEN],
plural double a[LEN][LEN], plural double b[LEN][LEN]);
void decomp(int nblk, int ipvtpe[NXPROC_C][LEN],
    int ipvtmem[NXPROC_C][LEN],
plural double amat[LEN][LEN]);
void dpumap(void);
void errexit(char *string);
void exact(double xp, double yp, double time,
    double *tmpexct);
int find_row_with_max(plural double x, int column);
void shfunc(plural double h, int nk);
void initcond();
void initread(void);
void initwrite(void);
void
    invrt(int sby, int nblk, plural double a[LEN][LEN],
        int ipvtpe[NXPROC_C][LEN],
        int ipvtmem[NXPROC_C][LEN], plural double ah1[],
        plural double ah2[],
        plural double xmat[LEN][LEN]);
void plud_invrt(int sby, int nblk, plural double a[LEN][LEN],
    int ipvtpe[NXPROC_C][LEN],
    int ipvtmem[NXPROC_C][LEN],
    plural double ah1[],
    plural double ah2[],
    plural double xmat[LEN][LEN]);
void gj_invrt(int sby, int nblk, plural double a[LEN][LEN],
plural double b[LEN][LEN]);
void intd(plural double xp, plural double yp, int bndcoll,
    int i);
void int4(plural double xp, plural double yp);
void matvec(void);
void
    mmult(int arows, int acols, int bcols, plural
        double A[LEN][LEN], plural double B[LEN][LEN],
        plural double C[LEN][LEN]);
void mmult_spl(int arows, int acols, int bcols,
    plural double A[LEN][LEN],
    plural double B[LEN][LEN],
    plural double C[LEN][LEN]);
void
    msolve(int nblk, int ipvtpe[NXPROC_C][LEN],
        int ipvtmem[NXPROC_C][LEN],
        plural double a[LEN][LEN],
        plural double b[LEN][LEN]);
int nsgn(int i);
plural double p_pwr(plural double x, plural int n);
void proc_args(int argc, char **argv);
double pwr(double x, int n);
void shape(int nk, plural double gg[], plural double h);
void solve(int nblk, int ipvtpe[NXPROC_C][LEN],
    int ipvtmem[NXPROC_C][LEN],
    plural double a[LEN][LEN], plural double b[LEN][LEN]);
double stopwatch(int resetflag);
plural double sum_to_diag(plural double x);
plural double sum_to_c0(plural double x);
FILE: bem.m

Boundary Element Method (BEM) Application
Jeffrey S. Clary
Iowa State University

This file contains the main program for the application.

---

```c
#include "bem.h"

main (int argc, char *argv[])
{
    plural double temp; /* Local temp for domain value */
    double tmpexct; /* "Exact" domain value */
    double cond; /* Condition number */
    int nstep; /* Current time step */
    int i, j, k; /* Loop counters */
    int done; /* Loop termination var */

    double time; /* Current time value */
    double elapsed; /* Temp for timing */
    double io_elapsed = 0.0; /* Temp for timing I/O */
    double iter_elapsed = 0.0; /* Temp for timing iteration */
    double total_elapsed = 0.0; /* Temp for timing total exec. */

    fprintf(stderr, "\n\n");
    proc_args(argc, argv);
    pi = 4.0 * atan ((double) 1.0);

    /* GET ANY ONE-TIME TIMER OVERHEAD OUT OF THE WAY */
    stopwatch(1);

    stopwatch(1);
    /* GET THE INPUT INTO SINGULAR ARRAYS AND VARIABLES */
    initread();
    if (echo)
        initwrite();

    if (debug)
```
195

{
    fprintf(stderr, "Geometry input done\n");
    fflush(stdout);
}

/* NOW CHUNK IT OUT TO THE DPU */
dpumap();
elapsed = stopwatch(0);
io_elapsed += elapsed;
total_elapsed += elapsed;

factor = 1.0 / (dt * diffk);

fprintf(stderr, "Problem size = %d\n", nodes);

/* COLLOCATE AND INTEGRATE */
stopwatch(1);
matvec();
elapsed = stopwatch(0);
total_elapsed += elapsed;
fprintf(stderr, "%0.11f\n", elapsed);

/* MUNGE THE ARRAYS */
stopwatch(1);
if (opt_invpipe)
    plud_invrt(nk-1, nodesblk, bcu, ipvtpe, ipvtmem, ah1, ah2, xmat);
else if (opt_invcol)
    col_invrt(nodesblk, bcu, ipvtpe, ipvtmem, xmat);
else if (opt_invgj)
    gj_invrt(nk-1, nodesblk, bcu, xmat);
else
    invrt(nk-1, nodesblk, bcu, ipvtpe, ipvtmem, ah1, ah2, xmat);
elapsed = stopwatch(0);
total_elapsed += elapsed;
fprintf(stderr, "%0.11f\n", elapsed);

stopwatch(1);
mmmult(nodesblk, nodesblk, nodesblk, bcu, bck, ymat);
mmmult(nodesblk, nodesblk, nodesblk, bcu, bc, zmat);
mmmult(nelxblk, nodesblk, nelxblk, dcu, zmat, bck);
mmult(nelxblk, nodesblk, nodesblk, dcu, ymat, bck);

for (i=0; i<nelxblk; i++)
    for (j=0; j<nelxblk; j++)
        bc[i][j] = bcu[i][j] - dc[i][j];

for (i=0; i<nelxblk; i++)
    for (j=0; j<nodesblk; j++)
        dc[i][j] = bck[i][j] - dck[i][j];

for (i=0; i<nelxblk; i++)
for (j=0; j<nelxblk; j++)
    if (i == j)
        {
            if (ixproc == iyproc)
                if (realdelem[i])
                    amat[i][j] = 2*pi + bc[i][j];
                else
                    amat[i][j] = 1.0;
            else
                amat[i][j] = bc[i][j];
        }
    else
        amat[i][j] = bc[i][j];
elapsed = stopwatch(0);
total_elapsed += elapsed;
fprintf(stderr, "%0.11f\n", elapsed);

/* LU DECOMPOSITION OF AMAT -- DONE ONLY ONCE */
stopwatch(1);
cond = decomp(nelxblk, ipvtpe, ipvtmem, amat);
elapsed = stopwatch(0);
total_elapsed += elapsed;
fprintf(stderr, "%0.11f\n", elapsed);

/* INIT INSERTS THE INITIAL CONDITION */
stopwatch(1);
initcond();
elapsed = stopwatch(0);
io_elapsed += elapsed;
total_elapsed += elapsed;

for (nstep=1; nstep<=nint; nstep++)
    {
        stopwatch(1);
        /* BOUND INSERTS THE TIME-DEPENDENT BOUNDARY CONDITIONS */
        bound();
elapsed = stopwatch(0);
io_elapsed += elapsed;
total_elapsed += elapsed;
        stopwatch(1);
        /* BVEC NEEDS TO GO INTO PE COLUMN ZERO FOR THE SOLVER */
        /* BCOND TIN AND TINM1 ARE ORGANIZED BY ROW */
        for (i=0; i<nelxblk; i++)
            {
                bvec[i] = 0.0;
                for (j=0; j<nodesblk; j++)
                    bvec[i] += dc[i][j] * bcond[j];
                for (j=0; j<nelxblk; j++)
                    bvec[i] += bc[i][j] * tinml[j];
for (i=0; i<nelxblk; i++)
    bvec[i] = sum_to_c0(bvec[i]);

solve(nelxblk, ipvtpe, ipvtmem, amat, bvec);

/* BVEC VALUES MUST BE TRANPOSED AND COPIED TO ROWS */
for (i=0; i<nelxblk; i++)
    {
        temp = bvec[i];
        if (ixproc==0)
            xnetcE[nxproc].temp = temp;
        if (ixproc==iyproc)
            xnetcS[nxproc].temp = temp;
        tin[i] = temp;
    }

/* BVAL VALUE GO INTO PE COLUMN 0 */
for (i=0; i<nodesblk; i++)
    {
        bval[i] = 0.0;
        for (j=0; j<nodesblk; j++)
            bval[i] -= ymat[i][j] * bcond[j];
        for (j=0; j<nelxblk; j++)
            bval[i] += zmat[i][j] * (tin[j] - tinml[j]);
    }
for (i=0; i<nodesblk; i++)
    bval[i] = sum_to_c0(bval[i]);

iter_elapsed += stopwatch(0);

/* PRINT SOME OUTPUT IF DESIRED */
if (nstep % npr == 0)
    {
        stopwatch(1);
        /* PRINT THE DOMAIN VALUES (PSI) */
        printf("Domain Solution\n");
        printf("Exact Solution\n");
        printf(" X Y Temp Tmpexct\n");
        for (done=0, i=0; i<nyproc && !done; i++)
            {
                for (j=0; j<nelxblk && !done; j++)
                    if (i*nelxblk+j < nelx)
                        {
                            time = nstep * dt;
                            exact(proc[0][i].r_cx[j], proc[0][i].r_cy[j],
                                    time, &tmpexct);
                            printf("%10.41f %10.41f %10.41f %10.41f\n",
                                    proc[0][i].r_cx[j], proc[0][i].r_cy[j],
                                    proc[0][i].tin[j], tmpexct);
                        }
            else


done = 1;
/* PRINT THE BOUNDARY VALUES (PHI) */
for (i=0; i<npblk; i++)
    for (j=0; j<nyproc; j++)
        for (k=0; k<nk-1; k++)
            if (1)
                printf("%d %d %0.4lf\n",
                    i*nyproc*(nk-1)+j*(nk-1)+k,
                    proc[j][0].c_nbdy[i*(nk-1)+k],
                    proc[j][0].bval[i*(nk-1)+k]);

    elapsed = stopwatch(0);
    io_elapsed += elapsed;
    total_elapsed += elapsed;
}
for (i=0; i<nelxblk; i++)
    tinml[i] = tin[i];
iter_elapsed /= (double)nint;
fprintf(stderr, "%.1lf\n", iter_elapsed);

fprintf(stderr, "%.1lf\n", io_elapsed);

total_elapsed += iter_elapsed;
fprintf(stderr, "%.1lf\n\n", total_elapsed);
if (logging)
    fclose(logfile);
}
#include "bem.h"

void proc_args(int argc, char **argv)
{
    while (((--argc) > 0))
    {
        if (!strcmp(argv[1], "-h"))
        {
            fprintf(stderr, "\n\nUsage: %s (option ...\n\nValid options:\n\n  -cse Do common subexpression elimination\n\n  -debug Turn on debugging output\n\n  -echo Echo problem input\n\n  -invgj Use Gauss-Jordan elim. for inversion\n\n  -invpipe Use pipelined LUD for inversion\n\n  -invcol Use column oriented LUD for inversion\n\n  -log <f> Write a debug log to file 'f'\n\n  -switch Use smart plural switch\n\n  -spl Do source level software pipelining\n\nexit(0);
        }

        argv++;
        if (!strcmp(*argv, "-cse"))
        {
            fprintf(stderr, "Common subexpression elimination on\n\nopt_cse = 1;
        }
        // Other arguments and flag processing...
    }
}
else if (!strcmp(*argv, "-debug"))
    debug = 1;
else if (!strcmp(*argv, "-echo"))
    echo = 1;
else if (!strcmp(*argv, "-invgj"))
    {
        fprintf(stderr, "Gauss-Jordan inversion\n");
        opt_invgj = 1;
    }
else if (!strcmp(*argv, "-invpipe"))
    {
        fprintf(stderr, "Pipelined LUD inversion\n");
        opt_invpipe = 1;
    }
else if (!strcmp(*argv, "-invcol"))
    {
        fprintf(stderr, "Column-oriented LUD inversion\n");
        opt_invcol = 1;
    }
else if (!strcmp(*argv, "-log"))
    {
        logging = 1;
        argv++;
        argc--;
        if ((logfile = fopen(*argv, "w")) == NULL)
            errexit("Can't open log file\n");
    }
else if (!strcmp(*argv, "-switch"))
    {
        fprintf(stderr, "Optimized switch statement on\n");
        opt_switch = 1;
    }
else if (!strcmp(*argv, "-spl"))
    {
        fprintf(stderr, "Software pipelining on\n");
        opt_spl = 1;
    }
else
    {
        sprintf(errbuf, "Unknown option %s: use -h for help\n", *argv);
        errexit(errbuf);
    }
FILE: cond.m

Boundary Element Method (BEM) Application
Jeffrey S. Clary
Iowa State University

This file contains functions to set the initial domain
and the time-dependent boundary conditions. Note that this is
problem-dependent information, and for other problem sets, these
functions would have to be rewritten. For a production code,
these functions could be replaced with ones that read values as
input.

#include "bem.h"

This function inserts the time-dependent boundary conditions.

void bound()
{
    int i;

    for (i=0; i<nodesblk; i++)
        bcond[i] = 0.0;
}

This function inserts the initial condition.

void initcond()
{
    int i;

    for (i=0; i<nelxblk*2; i++)
    {
        tinml[i] = p_sin(pi*r_cx[i]);
    }
}
/* File: domainint.m */
/* */
/* Boundary Element Method (BEM) Application */
/* Jeffrey S. Ciary */
/* Iowa State University */
/* */
/* This file contains routines for domain integration. */
/* */
/* This code is adapted directly from Dr. Ambar Mitra's Fortran */
/* integration code. As such, it is not heavily documented. */
/**************************************************************/
#include "bem.h"
#include <math.h>

/**************************************************************/
/* Global variables used in integration */
/**************************************************************/
static plural double aloga, alogb;
static plural double h, u, v, thu, thv, w;

/**************************************************************/
plural double calc_alxx(plural double arg)
{
    plural double rval;

    if (arg < -1.0)
        rval = pi;
    else if (arg > 1.0)
        rval = 0.0;
    else
        rval = p_acos(arg);

    return rval;
}

/**************************************************************/
/* This function determines the case for integration. */
/**************************************************************/
plural int chkcase(plural double xp, plural double yp,
                   plural double xl, plural double yl,
                   plural double x2, plural double y2,
                   plural double x3, plural double y3)
{
    double small = 0.0001;

    plural double vx1 = x1 - xp;
    plural double vy1 = y1 - yp;
    plural double vx2 = x2 - xp;
    plural double vy2 = y2 - yp;
plural double vx3 = x3 - xp;
plural double vy3 = y3 - yp;
plural double vl, v2, v3;
plural double arg1, arg2, arg3;
plural double a12, a123, a131;
plural int rval;
vl = p_sqrt(vxl*vxl + vyl*vyl);
v2 = p_sqrt(vx2*vx2 + vy2*vy2);
v3 = p_sqrt(vx3*vx3 + vy3*vy3);
if (vl>0.0 && v2>0.0)
  all2 = calc_alxx((vxl*vx2 + vyl*vy2) / (vl*v2));
if (v2>0.0 && v3>0.0)
  a123 = calc_alxx((vx2*vx3 + vy2*vy3) / (v2*v3));
if (vl>0.0 && v3>0.0)
  a131 = calc_alxx((vxl*vx3 + vyl*vy3) / (vl*v3));
if (vl < small)
  rval = 1;
else if (v2 < small)
  rval = 2;
else if (v3 < small)
  rval = 3;
else if (p_fabs(a12 - pi) < small)
  rval = 4;
else if (p_fabs(a123 - pi) < small)
  rval = 5;
else if (p_fabs(a131 - pi) < small)
  rval = 6;
else if (a12 < small)
  if (v1 <= v2)
    rval = 7;
  else
    rval = 8;
else if (a123 < small)
  if (v3 <= v2)
    rval = 9;
  else
    rval = 10;
else if (a131 < small)
  if (v1 <= v3)
    rval = 11;
  else
    rval = 12;
else if (p_fabs(a12+a123+a131-2*pi) < small)
  rval = 13;
else if (p_fabs(a12-a123-a131) < small)
  if ((vxl*vy2 - vyl*vx2) <= 0.0)
rval = 14;
else
    rval = 15;
else if (p_fabs(al23-al123-al31) < small)
    if ((vx2*vy3 - vx3*vy2) <= 0.0)
        rval = 16;
    else
        rval = 17;
else if (p_fabs(al31-al23-al12) < small)
    if ((vx3*vy1 - vx1*vy3) <= 0.0)
        rval = 18;
    else
        rval = 19;
return rval;

