Algorithms and Applications
in Parallel Computing

edited by

H. Power
Wessex Institute of Technology, UK

WIT*PRESS* Boston, Southampton
Computational Mechanics Publications

Portable Message Passing Implementation and Performance Analyses of Engineering Application Modules

J.C. Bruch, Jr.[1] and S. Boeriu[2]

[1]*Department of Mechanical and Environmental Engineering and Department of Mathematics, University of California, Santa Barbara, CA 93106*

[2]*Center for Computational Science and Engineering, University of California, Santa Barbara, CA 93106*

Abstract

Portability becomes an increasingly important and desirable property of parallel programs as the rate at which new machines are introduced increases. The principal reasons are an extended life of the application, and the flexibility to scale the application to different platforms.

A major concern in the design and development of parallel programs is the choice of communication paradigms. Message passing is probably the most widely used parallel programming model today. We will concentrate on portable message passing systems, as they offer the largest degree of portability and flexibility for developing parallel programs and in particular on the Message Passing Interface (MPI).

Two modules, based on: 1. finite difference scheme with SOR, applied to a wet chemical etching problem; and 2. adaptive mesh finite element analysis, applied to a free surface porous medium flow problem; initially written using the NX library (Intel Paragon) were ported to MPI and run on the Intel Paragon, Meiko CS-2, SP-2, and T3E.

A critical evaluation of the implementations and the problems related to porting the parallel applications from NX to MPI is

presented, followed by a detailed performance and scalability analysis. The Cray MPP Apprentice and IBM's VT (Visualization Tool) will be used to help analyze and understand execution behavior such as: communication patterns, processor load balance, computation versus communication ratios, timing characteristics, and processor idle time.

1 Introduction

Two problems from the class of free and moving boundary problems will be used as the working examples for the computer software that will be investigated. These problems are non-linear and have one boundary which is *a priori* unknown and is obtained in the process of solving the problem. Short descriptions and basic theoretical considerations for the problem formulations will be given in Sections 2–2.2 and 3–3.2. For more details, consult the references listed. The basic problem formulations will then be shown in the form of figures.

The porting of the parallel programs that solve these free and moving boundary problems to various parallel platforms and detailed performance and scalability analyses are discussed in Sections 2.3–2.6.2 and 3.3–3.6, respectively.

2 Wet chemical etching

The first problem from the class considered is that of wet chemical etching in semiconductor fabrication problems (Bruch *et al.* [1] and Vuik and Cuvelier [2]). An approximation for the physical problem is shown in Fig. 1. Here a gap of width $2a$ and length L is to be

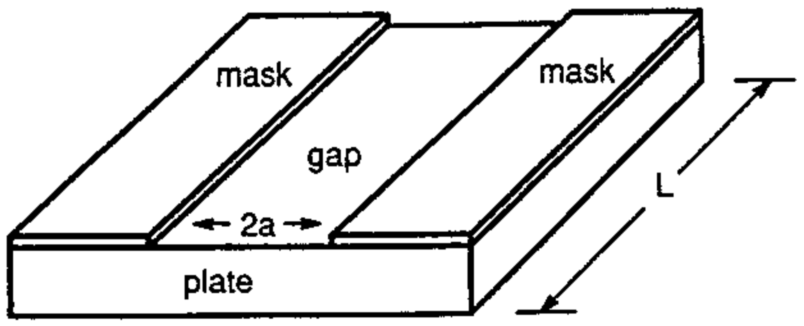


Figure 1: Physical problem.

etched in a flat plate. The remainder of the plate is covered with a protective (photoresist) layer. Since it is assumed that L is much larger than $2a$, the problem can be considered as two dimensional.

The following four simplifying assumptions are used to make the problem tractable. There is no convection in the etching medium; the etching process is isotropic; the thickness of the photoresist layer is infinitely small; and only one component of the etching liquid determines the process.

Figure 2 presents the mathematical model that is derived using the previous assumptions. The etching fluid $\Omega(t)$ is bounded by the outer boundary Γ_1, the photoresist layer $\Gamma_2(t)$, and the moving boundary $S(t)$. $D\backslash\Omega(t)$ denotes part of the solid. Let $\Omega(0)$ be a square region in the caustic fluid which is large enough so that increasing the size of $\Omega(0)$ will not change the etching process. Let D_1 be a rectangular region in the plate's cross section such that all the etching will occur in D_1 for $t \in [0, T]$. Denote by Γ_3 the slit of length $2a$, $\Gamma_3 = \partial\Omega(0) \cap \partial D_1$. Then the domain of the problem, D, is the union of the upper rectangular region $\Omega(0)$, the lower rectangular region D_1 in the plate, and Γ_3 along the boundary between the two, i.e. $D = \Omega(0) \cup D_1 \cup (\partial\Omega(0) \cap \partial D_1)$. Let $\Omega(t)$ be the open region in D at time $t \in (0, T)$ that is wet. Denote by Γ_1

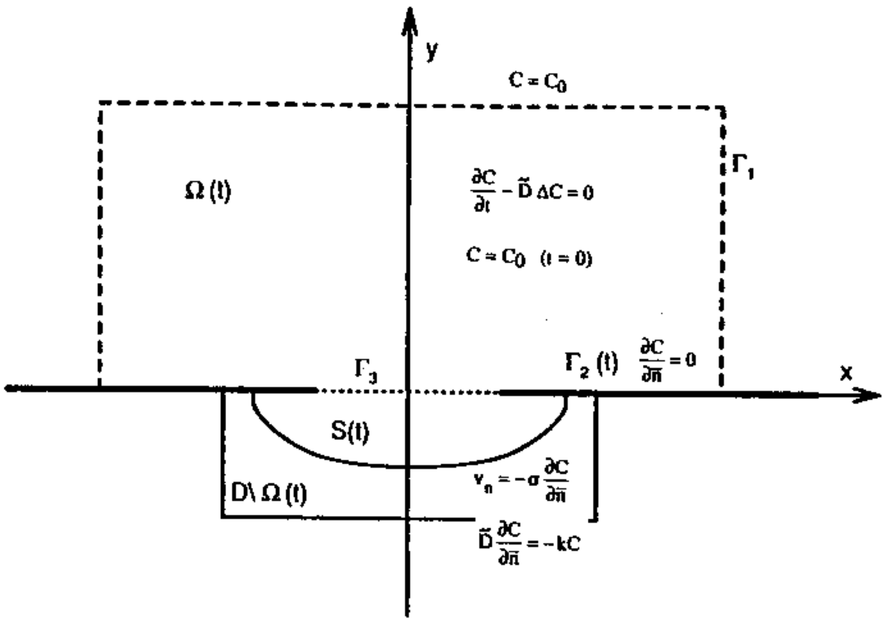

y
$C = C_0$
Γ_1
$\Omega(t)$
$\frac{\partial C}{\partial t} - \tilde{D}\Delta C = 0$
$C = C_0 \quad (t = 0)$
Γ_3
$\Gamma_2(t) \quad \frac{\partial C}{\partial \vec{n}} = 0$
x
S(t)
$D\backslash\Omega(t)$
$v_n = -\sigma\frac{\partial C}{\partial \vec{n}}$
$\tilde{D}\frac{\partial C}{\partial \vec{n}} = -kC$

Figure 2: Side view of physical problem showing mathematical solution setup.

the boundary of $\Omega(0)$ that is not along the mask or Γ_3, and $\Gamma_2(t)$ for the upper and lower surfaces of the mask that are exposed to the caustic fluid.

In Fig. 2, $C = C(x,y,t)$ is the concentration of the component in $\Omega(t)$ with $C_0 = C(x,y,0)$ being the initial concentration. $\bar{D}$ is the diffusion coefficient; v_n is the normal velocity of the boundary $S(t)$; σ is a material constant; k denotes the rate of reaction; and $\bar{n}$ is the outward unit normal vector.

Non-dimensionalizing the problem, two non-dimensional groups arise, $B = \tilde{D}/(\sigma C_0)$ and Sh $= ak/\bar{D}$, the Sherwood number. Herein k is assumed to tend to infinity which means the Sherwood number is large. Therefore, the lower boundary condition on $S(t)$ becomes $C = 0$. The upper boundary flux condition becomes

$$\frac{\partial C}{\partial \bar{n}} = -Bv_n \text{ on } S(t). \qquad (1)$$

2.1 Fixed domain formulation

Now applying the method of variational inequalities which uses a fixed domain method and a Baiocchi type transformation,

$$w(x,y,t) = \int_0^t \bar{C}(x,y,\tau)\,\mathrm{d}\tau \qquad (x,y)\in D\ ,\ t\in(0,T), \qquad (2)$$

where $\bar{C}(x,y,t)$ is an intermediate dependent variable which is equal to $C(x,y,t)$ in the etching fluid domain and has been extended continuously across the moving boundary, $S(t)$, into the fixed domain $D\backslash\Omega(t)$, yielding the fixed domain formulation shown in Fig. 3. (Note Δw stands for the Laplacian of w.) This is the problem that shall be solved numerically using a parallel supercomputer.

If w is a solution of the above problem, then $\bar{C} = \partial w/\partial t$ solves the original problem.

2.2 Numerical algorithm

The basic numerical algorithm is:

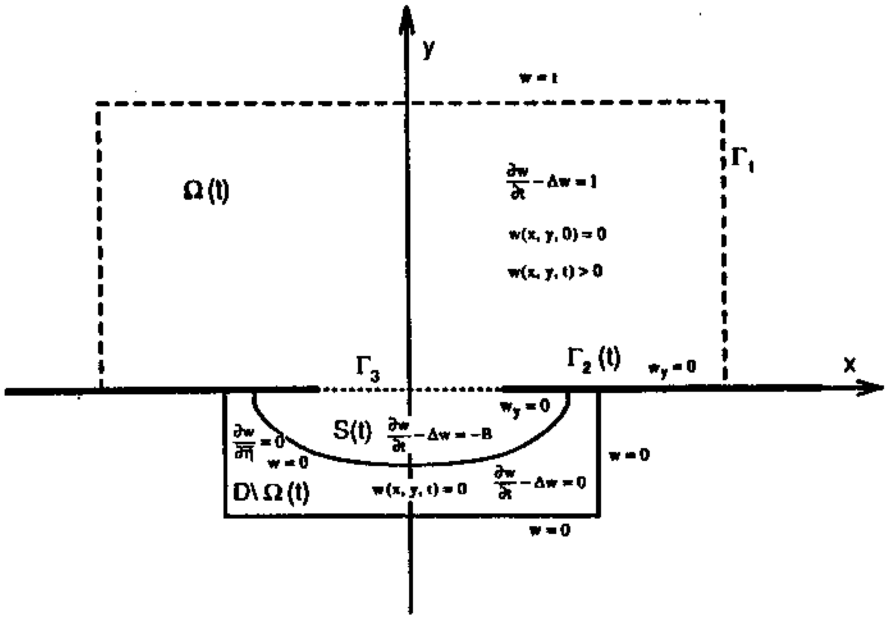


Figure 3: Fixed domain mathematical formulation.

$$\left(w_1\right)_{i,j,k}^{(n+1/2)} = \left(\frac{1}{\Delta t} + \frac{2}{(\Delta x)^2} + \frac{2}{(\Delta y)^2}\right)^{-1} \left\{C_0 + \frac{1}{\Delta t}\left(w_1\right)_{i,j,k-1} + \frac{1}{(\Delta x)^2}\left[\left(w_1\right)_{i-1,j,k}^{(n+1)} + \left(w_1\right)_{i+1,j,k}^{(n)}\right] + \frac{1}{(\Delta y)^2}\left[\left(w_1\right)_{i,j-1,k}^{(n+1)} + \left(w_1\right)_{i,j+1,k}^{(n)}\right]\right\} \tag{3}$$

with

$$\left(w_1\right)_{i,j,k}^{(n+1)} = \left(w_1\right)_{i,j,k}^{(n)} + \theta\left[\left(w_1\right)_{i,j,k}^{(n+1/2)} - \left(w_1\right)_{i,j,k}^{(n)}\right] \tag{4}$$

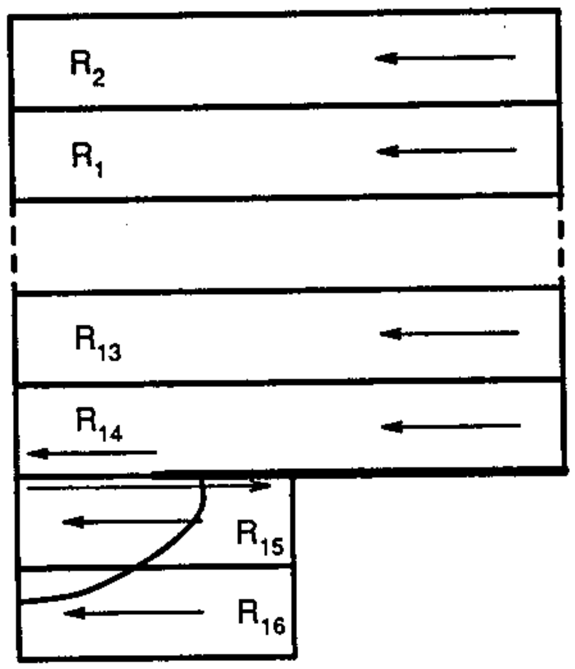


Figure 4: Domain decomposition of mathematical problem into sixteen subregions showing the flow of computations in each.

