Application of Parallel Processing to a Surface Patch / Wire Junction EFIE Code

L. C. Russell, J. W. Rockway
Naval Ocean Systems Center
San Diego, California 92152

ABSTRACT

A surface patch / wire junction EFIE method of moment algorithm, JUNCTION, developed by the University of Houston has been implemented in a transputer based parallel processing environment installed in a personal computer. This paper addresses transputer hardware and software options, the JUNCTION algorithm, techniques for parallelizing matrix analysis algorithms, and performance results. The transputer array was found to provide a flexible, low-cost, high performance desktop computing environment for method of moment analysis.

1.0 INTRODUCTION

This paper presents the application of parallel processing techniques to an existing computational electromagnetic (CEM) code. The goal was to demonstrate that high performance computing (HPC) can improve the utility and efficiency of computational techniques used in electromagnetic ship analysis [Li, 1988]. The parallel processing platform was an array of four transputers on a board inside an IBM-compatible PC. The transputers were run using the ParaSoft EXPRESS operating environment [ParaSoft, 1990]. This operating environment was chosen to allow portability of the code to other HPC platforms and minimize the number of required programming changes to the CEM code. The selected CEM code was the method of moment (MoM) algorithm JUNCTION developed by the University of Houston [Hwu et al, 1988; Wilton et al, 1988].

2.0 PARALLEL PROCESSING

The concept of parallel processing has been around for a long time but only recently was its utility generally recognized. By 1986 more than a dozen companies were either selling or in the process of building parallel processors [Tazelaar, 1988].

2.1 Transputer hardware

The hardware platform chosen was an array of four transputers on a motherboard which could be installed inside a PC. This decision was based on the low cost, ease of use, and flexibility of a transputer based system. Transputer systems provide an inexpensive entry into the world of parallel processing. For this entry level application a decision was made to purchase only four floating point transputer modules (TRAMs) each with one Megabyte of external RAM. These size 1 TRAMs are an industry standard and provide sufficient memory for most applications at a reasonable cost and in a compact package. The T800 transputer used has a peak performance of 1.5 Mflops.

2.2 Transputer Software

There are several different ways to program an array of transputers. Each method has trade-offs in portability, performance, and ease of use. The parallel language, Occam, was developed concurrently with the transputer. Parallel versions of high level languages such as Fortran, C, Pascal, and Modula 2 can be obtained. The parallel operating environment EXPRESS (ParaSoft Corporation) is available for transputer systems.

2.2.1 Occam

There are a number of advantages to using Occam for programming a transputer system. Since Occam was developed simultaneously with the transputer, it is one of the few languages designed for concurrency. Occam has the performance and efficiency of assembly language. Occam can be used as a harness to link modules written in other industry standard languages. Its other features include: well implemented timing capabilities, straightforward control of process scheduling, and a high level of structure.

The main disadvantage to using Occam is that programs written in Occam are not portable to other parallel computers. Existing codes written in other languages need to be completely recoded. Occam works only with the transputer. Other disadvantages are that Occam does not support many features of other high level languages (no recursion or dynamic memory allocation), use of it requires learning a new language, and Occam may not be supported in the future. In the trade-off space (portability versus performance versus ease of use) Occam scores high only in performance.

2.2.2 High Level Languages

The advantage of using a parallel version of a high level language such as Fortran, C, Pascal, or Modula 2 is that one can use a language with which one is already familiar. All that is required is to learn the parallel extensions. Existing sequential code can be ported over.

There are, however, a number of disadvantages to using a parallel version of a high level language. Programming an array of transputers requires the user to develop both a host program and a node program. The host program runs on the host processor and handles I/O calls and manages the other processors. The node program runs on all the other processors and does the bulk of the computations. The user is required to maintain two programs instead of the usual one program. This can make program development and maintenance somewhat complex. Performing I/O, making system calls, and measuring timing can be very difficult. Debuggers for parallel versions of high level languages are starting to appear, but they are not widely available. Debugging parallel programs is very difficult. In addition to all this, parallel versions of high level languages do not offer the performance that one can get from Occam.

2.2.3 ParaSoft EXPRESS

Parallel operating environments such as ParaSoft's EXPRESS allow the user the convenience of using a familiar high level language while mitigating some of the disadvantages to parallel high level languages which were mentioned in the previous section. EXPRESS is based on the CUBIX programming model which was developed at CalTech. EXPRESS is available for both Fortran and C. There is no need to develop both a host and node program. Instead only one program (which runs on all processors) needs to be developed and maintained. EXPRESS does not include a compiler so one of the parallel high level language compilers mentioned above is still needed to compile and link the code. During linking the EXPRESS library is linked into the user's source code. The EXPRESS library provides simple Fortran or C language function calls to handle I/O and system calls. EXPRESS also provides timing capabilities, a source level debugger, and a performance monitor. The number of processors being used does not have to be hardwired into the code. Instead this is specified at runtime by a switch in the run command. In addition, EXPRESS is available for a wide variety of parallel machines, not just the transputer. Currently EXPRESS is available for: the multi-headed IBM 3090 (AIX) and the multi-headed CRAY (UNICOS) mainframes; the Intel iPSC 2 and iPSC 860; the nCUBE 1 and 2; PC, MAC, and SUN hosts for transputers; and the IBM RS/6000, Silicon Graphics, and SUN workstations. All this leads to very portable source code.

The only significant disadvantage to EXPRESS is that it possibly degrades system performance to some extent. However, this disadvantage is more than offset by the advantage of having portable code. For these reasons EXPRESS by ParaSoft was chosen for the operating environment. Portability was the key feature since the ultimate goal of this effort was to run the code on a high performance computer. The language chosen was Fortran since this is the principal language of the JUNCTION method of moments code. For obvious reasons, it was desirable to make as few changes to the existing code as possible.

3.0 The JUNCTION METHOD OF MOMENTS CODE

The JUNCTION computer code invokes the method of moments to solve a coupled electric field integral equation (EFIE) for the currents induced on an arbitrary configuration of perfectly conducting bodies and wires [Wilton et al, 1989]. There are three principal advantages to the JUNCTION formulation. First, the EFIE formulation for the surfaces of bodies, in contrast to the magnetic field integral equation (MFIE), applies to open bodies. Second, JUNCTION allows voltage and load conditions to be easily specified at terminals defined on the structure. Third, the triangular patches of JUNCTION are the simplest planar surfaces which can be used to model arbitrary surfaces and boundaries, and triangular patches permit patch densities to be varied locally so as to model a rapidly varying current distribution.