/* This function does analytic domain integration over constant */
/* triangular elements. */
/* This is the original version, not modified with common */
/* subexpression elimination. */
plural double nocse_aint(plural double xp, plural double yp,
    plural double xu, plural double yu,
    plural double xv, plural double yv)
{
    plural double d1 = xu - xp;
    plural double d2 = xv - xp;
    plural double d3 = yu - yp;
    plural double d4 = yv - yp;
    plural double d5 = xv - xu;
    plural double d6 = yv - yu;
    plural double a = p_sqrt(d1*d1 + d3*d3);
    plural double b = p_sqrt(d2*d2 + d4*d4);
    plural double phi = p_atan2(d6, d5);
    plural double alpha = phi - 0.5*pi;
    plural double arg1, arg2;
    plural double val1, q2;
    h = p_sqrt(d5*d5 + d6*d6);
    if (a == 0.0) /* FORTRAN LABEL 5 */
    {
        arg1 = d4*p_cos(alpha) - d2*p_sin(alpha);
        arg2 = d2*p_cos(alpha) + d4*p_sin(alpha);
        /* Additional code */
    }
thu = thv = p_atan2(arg1, arg2);
aloga = 0.0;
alogb = p_log(b);
}
else if (b == 0.0) /* FORTRAN LABEL 8 */
{
  argl = d3*p_cos(alpha) - d1*p_sin(alpha);
  arg2 = d1*p_cos(alpha) + d3*p_sin(alpha);
  thu = thv = p_atan2(arg1, arg2);
aloga = p_log(a);
alogb = 0.0;
}
else
{
  argl = d3*p_cos(alpha) - d1*p_sin(alpha);
  arg2 = d1*p_cos(alpha) + d3*p_sin(alpha);
  thu = p_atan2(arg1, arg2);
  argl = d4*p_cos(alpha) - d2*p_sin(alpha);
  arg2 = d2*p_cos(alpha) + d4*p_sin(alpha);
  thv = p_atan2(arg1, arg2);
aloga = p_log(a);
alogb = p_log(b);
}
u = a*p_sin(thu);
v = b*p_sin(thu);
w = a*p_cos(thu);
val = p_tan(thv)*(alogb-0.5) - p_tan(thu)*(aloga-0.5);
q2 = (thv-thu) - (p_tan(thv) - p_tan(thu));

return 0.5 * w * w * (val+q2);
}
register plural double phi = p_atan2(d6, d5);

register plural double alpha = phi - 0.5*pi;

register plural double tmpl = p_cos(alpha);
register plural double tmp2 = p_sin(alpha);

register plural double d4m2 = d4*tmpl - d2*tmp2;
register plural double d2p4 = d2*tmpl + d4*tmp2;
register plural double d3m1 = d3*tmpl - d1*tmp2;
register plural double dlp3 = d1*tmpl + d3*tmp2;

register plural double arg1, arg2;
register plural double val, q2;

/* Why is this done in the original code? */
h = p_sqrt(d5*d5 + d6*i)

if (a == 0.0)
  aloga = 0.0;
else
  aloga = p_log(a);

if (b == 0.0)
  alogb = 0.0;
else
  alogb = p_log(b);

/* PLAYING LOOSE WITH VARIABLE MEANINGS HERE TO AVOID REPEATING THE P_ATAN2 CALL. OVERWRITING d??? VARIABLE WITH THE NECESSARY VALUE WHEN a OR b IS ZERO */
if (a == 0.0)
{
  d3m1 = d4m2;
  dlp3 = d2p4;
}
if (b == 0.0)
{
  d4m2 = d3m1;
  d2p4 = dlp3;
}

thu = p_atan2(d3m1, dlp3);
thv = p_atan2(d4m2, d2p4);

tmpl1 = p_sin(thu);
tmp2 = p_cos(thu);
u = a*tmpl1;
v = b*tmp1;
w = a*tmp2;
/* TRIG IDENTITY */
tmp1 = p_tan(thu);
*/
tmp2 = p_tan(thv);
val = tmp2*(alogb-0.5) - tmp1*(aloga-0.5);
q2 = (thv-thu) - (tmp2 - tmp1);
return 0.5 * w*w * (val+q2);

plural double aint(plural double xp, plural double yp,
       plural double xu, plural double yu,
       plural double xv, plural double yv)
{
    if (opt_cse)
        return cse_aint(xp, yp, xu, yu, xv, yv);
    else
        return nocse_aint(xp, yp, xu, yu, xv, yv);
}

plural double domint(plural double xp, plural double yp,
       plural double x1, plural double y1,
       plural double x2, plural double y2,
       plural double x3, plural double y3,
       plural int icase)
{
    register plural double xx1, yy1, xx2, yy2;
    register plural double rslt1, rslt2, rslt3;
    /* IF WE ARE USING THE PARALLEL SWITCH OPTIMIZATION, THEN SET */
    /* UP TEMPORARY INPUTS TO AINT WITHIN THE SWITCH, THEN CALL AINT */
    /* WITH THOSE TEMPORARY ARGS. THIS INCREASES PE UTILIZATION */
    if (opt_switch)
    {
        switch (icase)
{  
case 1:
case 4:
case 7:
case 11:
case 14:
case 17:
    xx1 = x2;
    yy1 = y2;
    xx2 = x3;
    yy2 = y3;
    break;

case 2:
case 8:
case 10:
case 19:
    xx1 = x3;
    yy1 = y3;
    xx2 = x1;
    yy2 = y1;
    break;

case 3:
case 5:
case 6:
case 9:
case 12:
case 13:
case 15:
case 16:
case 18:
    xx1 = x1;
    yy1 = y1;
    xx2 = x2;
    yy2 = y2;
    break;

default:
    break;
}

rsltl = aint(xp, yp, xx1, yy1, xx2, yy2);

switch (icase)
{
case 4:
case 5:
case 14:
case 16:
    xx1 = x3;
    yy1 = y3;

xx2 = x1;
yy2 = y1;
break;

case 6:
case 13:
case 18:
   xxl = x2;
   yy1 = y2;
   xx2 = x3;
   yy2 = y3;
   break;

case 7:
case 9:
   xx1 = x1;
   yy1 = y1;
   xx2 = x3;
   yy2 = y3;
   break;

case 8:
case 12:
case 15:
case 19:
   xxl = x3;
   yy1 = y3;
   xx2 = x2;
   yy2 = y2;
   break;

case 10:
case 11:
case 17:
   xxl = x2;
   yy1 = y2;
   xx2 = x1;
   yy2 = y1;
   break;

default:
   break;
)
if (icase >= 4)
   rslt2 = aint(xp, yp, xx1, yy1, xx2, yy2);

switch (icase)
{
   case 13:
      xx1 = x3;
      yy1 = y3;
xx2 = x1;
yy2 = y1;
break;

case 14:
case 19:
    xx1 = x2;
yy1 = y2;
    xx2 = x1;
yy2 = y1;
    break;

case 15:
case 17:
case 18:
    xx1 = x1;
yy1 = y1;
    xx2 = x3;
yy2 = y3;
    break;

case 16:
    xx1 = x3;
yy1 = y3;
    xx2 = x2;
yy2 = y2;
    break;

default:
    break;
}
if (lease >= 13)
    rslt3 = aint(xp, yp, xx1, yy1, xx2, yy2);

switch(icase)
{
case 1:
case 2:
case 3:
    return rslt1;
case 4:
case 5:
case 6:
    return rslt1 + rslt2;
case 7:
case 8:
case 9:
case 10:
case 11:
case 12:
    return rslt1 - rslt2;
case 13:
    return rslt1 + rslt2 + rslt3;
case 14:
case 16:
case 18:
    return rslt1 + rslt2 - rslt3;
case 15:
case 17:
case 19:
    return rslt1 - rslt2 - rslt3;
}

} /* OTHERWISE, WE ARE NOT DOING THE PARALLEL SWITCH OPTIMIZATION, */
/* SO JUST INVOKE THE SWITCH IN THE NAIVE WAY. NOTICE THAT THE */
/* CALLS TO AINT GET SERIALIZED. */
else
{
    switch (icase)
    {
        case 1: return aint(xp, yp, x2, y2, x3, y3);
        case 2: return aint(xp, yp, x3, y3, x1, y1);
        case 3: return aint(xp, yp, x1, y1, x2, y2);
        case 4: return aint(xp, yp, x2, y2, x3, y3) +
                  aint(xp, yp, x3, y3, x1, y1);
        case 5: return aint(xp, yp, x1, y1, x2, y2) +
                  aint(xp, yp, x3, y3, x1, y1);
        case 6: return aint(xp, yp, x1, y1, x2, y2) +
                  aint(xp, yp, x2, y2, x3, y3);
        case 7: return aint(xp, yp, x2, y2, x3, y3) -
                  aint(xp, yp, x1, y1, x3, y3);
        case 8: return aint(xp, yp, x3, y3, x1, y1) -
                  aint(xp, yp, x3, y3, x2, y2);
        case 9: return aint(xp, yp, x1, y1, x2, y2) -
                  aint(xp, yp, x1, y1, x3, y3);
        case 10: return aint(xp, yp, x3, y3, x1, y1) -
                  aint(xp, yp, x2, y2, x1, y1);
        case 11: return aint(xp, yp, x2, y2, x3, y3) -
                  aint(xp, yp, x2, y2, x1, y1);
        case 12: return aint(xp, yp, x1, y1, x2, y2) -
                  aint(xp, yp, x3, y3, x2, y2);
        case 13: return aint(xp, yp, x1, y1, x2, y2) +
                  aint(xp, yp, x2, y2, x3, y3) +
                  aint(xp, yp, x3, y3, x1, y1);
        case 14: return aint(xp, yp, x2, y2, x3, y3) +
                  aint(xp, yp, x3, y3, x1, y1) -
                  aint(xp, yp, x2, y2, x1, y1);
        case 15: return aint(xp, yp, x1, y1, x2, y2) -
                  aint(xp, yp, x3, y3, x2, y2) -
                  aint(xp, yp, x1, y1, x3, y3);
        case 16: return aint(xp, yp, x1, y1, x2, y2) +
                  aint(xp, yp, x3, y3, x1, y1) -
aint(xp, yp, x3, y3, x2, y2);
case 17: return aint(xp, yp, x2, y2, x3, y3) -
    aint(xp, yp, x2, y2, x1, y1) -
    aint(xp, yp, x1, y1, x3, y3);
case 18: return aint(xp, yp, x1, y1, x2, y2) +
    aint(xp, yp, x2, y2, x3, y3) -
    aint(xp, yp, x1, y1, x3, y3);
case 19: return aint(xp, yp, x3, y3, x1, y1) -
    aint(xp, yp, x3, y3, x2, y2) -
    aint(xp, yp, x2, y2, x1, y1);
}
}

/*****************************************************************************/
/* This is the driver function for domain integration. */
/*****************************************************************************/
void intd(plural double xp, plural double yp, int bndcoll, int i)
{
    int j;
    plural int icase;
    plural double xint;
    plural double xx1, xx2, xx3;
    plural double yy1, yy2, yy3;

    for (j=0; j<nelxblk; j++)
    {
        /* IGNORE ENTRIES PAST LAST DOMAIN ELEMENT */
        if (ixproc<nelx/nelxblk || (ixproc==nelx/nelxblk && j<nelx%nelxblk))
        {
            xx1 = r_cxl[j];
            xx2 = r_cx2[j];
            xx3 = r_cx3[j];
            yy1 = r_cyl[j];
            yy2 = r_cy2[j];
            yy3 = r_cy3[j];

            /* CHKCASE DETERMINES THE LOCATION OF (XP,YP) W.R.T. */
            /* THE TRIANGULAR ELEMENT */
            icase = chkcase(xp, yp, xx1, yy1, xx2, yy2, xx3, yy3);

            /* DOMINT PREPARES DATA FOR ANALYTIC INTEGRATION OVER */
            /* CONSTANT TRIANGULAR ELEMENTS. OUTPUT XINT CONTAINS */
            /* VALUE OF INTEGRAL */
            xint = domint(xp, yp, xx1, yy1, xx2, yy2, xx3, yy3, icase);

            if (bndcoll) /* BOUNDARY COLLOCATION */
                bc[i][j] = xint * factor;
            else /* DOMAIN COLLOCATION */
                dc[i][j] = xint * factor;
    }
/* FILE: dpumap.m */
/* */
/* */
/* Boundary Element Method (BEM) Application */
/* Jeffrey S. Clary */
/* Iowa State University */
/* */
/* This file contains code to distribute input values to the PE */
/* array. */
/* */
/* It is assumed that nxproc == nyproc */
/* ***************************************************************************/
#include "bem.h"
void printmap(void);

//*****************************************************************************/
/* This function distributes the input values from the singular */
/* arrays and maps it to the DPU. */
/* */
/* Code is missing to handle xalt, yalt. */
/* ***************************************************************************/
void dpumap()
{
    plural double xtmp, ytmp; /* Temporary bndry node coords */
    plural double xtmpl, ytmpl; /* Temporary domain node coords */
    plural double xtmp2, ytmp2;
    plural double xtmp3, ytnip3;
    plural int rtmp, ntmp, itmp; /* Other temps */
    int i, j, ifix, inext; /* Loop counters */
    int n, col, row, ofst;
    int npadded; /* Number of dummy elements added so far */

    /* COUNT HOW MANY DUMMY ELEMENTS WILL BE NEEDED */
    npadd = 0;
    for (i=0; i<np-1; i++)
        if (ijk[nk-1][i] != ijk[0][i+1])
            npadd++;
    if (ijk[nk-1][np-1] != ijk[0][0])
            npadd++;

    npblk = (np+npadd) / nyproc;
    if (npblk * nyproc != (np+npadd))
        npblk++;
    nodesblk = npblk * (nk-1);

    npadded = 0;
    /* SCATTER BY NK-1 THE BOUNDARY NODES DOWN COLUMN 0 */
for (i = 0; i < np; i++)
{
    ifix = i + npadded;
    row = ifix % nyproc;
    ofst = ifix / nyproc;
    proc[row][0].(c_realelem[ofst]) = 1;
    ofst *= (nk-1);
    for (j = 0; j < nk-1; j++, ofst++)
    {
        n = ijk[j][i];
        proc[row][0].(c_x[ofst]) = s_x[n];
        proc[row][0].(c_y[ofst]) = s_y[n];
        proc[row][0].(c_nbdy[ofst]) = s_nbdy[n];
        proc[row][0].(c_iflag[ofst]) = s_iflag[n];
    }
    /* CHECK WHETHER TO INSERT A DUMMY ELEMENT */
    /* If so, its 0th ELEMENT MUST BE INIT'D */
    inext = (i < np-1) ? i+1 : 0;
    if (ijk[nk-1][i] != ijk[0][inext])
    {
        npadded++;
        ifix = i + npadded;
        n = ijk[nk-1][i];
        row = ifix % nyproc;
        ofst = ifix / nyproc;
        proc[row][0].(c_realelem[ofst]) = 0;
        ofst *= (nk-1);
        proc[row][0].(c_x[ofst]) = s_x[n];
        proc[row][0].(c_y[ofst]) = s_y[n];
        proc[row][0].(c_nbdy[ofst]) = s_nbdy[n];
        proc[row][0].(c_iflag[ofst]) = s_iflag[n];
    }
    /* BROADCAST BOUNDARY NODES TO ALL OTHER COLUMNS */
    for (i = 0; i < nodesblk; i++)
    {
        if (ixproc == 0)
        {
            xtmp = c_x[i];
            ytmp = c_y[i];
            rtmp = c_realelem[i];
            ntmp = c_nbdy[i];
            itmp = c_iflag[i];
            xnetcE[nxproc-1].xtmp = xtmp;
            xnetcE[nxproc-1].ytmp = ytmp;
            xnetcE[nxproc-1].rtmp = rtmp;
            xnetcE[nxproc-1].ntmp = ntmp;
            xnetcE[nxproc-1].itmp = itmp;
        }
        c_x[i] = xtmp;
        c_y[i] = ytmp;
    }
c_realelem[i] = rtmp;
c_nbdy[i] = ntmp;
c_iflag[i] = itmp;
}

