A Guide to the Parallelization of a FEM Solver using the MPI Library

C. T. Wolfe
Lexmark International, Inc.
Digital Office Products
Lexington, KY 40550

S. D. Gedney
University of Kentucky
Department of Electrical Engineering
Lexington, KY 40506-0046

Abstract. The MPI (Message Passing Interface) library is a public domain communication library used in parallel computing systems. This paper shows all necessary functions needed to implement a message-passing algorithm with the MPI library. Through an example based on a frequency-domain finite-element solution of the vector wave equation, the basic functionality and application of the MPI library from design to implementation is presented.

I. Introduction

Many Computational Electromagnetic (CEM) schemes involve the formulation and solution of a large linear system of equations. One way to efficiently handle large linear systems is to use a multi-processor computer. Thus, by distributing the data and computational effort, an effective speedup proportional to the number of processors should be realized. The difficulty of using a multi-processor system is that it can require an additional layer of programming beyond that required by a traditional sequential machine. This is generally minimal when using a shared memory paradigm on a coarse-grained parallel system. On such a system, the code is typically parallelized through the use of compiler switches or directives. Unfortunately, respectable speedups are realized only with a few processors. Furthermore, compilers are generally limited to parallelizing loops, which can have limited effectiveness. Larger parallel systems, such as PC clusters, typically use a distributed memory paradigm. This provides the user with explicit control over the distribution of the data and computation among the processors. However, it also requires the programmer to design an effective parallel algorithm. Furthermore, he/she must have the tools to communicate between processors. The latter task is usually accomplished using a predefined message-passing library. A popular public domain library that is available is the Message Passing Interface (MPI) [1]. MPI provides a convenient inter-processor communication abstraction that relieves the programmer from having to worry about the underlying hardware.

This paper demonstrates the use of MPI for the iterative solution of the large sparse matrices arising from a Finite Element Method (FEM) solution of the vector wave equation [2][3][4]. The parallel algorithm is based on a Single Program Multiple Data (SPMD) paradigm that employs a domain decomposition of the mesh. The use of the MPI library for communication directives is detailed.

The ultimate goal of this paper is to familiarize the reader with the use of the MPI library. While the example chosen is the FEM solution, it is hoped that the introduction to the MPI library is complete enough to allow others to easily port more sophisticated codes to distributed memory multiprocessor systems.

II. The Finite Element Implementation

A vector electric field $\vec{E}$ produced by a source with time dependence e^{jwt} is distributed in a volume Ω bound by a surface $\partial\Omega$. The electric field satisfies the vector wave equation

$$\nabla \times m^{-1} \nabla \times \vec{E} - w^2 e \vec{E} = 0 \tag{1}$$

within Ω and the appropriate boundary conditions on $\partial\Omega$. The inner product of (1) with a test vector field $\vec{T}$ is performed. Utilizing Green's first identity results in the weak form equation [2]:

$$\iiint_{\Omega} \left[m^{-1}(\nabla \times \vec{T}) \cdot (\nabla \times \vec{E}) - w^2 e \, \vec{E} \cdot \vec{T} \right] d\Omega - \oiint_{\partial\Omega} m^{-1} \left[\vec{T} \times \nabla \times \vec{E} \right] \cdot n \, dS = 0 \tag{2}$$

A finite element solution for $\vec{E}$ is performed by discretizing the volume into element domains, and expanding the vector fields $\vec{E}$ and $\vec{T}$ with vector edge elements [2]. The first variation of (2) is then evaluated at the stationary point, leading to a sparse linear system of equations of the form [2][3][4]:

$$K\mathbf{e} = \mathbf{f} \tag{3}$$

where $\mathbf{e}$ is the vector of unknown coefficients weighting the trial functions in the region Ω, K is the stiffness matrix, and $\mathbf{f}$ is the forcing vector.

The stiffness matrix K is symmetric and highly sparse. The set of equations can be solved directly with banded storage [5] using LU decomposition or iteratively with compressed storage [6][7]. An advantage to solving the system of equations iteratively is that these methods use much less memory than the direct methods since a dense matrix factorization is not stored. Here the BiConjugate Gradient method (BCG) is used. The solution of (3) using the BCG algorithm in psuedo-code is provided in Fig. 1.