The method of moments is a numerical procedure for solving field integral equations [Harrington, 1968]. Basis functions are chosen to represent the unknown currents. Testing functions are chosen to enforce the integral equation on the surface of the conducting structure. With the choice of basis and testing functions a matrix approximating the integral equation is derived. In JUNCTION a system of N linear equations results, where $N = N_B + N_W + N_J$.

$$\begin{bmatrix} [Z^{BB}] & [Z^{BW}] & [Z^{BJ}] \\ [Z^{WB}] & [Z^{WW}] & [Z^{WJ}] \\ [Z^{JB}] & [Z^{JW}] & [Z^{JJ}] \end{bmatrix} \begin{bmatrix} [I^B] \\ [I^W] \\ [I^J] \end{bmatrix} = \begin{bmatrix} [E^B] \\ [E^W] \\ [E^J] \end{bmatrix}$$

where Z is the impedance matrix, I is the current vector, and E is the excitation vector. The superscripts are B for body elements, W for wire elements, and J for junctions between wires and bodies. The analytical expressions for each of the individual submatrices are different. Solution of the linear system of equations yields the set of unknown coefficients used in the representation of the surface, wire, and junction currents. Once these currents are known, the scattered field or any other electromagnetic quantity of interest may be determined.

4.0 IMPLEMENTATION OF SEQUENTIAL PROGRAM

The first step in parallelizing an existing program is to verify the sequential version of the program. This is done by running the program on a serial computer. Once this has been successfully done, the next step is to get the program to run on a single processor on the parallel computer. This is necessary to eliminate any problems caused by compiler differences between the serial compiler and the parallel compiler and also to "check out" the hardware. Only after this has been completed can the parallelization of the program be considered.

JUNCTION was broken into four separate programs: DATGN, GEOMETRY, CURRENT, and FIELD. DATGN creates input data set files. GEOMETRY calculates the geometrical parameters. CURRENT computes the currents and impedance. FIELD computes charges, near fields, far fields, radar cross sections, and power gain. The computationally intensive programs are CURRENT and FIELD. This paper is concerned with the parallelizing of CURRENT.

From a computational standpoint CURRENT consists of filling the impedance matrix and the excitation vector, and then solving for the current vector. The current vector gives the currents on the bodies, wires and junctions. The methods used are standard matrix manipulation techniques. Once the matrix is filled, Gaussian elimination is performed using LU decomposition with partial pivoting to factor the matrix. Finally, the current vector is solved for by using backward and forward substitution. The library subroutines used are the LINPACK routines CGEFA and CGESL [Dongarra et al, 1979]. CGEFA factors a general complex matrix. CGESL solves a complex matrix equation using the output from CGEFA.

5.0 PARALLEL MATRIX FACTOR AND SOLVE

5.1 Parallel LINPACK

When this effort was initiated, general parallel versions of the LINPACK subroutines were not available. This is because parallel subroutines are by their nature very machine specific and one of the goals of the LINPACK project was for the subroutines to be machine independent.

On sequential computers the number of computations required to factor a matrix into a product of triangular matrices is of the order (n^3) where n is the dimension of the matrix. In comparison, the number of computations required for the triangular solution are of the order (n^2). Because of this, unless multiple solutions are required, most effort has focused on optimizing the factorization algorithm.

On parallel computers good efficiency is more difficult to obtain for matrix solving compared to matrix factoring. This is because much more interprocessor communication is required during solving. Solving is inherently a finer grain process than factoring. Various parallel algorithms have been proposed. Each one has its advantages and disadvantages which are dependent on the number of processors being used, the size of the problem being solved, and the relative cost of communication and computation.

The matrix columns were distributed onto the processors using an interleaving technique known as column wrap mapping. Column wrap mapping has good load balancing properties and gives excellent performance when doing LU decomposition.

Intel Corporation provided the source code for Intel's parallel versions of DGEFA and DGESL from LINPACK. DGEFA and DGESL are double precision non-complex versions of CGEFA and CGESL, respectively. Included were two different matrix solving routines: a cyclic version and a wavefront version of DGESL. The differences between these versions are described below. These routines were all developed for Intel's iPSC parallel computer so a number of modifications were needed to implement them under EXPRESS on the transputer array. The matrix factoring routine was straightforward to parallelize and provided very high efficiencies.

The difficult problem was the solution of the lower triangular linear system

$$Lx = b,$$

where L is a lower triangular matrix of order n, b is a known vector of dimension n, and x is the unknown solution vector of dimension n. The sequential version of LINPACK solves this by forward substitution using the following doubly nested loop:

```
for $j = 1$ to $n$
  $x_j = b_j / L_{jj}$
  for $i = j+1$ to $n$
    $b_i = b_i - x_j L_{ij}$
```

There are a number of ways to parallelize this problem on a distributed memory parallel computer. Several algorithms have been developed. Two of the most popular are the *wavefront* algorithm and the *cyclic* algorithm [Heath et al, 1988]. These algorithms assume the matrix is column wrap mapped onto the processors.

The wavefront algorithm breaks the updating of b into segments and pipelines the segments through the processors in a wavefront fashion. The n-vector z is used to accumulate the updates of b so that the components of b remain distributed among the processors. The size of the circulated segments is an adjustable parameter that controls the granularity of the algorithm and therefore affects performance. The optimal value of the segment size depends on the characteristics of the hardware.