/* USE THE DIAGONAL ELEMENTS TO CREATE A TRANSPOSE OF NODE INFO */
for (i=0; i<nodesblk; i++)
{
    if (ixproc == iyproc)
    {
        xtmp = c_x[i];
ytmp = c_y[i];
rtmp = c_realelem[i];
ntmp = c_nbdy[i];
itmp = c_iflag[i];
xnetcS[nxproc-1].xtmp = xtmp;
xnetcS[nxproc-1].ytmp = ytmp;
xnetcS[nxproc-1].rtmp = rtmp;
xnetcS[nxproc-1].ntmp = ntmp;
xnetcS[nxproc-1].itmp = itmp;
    }
    r_x[i] = xtmp;
r_y[i] = ytmp;
r_realelem[i] = rtmp;
r_nbdy[i] = ntmp;
}

/* SPREAD THE DOMAIN COORDS ACROSS ROW 0 */
nelxblk = nelx / nxproc;
if (nelxblk * nxproc != nelx)
    nelxblk++;
for (i=0; i<nelxblk; i++)
    r_cxl[i] = r_cx2[i] = r_cx3[i] = r_cyl[i] = r_cy2[i] = r_cy3[i] = 0.0;
for (i=0, col=0, ofst=0; i<nelx; i++)
{
    proc[0][col].(r_cxl[ofst]) = s_x2[ijk2[0][i]];
    proc[0][col].(r_cx2[ofst]) = s_x2[ijk2[1][i]];
    proc[0][col].(r_cx3[ofst]) = s_x2[ijk2[2][i]];
    proc[0][col].(r_cyl[ofst]) = s_y2[ijk2[0][i]];
    proc[0][col].(r_cy2[ofst]) = s_y2[ijk2[1][i]];
    proc[0][col].(r_cy3[ofst]) = s_y2[ijk2[2][i]];
    proc[0][col].(realdelem[ofst]) = 1;
ofst++;
    if (ofst == nelxblk)
    {
        ofst = 0;
        col++;
    }
}

/* CALCULATE CENTROIDS */
for (i=0; i<nelxblk; i++)
{
    r_cx[i] = (r_cx1[i] + r_cx2[i] + r_cx3[i]) / 3.0;
    r_cy[i] = (r_cy1[i] + r_cy2[i] + r_cy3[i]) / 3.0;
}

/* BROADCAST DOMAIN NODES TO ALL OTHER ROWS */
for (i=0; i<nelxblk; i++)
{
    if (iyproc == 0)
    {
        xtmp = r_cx[i];
        ytmp = r_cy[i];
        xnetcS[nyproc-1].xtmp = xtmp;
        xnetcS[nyproc-1].ytmp = ytmp;
        xtmp1 = r_cx1[i];
        ytmp1 = r_cy1[i];
        xnetcS[nyproc-1].xtmp1 = xtmp1;
        xnetcS[nyproc-1].ytmp1 = ytmp1;
        xtmp2 = r_cx2[i];
        ytmp2 = r_cy2[i];
        xnetcS[nyproc-1].xtmp2 = xtmp2;
        xnetcS[nyproc-1].ytmp2 = ytmp2;
        xtmp3 = r_cx3[i];
        ytmp3 = r_cy3[i];
        xnetcS[nyproc-1].xtmp3 = xtmp3;
        xnetcS[nyproc-1].ytmp3 = ytmp3;
        rtmp = realdelem[i];
        xnetcS[nyproc-1].rtmp = rtmp;
    }
    r_cx[i] = xtmp;
    r_cy[i] = ytmp;
    r_cxl[i] = xtmp1;
    r_cyl[i] = ytmp1;
    r_cx2[i] = xtmp2;
    r_cy2[i] = ytmp2;
    r_cx3[i] = xtmp3;
    r_cy3[i] = ytmp3;
    realdelem[i] = rtmp;
}

/* USE THE DIAGONAL ELEMENTS TO CREATE A TRANSPOSE, JUST FOR CENTROID */
for (i=0; i<nelxblk; i++)
{
    if (ixproc == iyproc)
    {
        xtmp = r_cx[i];
        ytmp = r_cy[i];
        xnetcE[nyproc-1].xtmp = xtmp;
        xnetcE[nyproc-1].ytmp = ytmp;
        rtmp = realdelem[i];
    }
/* This function echoes the boundary and domain information from the PE array to verify that the mapping is correct. */

/*-----------------------------------------------*/

void printmap(void)
{
    int i, j, k, done;

    printf("nodesblk = %d\n", nodesblk);
    printf("npblk = %d\n", npblk);
    printf("nelxblk = %d\n", nelxblk);
    printf("npadd = %d\n", npadd);

    printf("Boundary Node list:\n");
    for (i=0; i<npblk; i++)
    {
        for (j=0; j<nxproc; j++)
        {
            for (k=0; k<nk-1; k++)
            {
                printf(" %5d: (%6 .21f, %6 .21f ) nbdy = %d",
                        i*nxproc+j, proc[0][j](r_x[i*(nk-1)+k]),
                        proc[0][j](r_y[i*(nk-1)+k]),
                        proc[0][j](r_nbdy[i*(nk-1)+k]));
                printf("%c
", (proc[0][j](r_realelem[i])) ? ' ' : '*');
            }
        }
    }

    printf("Domain Element List:\n");
    for (done=0, i=0; i<nxproc & & !done; i++)
    {
        for (j=0; j<nelxblk & & !done; j++)
        {
            if (i*nelxblk+j < nelx)
            {
                printf(" %5d: (%6.21f,%6.21f)\n",
                        i*nelxblk+j, proc[0][i](r_cx[j]),
                        proc[0][i](r_cy[j]));
                printf(" (%6.21f,%5.21f)\n",
                        proc[0][i](r_cx[j]),
                        proc[0][i](r_cy[j]));
            }
        }
    }
}

}
proc[0][i].(r_cx1[j]), proc[0][i].(r_cy1[j]));
printf("(%.21f,%.21f)\n",
    proc[0][i].(r_cx2[j]), proc[0][i].(r_cy2[j]));
printf("(%.21f,%.21f)\n",
    proc[0][i].(r_cx3[j]), proc[0][i].(r_cy3[j]));
    }
else
    done = 1;
    }
#include "bem.h"

void errexit(char *string)
{
    fprintf(stderr, "Error: %s\n", string);
    exit(-1);
}
/* FILE: exact.m */
/* */
/* Boundary Element Method (BEM) Application */
/* Jeffrey S. Clary */
/* Iowa State University */
/* */
/* This file contains code for finding what the "exact" domain psi */
/* values should be. */
/************************** *******************************************/
#include "bem.h"

/************************** *******************************************/
/* This function calculates the "exact" domain PSI value for some */
/* (x,y) point at some time. It is intended as a check against the */
/* values calculated by the BEM process. For different problems */
/* the routine would have to be rewritten, or modified to read its */
/* values from input. */
/************************** *******************************************/
void exact(double xp, double yp, double time, double *tmpexct)
{
  *tmpexct = sin(pi*xp);
}

/* FILE: global.m */
/* Boundary Element Method (BEM) Application */
/* Jeffrey S. Clary */
/* Iowa State University */
/* This file contains definitions of global variables. See bem.h */
/* for variable descriptions. */
/#include "bem.h"

double pi;
int debug = 0;
int echo = 0;
int opt_cse = 0;
int opt_switch = 0;
int opt_invpipe = 0;
int opt_invcol = 0;
int opt_invgj = 0;
int opt_spl = 0;
int logging = 0;
FILE *logfile;
char errbuf[80];

int ijk[PAN][SLEN];
int ijk2[3][SLEN];
double s_x[SLEN], s_y[SLEN];
double s_x2[SLEN], s_y2[SLEN];
double s_xalt[MAXALT], s_yalt[MAXALT];
int s_nbdy[SLEN], s_iflag[SLEN];

int ipvtpe[NXPROC_C][LEN];
int ipvtmem[NXPROC_C][LEN];

int np;
int nodes;
int nk;
int nflag;
int nelx;
int nx;
int nint;
int npr;

int nelxblk;
int npblk;
int nodesblk;
int npadd;
int nodesadd;
plural double ahl[LEN];
plural double ah2[LEN];
plural double r_x[LEN], r_y[LEN];
plural double c_x[LEN], c_y[LEN];
plural double bcond[LEN];
plural double bval[LEN];
plural double xalt[LEN];
plural double yalt[LEN];
plural int r_nbday[LEN];
plural int c_nbday[LEN];
plural int c_iflag[LEN];
plural int r_realelem[LEN];
plural int c_realelem[LEN];
plural double tin[LEN];
plural double tinnl[LEN];
plural double bvec[LEN];
plural double c_cx[LEN];
plural double c_cy[LEN];
plural double r_cx[LEN];
plural double r_cxl[LEN];
plural double r_cx2[LEN];
plural double r_cx3[LEN];
plural double r_cy[LEN];
plural double r_cyl[LEN];
plural double r_cy2[LEN];
plural double r_cy3[LEN];
plural int realdelein[LEN];
plural double bcu[LEN][LEN];
plural double bck[LEN][LEN];
plural double dcu[LEN][LEN];
plural double dck[LEN][LEN];
plural double bc[LEN][LEN];
plural double dc[LEN][LEN];
plural double xmat[LEN][LEN];
plural double ymat[LEN][LEN];
plural double zmat[LEN][LEN];
plural double amat[LEN][LEN];

double dt;
double diffk;

double factor;
static plural double aloga, alogb;
static plural double h, u, v, thu, thv, w;

#include "bem.h"
#include <math.h>

plural double binom(int m, int i)
{
    plural double comb = 1.0;
    int im, ii, mmi, immi;
    for (im=1; im<=m; im++)
        comb *= (double) im;
    for (ii=1; ii<=i; ii++)
        comb /= (double) ii;
    mmi = m - i;
    for (immi=1; immi<=mmi; immi++)
        comb /= (double) immi;
    comb *= p_pwr(u,mmi);
    return comb;
}

plural double sfunc(int k)
{
    plural double result;
    plural double sum = 0.0;
    double pimeg = -pi;
    int l;
    int i, i2, i2m1;
    if (k == (k/2)*2)
\begin{verbatim}
{
l = k/2;
result = thv-thu;
if (p_fabs(w) < 1.0e-12) result = 0.0;
if (result < pineg) result += 2.0*pi;
if (result > pi) result -= 2.0*pi;

result *= p_pwr(w,2*(l-1));
for (i=1; i<=l; i++)
{
    i2ml = i*2-l;
    sum += nsngn(i) * (p_pwr(v,i2ml)-p_pwr(u,i2ml))
        * p_pwr(w,2*(l-i))/i2ml;
}
}

else
{
l = (k-1)/2;
result = (alogb-aloga)*p_pwr(w,2*l);
for (i=1;i<=l; i++)
{
    i2 = 2*i;
    sum += nsngn(i) * (p_pwr(v,i2)-p_pwr(u,i2)) * p_pwr(w,2*(l-i))/i2;
}

return nsngn(l) * (result + sum);
}

/*-----------------------------------------------*/
/*-----------------------------------------------*/
plural double vfunc(int k)
{
    plural double result;
    plural double sum = 0.0;
    double pineg = -pi;
    int l;
    int i, i2, i2ml;

    if (k == (k/2)*2)
    {
        l = k/2;
        result = thv-thu;
        if (p_fabs(w) < 1.0e-12) result = 0.0;
        if (result < pineg) result += 2.0*pi;
        if (result > pi) result -= 2.0*pi;
        result *= p_pwr(w,2*l);
        for (i=1; i<=l; i++)
        {
            i2ml = 2*i-l;
            sum += nsngn(i) * (p_pwr(v,i2ml)-p_pwr(u,i2ml))
                * p_pwr(w,2*(l-i)+1)/i2ml;
        }
    }

    return nsngn(l) * (result + sum);
}
\end{verbatim}
else
{
  l = (k-1)/2;
  result = (alogb-aloga)*p_pwr(w,2*l+1);
  for (i=l;i<=l; i++)
  {
    i2 = 2*i;
    sum += nsgn(i)*(p_pwr(v,i2)-p_pwr(u,i2))
             *p_pwr(w,2*(l-i)+l)/i2;
  }
  return nsgn(l) * (result + sum);
}

/*-------------------------------------------------------------*/
/*  */
plural double glint(int ns)
{
  return (p_pwr(v,ns+1)*alogb-p_pwr(u,ns+1)
           *aloga-sfunc(ns+2))/(ns+1);
}

/*-------------------------------------------------------------*/
/*  */
plural double glpintdnt ns)
{
  return vfunc(ns);
}

/*-------------------------------------------------------------*/
/*  */
/* Calculates integral of log r. */
/*-------------------------------------------------------------*/
plural double lim(int ink)
{
  plural double xint = 0.0;
  int m = ink-1;
  int mps;
  int ns;

  for (ns=0; ns<ink; ns++)
  {
    mps = m + ns;
    xint += nsgn(mps) * binom(m,ns) * glint(ns);
  }
  return xint;
}

/*-------------------------------------------------------------*/
/*  */
/* Calculates integral of 1/r. */
/**-----------------------------------------------**/
plural double l3m(int ink)
{
    plural double xint = 0.0;
    int m = ink - 1;
    int mps;
    int ns;

    for (ns=0; ns<ink; ns++)
    {
        mps = m + ns;
        xint += nsgn(mps) * binom(m,ns) * glpint(ns);
    }
    return xint;
}

/*-----------------------------------------------*/
/* This function performs the same function as drivr(), but has */
/* hand-crafted common subexpression elimination. */
/*-----------------------------------------------*/
void cse_drivr(plural double xp, plural double yp,
    plural double xfst, plural double yfst,
    plural double xlst, plural double ylst, int nk,
    plural double gl[], plural double glp[])
{
    register plural double d1 = xfst - xp;
    register plural double d2 = xlst - xp;
    register plural double d3 = yfst - yp;
    register plural double d4 = ylst - yp;
    register plural double d5 = xlst - xfst;
    register plural double d6 = ylst - yfst;

    register plural double a = p_sqrt(d1*d1+d3*d3);
    register plural double b = p_sqrt(d2*d2+d4*d4);

    register plural double phi = p_atan2(d6,d5);
    register plural double alpha = phi - 0.5*pi;

    register plural double tmp1 = p_cos(alpha);
    register plural double tmp2 = p_sin(alpha);

    register plural double d4m2 = d4*tmp1 - d2*tmp2;
    register plural double d2p4 = d2*tmp1 + d4*tmp2;
    register plural double d3m1 = d3*tmp1 - d1*tmp2;
    register plural double dip3 = d1*tmp1 + d3*tmp2;

    register int ink;

    h = p_sqrt(d5*d5+d6*d6);
if (a == 0.0)
    aloga = 0.0;
else
    aloga = p_log(a);

if (b == 0.0)
    alogb = 0.0;
else
    alogb = p_log(b);

/* PLAYING LOOSE WITH VARIABLE MEANINGS HERE TO AVOID REPEATING
   P_ATAN CALL UNNECESSARILY. OVERWRITING d??? VARIABLES WITH THE
   NECESSARY VALUE WHEN a OR b IS ZERO */
if (a == 0.0) /* LABEL 5 IN FORTRAN */
{
    d3ml = d4m2;
    dlp3 = d2p4;
}
if (b == 0.0) /* LABEL 8 IN FORTRAN */
{
    d4m2 = d3ml;
    d2p4 = dlp3;
}

thu = p_atan2(d3ml, dlp3);
thv = p_atan2(d4m2, d2p4);