in $\Omega(0)$ and

$$\left(w_2\right)_{i,j,k}^{(n+1/2)} = \left(\frac{1}{\Delta t} + \frac{32}{(\Delta x)^2} + \frac{32}{(\Delta y)^2}\right)^{-1} \left\{-B + \frac{1}{\Delta t}\left(w_2\right)_{i,j,k-1} + \frac{16}{(\Delta x)^2}\left[\left(w_2\right)_{i-1,j,k}^{(n+1)} + \left(w_2\right)_{i+1,j,k}^{(n)}\right] + \frac{16}{(\Delta y)^2}\left[\left(w_2\right)_{i,j-1,k}^{(n+1)} + \left(w_2\right)_{i,j+1,k}^{(n)}\right]\right\} \tag{5}$$

with

$$\left(w_2\right)_{i,j,k}^{(n+1)} = \max\left\{0, \left(w_2\right)_{i,j,k}^{(n)} + \theta\left[\left(w_2\right)_{i,j,k}^{(n+1/2)} - \left(w_2\right)_{i,j,k}^{(n)}\right]\right\} \tag{6}$$

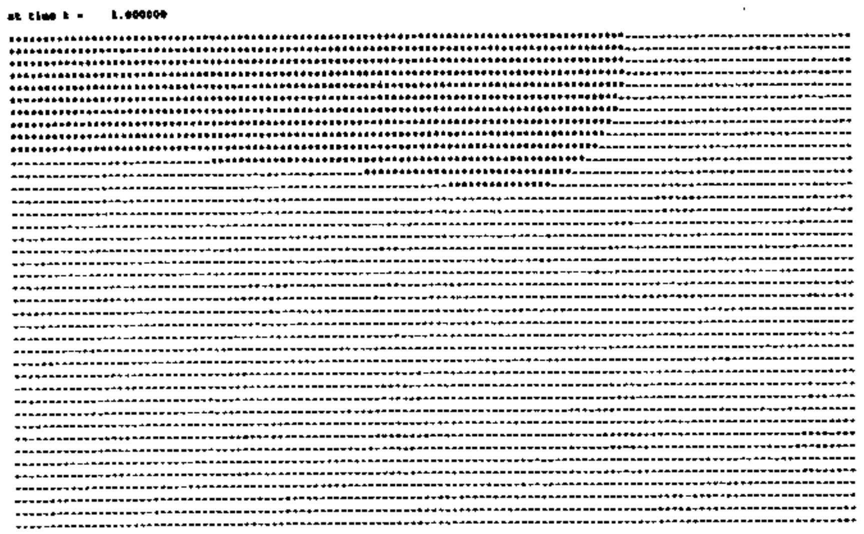


Figure 5: Computed results for the etched region (asterisks) at $t = 1$.

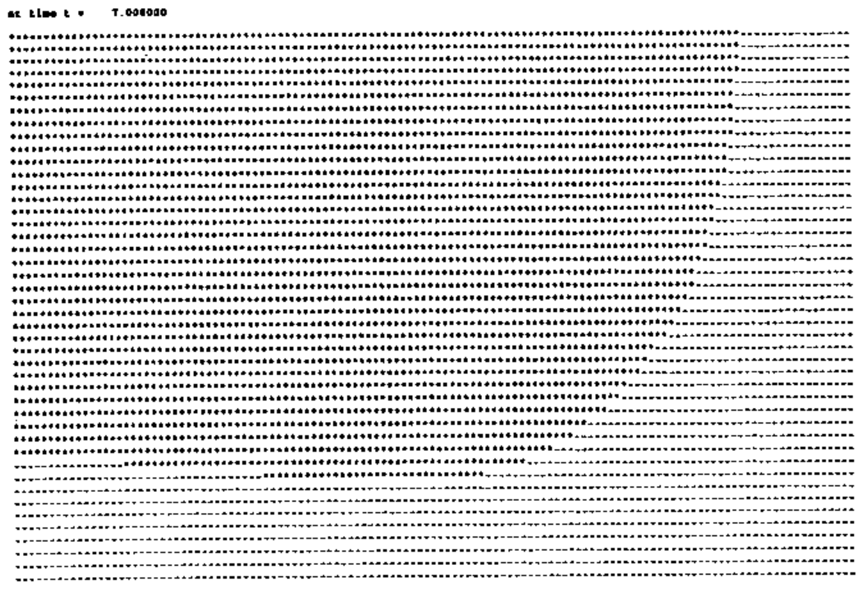


Figure 6: Computed results for the etched region (asterisks) at $t = 7$.

in D_1, where use was made of a first order backwards difference formula for $\partial w/\partial t$, second order central difference formulas for Δw, n denotes the SOR iterate, and θ is the SOR relaxation factor. Since there is a line of symmetry along the y-axis through the middle of the region, only the right half of the region will be solved numerically.

The parallel scheme presented in Bruch *et al.* [1] is used on the problem divided into horizontal strips, such as the subregion case seen in Fig. 4. For the details concerning the parallel algorithm see Bruch *et al.* [1]. Figures 5 and 6 show the computed results at times $t = 1$ and $t = 7$, respectively, for the etching process using 4 processors.

2.3 Porting the 'etch' module to MPI

The module for the etching problem was initially written in Fortran-77 using the NX communication library [3] and run on the Intel iPSC/860 and Intel Paragon. NX is a high performance library, being, however, customized for a particular platform.

As the computer hardware changes at a rapid pace, porting the modules to the MPI library seems the best option available. The MPI library [4], [5] is a standard and with its portability, efficiency and functionality, is also the best library for multidisciplinary applications.

The 'etch' module was ported to MPI and run on the Meiko CS-2 [6], IBM SP-2 [7] and Cray T3E [8].

As most of the NX routines have MPI counterparts, porting is relatively straightforward. Some difficulties might be present where hardware/software is different: host/node programs, cooperative I/O and Intel-specific optimization [4], [9]. As the 'etch' module was initially a host/node program (running on Intel iPSC/860), the host/node logic and structure was changed. Good correspondence exists between nonblocking NX and MPI point-to-point communication.

Data in the NX library are untyped - the message buffers are a string of bytes and the length of message is specified in bytes. Most data in MPI are typed - the message buffers are arrays of integers, reals, doubles, or more complicated structures. The length of the message is specified by the number of data elements. In NX, the synchronous send has the form:

```
    csend(tag, buf, size, dest, ptype)
    example:    real tmp(330)
                csend(250, tmp, 330, i, 0)
```

In MPI, the basic send has the form:

```
    MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    example:    real tmp(330)
                MPI_Send(tmp, 330, MPI_Real, i, 250,
                MPI_COMM_WORLD, ierr)
```

All the NX collective operations have MPI counterparts. While NX provides access to basic parallel I/O capabilities, the MPI does not currently define any parallel I/O calls.

2.4 Test case

The mathematical problem considered for the test case is shown in Fig. 3 and the Input entries are shown in Table 1.

Table 1. Input data for etch

maxrow1	= 280	maxrow2	= 80
maxcol1	= 321	maxcol2	= 161
maxtime	= 5	Δt	= 1.0
θ	= 1.935	B	= 10.00

where:

maxrow1	=	number of rows in the top region,
maxcol1	=	number of columns in the top region,
maxrow2	=	number of rows in the bottom region,
maxcol2	=	number of columns in the bottom region,
maxtime	=	number of time steps,
Δt	=	size of time step,
θ	=	successive over-relaxation factor,
B	=	non-dimensional number.

The domain decomposition for the 16 node case is shown in Fig. 4. The load balancing for the test case is shown in Table 2.

Table 2. Load balancing information for the test case

Nodes	2	4	8	16	32	64
bottom nodes	1	1	1	2	4	8
bottom points	12880	12880	12880	6440	3220	1610
top nodes	1	3	7	14	28	56
top points	89880	30174	12840	6420	3210	1605

2.5 Performance considerations

The 'etch' module was run on the T3E, SP-2 and Meiko CS-2. The speed-up on the T3E is shown in Table 3 and Fig. 7 for the third time step.

Table 3. Ideal and obtained speed-up

Nr. of PE's	Ideal Speed-up	Obtained Speed-up
1	1.0	1.0
4	4.0	3.66
8	8.0	7.20
16	16.0	13.85
32	32.0	26.12
64	64.0	47.40

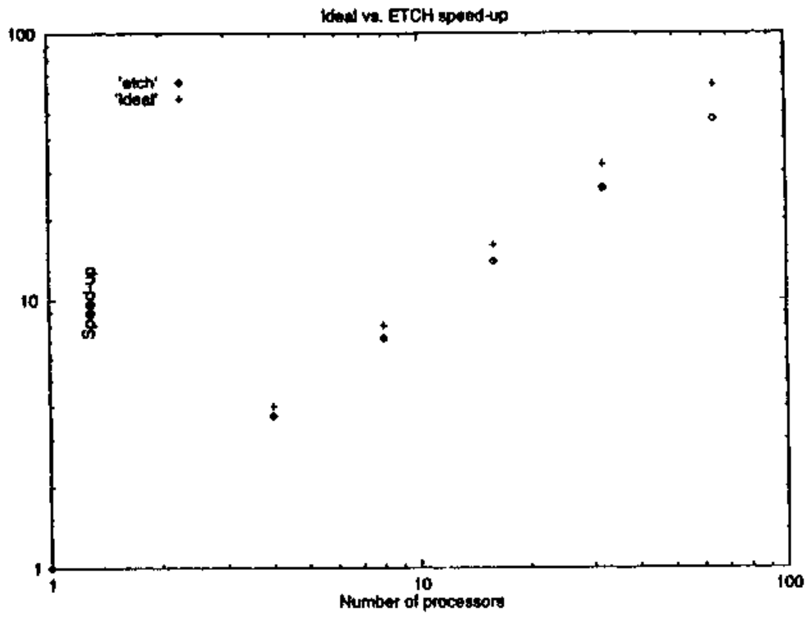


Figure 7: Ideal versus obtained speed-up.

2.6 Performance tools

The Cray's MPP Apprentice and the IBM's VT tools were used to find and correct performance anomalies and inefficiencies in the module.

2.6.1 MPP Apprentice

MPP Apprentice [10] is a performance tool that can help tune the performance of the application run on a Cray platform through an X Window System user interface.

The MPP Apprentice has the following characteristics:

- It is a postexecution performance analysis tool, that provides information about the program by examining data files that were created at compile time and run time.

- It uses program summary information, not event traces. The MPP Apprentice tool records the amount of time spent in execution of each portion of the code (called elapsed time).

- It shows the total execution time, time to execute a subroutine, communication time and number of instruction executed.

To use the MPP Apprentice:

- compile with the Apprentice option
- link the object files
- execute the program
- invoke the Apprentice tool

When accessed, the following windows and displays are available:

- Navigational display (Fig. 8). It shows the total time of each code object and permits navigation through the code structure. For each code object displayed, there is a graphic representation of the time spent in overhead, parallel work and Input/Output operations. The display includes the toggle buttons 'Include' and 'Exclude', which enable you to show data for all instrumented routines in the code. When 'Exclude' is selected, the total time is the time for the code object excluding time spent in called subroutines. When 'Include' is selected, the total time is for the code object including time spent in called routines. The Navigational display lists all functions and subroutines in the program and the time spent in each. Functions and subroutines are shown in decreasing order of time spent.

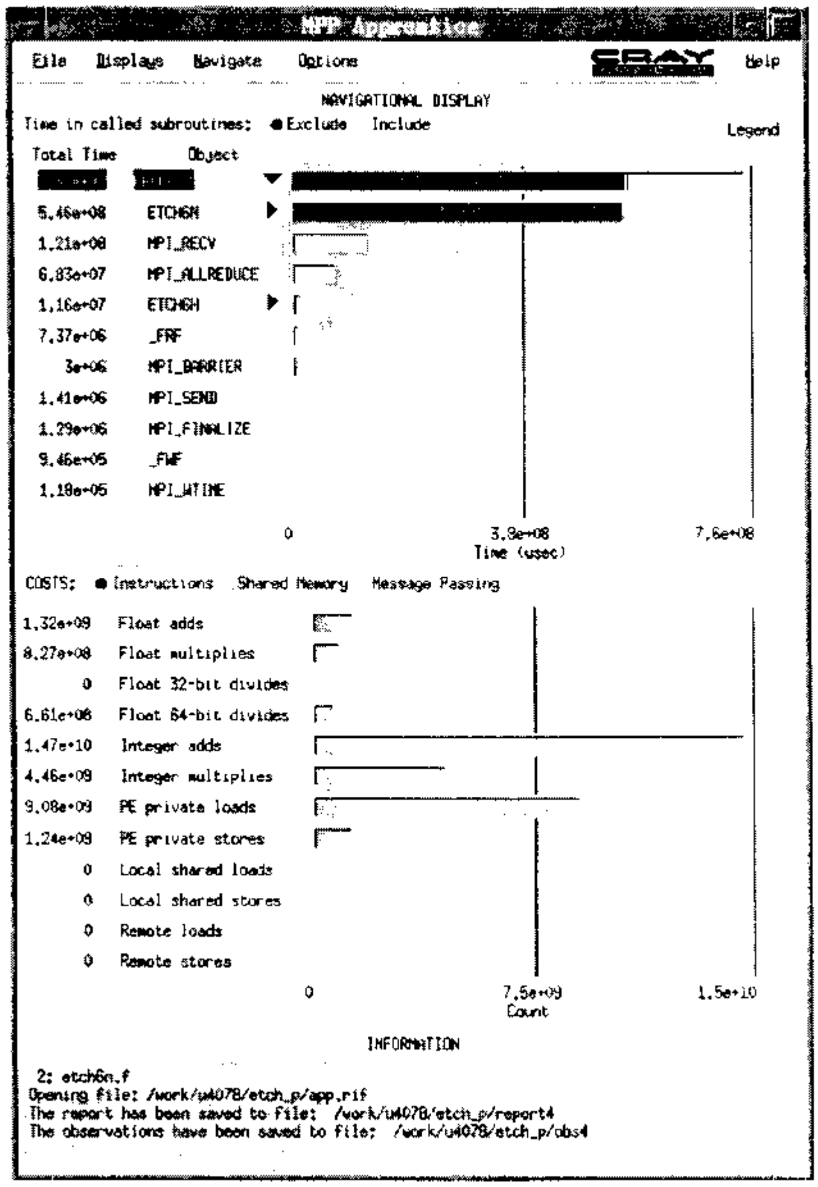
- Legend button (Fig. 9). It is shown as bar graphs of various colors.