The wavefront algorithm is written in pseudo-code as:

```
for $j \in$ *mycols*
  for $k = 1$ to # *segments*
    receive *segment*
    if $k = 1$ then
      $x_j = (b_j - z_j)/L_{jj}$
      *segment* = *segment* - $\{z_j\}$
    for $z_i \in$ *segment*
      $z_i = z_i + x_j L_{ij}$
    if |*segment*| > 0 then
      send *segment* to processor *map(j + 1)*
```

The cyclic algorithm is similar to the wavefront algorithm in that they both send a segment z between processors. However, the cyclic algorithm circulates a single segment of the fixed size p-1, where p is the number of processors. The name *cyclic* comes from the segment cycling through all other processors before returning to a given processor. The updates computed by the processor while the segment is circulating elsewhere are stored in the vector t. The cyclic algorithm is written in pseudo-code as:

```
for $j \in$ *mycols*
  receive *segment*
  $x_j = (b_j - z_j - t_j)/L_{jj}$
  *segment* = *segment* - $\{z_j\}$
  for $z_i \in$ *segment*
    $z_i = z_i + t_i + x_j L_{ij}$
  $z_{j+p-1} = t_{j+p-1} + x_j L_{j+p-1,j}$
```

```
segment = segment ∪ {$z_{j+p-1}$}
send segment to processor map(j+1)
for $i = j + p$ to $n$
  $t_i = t_i + x_j L_{ij}$
```

Theoretical analysis shows that the cyclic algorithm performs best on a small number of processors whereas the wavefront algorithm performs best on a large number of processors.

5.2 Implementation of parallel LINPACK under EXPRESS

A number of modifications were needed to the Intel parallel LINPACK routines in order to implement them on the transputer array operating under ParaSoft EXPRESS. These modifications fell into three categories:

1. converting double precision variables to complex variables,
2. using complex LINPACK routines instead of the equivalent double precision LINPACK routines,
3. using EXPRESS communication functions/routines instead of iPSC communication functions/routines.

5.3 Results

Performance measurements were made of both the parallel matrix factor routine and the two parallel matrix solve routines. Timings were made using one, two, three and four processors. The timing measurements were converted to the standard performance measurement parameters, speed-up and efficiency. Speed-up is the ratio of the execution time of the algorithm on a single processor to the execution time of the parallel algorithm on the *n* processors. Efficiency is the speed-up divided by *n* and expressed as percent.

Due to memory limitations, examples with more than 200 unknowns could not be run on a single transputer. Several sample problems were developed to test out the parallel factor and solve routines. The four sample problems had the following number of unknowns: 104, 116, 155, and 189.

5.3.1 Factor

Figure 5-1 shows the timing results on the parallel version of CGEFA for the four sample problems. Figure 5-2 shows speed-up and figure 5-3 shows efficiency. The important issue is that as the number of unknowns increases the parallel matrix factoring routine becomes more efficient. This is expected since small problems do not achieve good load balancing and cannot benefit greatly from parallel computation. An efficiency of over 90% is considered very good.

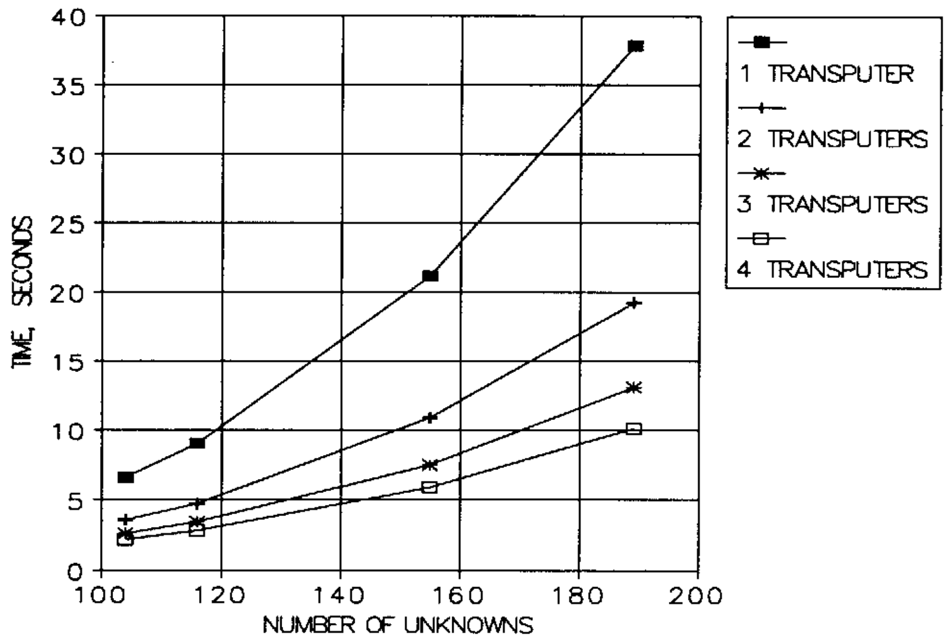

Figure 5-1. Timing for parallel matrix factoring routine.

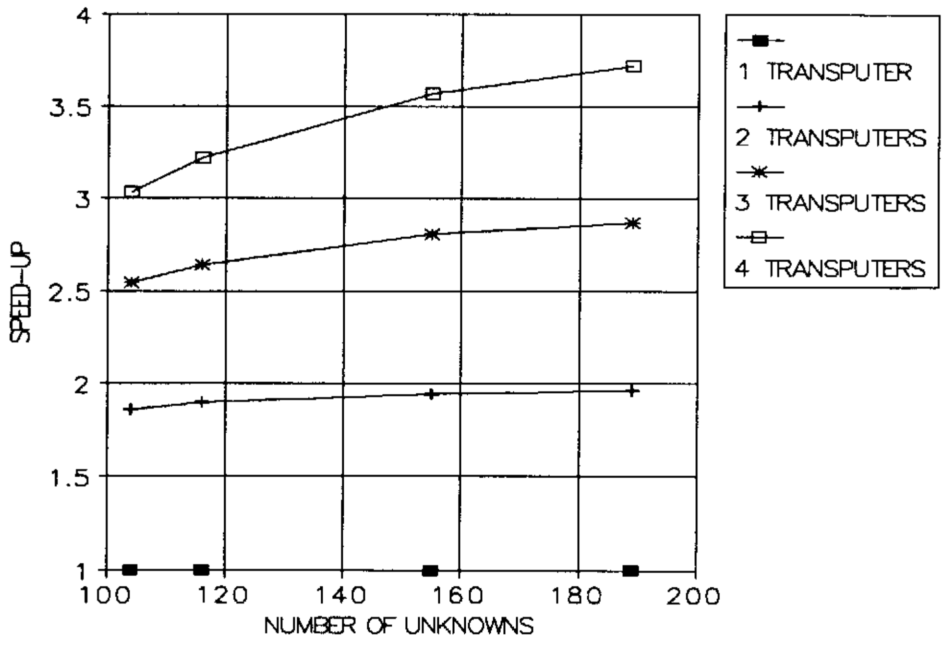

Figure 5-2. Speedup achieved for parallel matrix factoring routine.