/* LABEL 9 FORTRAN */
u = a*p_sin(thu);
v = b*p_sin(thv);
w = a*p_cos(thu);
for (ink=0; ink<nk; ink++)
{
    gl[ink] = llm(ink+1);
    glp[ink] = 13m(ink+1);
}

/* ----------------------------- */
/* If hand-crafted common subexpression elimination is enabled, */
/* this routine calls the CSE version of itself, cse_driver(). */
/* ----------------------------- */
void drivr(plural double xp, plural double yp,
           plural double xfst, plural double yfst,
           plural double xlst, plural double ylst, int nk,
           plural double gl[], plural double glp[])
{
    plural double d1, d2, d3, d4, d5, d6;
    plural double a, b;
    plural double alpha, phi;
plural double arg1, arg2;
int ink;

if (opt_cse)
{
    cse_drivr(xp, yp, xfst, yfst, xlst, ylst, nk, gl, glp);
    return;
}

d1 = xfst - xp;
d2 = xlst - xp;
d3 = yfst - yp;
d4 = ylst - yp;
d5 = xlst - xfst;
d6 = ylst - yfst;

h = p_sqrt(d5*d5+d6*d6);
a = p_sqrt(d1*d1+d3*d3);
b = p_sqrt(d2*d2+d4*d4);
phi = p_atan2(d6,d5);
alp = phi - 0.5 * pi;

if (a == 0.0)
{
    /* LABEL 5 IN FORTRAN */
    arg1 = d4*p_cos(alp)-d2*p_sin(alp);
    arg2 = d2*p_cos(alp)+d4*p_sin(alp);
thv = p_atan2(arg1,arg2);
    thu = thu;
    aloga = 0.0;
    alogb = p_log(b);
}
else if (b == 0.0)
{
    /* LABEL 8 IN FORTRAN */
    arg1 = d3*p_cos(alp) - d1*p_sin(alp);
    arg2 = d1*p_cos(alp) + d3*p_sin(alp);
    thu = p_atan2(arg1,arg2);
    thv = thu;
    aloga = p_log(a);
    alogb = 0.0;
}
else
{
    arg1 = d3*p_cos(alp) - d1*p_sin(alp);
    arg2 = d1*p_cos(alp) + d3*p_sin(alp);
    thu = p_atan2(arg1,arg2);
    arg1 = d4*p_cos(alp) - d2*p_sin(alp);
    arg2 = d2*p_cos(alp) + d4*p_sin(alp);
    thu = p_atan2(arg1,arg2);
    aloga = p_log(a);
\[
\text{alogb = p\_log}(b);
\]

/* LABEL 9 FORTRAN */
\[
u = a\times\text{p\_sin}(\text{thu});
\]
\[
v = b\times\text{p\_sin}(\text{thv});
\]
\[
w = a\times\text{p\_cos}(\text{thu});
\]

for (ink=0; ink<nk; ink++)
{
    gl[ink] = llm(ink+1);
    glp[ink] = l3m(ink+1);
}

/* This function is the driver for boundary integration. */
/* By the time we get here, assume that all the "dummy" elements */
/* have been constructed, so with each element is stored its nodes */
/* 0 through nk-2. (It's node nk-1 is node 0 of the next element */
/* to east.) Assume x and y are the nodal coordinates decomposed */
/* and copied into each ROW. xp and yp are the coordinates of the */
/* nodes we are collocating on, copied into each COLUMN. */

void int4(plural double xp, plural double yp)
{
    plural double tmpl, tmp2;
    plural double gl[6], glp[6];
    plural double xfst, yfst, xlst, ylst;
    int i,j;
    int nodeidx;

    for (i=0; i<nodesblk; i++)
        ahl[i] = ah2[i] = 0.0;
    for (i=0; i<npblk; i++)
    {
        /* ONLY PROCESS "REAL" ELEMENTS...DUMMY ELEMS CONTRIBUTE NOTHING */
        if (r_realelem[i])
            nodeidx = i*(nk-1);
            /* GET THE RIGHT X,Y VALUES */
            xfst = r_x[nodeidx];
            yfst = r_y[nodeidx];

            /* ELEMS SCATTERED...LAST NODE ALWAYS ON NEXT PE */
            all tmpl = r_x[nodeidx];
            xlst = xnetE[1].tmpl;
            all tmp2 = r_y[nodeidx];
            ylst = xnetE[1].tmp2;
drivr(xp, yp, xfst, yfst, xlst, ylst, nk, gl, glp);
shfunc(h, nk);
shape(nk, gl, h);
shape(nk, glp, h);

/* FOR EACH FACE, NODES 0 TO NK-2 ARE STORED WITH THE FACE */
/* NODE NK-1 IS STORED ON THE NEXT FACE TO THE EAST. */
for (j=0; j<nk-1; j++)
{
    ahl[nodeidx+j] -= glp[j];
    ah2[nodeidx+j] += gl[j];
}

/* NEXT FACE IS ON THE NEXT PROCESSOR */
/* HERE IS A PROBLEM. AS WRITTEN THIS WILL NOT WORK */
/* IF THE NUMBER OF BNDRY ELEMENTS IS NOT A MULTIPLE OF */
/* THE PE ARRAY LENGTH. THE PROBLEM IS THE LAST ELEM */
/* WILL NOT HAVE ITS "NEXT" ELEM EXACTLY ONE PE TO THE */
/* EAST. THIS CAN BE FIXED WITH SPECIAL CASE CODE, */
/* BUT WAS NOT NEEDED FOR THE ORIGINAL PROJECT. */
all tmpl = tinp2 = 0.0;
xnetE[1].tmpl = glp[j];
xnetE[1].tmp2 = gl[j];
all
{
    ahl[nodeidx] -= tmpl;
    ah2[nodeidx] += tmp2;
}
}
#include "bem.h"

plural double arow[LEN]; /* Temporary matrices used for Gauss- 
plural double brow[LEN]; /* Jordan elimination. */

/* This function inverts a matrix by LU decomposing it and solving */
/* for the columns of the identity matrix. It is terribly slow */
/* because the solves are done sequentially. */
/* NOTE the original matrix "a" is overwritten by its inverse. */
void invrt(int sby, int nblk, plural double a[LEN][LEN],
           int ipvtpe[NXPROC_C][LEN], int ipvtmem[NXPROC_C][LEN],
           plural double ahl[], plural double ah2[],
           plural double xmat[LEN][LEN])
{
  int ix, i, j;
  double cond;
  plural double tmp;

  cond = decomp(nblk, ipvtpe, ipvtmem, a);

  if (debug) fprintf(stderr, "Condition number = %lf in invrt\n", cond);

  for (ix=0; ix<nxproc; ix++)
  {
    for (i=0; i<nblk; i++)
    {
      for (j=0; j<nblk; j++)
      {
        ahl[j] = 0.0;
        if (ixproc==0 && iyproc==ix && j==i)
          ahl[j] = 1.0;
      }
      solve(nblk, ipvtpe, ipvtmem, a, ahl); /* SOLUTION IS IN PE COLUMN 0; WE NEED IT IN IX */
      for (j=0; j<nblk; j++)
      {
        tmp = ahl[j];
        if (ixproc==0)
xnetpE[ix].tmp = tmp;
if (ixproc==ix)
    xmat[j][i] = tmp;
}
}

for (i=0; i<nblk; i++)
    for (j=0; j<nblk; j++)
        a[i][j] = xmat[i][j];

/*---------------------------------------------------------------*/
/* This matrix inversion function is like the previous, except that */
/* the solves for the columns of the identity matrix are pipelined. */
/* */
/* NOTE the original matrix "a" is overwritten by its inverse. */
/*---------------------------------------------------------------*/
void plud_invrt(int sbly, int nblk, plural double a[LEN][LEN],
                int ipvtpe[NXPROC_C][LEN],
                int ipvtmem[NXPROC_C][LEN], plural double ahl[],
                plural double ah2[], plural double xmat[LEN][LEN])
{
    int i,j;
    double cond;
    plural double tmp;

    cond = decomp(nblk, ipvtpe, ipvtmem, a);
    if (debug) fprintf(stderr,
                       "Condition number = %lf in invrt\n", cond);

    for (i=0; i<nblk; i++)
    {
        for (j=0; j<nblk; j++)
        {
            ahl[j] = 0.0;
            if (ixproc==i & j==i)
                ahl[j] = 1.0;
        }

        m3olve(nblk, ipvtpe, ipvtmem, a, ahl);
        for (j=0; j<nblk; j++)
            xmat[j][i] = ahl[j];
    }

    for (i=0; i<nblk; i++)
        for (j=0; j<nblk; j++)
            a[i][j] = xmat[i][j];

/*---------------------------------------------------------------*/
/* This is a software-pipelined routine for swapping two rows of */
/* a matrix. Used by the Gauss-Jordan inversion function. */
/ *----------------------------------------------------------------------*/
void swaprow_spl(int nblk, plural double a[LEN][LEN], int pel, int pe2, int idx1, int idx2)
{
    register plural double ptmp;
    register int j;

    register plural double a1_p0, a1_p1;
    register plural double a2_p0, a2_p1;

    if (pel != pe2)
        /* PE ROW IS DIFFERENT */
        {
            a1_p0 = a[idx1][0];
            a2_p0 = a[idx2][0];
            for (j=0; j<nblk-1; j++)
            {
                a1_p1 = a[idx1][j+1];
                a2_p1 = a[idx2][j+1];
                /* FIRST COPY ELEMENT TO TEMP ON OTHER ROW */
                if (pe2-pel < 0)
                {
                    if (iyproc == pe2)
                        xnetS[pel-pe2].ptmp = a2_p0;
                    if (iyproc == pel)
                        xnetN[pel-pe2].ptmp = a1_p0;
                }
                else
                {
                    if (iyproc == pe2)
                        xnetN[pe2-pel].ptmp = a2_p0;
                    if (iyproc == pel)
                        xnetS[pe2-pel].ptmp = a1_p0;
                }
                /* THEN COPY IT BACK INTO THE ARRAY */
                if (iyproc == pe2)
                    a[idx2][j] = ptmp;
                if (iyproc == pel)
                    a[idx1][j] = ptmp;
                a1_p0 = a1_p1;
                a2_p0 = a2_p1;
            }
        }/* FIRST COPY ELEMENT TO TEMP ON OTHER ROW */
        if (pe2-pel < 0)
        {
            if (iyproc == pe2)
                xnetS[pel-pe2].ptmp = a2_p0;
            if (iyproc == pel)
                xnetN[pel-pe2].ptmp = a1_p0;
        }
        else
if (iyproc == pe2)
    xnetN[pe2-pel].ptmp = a2_p0;
if (iyproc == pel)
    xnetS[pe2-pel].ptmp = al_p0;
}
/* THEN COPY IT BACK INTO THE ARRAY */
if (iyproc == pe2)
    a[idx2][j] = ptmp;
if (iyproc == pel)
    a[idx1][j] = ptmp;
else if (idx1 != idx2)
    /* PE ROW IS SAME BUT IDX DIFFERENT */
    { if (iyproc == pel)
    {
        a1_p0 = a[idx1][0];
        a2_p0 = a[idx2][0];
        for (j=0; j<nblk-1; j++)
        {
            a1_pl = a[idx1][j+1];
            a2_pl = a[idx2][j+1];
            ptmp = a[idx1][j];
            a[idx1][j] = a[idx2][j];
            a[idx2][j] = ptmp;
            a1_p0 = a1_pl;
            a2_p0 = a2_pl;
        }
        ptmp = a[idx1][j];
        a[idx1][j] = a[idx2][j];
        a[idx2][j] = ptmp;
    }
    }
/*-----------------------------------------------*/
/* This function implements inversion as Gauss-Jordan elimination. */
/* This version includes source-level software pipelining. */
/*-----------------------------------------------*/
void gj_invrt_spl(int sby, int nblk,
    plural double a[LEN][LEN], plural double b[LEN][LEN])
{
    register int ix, tidx;
    register int i,j,k,m;
    register plural double atmp, btmp;

    register plural int bigpe;
    register plural int bigidx;
    register plural double bigval;
register int pivotpe;
register int pivotidx;
register double pivotval;

register plural double a_pO, a_pl;
register plural double b_pO, b_pl;
register plural double arow_pO, arow_pl;
register plural double brow_pO, brow_pl;

/* INITIALIZE b TO THE IDENTITY MATRIX */
for (i=0; i<nblk; i++)
    for (j=0; j<nblk; j++)
        b[i][j] = 0.0;
for (i=0; i<nblk; i++)
    if (ixproc==iyproc)
        b[i][i] = 1.0;

/* PERFORM GAUSS-JORDAN ELIMINATION */
for (i=0; i<nblk; i++)
{
    for (ix=0; ix<nxproc; ix++)
    {
        /* FIND PIVOT */
        if (ixproc == ix)
        {
            /* FIRST EACH PE FINDS ITS MAX */
            bigidx = i;
            if (iyproc >= ix)
                bigval = p_fabs(a[i][i]);
            else
                bigval = 0.0;

            a_pO = a[i+1][0];
            for (j=i+1; j<nblk; j++)
            {
                a_pl = a[j+1][i];
                a_p0 = p_fabs(a_p0);
                if (a_p0 > bigval)
                {
                    bigval = a_p0;
                    bigidx = j;
                }
                a_p0 = a_p0;
            }
            a_p0 = p_fabs(a_p0);
            if (a_p0 > bigval)
            {
                bigval = a_p0;
                bigidx = j;
            }
    }
/* NOW FIND THE PE ROW */
pivotpe = find_row_with_max(bigval, ix);
pivotval = proc[pivotpe][ix].bigval;
pivotidx = proc[pivotpe][ix].bigidx;
}

if (pivotval == 0.0)
{
    fprintf(stderr, "gj inversion error, 0 pivot\n");
    exit(-1);
}

/* FINALLY DO THE SWAP */
swaprow_spl(nblk, a, ix, pivotpe, i, pivotidx);
swaprow_spl(nblk, b, ix, pivotpe, i, pivotidx);

/* DIVIDE PIVOT ROW BY PIVOT ELEMENT */
atmp = proc[ix][ix].a[i][i];
if (iyproc == ix)
{
    a_p0 = a[i][0];
    b_p0 = b[i][0];
    a_p0 /= atmp;
    b_p0 /= atmp;
    for (j=0; j<nblk-1; j++)
    {
        a_pl = a[i][j+1];
        b_pl = b[i][j+1];
        a[i][j] = a_pl/a_p0;
        b[i][j] = b_pl/b_p0;
        a_p0 = a_pl /= atmp;
        b_p0 = b_pl /= atmp;
    }
    a[i][j] = a_p0;
    b[i][j] = b_p0;
}

/* BROADCAST PIVOT a AND b TO ALL ROWS AND SUBTRACT */
if (iyproc == ix)
{
    a_p0 = a[i][0];
    b_p0 = b[i][0];
    for (j=0; j<nblk-1; j++)
    {
        a_pl = a[i][j+1];
        b_pl = b[i][j+1];
        xnetcN[nyproc].atmp = a_p0;
        xnetcN[nyproc].btmp = b_p0;
        all arow[j] = atmp;
        all brow[j] = btmp;
        a_p0 = a_pl;
        }
b_p0 = b_pl;
}
xnetcN[nyproc].atmp = a_p0;
xnetcN[nyproc].btmp = b_p0;
all arow[j] = atmp;
all brow[j] = btmp;
*/
/* NOW UPDATE a AND b (EXCEPT FOR PIVOT ROW) */
for (j=0; j<nblk; j++)
{
  if (ixproc == ix)
  {
    xnetcE[nxproc].atmp = a[j][i];
  }
  if ((iyproc != ix) || (i != j))
  {
    arow_p0 = arow[0];
    brow_p0 = brow[0];
    a_p0 = a[j][0];
    b_p0 = b[j][0];
    for (k=0; k<nblk-l; k++)
    {
      arow_p1 = arow[k+1];
      brow_p1 = brow[k+1];
      a_p1 = a[j][k+1];
      b_p1 = b[j][k+1];
      a_p0 -= atmp*arow_p0;
      b_p0 -= atmp*brow_p0;
      a[j][k] = a_p0;
      b[j][k] = b_p0;
      a_p0 = a_p1;
      b_p0 = b_p1;
      arow_p0 = arow_p1;
      brow_p0 = brow_p1;
    }
    a_p0 -= atmp*arow_p0;
    b_p0 -= atmp*brow_p0;
    a[j][k] = a_p0;
    b[j][k] = b_p0;
  } /*
}
for (i=0; i<nblk; i++)
  for (j=0; j<nblk; j++)
    a[i][j] = b[i][j];