- Costs display (Fig. 8). This display is actually three separate displays and appears in the middle of the MPP main window. It consists of the following displays, each of which can be accessed by clicking on the appropriate button in the display title:

  - Instructions display
  - Shared Memory display
  - Message Passing dispay

- Information display (Fig. 8), which appears at the bottom of the MPP Apprentice main window and displays informational and error messages.

The MPP Apprentice tool can analyse and offer specific guidance for problems such as:

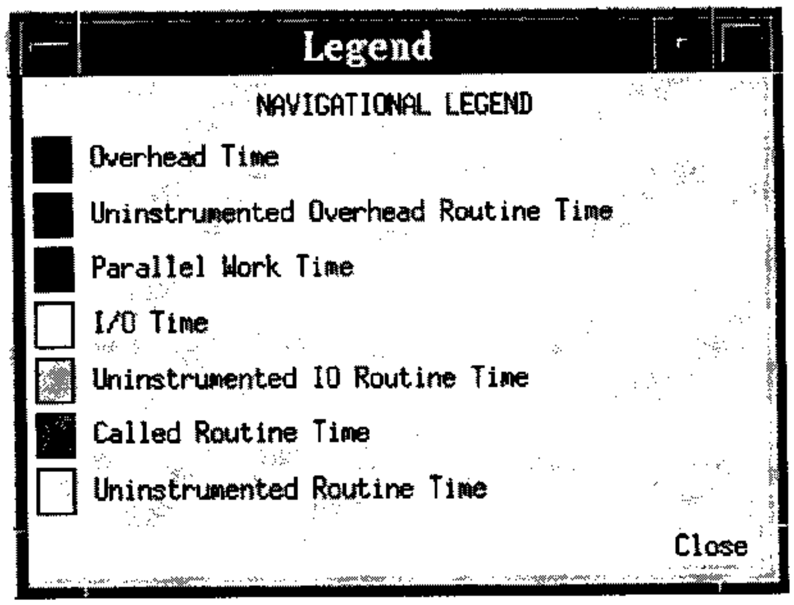


```
Total Time    Object
5.46e+08      ETCHGN
1.21e+08      MPI_RECV
6.83e+07      MPI_ALLREDUCE
1.16e+07      ETCHGH
7.37e+06      _FRF
   3e+06      MPI_BARRIER
1.41e+06      MPI_SEND
1.29e+06      MPI_FINALIZE
9.46e+05      _FWF
1.18e+05      MPI_WTIME

              0          3.9e+08          7.6e+08
                        Time (usec)

COSTS:  ● Instructions   Shared Memory   Message Passing

1.32e+09   Float adds
8.27e+08   Float multiplies
       0   Float 32-bit divides
6.61e+08   Float 64-bit divides
1.47e+10   Integer adds
4.46e+09   Integer multiplies
9.08e+09   PE private loads
1.24e+09   PE private stores
       0   Local shared loads
       0   Local shared stores
       0   Remote loads
       0   Remote stores

              0          7.5e+09          1.5e+10
                          Count

 2: etch6n.f
Opening file: /work/u4078/etch_p/app.rif
The report has been saved to file:  /work/u4078/etch_p/report4
The observations have been saved to file:  /work/u4078/etch_p/obs4
```

Figure 8: MPP Apprentice. Navigational: costs and information display.

- Load imbalance
- Excessive serialization
- Excessive communication
- Network contention
- Poor use of the memory hierarchy

Load imbalance problems can occur in many ways, and all result in processing elements (PE) being blocked on some form of synchronization. One example of this is when a computation includes a global reduction of some value across all PEs. If one PE is delayed reaching the global reduction, all other PEs are blocked until the delayed PE can participate. This is one problem we can identify for the 'etch' module: the MPI_ALLREDUCE has a relatively high value (Table 4). We use MPI_ALLREDUCE to check for some convergence criteria and this check is used starting with the first iteration, which is clearly excessive. Study and testing is under way to improve this situation. More information is obtained by using the IBM's VT and is presented and discussed in Section 2.6.2.



■	Overhead Time
■	Uninstrumented Overhead Routine Time
■	Parallel Work Time
□	I/O Time
▨	Uninstrumented IO Routine Time
■	Called Routine Time
□	Uninstrumented Routine Time

Figure 9: MPP Apprentice legend.

Communication Problems. In message passing, one processing element (PE) explicitly sends a message to another PE, who explicitly receives it. When the sending PE has more work to do than the receiving PE, the receiving PE might become idle until the message arrives. This is a second problem we can identify for the 'etch' module: also MPI_RECV has a relatively high value (Table 4). Study and testing is under way to improve this situation (using user-specified Send - MPI_BSend, nonblocking Send - MPI_ISend, etc). More information is obtained by using the IBM's VT and is presented and discussed in Section 2.6.2.

Table 4. Report window

INSTRUMENTED SUBROUTINE ETCH			
Exclusive Time		2.6564	microseconds
Inclusive Time		7.52662e+08	microseconds
Time in Called Routines		7.52662e+08	microseconds
Parallel Work Time		2.6564	microseconds
INSTRUMENTED SUBROUTINE ETCH6H			
Exclusive Time		1.16211e+07	microseconds
Inclusive Time		1.80862e+08	microseconds
Time in Called Routines		1.69241e+08	microseconds
Parallel Work Time		3.46205e+06	microseconds
INSTRUMENTED SUBROUTINE ETCH6N			
Exclusive Time		5.45637e+08	microseconds
Inclusive Time		7.39061e+08	microseconds
Time in Called Routines		1.93424e+08	microseconds
Parallel Work Time		5.45477e+08	microseconds
UNINSTRUMENTED ROUTINES	TYPE	TIME	
f$init	Misc	3798.74	microseconds
MPI_INIT	Misc	6461.19	microseconds
MPI_COMM_RANK	Misc	13.092	microseconds
MPI_COMM_SIZE	Misc	13.332	microseconds
MPI_FINALIZE	Misc	1.29491e+06	microseconds
_fcd_copy	Misc	62336.2	microseconds
_STOP	Misc	0	microseconds
_FWF	IO	945780	microseconds
_FRF	IO	7.37313e+06	microseconds
_OPEN	Misc	80971	microseconds
_CLOSE	Misc	4004.05	microseconds
$sldiv	Misc	3.05636	microseconds
MPI_SEND	Misc	1.4102e+06	microseconds
MPI_WTIME	Misc	118153	microseconds
MPI_RECV	Misc	1.21083e+08	microseconds
MPI_BARRIER	Misc	2.99616e+06	microseconds
MPI_ALLREDUCE	Misc	6.83446e+07	microseconds

2.6.2 VT - the Parallel Environment's Visualization Tool

VT [11], [12] is essentially an animation of the communication and system events that occurred while the program was running. The presentation of data graphically rather than textually allows one to quickly discern differences among parallel tasks.

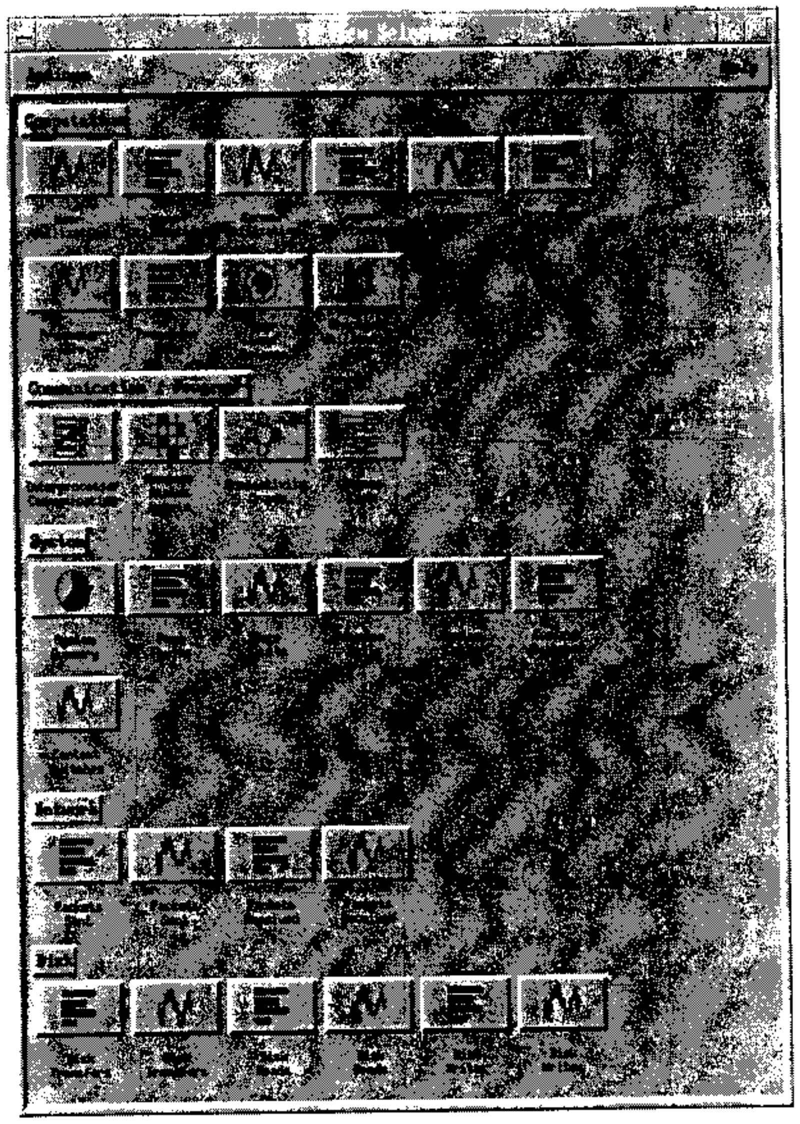


Figure 10: View Selector window.

The components of VT are:

- Trace Analysis (Trace Visualization)

The information is graphically displayed. You can select a variety of preconfigured displays by pressing a button. The displays are time synchronized so you can understand the sequence of program activity.

- Performance Monitor

The Performance Monitor is used while running the application program to obtain a basic understanding of the system resources being used.

The VT views enable you to visualize:

- communication among processor nodes
- the type and duration of communication events
- a parallel program's source code as it relates to the executable's run
- CPU utilization of processor nodes
- load balance

The VT arranges the views into the following categories (Fig. 10):

- Computation
- Communication/Program
- System
- Network
- Disk

To use the VT tool:

- compile the program with the -g option
- set the trace level and the sampling interval to the desired values
- run the program to generate a trace file
- start the VT session

The View Selector window (Fig. 10) and the Trace Visualization

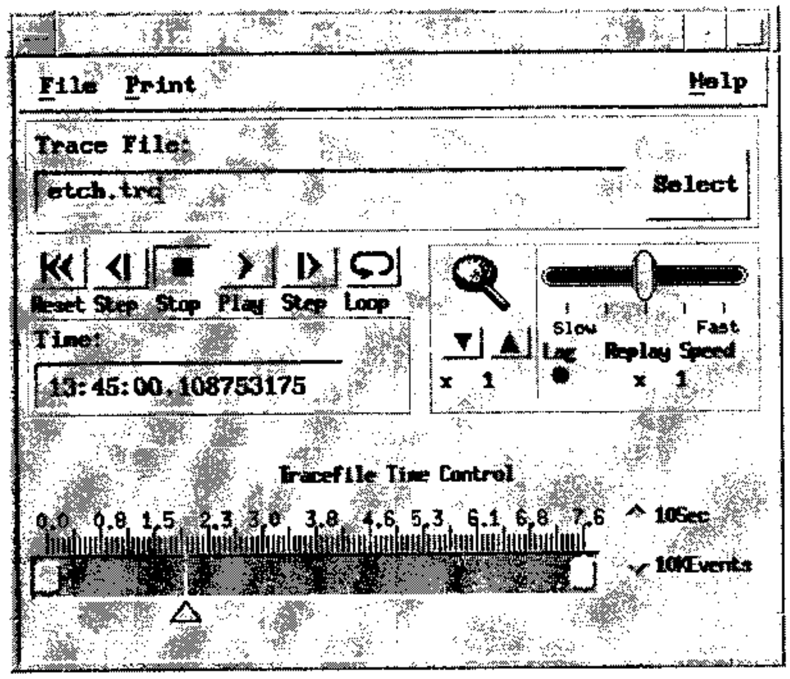


Figure 11: VT window for trace visualization.

window (Fig. 11) automatically opens at the start of a VT session.