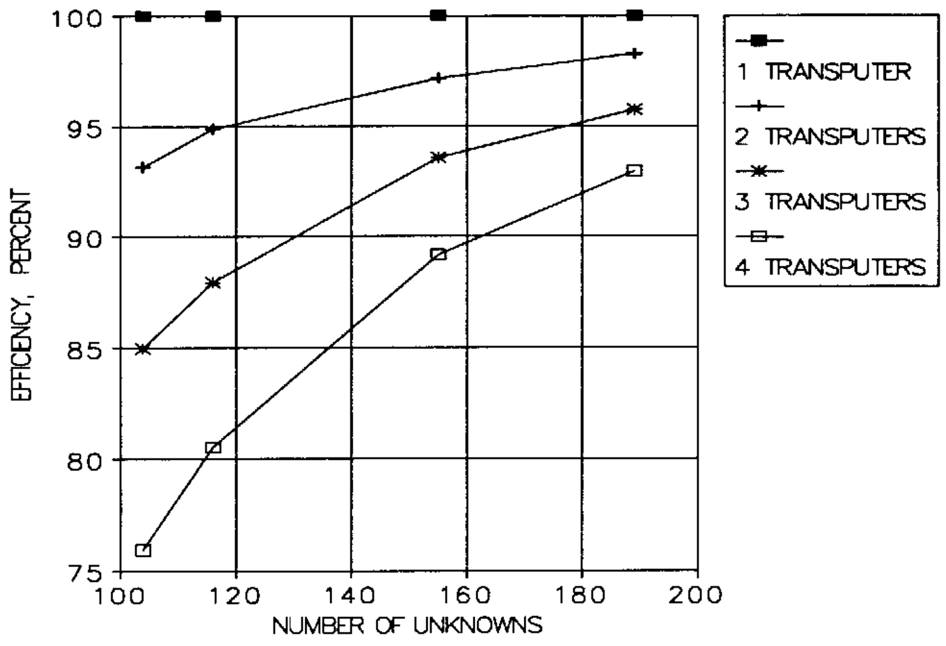

Figure 5-3. Efficiency achieved for parallel matrix factoring routine.

5.3.2 Solve

The performance of the two parallel matrix solve routines were measured for all the example problems. The results are shown here for the largest problem, the one with 189 unknowns. Table 5-1 shows the results for the cyclic algorithm.

Table 5-1. Performance results for cyclic algorithm for 189 unknowns.

Number of Processors	Time, seconds	Speedup	Efficiency
1	0.621	1.00	100%
2	0.367	1.69	84.5%
3	0.257	2.42	80.7%
4	0.207	3.00	75.0%

The performance results for the wavefront algorithm are complicated by the presence of the adjustable segment size parameter. This segment size parameter can range from 1 to the number of unknowns. Table 5-2 shows the optimal performance segment size and time. The optimal performance segment size is a function of the number of processors (as well as the number of unknowns). In addition, by comparing the timing results for the wavefront algorithm with the results for the cyclic algorithm it can be seen that even with the optimum segment size the performance for the wavefront algorithm is much worse than the cyclic algorithm. For these two reasons the wavefront algorithm was rejected for use on the size of problem which could be run on the four processor transputer array. However, it must be noted that the performance results could turn out to be very different on a high performance computer with many more processors running larger problems.

Table 5-2. Performance results for wavefront algorithm for 189 unknowns.

Number of Processors	Optimal segment size	Time, seconds
1	95	0.741
2	72	0.555
3	34	0.448
4	24	0.380

6.0 PARALLEL MATRIX FILLING

Developing a highly efficient general routine for parallel matrix filling was much more difficult in comparison to the work required to parallelize the matrix factor and solve routines. This was due to several reasons: the large variety of problem geometries being solved, shared calculations between matrix columns, memory limitations, and concurrent filling of the excitation vector.

6.1 Sequential matrix filling

In JUNCTION the columns of the matrix correspond to source points on the structure. The rows of the matrix correspond to observation points on the structure. The body columns/rows come before the wire columns/rows, but otherwise the ordering of the columns/rows depends on how the problem was specified in the input data portion of the code. The junction points are scattered within the wire portion of the matrix. Wire columns correspond to nodes on the wires whereas body columns correspond to sides of triangular patches on the surfaces. A junction is a node on a wire, hence its location is in the wire portion of the matrix.

6.2 Column mapping techniques

There are a wide variety of techniques which can be used to map the columns of the matrix onto the processors. A column wrap mapping technique was used for matrix factoring and solving as described in the previous chapter.

It quickly became apparent that column wrap mapping was not necessarily the best method to use for parallelizing the matrix filling portion of the code. Adjacent columns of the matrix are often associated with each other. For bodies adjacent columns are usually either on the same face or on an attached face. For wires adjacent columns often refer to nodes which are joined by the same segment. In JUNCTION calculations are made with respect to faces and segments, not edges and nodes. This avoids duplication of work in a serial computation. Alternative methods for mapping the columns onto the processors were needed. (Note: mapping the rows onto the processors, instead of the columns, was never considered since matrix factoring required the columns to be already mapped onto the processors).

Alternative mapping techniques were devised and evaluated. The techniques and performance results are described below in the section on structural dependencies. It was found that using a different mapping technique to fill the matrix did not compromise the output from matrix solving. The only effect was to scramble the output vector. The output vector is easily and rapidly unscrambled at the end of the matrix solve routines. Unscrambling the output vector at the end is much faster than unscrambling the matrix before the matrix factor routine, since much less inter-processor communication is required for the former.

6.3 Structural dependencies

The performance of the parallel matrix filling algorithm was found to be very dependent on the structure being analyzed. The first types of problems analyzed were for *homogeneous* structures. Homogeneous structures are defined as having either all body elements or all wire elements. Next, *heterogeneous* structures were evaluated. Heterogeneous structures are defined as having a mixture of body, wire, and junction elements.