/*-----------------------------------------------*/
/* This is a non software-pipelined routine for swapping two rows */
/* of a matrix. Used by the Gauss-Jordan inversion function. */

#include "symbtab.h"

void swaprow(int nblk, plural double a[LEN][LEN], int pe1, int pe2,
             int idx1, int idx2)
{
    plural double ptmp;
    int j;

    if (pe1 != pe2)
        /* PE ROW IS DIFFERENT */
        {
            for (j=0; j<nblk; j++)
                /* FIRST COPY ELEMENT TO TEMP ON OTHER ROW */
                if (pe2-pe1 < 0)
                    /* PE ROW IS DIFFERENT */
                    {
                        if (iyproc == pe2)
                            xnetS[pe1-pe2].ptmp = a[idx2][j];
                        if (iyproc == pel)
                            xnetN[pe1-pe2].ptmp = a[idx1][j];
                    }
                else
                    {
                        if (iyproc == pe2)
                            xnetN[pe2-pe1].ptmp = a[idx2][j];
                        if (iyproc == pel)
                            xnetS[pe2-pe1].ptmp = a[idx1][j];
                    }
        }
        /* THEN COPY IT BACK INTO THE ARRAY */
        if (iyproc == pe2)
            a[idx2][j] = ptmp;
        if (iyproc == pe1)
            a[idx1][j] = ptmp;
    }
else if (idx1 != idx2)
    /* PE ROW IS SAME BUT IDX DIFFERENT */
    {
        if (iyproc == pel)
            for (j=0; j<nblk; j++)
                {
                    ptmp = a[idx1][j];
                    a[idx1][j] = a[idx2][j];
                    a[idx2][j] = ptmp;
                }
    }

/* This function implements inversion as Gauss-Jordan elimination. */
/* This version includes no source-level software pipelining. */
```c
void gj_invrt(int sby, int nblk,
        plural double a[LEN][LEN], plural double b[LEN][LEN])
{
    int ix, is;
    int i,j,k,m;
    plural double atmp, btmp;

    plural int bigpe;
    plural int bigidx;
    plural double bigval;

    int pivotpe;
    int pivotidx;
    double pivotval;

    if (opt_spl)
    {
        gj_invrt_spl(sby, nblk, a, b);
        return;
    }

    /* INITIALIZE b TO THE IDENTITY MATRIX */
    for (i=0; i<nblk; i++)
        for (j=0; j<nblk; j++)
            b[i][j] = 0.0;
    for (i=0; i<nblk; i++)
        if (ixproc==iyproc)
            b[i][i] = 1.0;

    /* PERFORM GAUSS-JORDAN ELIMINATION */
    for (i=0; i<nblk; i++)
    {
        for (ix=0; ix<nxproc; ix++)
        {
            /* FIND PIVOT */
            if (ixproc == ix)
            {
                /* FIRST EACH PE FINDS ITS MAX */
                bigidx = i;
                if (iyproc >= ix)
                    bigval = p_fabs(a[i][i]);
                else
                    bigval = 0.0;

                for (j=i+1; j<nblk; j++)
                {
                    if (p_fabs(a[j][i]) > bigval)
                    {
                        bigval = p_fabs(a[j][i]);
                    }
                }
            }
        }
    }
```

bigidx = j;
}

/* NOW FIND THE PE ROW */
pivotpe = find_row_with_max(bigval, ix);
pivotval = proc[pivotpe][ix].bigval;
pivotidx = proc[pivotpe][ix].bigidx;
}

if (pivotval == 0.0)
{
    fprintf(stderr, "gj inversion error, 0 pivot\n");
    exit(-1);
}

/* FINALLY DO THE SWAP */
swaprow(nblk, a, ix, pivotpe, i, pivotidx);
swaprow(nblk, b, ix, pivotpe, i, pivotidx);

/* DIVIDE PIVOT ROW BY PIVOT ELEMENT */
atmp = proc[ix][ix].a[i][i];
if (iyproc == ix)
    for (j=0; j<nblk; j++)
      { 
      a[i][j] /= atmp;
      b[i][j] /= atmp;
      }

/* BROADCAST PIVOT a AND b TO ALL ROWS AND SUBTRACT */
if (iyproc == ix)
{
    for (j=0; j<nblk; j++)
      {
      xnetcN[nyproc].atmp = a[i][j];
      all arow[j] = atmp;
      xnetcN[nyproc].btmp = b[i][j];
      all brow[j] = btmp;
      }
}

/* NOW UPDATE a AND b (EXCEPT FOR PIVOT ROW) */
for (j=0; j<nblk; j++)
{
    if (ixproc == ix)
        xnetcE[nxproc].atmp = a[j][i];
    if ((iyproc != ix) || (i != j))
      {
      for (k=0; k<nblk; k++)
            { 
             a[j][k] -= atmp*arow[k];
             }
for (i=0; i<nblk; i++)
    for (j=0; j<nblk; j++)
        a[i][j] = b[i][j];

/* This function implements matrix inversion by LU decomposing the */
/* matrix and then solving for the columns of the identity matrix. */
/* It performs nxproc solves simultaneously, and achieves better */
/* processor utilization than the pipelined LU solve inversion */
/* function. */

void col_invrt(int nblk, plural double a[LEN][LEN],
               int ipvtpe[NXPROC_C][LEN], int ipvtmem[NXPROC_C][LEN],
               plural double xmat[LEN][LEN])
{
    int ix, i, j;
    double cond;

    cond = decomp(nblk, ipvtpe, ipvtmem, a);

    if (debug) fprintf(stderr, "Condition number = \%lf in invrt\n", cond);

    /* CONSTRUCT THE IDENTITY MATRIX IN XMAT */
    for (i=0; i<nblk; i++)
        for (j=0; j<nblk; j++)
            xmat[i][j] = 0.0;
    if (ixproc == iyproc)
        for (i=0; i<nblk; i++)
            xmat[i][i] = 1.0;

    col_solve(nblk, ipvtpe, ipvtmem, a, xmat);

    for (i=0; i<nblk; i++)
        for (j=0; j<nblk; j++)
            a[i][j] = xmat[i][j];
}
/** FILE: io.m */
/* Boundary Element Method (BEM) Application */
/* Jeffrey S. Clary */
/* Iowa State University */
/* *
/* This file contains the routines to read the geometry into *
/* singular arrays, and echo it. Other routines will handle the *
/* job of mapping it onto the DPU. *
/* */
/* Assumptions about input data: *
/* - boundary elements are listed in ijk[] in counterclockwise *
/* order around the domain. */

#include "bem.h"

/* Read the problem geometry into scalar arrays. */

void initread()
{
    int i, ir;

    /* READ BOUNDARY INFORMATION */
    /* NUMBER OF PANELS, NUMBER OF NODES */
    if (scanf("%d %d", &np, &nodes) != 2)
        errexit("in initread getting number of panels and nodes");
    /* FOR EACH NODE COORDS, BOUND TYPE, AND DOUBLE PT. FLAG */
    for (ir=0; ir<nodes; ir++)
        if (scanf("%lf %lf %d %d", &s_x[ir], &s_y[ir], &s_nbdy[ir], &s_iflag[ir]) != 4)
            sprintf(errbuf, "in initread getting (x,y,nbdy,iflag) {ir=%d}", ir);
            errexit(errbuf);
    
    /* NUMBER OF NODES PER PANEL */
    if (scanf("%d", &nk) != 1)
        errexit("in initread getting nk");
    /* CONNECTIVITY MATRIX */

    for (ir=0; ir<np; ir++)
        for (i=0; i<nk; i++)
            if (scanf("%d", &ijk[i][ir]) != 1)
                sprintf(errbuf, "in initread getting ijk[%d][%d]", i, ir);
                errexit(errbuf);
/* NFLAG IS # OF PTS. WITH IFLAG SET */
if (scanf("%d", &nflag) != 1)
    errexit("in initread getting nflag");
/* ALTERNATE COLLOCATION PTS. */
for (ir=0; ir<nflag; ir++)
    if (scanf("%lf %lf", &cs_xalt[ir], &cs_yalt[ir]) != 2)
    
        sprintf(errbuf, "initread getting (xalt[%d],yalt[%d])", ir, ir);
        errexit(errbuf);

/* READ DOMAIN INFORMATION */
/* */
if (scanf("%d %d", &nelx, &nx) != 2)
    errexit("in initread getting nelx and nx");
for (ir=0; ir<nx; ir++)
    if (scanf("%lf %lf", &s_x2[ir], &s_y2[ir]) != 2)
    
        sprintf(errbuf, "initread getting (x2[%d],y2[%d])", ir, ir);
        errexit(errbuf);
for (ir=0; ir<nelx; ir++)
    for (i=0; i<3; i++)
        if (scanf("%d", &ci_jk[i][ir]) != 1)
    
        sprintf(errbuf, "initread getting ijk[%d][%d]", i, ir);
        errexit(errbuf);

/* NINT = NUMBER OF TIME-MARCHING STEPS */
/* DT = TIME STEP */
/* IORDER = ORDER OF TIME FINITE-DIFFERENCING */
if (scanf("%d %lf %lf", &nint, &dt, &diffk) != 3)
    errexit("in initread getting nint, dt, and diffk");
if (scanf("%d", &npr) != 1)
    errexit("in initread getting npr");

/*--------------------------------------------------------------------------------*/
/* Write out the geometry from the scalar arrays. */
/*--------------------------------------------------------------------------------*/
void initwrite()
{
    int i, ir;

    printf("Number panels = %8d\n", np);
    printf("Nodes = %8d\n", nodes);
    printf("Nodes/Panel = %8d\n", nk);
    
}
printf("# Iflags set = %8d\n", nflag);
printf("Nx = %8d\n", nx);
printf("Nelx = %8d\n", nelx);
printf("Nint = %8d\n", nint);
printf("Dt = %11.21f\n", dt);
printf("Diffusivity = %11.21f\n", diffk);

printf("\nBoundary Information\n");
printf(" Node X Y Nbdy Iflag\n");
for (ir=0; ir<nodes; ir++)
    printf("%4d %10.21f %11.21f %6d %8d\n", ir, s_x[ir], s_y[ir], s_nbdy[ir], s_iflag[ir]);

printf("\nConnectivity Information\n");
printf(" Panel Local Node # Global Node #\n");
for (ir=0; ir<np; ir++)
    for (i=0; i<nk; i++)
        printf("%6d %10d %18d\n", ir, i, ijk[i][ir]);

printf("\nAlternate collocation points\n");
printf(" Point # X Y\n");
for (ir=0; ir<nflag; ir++)
    printf("%5d %10.21f %10.21f\n", ir, s_xalt[ir], s_yalt[ir]);

printf("\nDomain Information\n");
printf(" Point2 # X2 Y2\n");
for (ir=0; ir<nx; ir++)
    printf("%6d %13.21f %10.21f\n", ir, s_x2[ir], s_y2[ir]);

printf("\nConnectivity Information\n");
printf(" Panel Local Node # Global Node #\n");
for (ir=0; ir<nelx; ir++)
    for (i=0; i<3; i++)
        printf("%6d %10d %16d\n", ir, i, ijk2[i][ir]);
printf("\n");}
This function inserts boundary conditions.

\[
\begin{align*}
\text{nbdy == 1} & \implies \psi \text{ unknown} \\
\text{nbdy == 2} & \implies \psi_i \text{ unknown}
\end{align*}
\]

```c
void bdry(plural double a[LEN][LEN], plural double b[LEN][LEN],
int bndcoll, int i)
{
    register int j;

    for (j=0; j<nodesblk; j++)
    {
        if (r_nbdy[j] == 1)
        {
            a[i][j] = ah1[j];
            b[i][j] = ah2[j];
        }
        else
        {
            a[i][j] = ah2[j];
            b[i][j] = ah1[j];
        }
    }
}
```

This is the driver function for collocation.

```c
void matvec(void)
{
    int i, j, icc;

    plural double cc;
    plural double xp, yp;

    plural double scc;
```
/* COLLOCATING AT BOUNDARY NODES */
for (i=0; i<nodesblk; i++)
{
    /* ADDED DUMMY NODES MUST BE HANDLED DIFFERENTLY */
    /* AS DO PE MEMORY LOCs PAST END OF LOGICAL MATRIX */
    /* WHEN NODES NOT A MULTIPLE OF NXPROC */
    if ( ( !c_realelem[i/(nk-1)] && i%(nk-1)!=0)
        || (iyproc>nodes/nodesblk || (iyproc==nodes/nodesblk
            && i>=(nodes%nodesblk))))
    {
        for (j=0; j<nodesblk; j++)
        {
            bcu[i][j] = 0.0;
            bck[i][j] = 0.0;
        }
        for (j=0; j<nelxblk; j++)
        {
            bc[i][j] = 0.0;
        }
        if (iyproc == ixproc)
        {
            bcu[i][i] = 1.0;
            bck[i][i] = 1.0;
            bc[i][i] = 1.0;
        } 
        else /* THIS IS A REAL DATA NODE */
        {
            if (c_iflag[i] == 0)
            {
                xp = c_x[i];
                yp = c_y[i];
            } 
            else /* DOUBLE-NODE COLLOC. AT DIRECHLET-DIRECHLET CORNER */
            {
                xp = xalt[i];
                yp = yalt[i];
            }
        }
    } /* INTEGRATION ON THE BOUNDARY */
    int4(xp, yp);
    /* CALCULATION OF RIGID BODY TERM */
    all cc = 0.0;
    for (icc=0; icc<nodesblk; icc++)
    {
        cc += ahl[icc];
    }
    all see = - sum_to_diag(cc);
    if (ixproc == iyproc)
    {
        ahl[i] = see;
    } /* THE FOLLOWING IS NECESSARY FOR EXTERNAL PROBLEMS */
if (ahl[i] < 0.0)
    ahl[i] += 2*pi;

/* BDRY INSERTS THE APPROPRIATE BOUNDARY CONDITIONS */
/* BCU IS THE COEFF. MATRIX MULTIPLYING THE UNKNOWNS ON */
/* THE BOUNDARY. SIMILARLY BCK IS THE MATRIX MULTIPLYING */
/* THE KNOWNS ON THE BOUNDARY. BCU AND BCK ARE THE */
/* MATRICES FOR BOUNDARY COLLOCATION. */
bdry(bcu, bck, 1, i);

/* INTEGRATION OVER THE DOMAIN. (INTD DRIVES DOMAIN */
/* INTEGRATION ROUTINES */
intd(xp, yp, 1, i);
}

if (debug)
{
    printf("boundary collocation done\n");
    fflush(stdout);
}

/* COLLOCATING AT DOMAIN NODES */
for (i=0; i<nelxblk; i++)
{
    /* DON'T PROCESS PAST END OF MATRIX */
    if (realdelem[i])
    {
        xp = c_cx[i];
        yp = c_cy[i];

        /* INTEGRATION ON THE BOUNDARY */
        intd(xp, yp);
        /* DCU IS THE COEFF. MATRIX MULTIPLYING THE UNKNOWNS ON */
        /* THE BOUNDARY. SIMILARLY DCK IS THE MATRIX MULTIPLYING */
        /* THE KNOWNS ON THE BOUNDARY. DCU AND DCK ARE THE */
        /* MATRICES FOR DOMAIN COLLOCATION. */
        bdry(dcu, dck, 0, i);

        /* INTEGRATION OVER THE DOMAIN. (INTD DRIVES DOMAIN */
        /* INTEGRATION ROUTINES */
        intd(xp, yp, 0, i);
    }
    else
    /* FIX UP THE UNUSED ARRAY SLOTS AT THE END AS AN IDENTITY */
    /* MATRIX TO KEEP TO SIMPLIFY CODE AT THE EXPENSE OF */
    /* EXECUTION TIME ON PROBLEMS NOT MULTIPLE OF PE ARRAY */
    {
        for (j=0; j<nodesblk; j++)
        {...
dcu[i][j] = 0.0;
dck[i][j] = 0.0;
}
for (j=0; j<nelxblk; j++)
{
    dc[i][j] = 0.0;
}
}
# FILE: mmult.m
/*
* Boundary Element Method (BEM) Application
* Jeffrey S. Clary
* Iowa State University
* This file contains matrix multiplication functions. The matmult algorithm used was derived using the systolic method of Kothari et al.