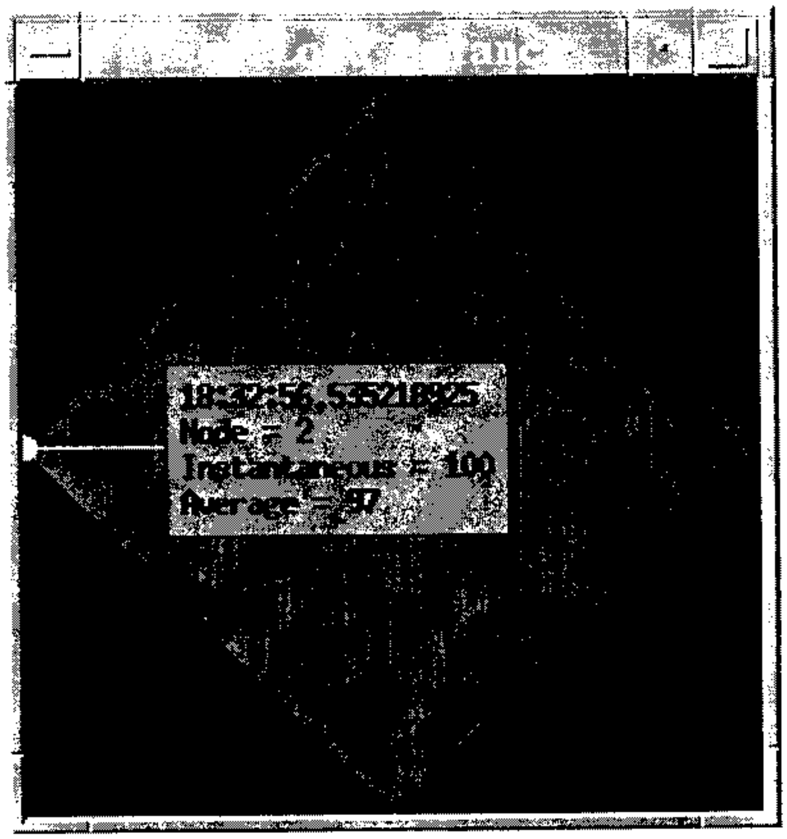
We selected the:

- User Load Balance
- Interprocessor Communication
- Source Code

User Load Balance.

This view uses three overlapping polygons to show the CPU utilization for each of the processor nodes, and the overall processor load balance. The largest of the polygons represents 100% utilization for all of the processor nodes.

VT draws the second polygon inside the first. This polygon represents the instantaneous CPU utilization for each of the processor nodes. On each node's spoke, VT draws a point which represents the current CPU utilization for that node. VT then connects the points to



```
18:32:56.535218925
Node = 2
Instantaneous = 100
Average = 97
```

Figure 12: User load balance.

form a polygon with a solid fill pattern. The more regular the polygon, the better the processor load balance.

The third polygon is similar to the second. It shows the average CPU utilization and has a hatched fill pattern.

The user load balance for the 'etch' module is shown in Fig. 12 and in Table 5.

Table 5. User load balance for 'etch'

PE	Instantaneous	Average
0	100%	82%
1	100%	96%
2	100%	97%
3	100%	98%

PE 0 has a load balance of 82% compared to the other PE's with aproximately 97%. As the first (100% utilization) polygon is identical to the second polygon (instantaneous CPU utilization - 100%), they are overlapped in Fig. 12.

Interprocessor communication

This view uses a bar chart to visualize the type and duration of communication events. Each bar represents a processor node on which the program was run, and the chart's horizontal axis represents a range of time. A label to the right of each bar shows the node number, and is colored based on the current state of that node. Each bar in the chart will be made up of a number of colored blocks. Each block represents a communication event involving the processor. The size of the block represents the event's duration, and the color indicates the type of event. For example, blocking sends are shown in one color, non-blocking sends in another, broadcasts in another, and so on. When messages are sent between processor nodes, a message line is drawn between the appropriate bars in the chart and the node labels light up. When nodes are involved in collective communication, a polygon is drawn covering the collective communication events.

In Fig. 13 the MPI_ALLReduce are in light blue, the MPI_Send (blocking) in dark blue, the MPI_Recv (blocking) in red, and No_Communication in light grey.

Lines are drawn when the communication is known to have been completed. They are drawn from the start of the event which initiated the communication to the end of the event which completed it. Thus the lines are drawn from the start of blocking or non-blocking sends to the ends of blocking receives, waits and status events. Lines are only drawn when the wait or status for any non-blocking calls involved in the communication have been completed.

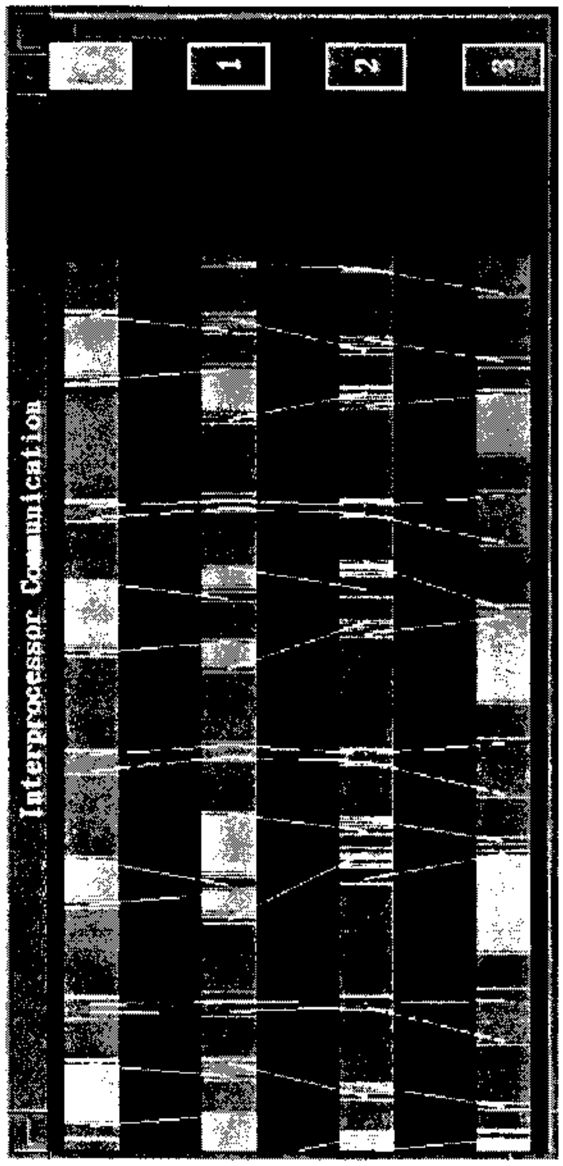
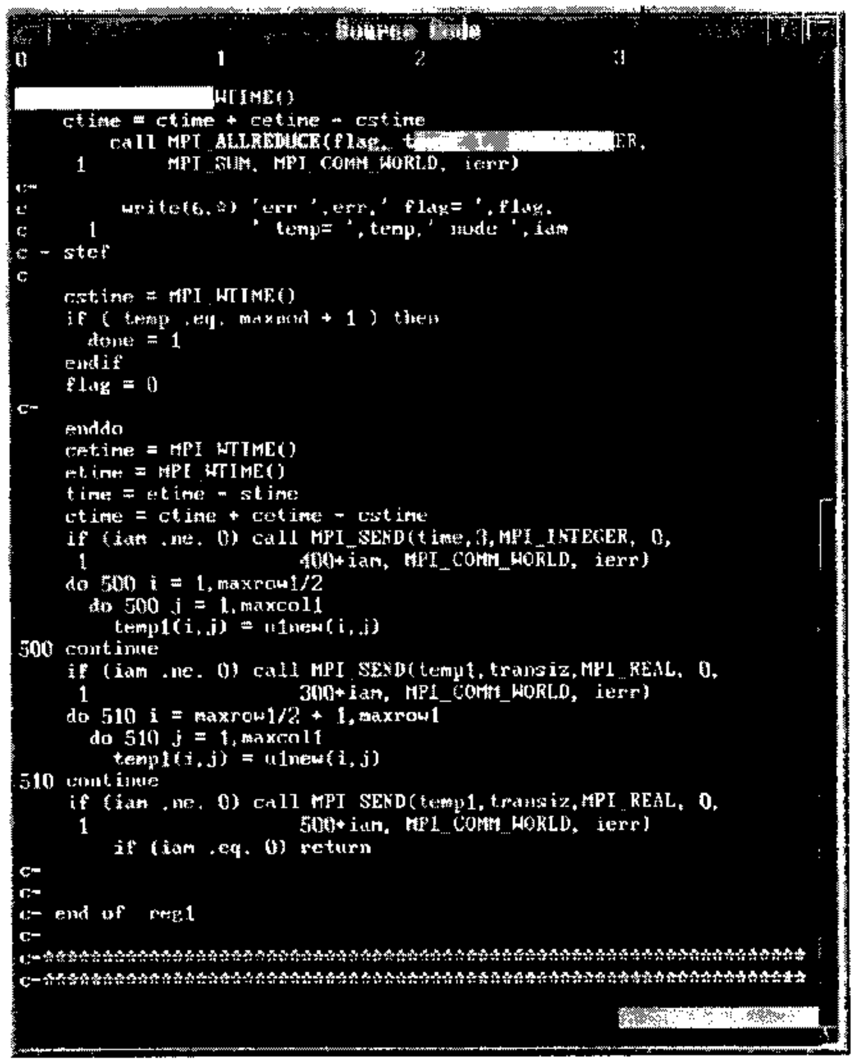


Figure 13: Interprocessor communication.

Source Code

This view shows the Fortran (or C or C++) source code of the program associated with the most recent trace event. A series of colored bars across the top of the display represent the different program tasks. As we play back the trace file, the bars move through the code to show each task's position in relation to the source. To be



```
[illegible]WTIME()
    ctime = ctime + cetime - cstime
        call MPI_ALLREDUCE(flag, t[illegible]ER,
     1        MPI_SUM, MPI_COMM_WORLD, ierr)
c-
c         write(6,*) 'err ',err,' flag= ',flag,
c    1                ' temp= ',temp,' node ',iam
c - stef
c
    cstime = MPI_WTIME()
    if ( temp .eq. maxnod + 1 ) then
      done = 1
    endif
    flag = 0
c-
    enddo
    cetime = MPI_WTIME()
    etime = MPI_WTIME()
    time = etime - stime
    ctime = ctime + cetime - cstime
    if (iam .ne. 0) call MPI_SEND(time,3,MPI_INTEGER, 0,
     1                       400+iam, MPI_COMM_WORLD, ierr)
    do 500 i = 1,maxrow1/2
      do 500 j = 1,maxcol1
        temp1(i,j) = u1new(i,j)
500 continue
    if (iam .ne. 0) call MPI_SEND(temp1,transiz,MPI_REAL, 0,
     1                       300+iam, MPI_COMM_WORLD, ierr)
    do 510 i = maxrow1/2 + 1,maxrow1
      do 510 j = 1,maxcol1
        temp1(i,j) = u1new(i,j)
510 continue
    if (iam .ne. 0) call MPI_SEND(temp1,transiz,MPI_REAL, 0,
     1                       500+iam, MPI_COMM_WORLD, ierr)
        if (iam .eq. 0) return
c-
c-
c- end of  reg1
c-
c-**************************************************************
c-**************************************************************
```

Figure 14: Source code corresponding to the Interprocessor Communication in Fig. 13.

more specific, for each time task entered into a communication function such as blocking send, an environment initialization, or an application marker call, its bar moves to that line in the source.

Figure 14 shows the Source Code corresponding to the Interprocessor Communication shown in Fig. 13. We can clearly see that the MPI_ALLReduce acts as a barrier, slowing down the execution.