$k = 0$
$\mathbf{r}_0 = \mathbf{f} - K\mathbf{e}_0$
$d_{denom} = \sqrt{< \mathbf{r}_0, \mathbf{r}_0 >}$
$b_0 = < \mathbf{r}_0^*, \mathbf{r}_0 >$
while $(d > e)$
  $k = k + 1$
  if $k == 1$
    $\mathbf{p}_1 = \mathbf{r}_0$
  else
    $b_{k-1} = < \mathbf{r}_{k-1}^*, \mathbf{r}_{k-1} >$
    $g_k = b_{k-1} / b_{k-2}$
    $\mathbf{p}_k = \mathbf{r}_{k-1} + g_k \mathbf{p}_{k-1}$
  endif
  $\mathbf{v}_k = K\mathbf{p}_k$
  $a_k = b_{k-1} / < \mathbf{p}_k^*, \mathbf{v}_k >$
  $\mathbf{e}_k = \mathbf{e}_{k-1} + a_k \mathbf{p}_k$
  $\mathbf{r}_k = \mathbf{r}_{k-1} - a_k \mathbf{v}_k$
  $d_{num} = \sqrt{< \mathbf{r}_k, \mathbf{r}_k >}$
  $d = d_{num} / d_{denom}$
end while

Fig. 1 The BCG algorithm for a symmetric complex linear system [2]

In Fig. 1, the lower case letters denote a scalar quantity, the value e is chosen for the desired accuracy of the solution, and the vector inner product $< \mathbf{x}, \mathbf{y} >$ is expressed as:

$$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^H \cdot \mathbf{y} = \sum_i x_i^* y_i \tag{4}$$

Analyzing some subtleties in this algorithm, temporary storage is needed for the vectors $\mathbf{r}_k$, $\mathbf{p}_k$, and $\mathbf{v}_k$ as well as scalar quantities d, d_{num}, d_{denom}, a, b_{k-1}, b_{k-2}, and g. When converting this serial algorithm to a parallel one, it is understood that the vectors and matrix are stored in a distributed manner. Thus the inner products and the matrix-vector product must be implemented carefully. This is addressed in the next section.

III. Converting to a Parallel Algorithm

The objective of computing in a parallel fashion is to reduce the amount of execution time by distributing the computational effort. Thus, both the data and the floating-point operations must be partitioned. For partial differential equation (PDE) based solutions methods, this is typically done by a spatial decomposition of the mesh. Some examples of spatial decomposition methods include the recursive inertia partitioning (RIP) algorithm [8], spectral bisection methods [9][10], and the METIS algorithm [11][12]. The most effective algorithm is dependent upon the problem. For general FEM solutions, we have found the most robust technique to be the METIS algorithm. Software employing the METIS algorithm for mesh partitioning is currently free, and can be found at the web site http://www-users.cs.umn.edu/~karypis/metis/metis/.

With the mesh partitioned among the processors, each processor will store the portion of the matrix K and the entries of the unknown coefficient vector $\mathbf{e}$ associated with the edges of the mesh assigned to the processor. The computation of the stiffness matrix K is done completely in parallel and requires no interaction between processors. For the BCG algorithm, the residual and direction vectors $\mathbf{r}$ and $\mathbf{p}$ and the temporary vector $\mathbf{v}$ are distributed in an identical manner as $\mathbf{e}$. The scalar quantities d, a, and g are stored globally (that is redundantly stored on all processors).

Observing the BCG algorithm in Fig. 1, there are two operations that require inter-processor communication: *a*) a matrix-vector product resulting in a distributed vector v, and *b*) inner products that result in a global scalar variable. Assume that each processor contains a block of K, designated as K_i, and a portion of the vector, designated as $\mathbf{e}_i$. Note that entries of $\mathbf{e}_i$ shared by multiple processors are stored redundantly. The matrix vector product is then performed by the superposition:

$$\mathbf{v} = K\mathbf{e} = \sum_{i=1}^{P} K_i \mathbf{e}_i \tag{5}$$

Each domain computes its local matrix-vector product $K_i\mathbf{e}_i$. For unknowns that are shared, each processor communicates their partial products with all other processors sharing this unknown. Each processor then sums the partial products. At this point each processor contains a copy of $\mathbf{v}$ for each variable that is held common with adjacent domains. It is noted that the communication used for this operation is local, requiring communication only with adjacent processors with which it shares data.