6.3.1 Homogeneous structures

Six types of structures were considered: straight wires, cylinders, plates, cones, disks, or spheres. The performance of the parallel matrix filling routines was evaluated for homogeneous problems made up of each of these types of structures. Two column mapping techniques were compared for each structure type. The two techniques used were *column wrap mapping* and *column block mapping*. Column wrap mapping consists of interleaving the columns onto the processors. It is described in detail in Section 6.2. Column block mapping consists of distributing the columns onto the processors using contiguous blocks of columns rather than individual columns. As an example, if 4 processors were being used to fill a matrix with 100 columns, using column block mapping would mean that the first processor would fill the first 25 columns, the second processor would fill the second 25 columns, and so on.

Four sample problems were developed for each structure type. The number of unknowns ranged from 40 to 200. Each sample problem was run on both a single transputer and four transputers. Measurements were made of both matrix filling time and matrix solving time for both column wrap mapping and column block mapping.

Table 6-3 shows the measured times and calculated efficiencies for matrix filling for the different structures using wrap and block mapping.

For a single processor it is seen that an all wire homogeneous matrix takes longer to fill than an all body homogeneous matrix of the same size. On average an all wire matrix took about 38% longer to fill than the same size all body matrix. Improvements are being been made in JUNCTION to improve the speed of calculation for wires. The cylinder, plate, cone, disk and sphere matrices all took about the same time to fill a given size matrix.

When the sample problems were run on four processors, distinct structural dependencies emerged for filling times and efficiencies. It is important to note that column block mapping always performed better than column wrap mapping. Column block mapping filling efficiencies were almost always greater than 70%, whereas column wrap mapping filling efficiencies were always between 40 and 60%. It is also interesting to note that the different structures performed differently in filling efficiencies. For column wrap mapping wires performed best followed by plates, cones, spheres, disks, and, then, cylinders. For column block mapping wires again performed best followed by cylinders, plates, cones, spheres, and, then, disks. These results are due to the nature of the physical connectivity of the various structures and how their elements are distributed to the columns

Table 6-3. Matrix filling times/efficiencies for various structures.

Structure Type	Number of Unknowns	1 Processor	4 Processors			
			Wrap Mapping		Block Mapping	
		TIME,sec	TIME	EFFIC	TIME	EFFIC
WIRE	47	7.38	3.32	55.6%	1.97	93.7%
	97	30.78	14.06	54.7%	8.08	95.2%
	147	70.51	31.54	55.9%	17.95	98.2%
	197	126.06	56.91	55.4%	32.22	97.8%
CYLINDER	56	8.56	4.97	43.1%	2.80	76.4%
	104	27.04	15.78	42.8%	7.94	85.1%
	152	55.95	32.69	42.8%	15.68	89.2%
	200	95.17	55.62	42.8%	25.99	91.5%
PLATE	40	5.12	2.66	48.1%	1.66	77.1%
	96	25.11	12.69	49.5%	7.88	79.7%
	133	45.7	25.95	44.0%	14.01	81.5%
	176	76.87	39.15	49.1%	21.92	87.7%
CONE	52	7.15	4.08	43.8%	2.54	70.4%
	100	24.09	13.88	43.4%	7.36	81.8%
	155	55.85	29.12	47.9%	16.56	84.3%
	200	90.52	46.58	48.6%	25.31	89.4%
DISK	49	6.60	3.77	43.8%	2.77	59.6%
	98	25.19	13.7	46.0%	10.42	60.4%
	150	55.65	29.84	46.6%	20.05	69.4%
	200	97.01	53.12	45.7%	34.33	70.6%
SPHERE	60	9.06	4.82	47.0%	3.23	70.1%
	90	19.75	10.65	46.4%	6.85	72.1%
	168	66.56	37.98	43.8%	20.27	82.1%
	198	91.84	47.99	47.8%	28.96	79.3%

of the matrix. Again, in JUNCTION calculations are made with respect to faces and segments. To avoid duplication of computation on the processors it is advantageous to have all the edge computations for a given face to be on the same processor. For a wire both nodes of a given segment should be on the same processor.

Table 6-4 shows the measured times and calculated efficiency for matrix factoring and solving for the different structures.

In matrix factoring and solving there are no structural dependencies. It should also be noted that as the number of unknowns increases, the factoring and solving efficiencies increase. For 200 unknowns factoring and solving efficiency is greater than 95% percent. Matrix filling efficiency also increases as the number of unknowns increases.

Table 6-4. Matrix factor and solve times/efficiency for various structures.

Structure Type	Number of Unknowns	1 Processor	4 Processors	
		TIME,sec	TIME	EFFIC
WIRE	47	0.66	0.22	75.0%
	97	5.34	1.48	90.2%
	147	18.09	4.81	94.0%
	197	42.94	11.19	95.9%
CYLINDER	56	1.13	0.35	80.7%
	104	6.70	1.81	92.5%
	152	20.22	5.34	94.7%
	200	45.36	11.79	96.2%
PLATE	40	0.42	0.15	70.0%
	96	5.12	1.44	88.9%
	133	13.35	3.58	93.2%
	176	30.55	8.04	95.0%
CONE	52	0.92	0.29	79.3%
	100	5.97	1.63	91.6%
	155	21.45	5.63	95.2%
	200	45.41	11.79	96.3%
DISK	49	0.78	0.25	78.0%
	98	5.60	1.54	90.9%
	150	19.48	5.11	95.3%
	200	45.32	11.78	96.2%
SPHERE	60	1.37	0.41	83.5%
	90	4.38	1.21	90.5%
	168	27.10	7.11	95.3%
	198	44.02	11.41	96.5%

6.3.2 Heterogeneous structures

The heterogeneous structures that were evaluated consisted mostly of square plates with numerous attached wires. This allowed the evaluation of matrices with various numbers of body, wire, and junction columns.

After evaluating a large number of heterogeneous problems a number of observations were made. These observations are summarized below:

- The junction columns are scattered within the wire portion of the matrix. The distribution of the junction columns depends on how the problem is initially specified. Fortunately, it is fairly straightforward to convert from junction number to matrix column number.
- A junction can connect to between one and six faces on a body. A quirk of JUNCTION is that the code will only recognize a maximum of one junction attached to a given face.