Figure 15 shows the Interprocessor Communication for the final

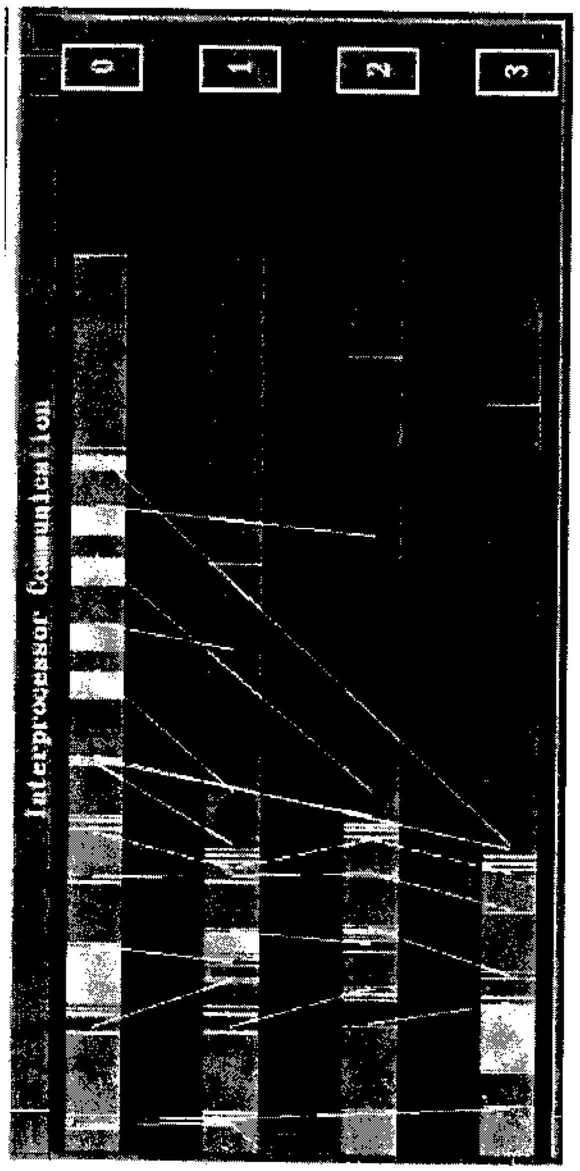
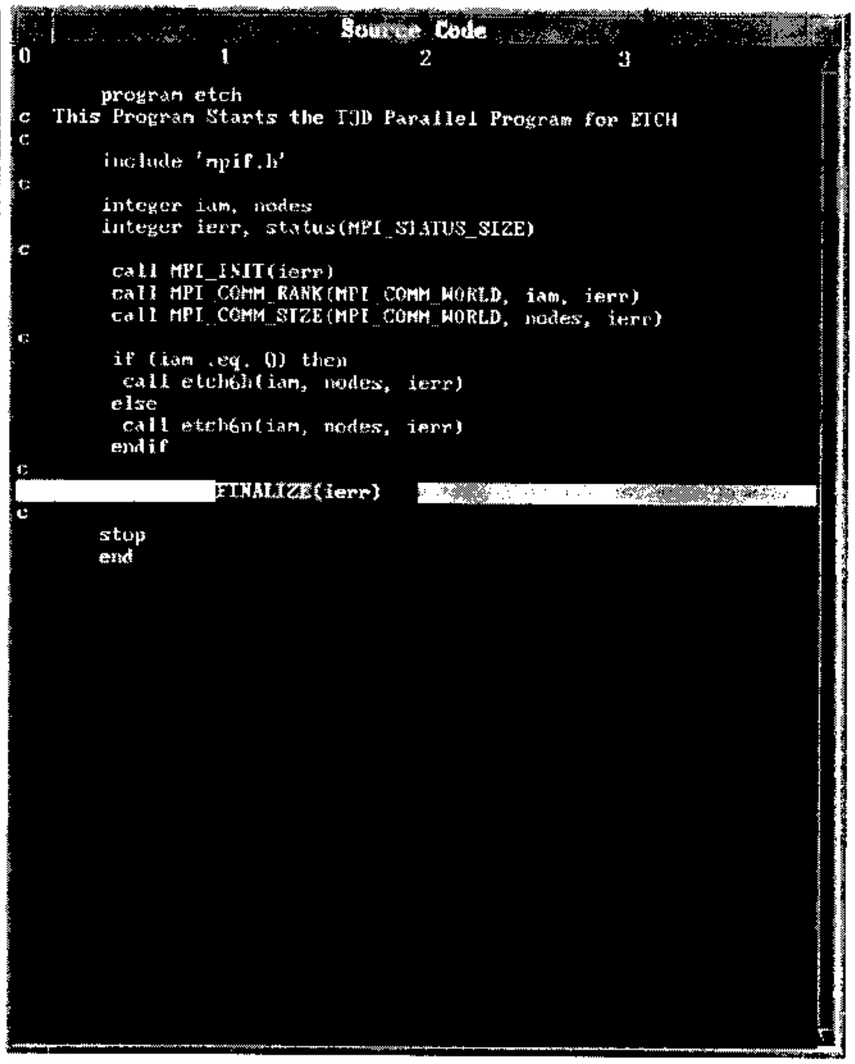


Figure 15: Final part of the Interprocessor Communication.

part of the 'etch' module, corresponding to the Source Code shown in Fig. 16.

The challenge of the message passing model is in reducing message traffic while ensuring that the correct and updated values of the passed data are promptly available to the tasks when required.



```
      program etch
c  This Program Starts the TJD Parallel Program for ETCH
c
      include 'mpif.h'
c
      integer iam, nodes
      integer ierr, status(MPI_STATUS_SIZE)
c
       call MPI_INIT(ierr)
       call MPI_COMM_RANK(MPI_COMM_WORLD, iam, ierr)
       call MPI_COMM_SIZE(MPI_COMM_WORLD, nodes, ierr)
c
       if (iam .eq. 0) then
        call etch6h(iam, nodes, ierr)
       else
        call etch6n(iam, nodes, ierr)
       endif
c
                  FINALIZE(ierr)
c
      stop
      end
```

Figure 16: Source Code corresponding to the Interprocessor Communication in Fig. 15.

Optimizing message traffic is one way of boosting performance.

Optimization of the parallel code is usually carried out in an iterative process involving several tools to investigate performance issues.

3 Free boundary problem

The second problem studied is the free surface seepage problem shown in Fig. 17. The following assumptions are made: the soil in the flowfield is homogeneous and isotropic; capillary and evaporation effects are neglected; the flow obeys Darcy's Law; the flow is two-dimensional and at steady state. Because of the assumptions made, the problem is described by the velocity potential function, φ, whose governing differential equation and boundary conditions are also shown in Fig. 17. The relevant dimensions are taken to be: $x_1 = 40$, $y_1 = 10$ and $y_2 = 3$. In Fig. 17, Ω is the seepage region abdf. The location of the curve $\overline{fd}$, $y = \bar{f}(x)$, is unknown *a priori.*

A fixed domain formulation for this problem can be obtained by using the Baiocchi method and transformation. (See Baiocchi and Capelo [13], Kinderlehrer and Stampacchia [14], Oden and Kikuchi [15], Bruch [16] and Crank [17]). In this approach the *a priori* unknown solution region is extended across the free surface into a known region. The dependent variable is also continuously, similarly extended. Then a new dependent variable is defined using Baiocchi's transformation within these regions. The resulting problem formulation leads to a 'complementarity system' associated with its respective variational or quasi-variational inequality formulation. This method has proven effective not only from the purely theoretical point of view, but also from the point of view of yielding new, simple, and efficient numerical solution schemes.

Figure 18 shows the governing equations and boundary conditions that describe the fixed domain formulation of the problem presented in Fig. 17. D is the region abef. The variable w is the Baiocchi transformation of the extended potential function, i.e.,

$$w(x,y) = \int_y^{y_1} \left[\tilde{\varphi}(x,\bar{\eta}) - \bar{\eta}\right] d\bar{\eta}\,, \tag{7}$$

where

$$\begin{aligned} \tilde{\varphi}(x,y) &= \varphi(x,y) && \text{in } \overline{\Omega} \\ \tilde{\varphi}(x,y) &= y && \text{in } \overline{D} - \overline{\Omega} \end{aligned} \tag{8}$$

The detailed derivations of these equations are given in Bruch [16].

The problem shown in Fig. 18 can be written as a 'complementarity system' and its corresponding variational inequality formulation. Then the following theorem can be stated:

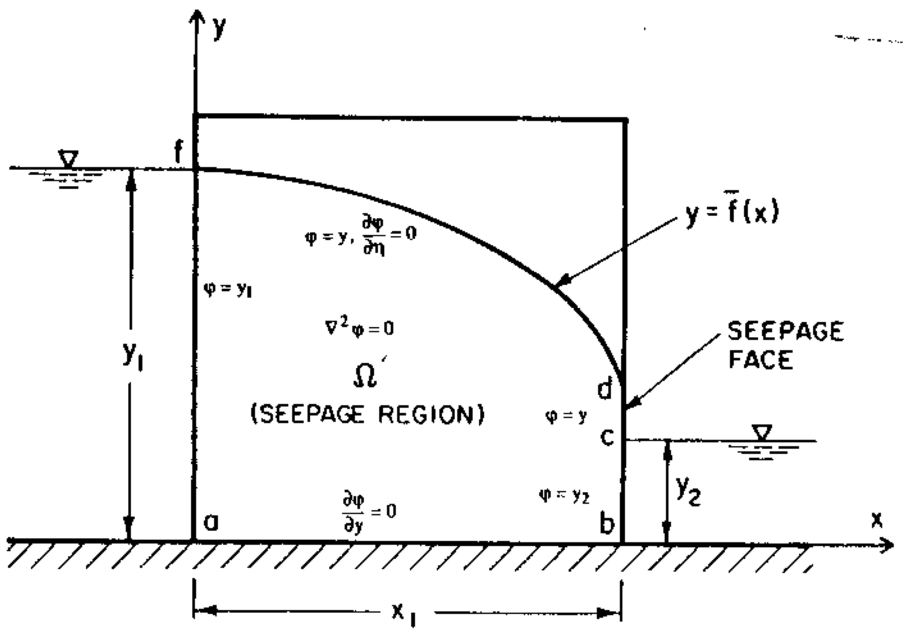


Figure 17: The example physical problem (free boundary seepage).

Let $\overline{p}$ (x,y) be the Dirichlet data in Fig. 18 and define

$$K = \left\{ v(x,y) \mid v \in H^1(D), v|_{\partial D} = \overline{p}, v \geq 0 \text{ a.e. on } D \right\},$$

a closed convex set, $K \subset H^1(D)$.

Theorem: If $w \in K$ satisfies the governing equations and boundary conditions shown in Fig. 18, then it also satisfies the variational inequality:

$$a(w, v-w) \geq L(v-w) \qquad \forall\, v \in K, \tag{9}$$

where

$$\begin{aligned} a(w, v-w) &= \iint_D \nabla w \cdot \nabla(v-w)\,\mathrm{d}x\,\mathrm{d}y \\ &= \iint_D \left\{ w_x(v_x - w_x) + w_y(v_y - w_y) \right\} \mathrm{d}x\,\mathrm{d}y \end{aligned} \tag{10}$$

and

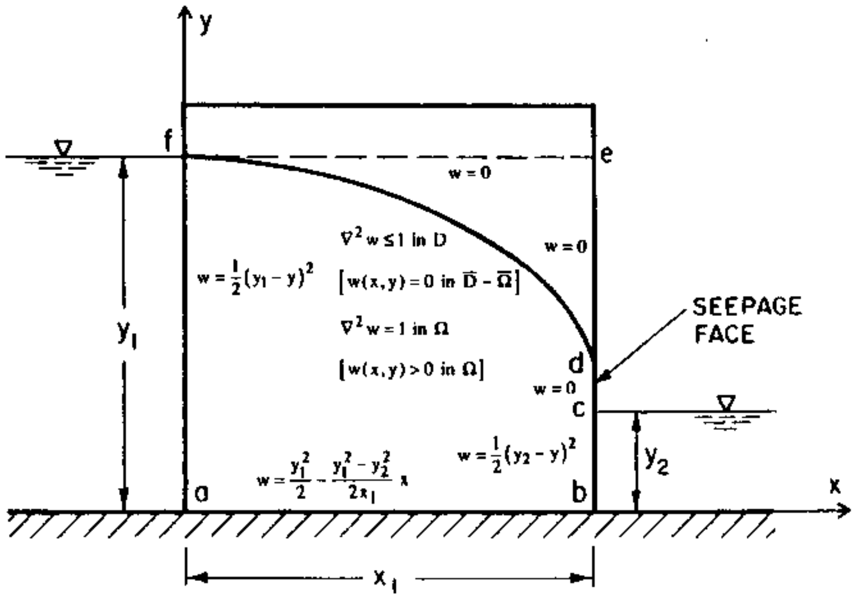

y
f
e
w = 0
$\nabla^2 w \le 1$ in D
w = 0
$w = \frac{1}{2}(y_1 - y)^2$
$[w(x,y) = 0$ in $\bar{D} - \bar{\Omega}]$
SEEPAGE FACE
y_1
$\nabla^2 w = 1$ in Ω
$[w(x,y) > 0$ in $\Omega]$
d
w = 0
c
$w = \frac{1}{2}(y_2 - y)^2$
y_2
$w = \frac{y_1^2}{2} - \frac{y_1^2 - y_2^2}{2x_1} x$
a
b
x
x_1

Figure 18: The example physical problem (free boundary seepage) for numerical implementation.

$$L(v-w) = -\iint_D (v-w)\mathrm{d}x\ \mathrm{d}y. \tag{11}$$

The finding of $w \in K$ is equivalent to solving the minimization problem

$$J(w) \le J(v) \qquad \forall v \in K, \tag{12}$$

where

$$J(v) = a(v,v) + 2(f,v) \tag{13}$$

in which $a(v, v)$ is a bilinear form, continuous, symmetric, positive definite on R and $f \in R$, i.e.,

$$a(v,v) = \iint_D \nabla v \cdot \nabla v\ \mathrm{d}x\ \mathrm{d}y, \tag{14}$$

$$(f,v) = \iint_D fv\ \mathrm{d}x\ \mathrm{d}y. \tag{15}$$

For this example problem, $f = 1$. The functional J has one and only one minimum in a closed convex set.

The minimum is found using the following finite element algorithm:

$$u_i^{(n+1/2)} = -\frac{1}{a_{ii}}\left(\sum_{j=1}^{i-1} a_{ij}u_j^{(n+1)} + \sum_{j=i+1}^{N} a_{ij}u_j^{(n)} + f_i\right), \qquad (16)$$

$$\begin{aligned} u_i^{(n+1)} &= P_i\left(u_i^{(n)} + \theta\left(u_i^{(n+1/2)} - u_i^{(n)}\right)\right) \\ &= \max\left(0, u_i^{(n)} + \theta\left(u_i^{(n+1/2)} - u_i^{(n)}\right)\right), \end{aligned} \qquad (17)$$

where $a_{ij} = a(N_i, N_j)$, $f_i = (f, N_i)$, N_i is the canonical basis of R^N, P_i is the projection on the convex set, $i = 1, \ldots, N$, N is the number of nodal points and θ is the relaxation factor. Linear triangular elements will be used in the discretization. It should be noted that the projection operation in the numerical scheme must be applied during the iteration process. It cannot be applied after the iteration process has been completed since if it were, an incorrect solution would be obtained. For the numerical results given herein, the SOR relaxation factor was 1.85, while the stopping error criterion was 10^{-4} for the maximum absolute difference between iterates at a mesh point.