#include "bem.h"

/* Macro to shift a matrix using xnet communication. */
/* Macro to shift a matrix using xnet communication. Source-level software pipelined version. */
#define shift(command,X,rows,cols) \
{ \
  register unsigned i,j; \
  register plural double xtmp; \
  for (i=0; i<rows; i++) \
    for (j=0; j<cols; j++) 
      { 
        xtmp = X[i][j]; 
        command[1].xtmp = xtmp; 
        X[i][j] = xtmp; 
      }
} 

#define shift_spl(command,X,rows,cols) \
{ \
  register plural double x_pO, x_pl; \
  register unsigned i,j; \
  x_pO = X[i][0]; \
  for (j=1; j<cols; j++) 
    { 
      x_pl = X[i][j]; 
      command[1].x_pO = x_pO; 
      X[i][j-1] = x_pO; 
      x_pO = x_pl; 
    } 
  command[1].x_pO = x_pO; 
  X[i][j-1] = x_pO; 
}
This function computes the matrix product \( C = A \times B \).

The arrays are treated as if they are 2D block decomposed with \( \text{arows} \), \( \text{acols} \) specifying the number of rows, cols per submatrix, and \( \text{bcols} \) specifying the number of columns per submatrix. It turns out that it is OK if the matrices are actually scatter decomposed. The same computations get performed, just in a different order.

It is assumed that the # of cols in A == # rows in B.

This is the source-level software pipelined version.

```c
void mmmult_spl(int arows, int acols, int bcols,
    plural double A[LEN][LEN],
    plural double B[LEN][LEN],
    plural double C[LEN][LEN])
{
    register plural double ctmp;
    register plural double a_p0, a_pl;
    register plural double b_p0, b_pl;

    register unsigned iter, i, j, k;

    /* ZERO OUT C MATRIX */
    for (i=0; i<arows; i++)
        for (j=0; j<bcols; j++)
            C[i][j] = 0.0;

    /* SHIFT A SO THAT DIAGONAL ELEMENTS ARE AT RIGHT EDGE */
    for (i=nyproc; i>0; i--)
        if (iyproc < i)
            shift_spl(xnetE, A, arows, acols)
    
    /* SHIFT B SO THAT DIAGONAL ELEMENTS ARE AT BOTTOM EDGE */
    for (i=nxproc; i>0; i--)
        if (ixproc < i)
            shift_spl(xnetS, B, acols, bcols);
```
/ * ITERATE FOR LENGTH OF PE ARRAY */
   for (iter=nxproc; iter; iter--)
   {
      /* EACH PE CALC C=A*B ON ITS SUBMATRIX */
      for (i=0; i<arows; i++)
      {
         for (j=0; j<bcols; j++)
         {
            /* DOTPRODUCT */
            a_p0 = A[i][0];
            b_p0 = B[0][j];
            ctmp = 0.0;
            for (k=l; k<acols; k++)
            {
               a_pl = A[i][k];
               b_pl = B[k][j];
               ctmp += a_p0 * b_p0;
               a_p0 = a_pl;
               b_p0 = b_pl;
            }
            ctmp += a_p0 * b_p0;
            C[i][j] += ctmp;
         }
      }
      /* SHIFT A,B ACCORDING TO SYSTOLIC ALGORITHM */
      shift_spl(xnetW, A, arows, acols)
      shift_spl(xnetN, B, acols, bcols)
   }
   /* Put A BACK THE WAY WE FOUND IT */
   for (i=nyproc; i>0; i--)
      if (iyproc < i)
         shift_spl(xnetW, A, arows, acols)
   /* Put B BACK THE WAY WE FOUND IT */
   for (i=nxproc; i>0; i--)
      if (ixproc < i)
         shift_spl(xnetN, B, acols, bcols)
   
/* This function computes the matrix product C = A * B. */
/* This is the non source-level software pipelined version. */
#include <stdio.h>
#include <stdlib.h>

void mmmult(int arows, int acols, int bcols,
            plural double A[LEN][LEN],
            plural double B[LEN][LEN],
            plural double C[LEN][LEN])
{
   register plural double ctmp;
}
register unsigned iter,i,j,k;

if (opt_spl)
{
    mmult_spl(arows, acols, bcols, A, B, C);
    return;
}

/* ZERO OUT C MATRIX */
for (i=0; i<arows; i++)
    for (j=0; j<bcols; j++)
        C[i][j] = 0.0;

/* SHIFT A SO THAT DIAGONAL ELEMENTS ARE AT RIGHT EDGE */
for (i=nyproc; i>0; i--)
    if (iyproc < i)
        shift(xnetE, A, arows, acols)

/* SHIFT B SO THAT DIAGONAL ELEMENTS ARE AT BOTTOM EDGE */
for (i=nxproc; i>0; i--)
    if (ixproc < i)
        shift(xnetS, B, acols, bcols)

/* ITERATE FOR LENGTH OF PE ARRAY */
for (iter=nxproc; iter; iter--)
{
    /* EACH PE CALC C=A*B ON ITS SUBMATRIX */
    for (i=0; i<arows; i++)
        for (j=0; j<bcols; j++)
        {
            ctmp = 0;
/* DOTPRODUCT */
            for (k=0; k<acols; k++)
                ctmp += A[i][k]*B[k][j];
            C[i][j] += ctmp;
        }

    /* SHIFT A,B ACCORDING TO SYSTOLIC ALGORITHM */
    shift(xnetW, A, arows, acols)
    shift(xnetN, B, acols, bcols)
}
/* Put A BACK THE WAY WE FOUND IT */
for (i=nyproc; i>0; i--)
    if (iyproc < i)
        shift(xnetW, A, arows, acols)

/* Put B BACK THE WAY WE FOUND IT */
for (i=nxproc; i>0; i--)
    if (ixproc < i)
        shift(xnetN, B, acols, bcols)
### Boundary Element Method (BEM) Application

This file contains functions implementing shape functions of the BEM.

```c
#include "bem.h"

static plural double sh[PAN][PAN];

void shfunc(plural double h, int nk)
{
    plural double hsq = h * h;
    plural double hcu = hsq * h;

    switch (nk)
    {
    case 2:
        sh[0][0] = 1.0;
        sh[0][1] = -1.0 / h;
        sh[1][0] = 0.0;
        sh[1][1] = -sh[0][1];
        break;
    case 3:
        sh[0][0] = 1.0;
        sh[0][1] = -3.0 / h;
        sh[0][2] = 2.0 / hsq;
        sh[1][0] = 0.0;
        sh[1][1] = 4.0 / h;
        sh[1][2] = -4.0 / hsq;
        sh[2][0] = 0.0;
        sh[2][1] = -1.0 / h;
        sh[2][2] = 2.0 / hsq;
        break;
    case 4:
        sh[0][0] = 1.0;
        sh[0][1] = -5.5 / h;
        sh[0][2] = 9.0 / hsq;
        sh[0][3] = -4.5 / hcu;
        sh[1][0] = 0.0;
        sh[1][1] = 9.0 / h;
        sh[1][2] = -22.5 / hsq;
    ```
```
sh[1][3] = 13.5 / hcu;
sh[2][0] = 0.0;
sh[2][1] = -4.5 / h;
sh[2][2] = 18.0 / hsq;
sh[2][3] = -13.5 / hcu;
sh[3][0] = 0.0;
sh[3][1] = 1.0 / h;
sh[3][2] = -4.5 / hsq;
sh[3][3] = 4.5 / hcu;
break;
}

/*---------------------------------------------*/
/* This is the shape function driver function. */
/*---------------------------------------------*/
void shape(int nk, plural double gg[], plural double h)
{
  plural double ggd[PAN];
  int i,j;

  /* Moved to int4() for efficiency
  shfunc(h, nk);
  */

  for (i=0; i<nk; i++)
  {
    ggd[i] = 0.0;
    for (j=0; j<nk; j++)
      ggd[i] += gg[j] * sh[i][j];
  }

  for (i=0; i<nk; i++)
    gg[i] = ggd[i];
}
/* File: solver.m */
/* Boundary Element Method (BEM) Application */
/* Jeffrey S. Clary */
/* Iowa State University */
/* This file contains the functions for LU decomposition and */
/* solving. */
/* The algorithm was adapted from an algorithm by Youngtae Kim. */
/* These routines assume that the size of the matrix to be solved */
/* is nblk*nxproc, and that the PE array is SQUARE. */
/*******************************************************************/
#include "bem.h"
plural double acol[LEN];

/*******************************************************/
/* This function implements lu decomposition with source-level */
/* software pipelining. */
/*******************************************************/
double decomp_spl(int nblk, int ipvtpe[NXPROC_C][LEN],
    int ipvtmem[NXPROC_C][LEN], plural double lu[LEN][LEN])
{
    register plural double ptmp, prow[LEN];
    register plural double prow_p0, prow_pl;
    register plural double lu_p0, lu_pl;
    register plural double rowmult;
    register unsigned ix,i,j,k;
    register plural int bigpe;
    register plural int bigidx;
    register plural double bigval;
    register int pivotpe;
    register int pivotidx;
    register double pivotval;

    /* TWO LOOPS TO PROCESS EACH PE COL AND EACH ELEM OF BLOCK */
    for (i=0; i<nblk; i++)
    {
        for (ix=0; ix<nxproc; ix++)
        {
            /* pivot finding is kludged till we find a better way */
            /* FIND PIVOT AND SWAP IF NECESSARY */
            /* FIRST EACH PROC FINDS ITS OWN BIGGEST */
            if (ixproc==ix)
            {
                bigidx = i;
            }
if (iyproc >= ix)
    bigval = p_fabs(lu[i][i]);
else
    bigval = 0.0;

/* WHEN ONLY ONE ELEM PER PROC IS LEFT, WE MUST NOT
PERFORM THIS LOOP */
if (i+1 < nblk)
{
    lu_p0 = lu[i+1][i];
    for (j=i+1; j<nblk-1; j++)
    {
        lu_pl = lu[j+1][i];
        lu_p0 = p_fabs(lu_p0);
        if (lu_p0 > bigval)
            {
            bigidx = j;
            bigval = lu_p0;
            }
        lu_p0 = lu_pl;
    }
    if (lu_p0 > bigval)
        {
        bigidx = j;
        bigval = lu_p0;
        }
}
/* NOW FIND THE PE ROW */
pivotpe = find_row_with_max(bigval,ix);
pivotval=proc[pivotpe][ix].bigval;
pivotidx = proc[pivotpe][ix].bigidx;

if (pivotval == 0.0)
{
    fprintf(stderr, "error, 0 pivot\n");
    exit(-1);
}
/* ipvtpe is the pe offset to the pivot */
/* ipvtmem is the mem offset to the pivot */
ipvtpe[ix][i] = pivotpe - ix;
ipvtmem[ix][i] = pivotidx - i;

/* FINALLY DO THE SWAP */
/* SWAP ONLY AFFECTS PIVOT AND TO RIGHT OF PIVOT */
if (pivotpe != ix)
/* PE ROW IS DIFFERENT */
{
    /* FIRST COPY PIVOT ROW TO prow ON CURRENT PE ROW */
    lu_p0 = lu[pivotidx][i];
for (j=i; j<nblk-1; j++)
{
    lu_pl = lu[pivotidx][j+1];
    if (j>i || ixproc==ix)
    {
        if (iyproc == pivotpe)
        {
            ptmp = lu_p0;
            xnetcN[pivotpe-ix].ptmp = ptmp;
        }
        if (iyproc == ix)
            prow[j] = ptmp;
    }
    lu_p0 = lu_pl;
}
if (j>i || ixproc==ix)
{
    if (iyproc == pivotpe)
    {
        ptmp = lu_p0;
        xnetcN[pivotpe-ix].ptmp = ptmp;
    }
    if (iyproc == ix)
        prow[j] = ptmp;
}

/* NEXT COPY CURRENT ROW TO PIVOT ROW */
lu_p0 = lu[i][i];
for (j=i; j<nblk-1; j++)
{
    lu_pl = lu[i][j+1];
    if (j>i || ixproc==ix)
    {
        if (iyproc == ix)
        {
            ptmp = lu_p0;
            xnetcS[pivotpe-ix].ptmp = ptmp;
        }
        if (iyproc == pivotpe)
            lu[pivotidx][j] = ptmp;
    }
    lu_p0 = lu_pl;
}
if (j>i || ixproc==ix)
{
    if (iyproc == ix)
    {
        ptmp = lu_p0;
        xnetcS[pivotpe-ix].ptmp = ptmp;
    }
    if (iyproc == pivotpe)
```

lu[pivotidx][j] = ptmp;

/* FINALLY COPY TMP BACK TO CURRENT ROW, COMPLETING SWAP */
if (iyproc == ix)
    for (j=i; j<nblk; j++)
        if (j>i)
            lu[i][j] = prow[j];
        else if (ixproc>=ix)
            lu[i][j] = prow[j];
else if (pivotidx != i)
    /* PE ROW IS SAME BUT IDX DIFFERENT */
    {
        if (iyproc == ix)
            for (j=i; j<nblk; j++)
                if (j>i || ixproc>=ix)
                    {
                        ptmp = lu[i][j];
                        prow[j] = lu[i][j] = lu[pivotidx][j];
                        lu[pivotidx][j] = ptmp;
                    }
    }
else
    /* NO PIVOT NEEDED */
    {
        if (iyproc==ix)
            for (j=i; j<nblk; j++)
                if (j>i)
                    prow[j] = lu[i][j];
                else if (ixproc>=ix)
                    prow[j] = lu[i][j];
    }

/* BROADCAST THE PIVOT ROW DOWN THE COLUMNS */
/* ALL VALUES OF prow THAT REPRESENT ELEMS TO THE LEFT */
/* OF THE CURRENT PIVOT REMAIN 0.0 */
if (iyproc == ix)
    {
        /* INVERT THE DIAGONAL ELEMENT SO WE CAN USE MULTIPLICATION */
        /* INSTEAD OF DIVISION LATER */
        if (ixproc == ix)
            prow[i] = 1.0 / prow[i];
        prow_p0 = prow[i];
        for (j=i; j<nblk-1; j++)
            {
                prow_p1 = prow[j+1];
                ptmp = 0.0;
```
if (j>i)
    ptmp = prow_p0;
else if (ixproc==ix)
    ptmp = prow_p0;
xnetcS[nyproc].ptmp = ptmp;
all prow[j] = ptmp;
prow_p0 = prow_pl;
}
ptmp = 0.0;
if (j>i)
    ptmp = prow_p0;
else if (ixproc==ix)
    ptmp = prow_p0;
xnetcS[nyproc].ptmp = ptmp;
all prow[j] = ptmp;
}