- To achieve maximum efficiency the filling of a junction column should be done by the same processor which is filling the columns associated with the faces to which the junction is attached, since a large part of the calculations are in common. The problem is that the associated body columns are sometimes scattered all through the matrix.
- The time to fill a matrix is a function of many factors including: relative/absolute number of wire, body and junction unknowns; number of separate wires; number of segments on each wire; locations of wires and junctions relative to a body; number of faces connected to each junction.
- Developing a general method for balancing the work load and achieving optimum efficiency for a heterogeneous problem is very complicated, even for the simple problems studied involving a single plate with various attached wires. If the choice is between block and wrap mapping, block mapping gives better results. (For the problems studied, block mapping was 62 - 93% efficient, whereas wrap mapping was 41 - 52% efficient).

Based on the above observations a decision was made to implement two additional column mapping techniques. Both techniques are modifications of the standard block mapping technique.

The first technique is called *random mapping*. Random mapping is block mapping with one twist: instead of the junction columns staying grouped with the wire columns, the junction columns are filled by the processors which are filling the body columns of the matrix. The columns are redistributed to keep a balanced number of columns on each processor. As an example, suppose a matrix has 50 body columns, 50 wire columns, and 6 junctions, and the problem is being run on 4 processors. Under standard column block mapping the first 25 body columns would be filled by the first processor, the second 25 body columns would be filled by the second processor, the first 25 wire columns would be filled by the third processor, and the second 25 wire columns would be filled by the fourth processor. The junction columns, being wire columns also, would be distributed on the third and fourth processors. Under random mapping the six junction columns would first be distributed to the first two processors in the following manner: junction 1 column to first processor, junction 2 column to second processor, junction 3 column to first processor, junction 4 column to second processor, and so on. The first and second processors would each have 3 junction columns. Block processing would then be used to distribute the body and remaining wire columns, so that each processor would finish up with 25 total columns as before.

The second technique is called *1st order mapping*. This is similar to the random mapping described above except now the junction columns are assigned to the body processors using knowledge of which processor will be calculating the majority of columns associated with the faces to which that junction is attached. This could potentially cause more junction columns to end up on one of the body processors than another, but the improvement in performance could be significant.

An analysis of matrix filling efficiency was run on four processors for the problem of a square plate with 8 wires attached to it. The problem involved 96 body unknowns, 96 wire unknowns, and 8 junctions. The four different column mapping techniques were used. The results are shown in table 6-5.

The results show that random mapping only gives a slight improvement over block mapping, but 1st order mapping gives a significant improvement over block mapping.

Table 6-5. Efficiencies of various mapping techniques.

Mapping Technique	Efficiency
WRAP	40.5%
BLOCK	61.8%
RANDOM	63.9%
1ST ORDER	76.9%

7.0 RESULTS AND CONCLUSIONS

7.1 Performance Comparisons

Comparisons were run for the micro-Vax, a single transputer, the four transputer array, and the Convex Model C-220. The Convex is a mini-supercomputer which can do some automatic vectorization.

Sample data sets were generated for both bodies and wires. Eight body data sets were generated with 104, 152, 200, 248, 296, 356, 404, and 452 unknowns. Eight wire data sets were generated with 47, 97, 147, 197, 247, 297, 347, and 397 unknowns. For each data set and each hardware platform timing measurements were made for both matrix filling and matrix factoring and solving. On the four transputer array the data sets were run using both column wrap mapping and column block mapping. On the Convex measurements were made of both CPU time and actual elapsed time. At the time the measurements were made there were 13 other users on the Convex and elapsed time averaged about 3.5 times longer than CPU time. This should be kept in mind when making comparisons between the various hardware platforms. Results are shown in tables 7-1 through 7-4.

Table 7-1. Time to fill all-body matrix, seconds.

Number	COMPUTER PLATFORM					
of	Micro-Vax	1 Xputer	4 Xputer		CONVEX	
Unknowns			Wrap Map	Block Map	CPU	Elapsed
104	73.7	26.6	16.7	8.8	5.1	20
152	151.7	55.0	33.4	16.5	10.7	26
200	257.5	93.8	56.1	26.6	18.3	69
248	394.0		85.4	39.7	27.9	105
296	555.9		120.1	54.9	39.4	145
356	801.6		172.8	78.0	56.9	201
404	1027.8		221.4	99.1	73.1	260
452	1282.3		276.1	122.9	91.4	329

Table 7-2. Time to factor & solve all-body matrix, seconds.

Number of Unknowns	COMPUTER PLATFORM					
	Micro-Vax	1 Xputer	4 Xputer		CONVEX	
			Wrap Map	Block Map	CPU	Elapsed
104	15.1	6.7	1.8	1.8	0.5	2
152	45.5	20.2	5.3	5.3	1.3	4
200	101.8	42.7	11.7	11.8	2.8	7
248	191.9		21.9	22.1	5.1	32
296	323.7		36.9	37.1	8.5	30
356	559.9		63.6	63.9	14.5	55
404	814.3		92.5	92.8	21.0	79
452	1137.2		129.0	129.4	29.1	102

Table 7-3. Time to fill all-wire matrix, seconds.

Number of Unknowns	COMPUTER PLATFORM					
	Micro-Vax	1 Xputer	4 Xputer		CONVEX	
			Wrap Map	Block Map	CPU	Elapsed
47	19.1	7.3	3.3	2.0	0.9	1
97	78.9	30.6	14.0	8.0	3.6	9
147	180.5	70.1	31.4	17.9	8.3	23
197	322.4	125.4	56.7	32.0	14.9	54
247	505.3		88.0	49.6	23.4	112
297	729.3		127.8	71.9	33.8	154
347	993.7		173.1	97.2	46.0	166
397	1302.1		227.8	127.8	60.2	172

The results show that the micro-Vax fills a wire matrix 32% slower than it fills a body matrix of the same size. Using column wrap mapping the four transputers fill a wire matrix 8% slower than they do a body matrix. In contrast, the Convex fills a wire matrix 15% faster than it does a body matrix! This may be due to the Convex's automatic vectorizing abilities. For comparison, a single transputer fills a wire matrix 36% slower than it fills a body matrix. The 8% figure for the four transputers can be accounted for by the fact that the wire filling efficiency (55%) is about 28% greater than the body filling efficiency (40%) for these problems and 28+8=36. The conclusion here is that for the JUNCTION code, sequentially speaking, wire filling takes about one third longer than body filling.