3.1 Adaptive mesh finite element analysis

Error estimation and local mesh refinement are two major concepts of adaptive mesh finite element analysis. In addition, a mesh refinement algorithm is required to perform remeshing after obtaining the error of a finite element system. The error estimate decides how the computed results deviate from the exact solution. Local mesh refinement testing is performed to determine how the mesh is to be refined. The remeshing algorithm is then used to automatically generate a refined mesh according to the error obtained.

The error estimation procedure used herein was introduced by Zienkiewicz and Zhu [18]. It allows an accurate assessment of errors while remaining so simple that it can readily be implemented as a post-processor involving minimal computation. This computationally simple error estimator with the modification introduced by Burkley and Bruch [19] and Burkley *et al.* [20] is used for the solution of the free surface flow through an earth dam using a parallel computer. The modification used by Burkley and Bruch [19] was that, instead of using the Zienkiewicz-Zhu procedure (a projection method) to calculate the nodal estimates for the exact nodal fluxes, they performed a simple averaging technique. That is, for each node they added up say the x-fluxes (which were constants since linear elements were used) from the elements that contained that node and divided that sum by the number of contributing elements. The same was done for the y-fluxes. A mesh generator and a mesh refiner were used to perform the finite element analysis and the post-processing of the mesh refinement. Isosceles right triangle elements were used for the purpose of this study. A simple mesh generator and refiner for these elements was implemented to generate the initial mesh. The concept behind the mesh generation and mesh refinement is simple: divide an element into two by refining across its longest side. Also, in this study, incompatible elements were not allowed. Therefore, a recursive process proposed by Rivara ([21], [22]) was used to avoid having such elements.

3.2 Parallel iterative scheme

Wang and Bruch [23] proposed parallel iterative Gauss-Seidel and SOR iterative schemes. Conventional Gauss-Seidel and SOR iteration schemes need to be performed sequentially. Reordering the equations alters these two schemes into fully parallel iterative schemes. Wang and Bruch [23] used this intuitive idea and implemented it on a free boundary seepage problem. Speed-ups were obtained that were superlinear (speed-up larger than the number of processors used).

The basic essence of the approach is as follows: after the computation domain is subdivided into subdomains, the problem domain boundary remains a boundary and the interfaces of a subdomain become new boundaries. Thus, the computation of values at interior mesh points for one subdomain is uncoupled from the other subdomains. Also, the computation of values at mesh points of an interface is uncoupled from the other interfaces. The iterative schemes use a combination of newly computed values and old values at mesh points surrounding a mesh point to compute the new value at that mesh point. Therefore, the iterative process can be performed for the interior mesh points of a subdomain using the old values at interface mesh points. Moreover, the values at interface mesh points can be updated using the newly computed values at interior mesh points by the iterative process. An example and explanation are given in Wang and Bruch [23].

Accordingly, this parallel iterative scheme simply reorders the computing sequence such that the values at interior mesh points are computed first, then those at the interface mesh points are computed. With this parallel scheme, all processors can compute concurrently for the new values at the interior mesh points. Also, all processors update the values at mesh points on the interface in parallel.

Figure 19 shows the domain subdivision meshes for the third adaptive mesh refinement. See Wang and Bruch [24].

3.3 Porting the 'fea' module to MPI

The module for the seepage problem was initially written in Fortran-77 using the NX communication library [3] and run on the Intel iPSC/860 and Intel Paragon. The 'fea' module was ported to MPI and run on the Meiko CS-2 [6], IBM SP-2 [7] and Cray T3E [8].

As discussed in Section 2.3, as most of the NX routines have MPI counterparts, porting was relatively straightforward:

- the 'fea' module has more collective operations than the 'etch' module. However, good correspondence exists between the NX and MPI counterparts.
- the 'fea' module was initially a host/node program, so the host/node logic and structure was changed.

3.4 Test case

The physical problem considered for the test case is shown in Fig. 18. As presented in Section 3, the SOR relaxation factor is 1.85, while the stopping error criterion is 10E-04 for the maximum absolute difference between iterates at a mesh point.

Every case is run in two steps:

- mesh generation, by running the 'gen' module, with the option of creating or refining a mesh.
- running the parallel module 'fea', using the mesh generated by the 'gen' module.

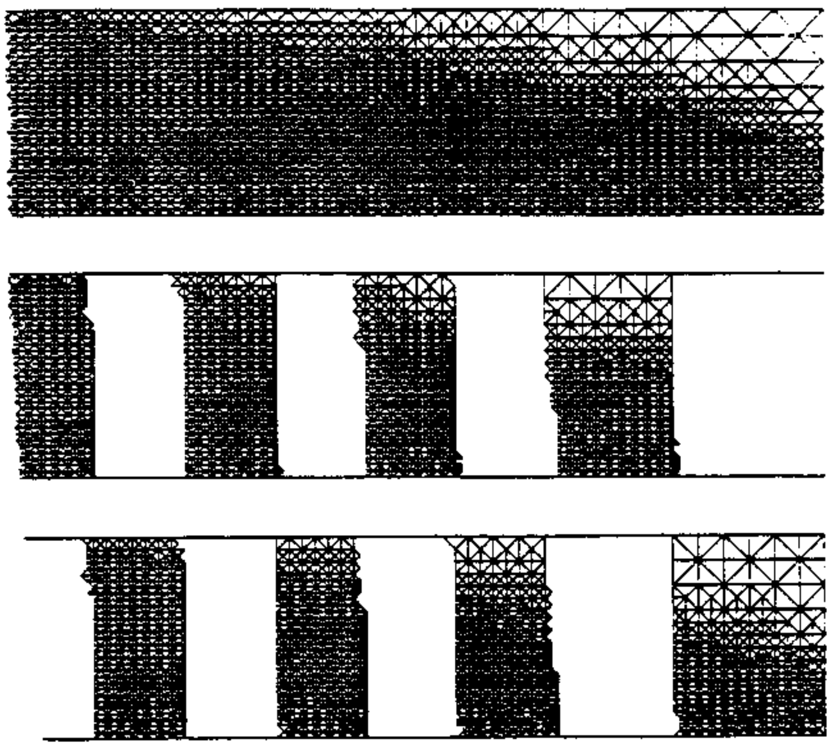


Figure 19: Domain subdividing for the third adaptive mesh refinement.

A typical output from the 'fea' module is shown in Table 6.

Table 6. FEA output

```
no. of proc.: 1
relaxation factor: 1.8500000000000001
number of nodes per processor: 1583
total ite #: 103
max sor time: 38.3094011545181 27
total error^2 is:  7.2697689141931852
total q^2 is:  5908.3187666771555
calculated percentage error is:  3.5077476619583 2907E-2
desired percentage error is:  5.0000000000000028E-2
effective index theta is:  1.8253362634148849
```

3.5 Performance considerations

The 'fea' module was run on the T3E, SP-2 and Meiko CS-2. The speed-up on the T3E is shown in Table 7 and Fig. 20 for the second adaptive mesh refinement solution (designated by Pass no. 3).

Table 7. Ideal and FEA speed-up

Nr. of PE's	Ideal Speed-up	FEA Speed-up
1	1.0	1.0
4	4.0	3.59
8	8.0	7.46
16	16.0	15.59
32	32.0	30.76

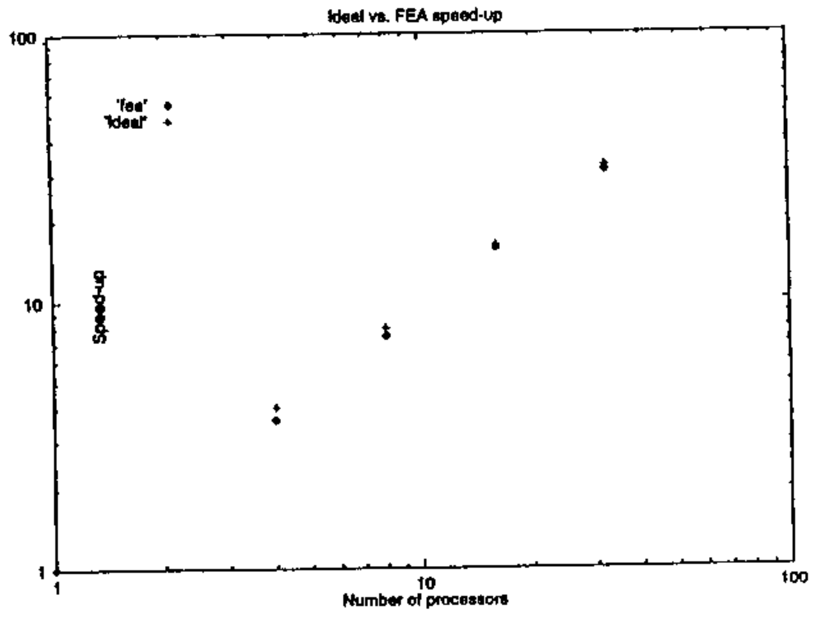


Figure 20: Ideal versus FEA speed-up.

3.6 Performance tools

The FEA speed-up is close to the ideal speed-up, which means that the module is running very efficiently on a parallel machine.

We used the MPP Apprentice [10] to analyze the load imbalance, excessive serialization, excessive communication, network contention, etc., for the 'fea' module.

Figure 21 shows the MPP Apprentice display after the second pass of the mesh refinement, with 164 finite element nodes per processor. Detailed data for this run, obtained from the Apprentice Report Window, is shown in Table 8.

Figure 22 shows the MPP Apprentice display after the seventh pass of the mesh refinement, with 1967 finite element nodes per processor. The corresponding data are shown in Table 9.

From Fig. 21 we can see a potential communication problem, i.e. excessive communication. The Parallel Work Time for the subroutine SOR, which should be the most time consuming routine, is (Table 8 - INSTRUMENTED SUBROUTINE SOR) 1.79E+06 microseconds from a total of 2.48E+07 microseconds. The number of mesh points (finite element nodes) per processor is 164.

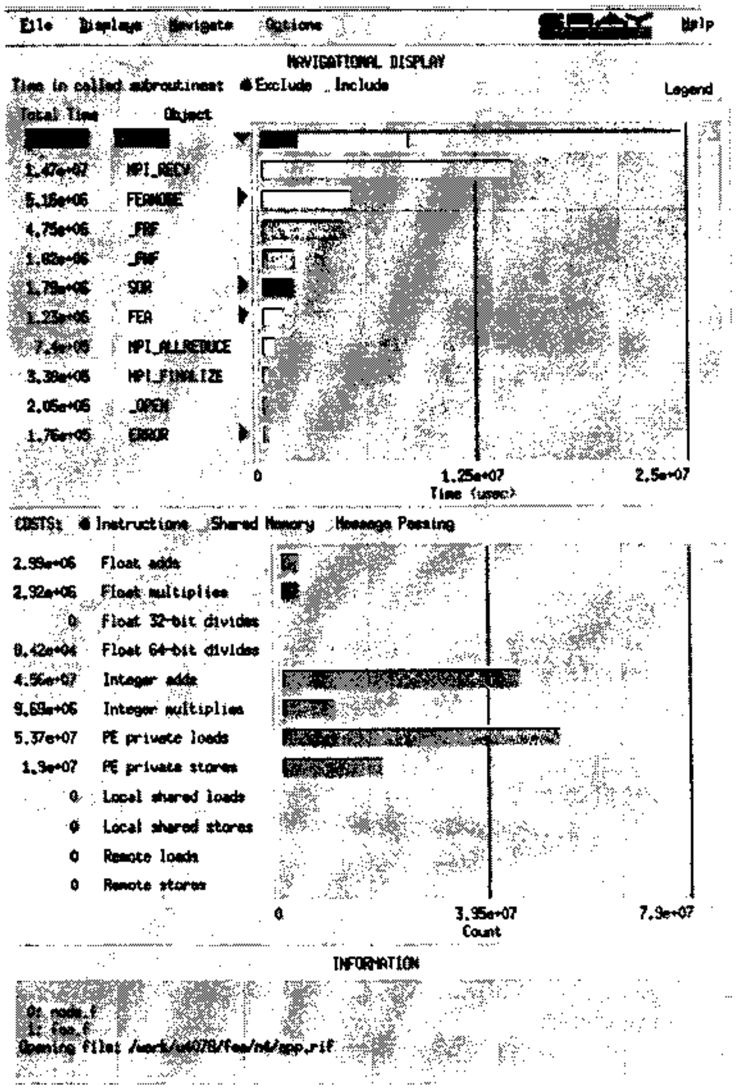
As we further refine the mesh, after the seventh pass, the number of mesh points per processor is 1967. The Parallel Work Time for the subroutine SOR, is (Table 9 - INSTRUMENTED SUBROUTINE SOR) 3.15E+08 from a total of 3.66E+08 microseconds. The result can be seen in Fig. 22.

A significant factor that affects the performance of a parallel application is the balance between communication and workload. At a low number of mesh points the overheads, latency, etc., exceeds the workload, creating an unbalance which affects the performance. At a number of 1967 mesh points per processor we obtain an very good performance. We actually see a good performance starting with 600 mesh points per processor.

The good communication load is mainly due to the fact that the most time consuming loop in the subroutine SOR does not have any message passing. The locations of the message passing routines can be easily visualized with the MPP Apprentices Source Code display.

A code object tree is an abridged representation of the source code that the MPP Aprentice tool uses to report statistics. It consists of a root node that represents the whole program, first level nodes, each of which represents one function or subroutine, and additional, lower-level nodes that recursively describe that function or subroutine. These nodes are also known as code objects.

If we select the 'Expose computing nodes in Navigational display' option in the Preference display of the MPP Apprentice, we obtain Figs. 23 and 24 showing the Source Code display and the Application Call Tree for the 'fea' module, respectively.