/* FOR EACH SUBROW BELOW CURRENT */
for (j=i; j<nblk; j++)
{
    /* BROADCAST CONSTANT MULTIPLIER ACROSS THE ROWS */
    if (ixproc == ix)
    {
        rowmult = - lu[j][i] * prow[i];
xnetcE[nxproc].rowmult = rowmult;
        if (j>i)
            lu[j][i] = rowmult;
        else if (iyproc>ix)
            lu[j][i] = rowmult;
    }
    if (j>i || iyproc>ix)
    {
        /* Add (MULTIPLIER * PIVOT_ROW) TO ROW */
prow_p0 = prow[i];
    lu_p0 = lu[j][i];
    for (k=i; k<nblk-1; k++)
    {
        prow_p1 = prow[k+1];
    lu_p1 = lu[j][k+1];
    lu_p0 += rowmult * prow_p0;
        if (k>i)
            lu[j][k] = lu_p0;
        else if (ixproc<ix)
            lu[j][k] = lu_p0;
prow_p0 = prow_p1;
    lu_p0 = lu_p1;
    }
    lu_p0 += rowmult * prow_p0;
    if (k>i)
        lu[j][k] = lu_p0;
    else if (ixproc>ix)

lu[j][k] = lu_p0;
}
}
}

return 0.0;
}

/*---------------------------------------------*/
/* This function solves for vector b (overwriting it with the */
/* solution vector, given an LU decomposed matrix a. This is the */
/* source-level software pipelined version. */
/*---------------------------------------------*/

void solve_spl(int nblk, int ipvtpe[NXPROC_C][LEN], int ipvtmem[NXPROC_C][LEN],
    plural double a[LEN][LEN], plural double b[LEN])
{
    register int i,j,k;
    register plural double a_p0, a_pl;
    register plural double b_p0, b_pl;
    register plural double btmp;
    register double tmp;
    register int ipvtpeelem;
    register int ipvtmemelem;

    /* FIRST FORWARD SOLVE L --- SWEEP RIGHT */
    for (j=0; j<nblk; j++)
    {
        for (i=0; i<nxproc; i++)
        {
            if (ixproc==i)
            {
                /* RECREATE PIVOT SWAP */
                tmp = proc[i][i].b[j];
                ipvtpeelem = ipvtpe[i][j];
                ipvtmemelem = ipvtmem[i][j];
                proc[i][i].b[j] = proc[i+ipvtpeelem][i].b[j+ipvtmemelem];
                proc[i+ipvtpeelem][i].b[j+ipvtmemelem] = tmp;

                /* BROADCAST CURRENT b DOWN COLUMN */
                if (iyproc==i)
                {
                    btmp = b[j];
                    xnetcS[nyproc].btmp = btmp;
                }

                /* UPDATE USING COLUMN OF a */
                /* SHOULD HANDLE TWO CASES HERE */
                a_p0 = a[j][j];
                for (k=j; k<nblk-1; k++)
                {
                    a_pl = a[k][j];
                    btmp = b[k];
                    for (i=0; i<nxproc; i++)
                    {
                        btmp = b[i][k] + a_pl*proc[i][k].b;
                        proc[i][k].b = btmp;
                    }
                }
            }
        }
    }
}


\[ a_{pl} = a[k+1][j]; \]

if \((k>j)\)

\[ b[k] += a_{p0} \times btmp; \]

else if \((iyproc>i)\)

\[ b[k] += a_{p0} \times btmp; \]

\[ a_{p0} = a_{pl}; \]

if \((k>j)\)

\[ b[k] += a_{p0} \times btmp; \]

else if \((iyproc>i)\)

\[ b[k] += a_{p0} \times btmp; \]

\/* SHIFT b VECTOR TO NEXT COLUMN EAST */

if \((i<nxproc-1 || j<nlbk-1)\)

\{ \n
\[ b_{p0} = b[0]; \]

for \((k=0; k<nlbk-1; k++)\)

\{ \n
\[ b_{pl} = b[k+1]; \]

\[ xnetE[i].b_{p0} = b_{p0}; \]

\[ b[k] = b_{p0}; \]

\[ b_{p0} = b_{pl}; \]

\} \n
\[ xnetE[i].b_{p0} = b_{p0}; \]

\[ b[k] = b_{p0}; \]

\}

\} \n
\} 

\} 

\} 

\} 

\} 

\} 

\}
a_pl = a[k+1][j];
/* if (k<j || (k==j & iyproc<i)) */
b[k] += a_p0 * btmp;
a_p0 = a_pl;
}
if (iyproc<i)
b[k] += a_p0 * btmp;

/* SHIFT b VECTOR TO NEXT COLUMN WEST */
if (i>0 || j>0)
all
{
b_p0 = b[0];
for (k=0; k<nblk-1; k++)
{
b_pl = b[k+1];
xnetW[l].b_p0 = b_p0;
b[k] = b_p0;
  b_p0 = b_pl;
}
xnetW[l].b_p0 = b_p0;
b[k] = b_p0;
}

/* This function implements pipelined LU solve, with source-level software pipelining. The input b vectors (one per PE column) are overwritten with their solutions. */

void msolve_spl(int nblk, int ipvtpe[NXPROC_C][LEN],
int ipvtmem[NXPROC_C][LEN], plural double a[LEN][LEN], plural double b[])
{
  register int i,j,k;
  register plural double btmp, b_p0, b_pl;
  register plural double a_p0, a_pl;
  register plural double tmp1, tmp2;
  register plural int ipvtpeofst;
  register plural int ipvtmemofst;

  /* FIRST FORWARD SOLVE L --- SWEEP RIGHT */
  for (j=0; j<nblk; j++)
  {
    for (i=0; i<nxproc+nxproc; i++)
    {
      if (ixproc<i & i>xproc+nxproc)}
/* RECREATE PIVOT SWAP */
ipvtpeofst = ipvtpe[ixproc][j];
ipvtraemofst = ipvtmem[ixproc][j];
if (ipvtpeofst!=0 II ipvtraemofst!=0)
{
  if (iyproc==ixproc)
    router[(iyproc+ipvtpeofst)*nxproc+ixproc].tmpl = b[j];
  if (iyproc==ixproc+ipvtpeofst)
    router[(iyproc-ipvtraemofst)*nxproc+ixproc].tmp2 =
      b[j+ipvtraemofst];
  if (iyproc==ixproc)
    b[j] = tmp2;
}

/* BROADCAST CURRENT b DOWN COLUMN */
if (iyproc==ixproc)
{
  btmp = b[j];
  xnetcS[nyproc].btmp = btmp;
}

/* UPDATE USING COLUMN OF a */
/* SHOULD HANDLE TWO CASES HERE */
a_p0 = a[j][j];
for (k=j; k<nblk-l; k++)
{
  a_pl = a[k+l][j];
  if (k>j)
    b[k] += a[k][j] * btmp;
  else if (iyproc>ixproc)
    b[k] += a[k][j] * btmp;
  a_p0 = a_pl;
}
if (k>j)
  b[k] += a[k][j] * btmp;
else if (iyproc>ixproc)
  b[k] += a[k][j] * btmp;

/* SHIFT b VECTOR TO NEXT COLUMN EAST */
if (i<nxproc+nxproc-l || j<nblk-l)
all
{
  b_p0 = b[0];
  for (k=0; k<nblk-l; k++)
  {
    b_pl = b[k+1];
    xnetE[1].b_p0 = b_p0;
    b[k] = b_p0;
  }
}
b_p0 = b_p1;
}
xnetE[1].b_p0 = b_p0;
b[k] = b_p0;
}
}

/* NOW BACKWARD SOLVE U --- SWEEP LEFT */
for (j=nblk-1; j>=0; j--)
{
  for (i=nxproc+nxproc-1; i>=0; i--)
  {
    if (ixproc+nxproc=i & ixproc<i)
    {
      /* BROADCAST CURRENT b UP COLUMN */
      if (iyproc==ixproc)
      {
        /* WHILE AT IT, DO THE DIVISION */
        b[j] /= a[j][j];
        btmp = - b[j];
        xnetcN[iyproc].btmp = btmp;
      }
      /* UPDATE USING COLUMN OF a */
      /* SHOULD HANDLE TWO CASES HERE */
      a_p0 = a[0][j];
      for (k=0; k<j; k++)
      {
        a_p1 = a[k+1][j];
        /* if (k<j || (j==k & iyproc<iyproc)) */
        b[k] += a_p0 * btmp;
        a_p0 = a_p1;
      }
      if (iyproc<iyproc)
        b[k] += a_p0 * btmp;
    }
    /* SHIFT b VECTOR TO NEXT COLUMN WEST */
    if (i>0 || j>0)
      all
      {
        b_p0 = b[0];
        for (k=0; k<nblk-1; k++)
        {
          b_p1 = b[k+1];
          xnetW[1].b_p0 = b_p0;
          b[k] = b_p0;
          b_p0 = b_p1;
        }
        xnetW[1].b_p0 = b_p0;
        b[k] = b_p0;
      }
double decomp(int nblk, int ipvtpe[NXPROC_C][LEN],
               int ipvtmem[NXPROC_C][LEN], plural double lu[LEN][LEN])
{
    plural double ptmp, prow[LEN];
    plural double rowmult;
    plural double bigtmp;
    double cond;
    unsigned ix,iy,i,j,k,m;

    plural int bigpe;
    plural int bigidx;
    plural double bigval;
    plural double ptmp, prow[LEN];
    plural double rowmult;
    plural double bigtmp;
    double cond;
    unsigned ix,iy,i,j,k,m;

    int pivotpe;
    int pivotidx;
    double pivotval;

    if (opt_spl)
        return decomp_spl(nblk, ipvtpe, ipvtmem, lu);

    /* TWO LOOPS TO PROCESS EACH PE COL AND EACH ELEM OF BLOCK */
    for (i=0; i<nblk; i++)
    {
        for (ix=0; ix<nxproc; ix++)
        {
            /* pivot finding is kludged till we find a better way */
            /* FIND PIVOT AND SWAP IF NECESSARY */
            /* FIRST EACH PROC FINDS ITS OWN BIGGEST */
            if (ixproc==ix)
            {
                bigidx = i;
                if (iyproc >= ix)
                    bigval = p_fabs(lu[i][i]);
                else
                    bigval = 0.0;
            }
            for (j=i+1; j<nblk; j++)
            {
                if (p_fabs(lu[j][i]) > bigval)
                {
                    bigidx = j;
                    bigval = p_fabs(lu[j][i]);
                }
        }
    }
}
/* NOW FIND THE PE ROW */
pivotpe = find_row_with_max(bigval,ix);
pivotval = proc[pivotpe][ix].bigval;
pivotidx = proc[pivotpe][ix].bigidx;
}

if (pivotval == 0.0)
{
    fprintf(stderr, "error, 0 pivot
");  
    exit(-1);
}

/* ipvtpe is the pe offset to the pivot */
/* ipvtmem is the mem offset to the pivot */
ipvtpe[ix][i] = pivotpe - ix;
ipvtmem[ix][i] = pivotidx - i;

/* FINALLY DO THE SWAP */
/* SWAP ONLY AFFECTS PIVOT AND TO RIGHT OF PIVOT */
if (pivotpe != ix)
    /* PE ROW IS DIFFERENT */
    {
    /* FIRST COPY PIVOT ROW TO prow ON CURRENT PE ROW */
    for (j=i; j<nblk; j++)
    {
        if (j > i || iproc >= ix)
        {
            if (iproc == pivotpe)
            {
                ptmp = lu[pivotidx][j];
xnetcN[pivotpe-ix].ptmp = ptmp;
            }
            if (iproc == ix)
            prow[j] = ptmp;
        }        }
    /* NEXT COPY CURRENT ROW TO PIVOT ROW */
    for (j=i; j<nblk; j++)
    {
        if (j > i || iproc >= ix)
        {
            if (iproc == ix)
            {
                ptmp = lu[i][j];
xnetcS[pivotpe-ix].ptmp = ptmp;
            }
            if (iproc == pivotpe)
            lu[pivotidx][j] = ptmp;
        }        }
    }
/* FINALLY COPY TMP BACK TO CURRENT ROW, COMPLETING SWAP */
if (iyproc == ix)
    for (j=i; j<nblk; j++)
        if (j>i)
            lu[i][j] = prow[j];
        else if (ixproc>=ix)
            lu[i][j] = prow[j];
else if (pivotidx != i)
    /* PE ROW IS SAME BUT IDX DIFFERENT */
    {
        if (iyproc == ix)
            for (j=i; j<nblk; j++)
            {
                if (j>i || ixproc>=ix)
                {
                    ptmp = lu[i][j];
                    prow[j] = lu[i][j] = lu[pivotidx][j];
                    lu[pivotidx][j] = ptmp;
                }
            }
        else
            /* NO PIVOT NEEDED */
            {
                if (iyproc==ix)
                    for (j=i; j<nblk; j++)
                    {
                        if (j>i)
                            prow[j] = lu[i][j];
                        else if (ixproc>=ix)
                            prow[j] = lu[i][j];
                    }
            }
    }
/* BROADCAST THE PIVOT ROW DOWN THE COLUMNS */
/* ALL VALUES OF prow THAT REPRESENT ELEMS TO THE LEFT */
/* OF THE CURRENT PIVOT REMAIN 0.0 */
if (iyproc == ix)
    {
        /* INVERT THE DIAGONAL ELEMENT SO WE CAN USE MULTIPLICATION */
        /* INSTEAD OF DIVISION LATER */
        if (ixproc == ix)
            prow[i] = 1.0 / prow[i];
        for (j=i; j<nblk; j++)
        {
            ptmp = 0.0;
            if (j>i)
                ptmp = prow[j];
            else if (ixproc==ix)
                ptmp = prow[j];
            xnetcS[nyproc].ptmp = ptmp;
all prow[j] = ptmp;
}

/* FOR EACH SUBROW BELOW CURRENT */
for (j=i; j<nblk; j++)
{
    /* BROADCAST CONSTANT MULTIPLIER ACROSS THE ROWS */
    if (ixproc == ix)
    {
        rowmult = - lu[j][i] * prow[i];
        xnetcE[nxproc].rowmult = rowmult;
        /* WHILE WE ARE AT IT, SAVE THE MULTIPLIER IN THE L MATRIX */
        if (j>i)
            lu[j][i] = rowmult;
        else if (iyproc>ix)
            lu[j][i] = rowmult;
    }
    if (j>i || iyproc>ix)
    {
        /* Add (MULTIPLIER * PIVOT_ROW) TO ROW */
        for (k=i; k<nblk; k++)
            if (k>i)
                lu[j][k] += rowmult * prow[k];
            else if (ixproc>ix)
                lu[j][k] += rowmult * prow[k];
    }
}

/* FIGURE OUT CONDITION NUMBER */
return cond;
}

/*--------------------------------------*/
/* This function solves for vector b (overwriting it with the */
/* solution vector, given an LU decomposed matrix a. This is the */
/* non source-level software pipelined version. */
/*--------------------------------------*/
void solve(int nblk, int ipvtpe[NXPROC_C][LEN], int ipvtmem[NXPROC_C][LEN],
    plural double a[LEN][LEN], plural double b[[]])
{
    register int i,j,k;
    register plural double btmp;
    double tmp;
    int ipvtpeelem;
    int ipvtmemelem;

if (opt_spl)
{
  solve_spl(nblk, ipvtpe, ipvtmem, a, b);
  return;
}

/* FIRST FORWARD SOLVE L --- SWEEP RIGHT */
for (j=0; j<nblk; j++)
{
  for (i=0; i<nxproc; i++)
  {
    if (ixproc==i)
    {
      /* RECREATE PIVOT SWAP */
      tmp = proc[i][i].b[j];
      ipvtpeelem = ipvtpe[i][j];
      ipvtmemelem = ipvtmem[i][j];
      proc[i][i].b[j] = proc[i+ipvtpeelem][i].b[j+ipvtmemelem];
      proc[i+ipvtpeelem][i].b[j+ipvtmemelem] = tmp;
      