Some of the comparison data is displayed graphically in figures 7-1 through 7-3. The data is for an all-body matrix. The figures show comparisons between the four transputer array, the Convex (CPU time), and projected results for a sixteen transputer array. A four transputer array with 1 Megabyte of RAM per node can solve problems with up to 450 unknowns. With a sixteen transputer array problems with more than 900 unknowns could be solved.

Table 7-4. Time to factor & solve all-wire matrix, seconds.

Number of Unknowns	COMPUTER PLATFORM					
	Micro-Vax	1 Xputer	4 Xputer		CONVEX	
			Wrap Map	Block Map	CPU	Elapsed
47	1.6	0.7	0.2	0.2	0.1	0
97	12.3	5.3	1.5	1.5	0.4	1
147	41.2	18.1	4.8	4.8	1.2	5
197	97.4	42.9	11.1	11.1	2.7	10
247	189.9		21.5	21.5	5.0	26
297	327.8		37.0	37.0	8.6	35
347	520.2		58.6	58.6	13.4	39
397	776.0		87.4	87.4	19.8	57

The plots show that a sixteen transputer array would perform better than the Convex for filling the matrix and almost as good as the Convex for factoring and solving the matrix (Note: a sixteen transputer array, including the motherboard, would cost about $15K). However, no attempt was made to improve the vectorization of JUNCTION for a more efficient computation on the Convex. It is envisioned that considerable improvement could be made in the Convex times for factoring and solving of the matrix.

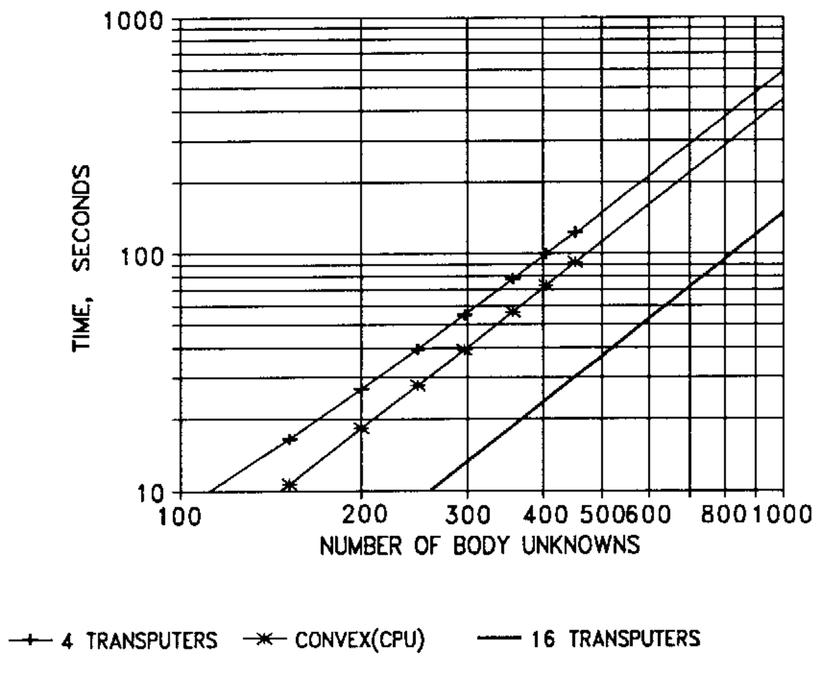

Figure 7-1. Matrix filling time comparisons, block mapping.

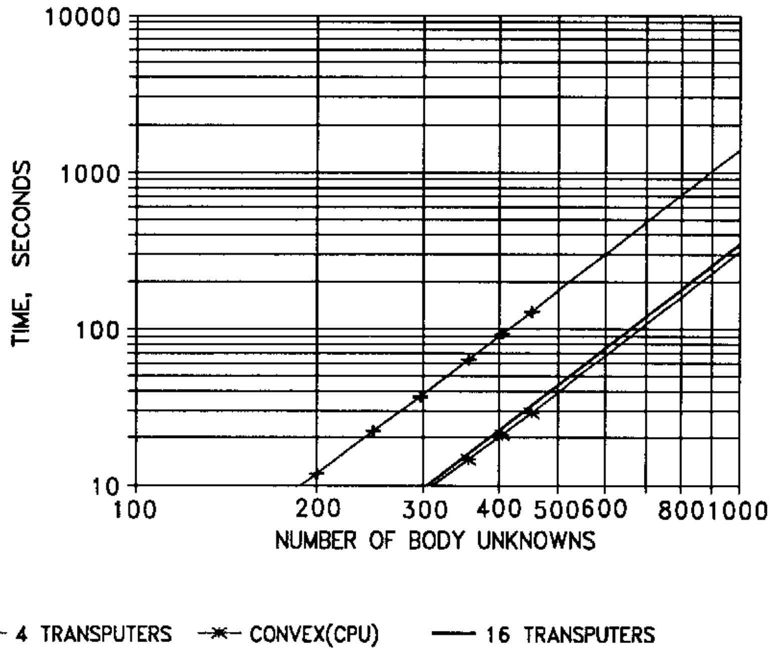

Figure 7-2. Matrix factoring and solving time comparisons.

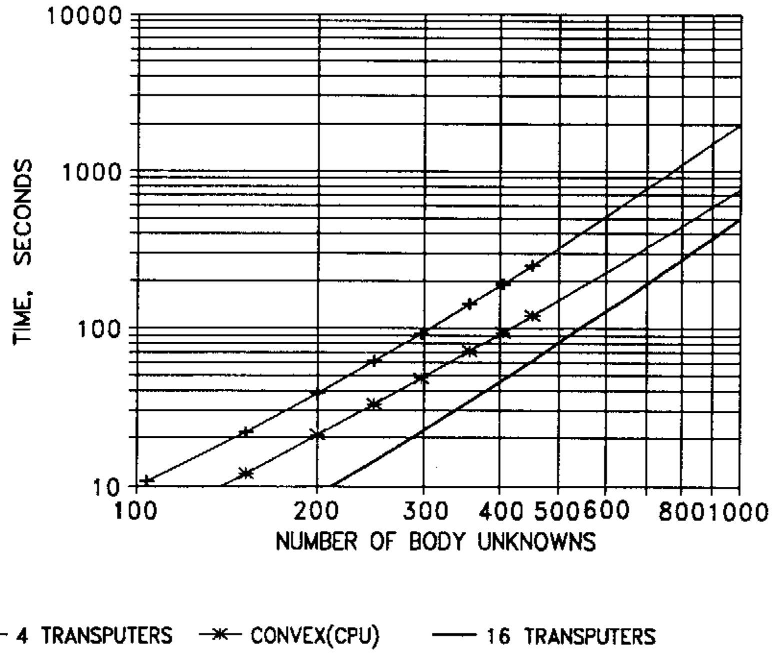

Figure 7-3. Total computation time comparisons, block mapping.