Time in called subroutines: ● Exclude Include

Total Time	Object
1.47e+07	MPI_RECV
5.16e+06	FEAMORE
4.75e+06	_FRF
1.82e+06	_FWF
1.79e+06	SOR
1.23e+06	FEA
7.4e+05	MPI_ALLREDUCE
3.39e+05	MPI_FINALIZE
2.05e+05	_OPEN
1.76e+05	ERROR

COSTS: ● Instructions Shared Memory Message Passing

2.99e+06	Float adds
2.92e+06	Float multiplies
0	Float 32-bit divides
8.42e+04	Float 64-bit divides
4.56e+07	Integer adds
9.69e+06	Integer multiplies
5.37e+07	PE private loads
1.3e+07	PE private stores
0	Local shared loads
0	Local shared stores
0	Remote loads
0	Remote stores

```
0: node.f
1: fea.f
Opening file: /work/u4078/fea/n4/app.rif
```

Figure 21: MPP Apprentice - Navigational: costs and information display for FEA with 164 nodes/processor.

Table 8. Pass number 2 report window

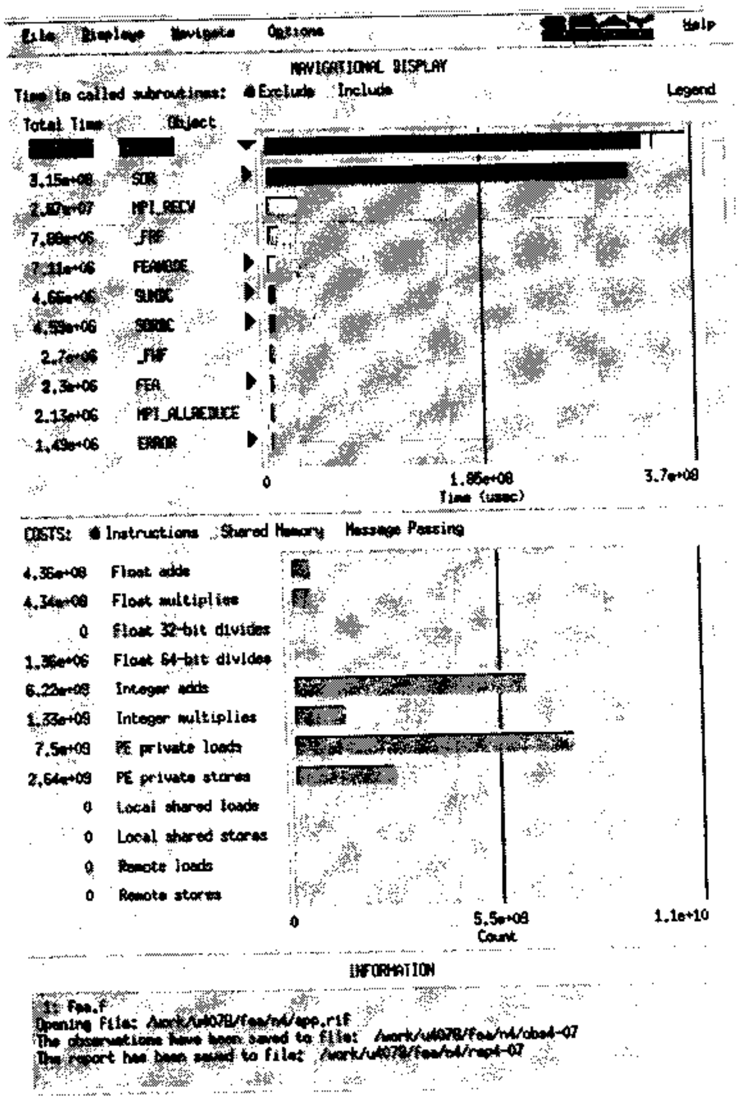
```
INSTRUMENTED SUBROUTINE FEANODE
    Exclusive Time               5.15942e+06 microseconds
    Inclusive Time               2.48204e+07 microseconds
    Time in Called Routines       1.9661e+07 microseconds
    Parallel Work Time               63.5203 microseconds

INSTRUMENTED SUBROUTINE OUTPUT
    Exclusive Time                   32939.6 microseconds
    Inclusive Time                   32939.6 microseconds
    Parallel Work Time               2996.26 microseconds

INSTRUMENTED SUBROUTINE SOR
    Exclusive Time               1.79072e+06 microseconds
    Inclusive Time               2.85131e+06 microseconds
    Time in Called Routines      1.06059e+06 microseconds
    Parallel Work Time           1.79072e+06 microseconds

INSTRUMENTED SUBROUTINE NEWPHI
    Exclusive Time                   1806.23 microseconds
    Inclusive Time                    334576 microseconds
    Time in Called Routines           332770 microseconds
    Parallel Work Time               1806.23 microseconds

INSTRUMENTED SUBROUTINE SUMBC
    Exclusive Time                   92819.5 microseconds
    Inclusive Time                    101936 microseconds
    Time in Called Routines          9116.97 microseconds
    Parallel Work Time               92819.5 microseconds

INSTRUMENTED SUBROUTINE SORBC
    Exclusive Time                    120846 microseconds
    Inclusive Time                    212941 microseconds
    Time in Called Routines          92095.2 microseconds
    Parallel Work Time                120846 microseconds
```

Table 8 (continued)

INSTRUMENTED SUBROUTINE TBCGRA	
Exclusive Time	258.754 microseconds
Inclusive Time	143779 microseconds
Time in Called Routines	143520 microseconds
Parallel Work Time	258.754 microseconds
INSTRUMENTED SUBROUTINE FEA	
Exclusive Time	1.22808e+06 microseconds
Inclusive Time	5.03778e+06 microseconds
Time in Called Routines	3.80969e+06 microseconds
Parallel Work Time	9427.59 microseconds
INSTRUMENTED SUBROUTINE STIFF	
Exclusive Time	15214.2 microseconds
Inclusive Time	27739.9 microseconds
Time in Called Routines	12525.7 microseconds
Parallel Work Time	15214.2 microseconds
INSTRUMENTED SUBROUTINE ERROR	
Exclusive Time	176379 microseconds
Inclusive Time	382480 microseconds
Time in Called Routines	206102 microseconds
Parallel Work Time	11671.1 microseconds
INSTRUMENTED SUBROUTINE GRAD	
Exclusive Time	12270.4 microseconds
Inclusive Time	156055 microseconds
Time in Called Routines	143784 microseconds
Parallel Work Time	12270.4 microseconds
INSTRUMENTED SUBROUTINE BAND	
Exclusive Time	4674.96 microseconds
Inclusive Time	4674.96 microseconds
Parallel Work Time	4674.96 microseconds

Table 8 (continued)

INSTRUMENTED SUBROUTINE ELSTMF		
Exclusive Time		12060.4 microseconds
Inclusive Time		12060.4 microseconds
Parallel Work Time		12060.4 microseconds
INSTRUMENTED SUBROUTINE MODIFY		
Exclusive Time		3581.14 microseconds
Inclusive Time		3581.14 microseconds
Parallel Work Time		616.278 microseconds
INSTRUMENTED SUBROUTINE INISTIFF		
Exclusive Time		1189.86 microseconds
Inclusive Time		39106 microseconds
Time in Called Routines		37916.1 microseconds
Parallel Work Time		1189.86 microseconds
UNINSTRUMENTED ROUTINES	TYPE	TIME
f$init	Misc	4004.33 microseconds
MPI_INIT	Misc	11059.7 microseconds
MPI_COMM_RANK	Misc	13.8686 microseconds
MPI_COMM_SIZE	Misc	12.4954 microseconds
_FWF	IO	1.82209e+06 microseconds
_fcd_copy	Misc	60.3106 microseconds
_fcd_blank	Misc	14.2386 microseconds
_OPEN	Misc	204981 microseconds
_FRF	IO	4.75354e+06 microseconds
MPI_RECV	Misc	1.46816e+07 microseconds
MPI_SEND	Misc	14507.7 microseconds
MPI_ALLREDUCE	Misc	739706 microseconds
_CLOSE	Misc	155853 microseconds
MPI_FINALIZE	Misc	337775 microseconds
_STOP	Misc	0 microseconds
$sldiv	Misc	16846.6 microseconds
MPI_WTIME	Misc	260.767 microseconds
_SQRT	Misc	556.838 microseconds



```
Time in called subroutines:  ● Exclude   Include

Total Time    Object
  3.15e+08    SOR
  2.07e+07    MPI_RECV
  7.88e+06    _FRF
  7.11e+06    FEANOSE
  4.66e+06    SUNDC
  4.53e+06    SBNDC
   2.7e+06    _FWF
   2.3e+06    FEA
  2.13e+06    MPI_ALLREDUCE
  1.49e+06    ERROR

COSTS:  ● Instructions   Shared Memory   Message Passing

4.36e+08   Float adds
4.34e+08   Float multiplies
       0   Float 32-bit divides
1.36e+06   Float 64-bit divides
6.22e+09   Integer adds
1.33e+09   Integer multiplies
  7.5e+09  PE private loads
2.64e+09   PE private stores
       0   Local shared loads
       0   Local shared stores
       0   Remote loads
       0   Remote stores
```

```
1: fea.f
Opening File: /work/u4078/fea/n4/app.rif
The observations have been saved to file:  /work/u4078/fea/n4/obs4-07
The report has been saved to file:  /work/u4078/fea/n4/rep4-07
```

Figure 22: MPP Apprentice - Navigational: costs and information display for FEA with 1967 nodes/processor.

Table 9. Pass number 7 report window

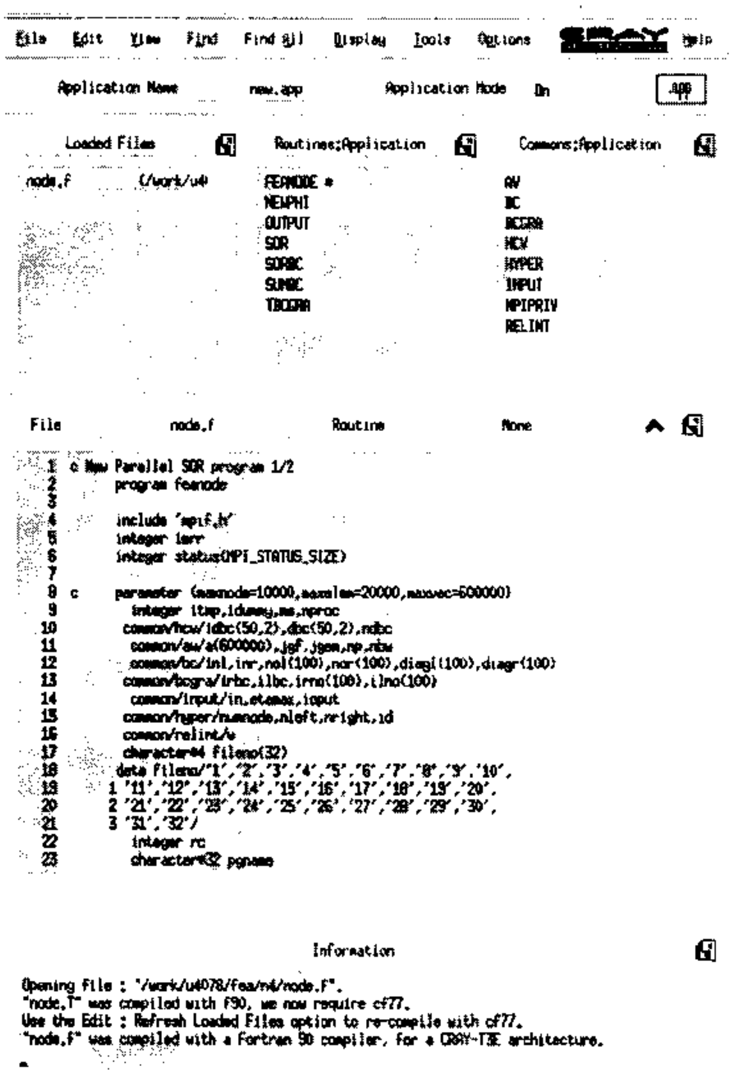
```
INSTRUMENTED SUBROUTINE FEANODE
    Exclusive Time              7.11197e+06 microseconds
    Inclusive Time              3.66807e+08 microseconds
    Time in Called Routines     3.59695e+08 microseconds
    Parallel Work Time              62.7437 microseconds

INSTRUMENTED SUBROUTINE OUTPUT
    Exclusive Time                   457429 microseconds
    Inclusive Time                   457429 microseconds
    Parallel Work Time              34092.8 microseconds

INSTRUMENTED SUBROUTINE SOR
    Exclusive Time              3.15226e+08 microseconds
    Inclusive Time              3.33253e+08 microseconds
    Time in Called Routines     1.80269e+07 microseconds
    Parallel Work Time          3.15226e+08 microseconds

INSTRUMENTED SUBROUTINE NEWPHI
    Exclusive Time                  10561.5 microseconds
    Inclusive Time               4.8587e+06 microseconds
    Time in Called Routines     4.84814e+06 microseconds
    Parallel Work Time              10561.5 microseconds

INSTRUMENTED SUBROUTINE SUMBC
    Exclusive Time              4.65843e+06 microseconds
    Inclusive Time              4.68698e+06 microseconds
    Time in Called Routines         28557.5 microseconds
    Parallel Work Time          4.65843e+06 microseconds

INSTRUMENTED SUBROUTINE SORBC
    Exclusive Time              4.59151e+06 microseconds
    Inclusive Time              6.20363e+06 microseconds
    Time in Called Routines     1.61212e+06 microseconds
    Parallel Work Time          4.59151e+06 microseconds
```