      /* BROADCAST CURRENT b DOWN COLUMN */
      if (iyproc==i)
      {
        btmp = b[j];
        xnetcS[nyproc].btmp = btmp;
      }
      /* UPDATE USING COLUMN OF a */
      /* SHOULD HANDLE TWO CASES HERE */
      for (k=j; k<nblk; k++)
      {
        if (k>j)
          b[k] += a[k][j] * btmp;
        else if (iyproc>i)
          b[k] += a[k][j] * btmp;
        }
    }

    /* SHIFT b VECTOR TO NEXT COLUMN EAST */
    if (i<nxproc-1 || j<nblk-1)
    all for (k=0; k<nblk; k++)
    {
      btmp = b[k];
      xnetE[l].btmp = btmp;
      b[k] = btmp;
    }
  }
}

/* NOW BACKWARD SOLVE U --- SWEEP LEFT */
for (j=nblk-1; j>=0; j--)
{
  for (i=nxproc-1; i>=0; i--)
  {

if (ixproc==i)
{
    /* BROADCAST CURRENT b UP COLUMN */
    if (iyproc==i)
    {
        /* WHILE AT IT, DO THE DIVISION */
        b[j] /= a[j][j];
        btmp = - b[j];
        xnetcN[nyproc].btmp = btmp;
    }
    /* UPDATE USING COLUMN OF a */
    /* SHOULD HANDLE TWO CASES HERE */
    for (k=0; k<=j; k++)
        if (k<j)
            b[k] += a[k][j] * btmp;
        else if (iyproc<i)
            b[k] += a[k][j] * btmp;
    /* SHIFT b VECTOR TO NEXT COLUMN WEST */
    if (i>0 || j>0)
        all for (k=0; k<nblk; k++)
        {
            btmp = b[k];
            xnetW[1].btmp = btmp;
            b[k] = btmp;
        }
}

/*- This function implements pipelined LU solve, without source-level*/
/* software pipelining.  The input b vectors (one per PE column) */
/* are overwritten with their solutions. */
/*-*/
void msolve(int nblk, int ipvtpe[NXPROC_C][LEN],
            int ipvtmem[NXPROC_C][LEN],
            plural double a[LEN][LEN], plural double b[])
{
    register int i,j,k;
    register plural double btmp;
    register plural double tmpl, tmp2;
    register plural int ipvtpeofst;
    register plural int ipvtmemofst;

    if (opt_spl)
    {
        msolve_spl(nblk, ipvtpe, ipvtmem, a, b);
        return;
    }

    /*...*/
/* FIRST FORWARD SOLVE L --- SWEEP RIGHT */
for (j=0; j<nblk; j++)
{
    for (i=0; i<nxproc+nxproc; i++)
    {
        if (ixproc<=i & ixproc+nxproc>i)
        {
            /* RECREATE PIVOT SWAP */
            ipvtpeofst = ipvtpe[ixproc][j];
            ipvtmemofst = ipvtmem[ixproc][j];
            if (ipvtpeofst!=0 || ipvtmemofst!=0)
            {
                if (iyproc==ixproc)
                    router[(iyproc+ipvtpeofst)*nxproc+ixproc].tmpl = b[j];
                if (iyproc==ixproc+ipvtpeofst)
                    router[(iyproc-ixproc)*nxproc+ixproc].tmp2 =
                        b[j+ipvtmemofst];
                b[j+ipvtmemofst] = tmpl;
            }
            if (iyproc==ixproc)
                b[j] = tmp2;
        }
    }
}
/* BROADCAST CURRENT b DOWN COLUMN */
if (iyproc==ixproc)
{
    btmp = b[j];
    xnetE[nyproc].btmp = btmp;
}
/* UPDATE USING COLUMN OF a */
/* SHOULD HANDLE TWO CASES HERE */
for (k=j; k<nblk; k++)
    if (k>j)
        b[k] += a[k][j] * btmp;
else if (iyproc>ixproc)
    b[k] += a[k][j] * btmp;
/* SHIFT b VECTOR TO NEXT COLUMN EAST */
if (i<nxproc+nxproc-1 || j<nblk-1)
    all for (k=0; k<nblk; k++)
    {
        btmp = b[k];
        xnetE[k].btmp = btmp;
        b[k] = btmp;
    }
/* NOW BACKWARD SOLVE U --- SWEEP LEFT */
for (j=nblk-1; j>=0; j--)
{
for (i=nxproc+nxproc-1; i>=0; i--)
{
if (ixproc+nxproc>=i && ixproc<i)
{
    /* BROADCAST CURRENT b UP COLUMN */
    if (iyproc==ixproc)
    {
        /* WHILE AT IT, DO THE DIVISION */
        b[j] /= a[j][j];
        btmp = - b[j];
        xnetcN[iyproc].btmp = btmp;
    }
    /* UPDATE USING COLUMN OF a */
    /* SHOULD HANDLE TWO CASES HERE */
    for (k=0; k<=j; k++)
    {
        if (k<j)
            b[k] += a[k][j] * btmp;
        else if (iyproc<ixproc)
            b[k] += a[k][j] * btmp;
    }
    /* SHIFT b VECTOR TO NEXT COLUMN WEST */
    if (i>0 || j>0)
        all for (k=0; k<nblk; k++)
        {
            btmp = b[k];
            xnetW[k].btmp = btmp;
            b[k] = btmp;
        }
}
}

/*----------------------------------------------------------*/
/* This function solves for each column of the matrix b, performing */
/* nxproc solves at a time. This is the source-level software */
/* pipelined version. */
/*----------------------------------------------------------*/
void col_solve_spl(int nblk, int ipvtpe[NPROC_C][LEN],
        int ipvtmem[NPROC_C][LEN],
        plural double a[LEN][LEN],
        plural double b[LEN][LEN])
{
    register plural double a_p0, a_p1;
    register int i,ix,j,k,ixofst;
    register plural double btmp;
    register plural double tmp1, tmp2;
    register int peofst;
register int memofst;

/* FIRST FORWARD SOLVE L --- SWEEP RIGHT */
for (i=0; i<nblk; i++)
{
    for (ix=0; ix<nxproc; ix++)
    {
        /* RE-CREATE PIVOT SWAP */
        peofst = ipvtpe[ix][i];
        memofst = ipvtmem[ix][i];
        if (peofst | memofst)
        {
            ixofst = ix + peofst;
            if (peofst < 0)
                peofst += nyproc;
            for (j=0; j<nblk; j++)
            {
                if (iyproc == ix)
                    xnetpS[peofst].tmpl = b[i][j];
                if (iyproc == ixofst)
                    [xnetpN[peofst].tmp2 = b[i+memofst][j];
                        b[i+memofst][j] = tmpl;
                }
                if (iyproc == ix)
                    b[i][j] = tmp2;
            }

        /* BROADCAST CURRENT COLUMN OF A ACROSS THE PE ARRAY */
        if (ixproc == ix)
        {
            a_p0 = a[i][i];
            for (j=i; j<nblk-1; j++)
            {
                a_pl = a[j+1][i];
                xnetcE[nxproc].tmpl = a_p0;
                all acol[j] = tmpl;
                a_p0 = a_pl;
            }
            xnetcE[nxproc].tmpl = a_p0;
            all acol[j] = tmpl;
        }

    /* PROCESS EACH COLUMN OF B IN TURN */
    for (j=0; j<nblk; j++)
    {
        /* BROADCAST CURRENT B DOWN COLUMN */
        if (iyproc == ix)
            xnetcS[nyproc].btmp = b[i][j];
}
/* UPDATE USING COLUMN OF A */
a_p0 = acol[i];
for (k=i; k<nblk-1; k++)
{
    a_pl = acol[k+1];
    if (k>i)
        b[k][j] += a_p0 * btmp;
    else if (iyproc>ix)
        b[k][j] += a_p0 * btmp;
    a_p0 = a_pl;
}
/* NOW BACKWARD SOLVE U --- SWEEP LEFT */
for (i=nblk-1; i>=0; i--)
{
    for (ix=nxproc-1; ix>=0; ix--)
    {
        /* BROADCAST CURRENT COLUMN OF A ACROSS THE PE ARRAY */
        if (ixproc == ix)
        {
            a_p0 = a[0][i];
            for (j=0; j<=i-1; j++)
            {
                a_pl = a[j+1][i];
                xnetcE[nxproc].tmpl = a_p0;
                all acol[j] = tmpl;
                a_p0 = a_pl;
            }
            xnetcE[nxproc].tmpl = a_p0;
            all acol[j] = tmpl;
        }
        /* PROCESS EACH COLUMN OF B IN TURN */
        for (j=0; j<nblk; j++)
        {
            /* BROADCAST CURRENT B UP COLUMN */
            if (iyproc == ix)
            {
                /* WHILE AT IT, DO DIVISION */
                b[i][j] /= acol[i];
                xnetcN[nyproc].btmp = -b[i][j];
            }
            /* UPDATE USING COLUMN OF A */
            a_p0 = acol[0];
        }
    }
}
for (k=0; k<=i-i; k++)
{
    a_pl = acol[k+1];
    if (k<i)
        b[k][j] += a_p0 * btmp;
    else if (iyproc<ix)
        b[k][j] += a_p0 * btmp;
    a_p0 = a_pl;
}
if (k<i)
    b[k][j] += a_p0 * btmp;
else if (iyproc<ix)
    b[k][j] += a_p0 * btmp;

/* This function solves for each column of the matrix b, performing */
/* nxproc solves at a time. This is the non source-level software */
/* pipelined version. */

void col_solve(int nblk, int ipvtpe[NXPROC_C][LEN], int
               ipvtmem[NXPROC_C][LEN],
               plural double a[LEN][LEN], plural double b[LEN][LEN])
{
    register int i,ix,j,k,ixofst;
    register int peofst, memofst;
    register plural double btmp;
    register plural double tmpl, tmp2;
    if (opt_spl)
    {
        col_solve_spl(nblk, ipvtpe, ipvtmem, a, b);
        return;
    }

    /* FIRST FORWARD SOLVE L --- SWEEP RIGHT */
    for (i=0; i<nblk; i++)
        {
            for (ix=0; ix<nxproc; ix++)
                {
                    /* RE-CREATE PIVOT SWAP */
                    peofst = ipvtpe[ix][i];
                    memofst = ipvtmem[ix][i];
                    if (peofst != memofst)
                        {
                            ixofst = ix + peofst;
                            if (peofst < 0)
peofst += nyproc;
for (j=0; j<nblk; j++)
{
  if (iyproc == ix)
    xnetpS[peofst].tmpl = b[i][j];
  if (iyproc == ixofst)
    { 
    xnetpN[peofst].tmp2 = b[i+memofst][j];
    b[i+memofst][j] = tmpl;
    }
  if (iyproc == ix)
    b[i][j] = tmp2;
}
/* BROADCAST CURRENT COLUMN OF A ACROSS THE PE ARRAY */
if (ixproc == ix)
{ 
  for (j=i; j<nblk; j++)
  { 
    xnetcE[nxproc].tmpl = a[j][i];
    all acol[j] = tmpl;
  }
}
/* PROCESS EACH COLUMN OF B IN TURN */
for (j=0; j<nblk; j++)
{ 
  /* BROADCAST CURRENT B DOWN COLUMN */
  if (iyproc == ix)
    { 
    xnetcS[nyproc].btmp = b[i][j];
    }
  /* UPDATE USING COLUMN OF A */
  for (k=i; k<nblk; k++)
  { 
    if (k> i)
      b[k][j] += acol[k] * btmp;
    else if (iyproc<ix)
    
      b[k][j] += acol[k] * btmp;
  }
}
/* NOW BACKWARD SOLVE U --- SWEEP LEFT */
for (i=nblk-1; i>=0; i--)
{ 
  for (ix=nxproc-1; ix>=0; ix--)
  { 
    /* BROADCAST CURRENT COLUMN OF A ACROSS THE PE ARRAY */
    if (ixproc == ix)
      { 
      }
for (j=0; j<=i; j++)
{
    xnetcE[nxproc].tmpl = a[j][i];
    all acol[j] = tmpl;
}

/* PROCESS EACH COLUMN OF B IN TURN */
for (j=0; j<nblk; j++)
{
    /* BROADCAST CURRENT B UP COLUMN */
    if (iyproc == ix)
    {
        /* WHILE AT IT, DO DIVISION */
        b[i][j] /= acol[i];
        xnetcN[nyproc].btmp = -b[i][j];
    }
    /* UPDATE USING COLUMN OF A */
    for (k=0; k<=i; k++)
    {
        if (k<i)
            b[k][j] += acol[k] * btmp;
        else if (iyproc<ix)
            b[k][j] += acol[k] * btmp;
    }
}
}
#include <sys/time.h>
#include <mp_resource.h>

struct mpRUsage_s ru;
double setpoint = 0.0;

double stopwatch(int resetflag)
{
    double rval;
    double usage;
    
    mpGetRUsage(RUSAGE_SELF, &ru);
    
    usage = ((double)ru.dpu.dr_dputime.tv_sec) +
             ((double)ru.dpu.dr_dputime.tv_usec) / 1000000.0;
    rval = usage - setpoint;
    
    if (resetflag)
        setpoint = usage;
    
    return rval;
}
/* FILE: util.m */
/* */
/* Boundary Element Method (BEM) Application */
/* Jeffrey S. Clary */
/* Iowa State University */
/* */
/* This file contains assorted utility functions. */
/****************************************************************************/

#include "bem.h"
#include <math.h>
/* */
/* This function returns 1 if its argument is even; -1 otherwise. */
/* */
int nsgn(int i)
{
    return (i & 0x01) ? -1 : 1;
}

/****************************************************************************/
/* This function returns x to the n'th power. */
/* */
plural double p_pwr(plural double x, plural int n)
{
    return (n==0) ? 1.0 : p_pow(x, (plural double) n);
}

/****************************************************************************/
/* This function returns x to the n'th power. */
/* */
double pwr(double x, int n)
{
    return (n==0) ? 1 : pow(x, (double) n);
}

/****************************************************************************/
/* This function sums its double argument into the diagonal PE on */
/* each PE row. */
/* */
/* Assumes a square PE array. */
/* */
plural double sum_to_diag(plural double x)
{
    plural int idx = (ixproc + nxproc - iyproc) % nxproc;
    plural int not_elim=-0, senders;
    int i;

    for (i=1; i<nxproc; i++)
    {
        ...
senders = (idx & i) & not_elim;
if (senders)
{
  xnetpw[i].x += x;
  not_elim = 0;
}
return x;

/*----------------------------------------------------------*/
/* This function sums its double argument into the 0th PE on */
/* each PE row. */
/* */
/* Assumes a square PE array. */
/* */
/*----------------------------------------------------------*/
plural double sum_to_c0(plural double x)
{
  plural int idx = ixproc;
  plural int not_elim=0, senders;
  int i;

  for (i=1; i<nxproc; i<<=1)
  {
    senders = (idx & i) & not_elim;
    if (senders)
    {
      xnetpw[i].x += x;
      not_elim = 0;
    }
  }
  return x;
}

/*----------------------------------------------------------*/
/* This function finds the maximum value of x in a PE column, and */
/* returns that PE row. */
/* */
/*----------------------------------------------------------*/
int find_row_with_max(plural double x, int column)
{
  register plural int idx = iyproc;
  register plural int not_elim=0, senders;
  register int i;

  register plural double x1, x2;
  register plural int idx1, idx2;

  x1 = x;
  idx1 = iyproc;
  for (i=1; i<nyproc; i<<=1)
  {
    /* */
  }

senders = (idx & i) & not_elim;
if (senders)
{
    xnetpN[i].x2 = x1;
    not_elim = 0;
}
if (not_elim & (x2 > x1))
{
    idx1 = xnetpS[i].idx1;
    x1 = x2;
}
return proc[0][column].idx1;