7.2 Observations

In the course of this effort a number of observations were made regarding transputers, ParaSoft EXPRESS, and parallel processing in general:

- ***A transputer array is an inexpensive, flexible parallel processing platform.*** An array of four transputer modules, each with 1 Megabyte of RAM, plus the PC motherboard cost less than $5K. The transputer array is very flexible since the number of processors can be easily expanded (just plug more modules in the sockets) and any Fortran or C code which runs on the PC can be modified to run on the transputer array.
- ***ParaSoft EXPRESS has been of great utility.*** Without EXPRESS it would have been very difficult to make the progress that was made in parallelizing the code. In addition, major modifications to the existing code would have been required, if not a complete rewriting of the code. EXPRESS will allow the porting of the code to another computer platform to be much more direct.
- ***Running an existing program on one transputer is very straightforward using EXPRESS.*** By adding just a few lines of code, then recompiling using the parallel compiler, and linking the object files to EXPRESS, an executable module which runs on a single transputer can be created for virtually any Fortran program which runs on the PC.
- ***The effort required to parallelize a code is algorithm dependent.*** Some algorithms can be parallelized just by adding a couple of lines of code. Other algorithms require extensive restructuring. Still other algorithms are not suited for implementation on a parallel computer and must be left as sequential code. It is not always obvious beforehand which algorithms will be efficient to parallelize and which won't.
- ***The optimal parallel code can be dependent on the number and type of processors as well as the size of the problem being solved.*** This was shown to be the case with the matrix solve algorithms. Two algorithms were tested: a cyclic implementation and a wavefront implementation. Although the cyclic implementation outperformed the wavefront implementation for the 4-transputer array, the literature suggests that the wavefront implementation will be the best performer for a massively parallel computing platform solving larger problems.

8.0 THE FUTURE

The state-of-the-art technology in the area of parallel processing changes monthly both for hardware platforms and software tools. The transputers used for this effort will soon be outdated, replaced by faster and more flexible processing units [Pountain, 1990]. New features are continuously being added to EXPRESS to make it more powerful and adaptable.

8.1 Future Hardware

The big news in the world of transputers is the development of the T9000. This next generation processor is being developed by the same company, INMOS, which did the original pioneering work on the development of the transputer [INMOS, 1988]. According to the manufacturer the key features of the T9000 are "a high performance pipelined superscalar processor and major support for multiprocessing applications. Peak performance will be more than 150 MIPS and 20 MFLOPS, representing a major advance in parallel computing and high speed communications...The design goals for T9000 were to enhance the transputer's position as the premier multiprocessing microprocessor, and to establish a new standard in single processor performance, while maintaining compatibility with existing transputer products." Of course, there will be some delay between the release of the T9000 and the development of compatible motherboards and software.

8.2 Future Software

A great deal of effort is going into developing programming software for parallel computing environments. This software includes parallel compilers, debuggers, performance analysis tools, visualization tools, dynamic load balancing tools, and automatic parallelization tools. Most of the presently existing software in these areas are very crude.

ParaSoft is in the process of improving EXPRESS. New tools which are being added include VTOOL, ASPAR, and EXDIST. VTOOL will allow memory access visualization. ASPAR provides automatic parallelization of sequential C programs. EXDIST will provide dynamic load balancing. ParaSoft's ultimate goal is to run EXPRESS within a heterogeneous parallel processing environment - known as a "meta" computer. A "meta" computer is made up of a number of architecturally different computers which are networked together and perform as a single entity.

REFERENCES

Dongarra, J.J., C.B. Moler, J.R. Bunch, and G.W. Stewart. 1979. *LINPACK User's Guide*, SIAM, Philadelphia, PA.

Harrington, R.F. 1968. *Field Computation by Moment Methods*, The Macmillan Company, NY.

Heath, M.T., and C.H. Romine. 1988. "Parallel Solution of Triangular Systems on Distributed-Memory Multiprocessors," *SIAM Journal of Scientific and Statistical Computing*, Volume 9, Number 3, May.

Hwu, S.U., and D. R. Wilton. 1988. "Electromagnetic Scattering and Radiation by Arbitrary Configurations of Conducting Bodies and Wires." NOSC TD 1325 (August). Naval Ocean Systems Center, San Diego, CA.

INMOS. 1988. *The Transputer Databook*, Bath Press Ltd, Bath, England.

Li, S.T., J.C. Logan, and J.W. Rockway. 1988. "Ship EM Design Technology," *Naval Engineers Journal*, May.

ParaSoft Corporation. 1990. *EXPRESS Fortran Reference Guide, Version 3.0,* Pasadena, CA.

ParaSoft Corporation. 1990. *EXPRESS Fortran User's Guide, Version 3.0,* Pasadena, CA.

Pountain, D. 1990. "Virtual Channels: The Next Generation of Transputers," *BYTE Magazine*, April.

Tazelaar, J.M. 1988. "Parallel Processing," *BYTE Magazine*, November.

Wilton, D.R., and S.U. Hwu. 1988. "JUNCTION Code User's Manual." NOSC TD 1324 (August). Naval Ocean Systems Center, San Diego, CA.

Wilton, D.R., and S.U. Hwu. 1989. "JUNCTION: A Computer Code for the Computation of Radiation and Scattering by Arbitrary Conducting Wire/Surface Configurations," 5th Annual Review of Progress in Applied Computational Electromagnetics (March).