Table 9 (continued)

INSTRUMENTED SUBROUTINE TBCGRA		
Exclusive Time	804.606	microseconds
Inclusive Time	94006.6	microseconds
Time in Called Routines	93202	microseconds
Parallel Work Time	804.606	microseconds
INSTRUMENTED SUBROUTINE FEA		
Exclusive Time	2.30259e+06	microseconds
Inclusive Time	3.39094e+08	microseconds
Time in Called Routines	3.36792e+08	microseconds
Parallel Work Time	602731	microseconds
INSTRUMENTED SUBROUTINE STIFF		
Exclusive Time	212689	microseconds
Inclusive Time	375786	microseconds
Time in Called Routines	163097	microseconds
Parallel Work Time	212689	microseconds
INSTRUMENTED SUBROUTINE ERROR		
Exclusive Time	1.48581e+06	microseconds
Inclusive Time	1.97804e+06	microseconds
Time in Called Routines	492234	microseconds
Parallel Work Time	154690	microseconds
INSTRUMENTED SUBROUTINE GRAD		
Exclusive Time	172634	microseconds
Inclusive Time	266647	microseconds
Time in Called Routines	94012.8	microseconds
Parallel Work Time	172634	microseconds
INSTRUMENTED SUBROUTINE BAND		
Exclusive Time	62328	microseconds
Inclusive Time	62328	microseconds
Parallel Work Time	62328	microseconds

Table 9 (continued)

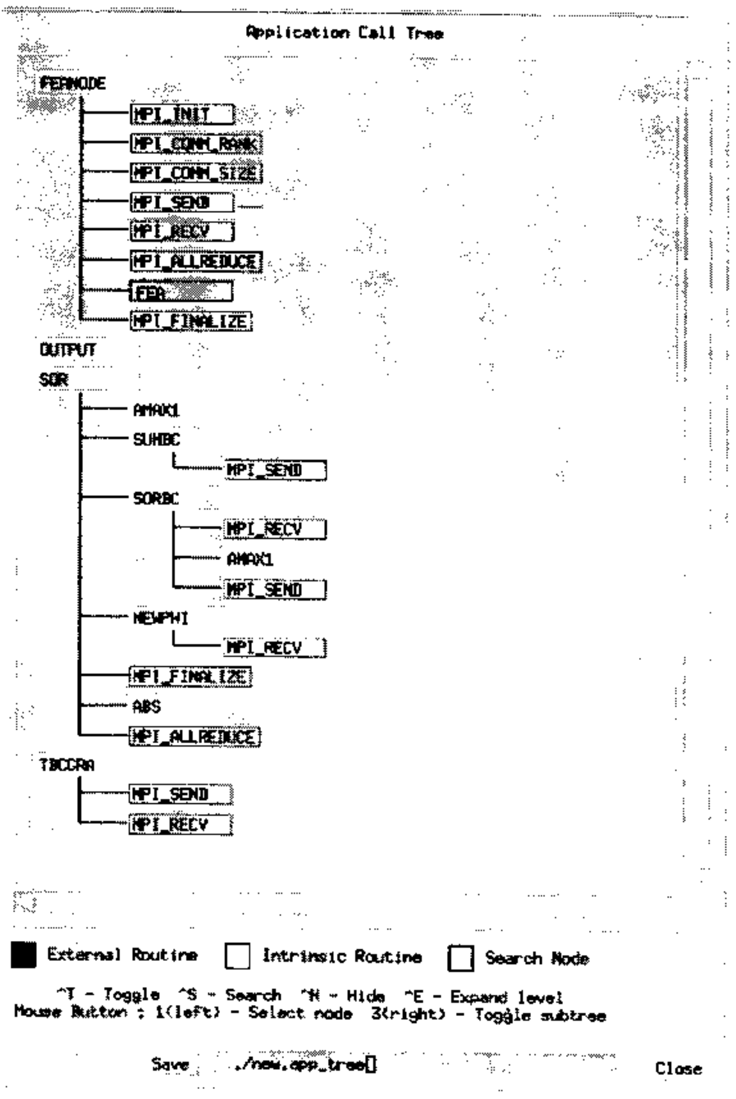
INSTRUMENTED SUBROUTINE ELSTMF		
Exclusive Time		158477 microseconds
Inclusive Time		158477 microseconds
Parallel Work Time		158477 microseconds
INSTRUMENTED SUBROUTINE MODIFY		
Exclusive Time		13414.9 microseconds
Inclusive Time		13414.9 microseconds
Parallel Work Time		2239.86 microseconds
INSTRUMENTED SUBROUTINE INISTIFF		
Exclusive Time		12022.6 microseconds
Inclusive Time		183218 microseconds
Time in Called Routines		171196 microseconds
Parallel Work Time		12022.6 microseconds
UNINSTRUMENTED ROUTINES	TYPE	TIME
f$init	Misc	3962.5 microseconds
MPI_INIT	Misc	10720.4 microseconds
MPI_COMM_RANK	Misc	14.4985 microseconds
MPI_COMM_SIZE	Misc	12.4754 microseconds
_FWF	IO	2.6991e+06 microseconds
_fcd_copy	Misc	63.3103 microseconds
_fcd_blank	Misc	14.5719 microseconds
_OPEN	Misc	415112 microseconds
_FRF	IO	7.8783e+06 microseconds
MPI_RECV	Misc	2.67223e+07 microseconds
MPI_SEND	Misc	33636.2 microseconds
MPI_ALLREDUCE	Misc	2.13027e+06 microseconds
_CLOSE	Misc	204651 microseconds
MPI_FINALIZE	Misc	334922 microseconds
_STOP	Misc	0 microseconds
$sldiv	Misc	461219 microseconds
MPI_WTIME	Misc	192.481 microseconds
_SQRT	Misc	6674.36 microseconds



```
c New Parallel SOR program 1/2
      program feanode

      include 'mpif.h'
      integer ierr
      integer status(MPI_STATUS_SIZE)

c     parameter (maxnode=10000,maxelem=20000,maxvec=600000)
        integer itmp,idummy,ms,nproc
       common/hcw/idbc(50,2),dbc(50,2),ndbc
        common/aw/a(600000),jgf,jgem,np,nbw
        common/bc/inl,inr,nol(100),nor(100),diagl(100),diagr(100)
       common/bcgra/irbc,ilbc,irno(100),ilno(100)
        common/input/in,etamax,iopt
       common/hyper/numnode,nleft,nright,id
       common/relint/w
       character*4 fileno(32)
      data fileno/'1','2','3','4','5','6','7','8','9','10',
     1 '11','12','13','14','15','16','17','18','19','20',
     2 '21','22','23','24','25','26','27','28','29','30',
     3 '31','32'/
        integer rc
        character*32 pgname
```

```
Opening file : '/work/u4078/fea/n4/node.f'.
"node.f" was compiled with f90, we now require cf77.
Use the Edit : Refresh Loaded Files option to re-compile with cf77.
"node.f" was compiled with a Fortran 90 compiler, for a CRAY-T3E architecture.
```

Figure 23: MPP Apprentice - Source Code display.



```
Application Call Tree

FEMNODE
   ├──MPI_INIT
   ├──MPI_COMM_RANK
   ├──MPI_COMM_SIZE
   ├──MPI_SEND
   ├──MPI_RECV
   ├──MPI_ALLREDUCE
   ├──FEA
   └──MPI_FINALIZE
OUTPUT
SOR
   ├──AMAX1
   ├──SUHBC
   │     └──MPI_SEND
   ├──SORBC
   │     ├──MPI_RECV
   │     ├──AMAX1
   │     └──MPI_SEND
   ├──NEWPHI
   │     └──MPI_RECV
   ├──MPI_FINALIZE
   ├──ABS
   └──MPI_ALLREDUCE
TBCCRA
   ├──MPI_SEND
   └──MPI_RECV
```

■ External Routine □ Intrinsic Routine □ Search Node

```
   ^T - Toggle  ^S - Search  ^H - Hide  ^E - Expand level
Mouse Button : 1(left) - Select node  3(right) - Toggle subtree
```

Figure 24: MPP Apprentice - Application Call Tree display.

4 Conclusions

The two modules, one based on a finite difference scheme with SOR and one on an adaptive mesh finite element approach, were ported to MPI and run successfully on the Cray T3E, IBM SP-2 and Meiko CS-2. In general, porting was straightforward, because MPI is for the most part a functional superset of existing message passing libraries.

These high performance computing systems provide a high potential for sustained performance. A significant factor that affects the performance of a parallel application is the balance between communication and workload. The challenge of the message passing model is in reducing message traffic over the interconnection network. To fully understand the performance behavior of such applications, analysis and visualization tools are needed. Two such tools, the Cray MPP Apprentice and the IBM VT, were used to analyze and visualize the performance of the two application modules. It was seen that optimization of the parallel codes can be carried out in an iterative process involving these tools to investigate performance issues.

Acknowledgements

The parallel algorithms used to solve the free and moving boundary problems were based upon work supported by the National Science Foundation Grant No. ECS-9006516.

This research was conducted using the resources of the Cornell Theory Center, San Diego Supercomputer Center and the Meiko Computing Facility within the Computer Science Department at UC Santa Barbara.

The Meiko CS-2 Computing Facility was acquired through NSF funding, with support from the College of Engineering and UCSB Office of Research, for research in parallel computing.

References

[1] Bruch, J.C., Jr., Papadopoulos, C.A., and Sloss, J.M., Parallel Computing Used in Solving Wet Chemical Etching Semiconductor Fabrication Problems, GAKUTO International Series, Mathematical Sciences and Applications, **1**, *Nonlinear Mathematical Problems in Industry*, pp. 281-292, 1993.

[2] Vuik, C. and Cuvelier, C., Numerical Solution of an Etching Problem, *J. Comput. Phys.*, **59**, pp. 247-263, 1985.

[3] Intel Corporation, *Paragon System Fortran Calls,* Reference Manual, 1995.

[4] Gropp W., Lusk E. and Skjellum A., *Using MPI. Portable Parallel Programming with the Message-Passing Interface,* The MIT Press, 1994.

[5] Snir M., Otto S., Huss-Lederman S., Walker D and Dongara J., *MPI: The Complete Reference,* The MIT Press, 1996.

[6] Meiko Scientific, *CS-2 Documentation Set,* Meiko Scientific, 1993.

[7] *Parallel Programming on the IBM SP,* Cornell Theory Center, http://www.tc.cornell.edu/UserDoc/SP/what.is.sp2.html.

[8] *Using the CRAY T3E at SDSC,* San Diego Supercomputer Center, http://www.sdsc.edu/Resources/frameresources.html.

[9] Saphir W., *Porting Parallel Applications from NX to MPI,* NASA Ames Research Center, 1994.

[10] *MPP Apprentice,* Cray Research Inc, IN-2511, 1994.

[11] *IBM AIX Parallel Environment,* IBM, SH26-7230-001.

[12] *Visualization Tool VT,* Cornell Theory Center, http://www.tc.cornell.edu/Edu/Tutor/Vt.trace.

[13] Baiocchi, C. and Capelo, A., *Variational and Quasivariational Inequalities,* John Wiley and Sons, New York, 1984.

[14] Kinderlehrer, D. and Stampacchia, G., *An Introduction to Variational Inequalities and Their Applications,* Academic Press, New York, 1980.

[15] Oden, J.T. and Kikuchi, N., Theory of Variational Inequalities with Applications to Problems of Flow Through Porous Media, *Int. J. of Engng. Sci.*, **18**, pp. 1173-1284, 1980.

[16] Bruch, Jr., J.C., A Survey of Free Boundary Value Problems in the Theory of Fluid Flow Through Porous Media: Variational Inequality Approach, *Advances in Water Resources*, Part I, **3**, pp. 65-80, Part II, **3**, pp. 115-124, 1980.

[17] Crank, J., *Free and Moving Boundary Problems*, Clarendon Press, Oxford, England, 1984.

[18] Zienkiewicz, O.C. and Zhu, J.Z., A Simple Error Estimator and Adaptive Procedure for Practical Engineering Analysis, *Int. J. Num. Meth. in Engr.*, **24**, pp. 337-357, 1987.

[19] Burkley, V.J. and Bruch, J.C., Jr., Adaptive Error Analysis in Seepage Problems, *Int. J. Num. Meth. Engng.*, **31**, pp. 1333-1356, 1991.

[20] Burkley, V.J., Bruch, J.C., Jr and Zienkiewicz, O.C., Adaptive Meshes Used in Solving a Free Surface Seepage Problem, *The Mathematics of Finite Elements and Applications*, **VII**, edited by J.R. Whiteman, Academic Press, pp. 101-110, 1991.

[21] Rivara, M.C., Algorithms for Refining Triangular Grids Suitable for Adaptive and Multigrid Techniques, *Int. J. Num. Meth. Engng.*, **20**, pp. 745-756, 1984.

[22] Rivara, M.C., A Grid Generator Based on 4-Triangles Conforming Mesh-Refinement Algorithms, *Int. J. Num. Meth. Engng.*, **24**, pp. 1343-1354, 1987.

[23] Wang, K.P. and Bruch, J.C., Jr., A SOR Iterative Algorithm for the Finite Difference and the Finite Element Methods that is Efficient and Parallelizable, *Advances in Engineering Software*, **21**(1), pp. 37-48, 1994.

[24] Wang, K.P. and Bruch, J.C., Jr., An Efficient Iterative Parallel Finite Element Computational Method, *The Mathematics of Finite Elements and Applications*, edited by J.R. Whiteman, John Wiley and Sons, Inc., Chapter 12, pp. 179-188, 1994.