The inner products of two distributed vectors are also computed via superposition. However, one must be careful since shared values of the vectors are store redundantly. This can be compensated for in one of two ways. The first is if the term $x_i y_i$ is shared by m processors, then these terms can be scaled by $1/m$ on all m processors. An alternate approach would be to assign only one of the m processors to be responsible for the term $x_i y_i$. In either case, each processor performs the local inner product, which results in a scalar value. This is then summed globally by aggregating all the results to a single processor (the root processor). The final result is then broadcast back to all processors, requiring a global communication.

IV. Introduction to the MPI Library

The MPI library enables data to be easily shared between processors within a given system. It consists of several abstractions that can be used to perform several types of functions. However, we will concentrate only on a subset of basic functions that will enable the reader to start using the MPI library quickly. The first step is to include the MPI library header file. This may be system dependent, but in general it takes the form:

include "mpif.h"

The next two routines initialize and terminate the MPI environment:

MPI_INIT(IERROR)
INTEGER IERROR
MPI_FINALIZE(IERROR)
INTEGER IERROR

No MPI routines should be executed before *MPI_INIT* or after *MPI_FINALIZE*.

Once the environment has been initialized, each processor needs to determine its node number. This can be done with the *MPI_COMM_RANK* routine that returns the node number within the *RANK* variable.

MPI_COMM_RANK(COMM,RANK,IERROR)
INTEGER COMM, RANK,IERROR

Within this routine, the variable *COMM* represents a group of processes and their communication context. One of the default communicators is *MPI_COMM_WORLD* that consists of all processes when an application begins. Once the environment is initialized and node numbers have been determined, it is possible to communicate with other nodes.

There are two types of "information passing" routines: blocking and non-blocking. A blocking (or synchronous) send function returns when the employed transmit buffers are ready for reuse. A non-blocking (or asynchronous) function returns immediately. When the non-blocking function is used, the associated transmit request should be tested at a later time to determine if the information transfer has completed.

Similarly there are blocking and non-blocking "information acquisition" calls. When a blocking receive call returns, the associated receive buffer is guaranteed to contain an informational message. For the non-blocking receive call, the function will return immediately, but the associated request must be tested at a later time to determine if the expected informational message has arrived.

The declarations for the asynchronous send and the synchronous receive functions are shown:

MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR

Although these functions use a buffer consisting of *COUNT* elements of type *DATATYPE*. It is often simpler to use a datatype that is defined by a single byte and then explicitly count the total number of bytes in the buffer. This method is more compatible with other message passing libraries.

The MPI library also contains functions for determining global sums that are needed for the inner product terms in our BCG solution. Since each node contains different parts of the total sum, a method for summing these values needs to be used. This is done in a two step process. We first sum the terms using the *MPI_REDUCE* call:

MPI_REDUCE(SENDBUF, RECVBUF, DATATYPE, OP, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER DATATYPE, OP, ROOT, COMM, IERROR

For our needs, the *OP* (or operation) to be performed is a sum (*MPI_SUM*). This function places the sum in the output buffer of the process of with a rank of *root*. This sum now needs to be distributed to all

other process nodes. This can be accomplished with a broadcast command from the *root* node to all other nodes.

MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
<type> BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR

The final function prototype that we will discuss is synchronization. Using the SPMD (Single Program Multiple Data) model of computation, issued nodes will be executing the same code, but with different data sets. However this does not guarantee that all process nodes are executing the same exact line of code at the same time. It may therefore be necessary to insure that all nodes are synchronized and waiting at the same location in the code. This is accomplished by using the *MPI_BARRIER* function.

MPI_BARRIER(COMM,IERROR)
INTEGER COMM, IERROR

This function blocks until all processes have called it. Only when all processes have called this function, will execution continue.

V. Implementation with the MPI Library

The previous section detailed all the functions necessary to implement a parallel version of the BCG solution. We will now show how the MPI library functions are used.

The first task is to initialize the environment and get the local node number (expressed in FORTRAN):

```
call MPI_INIT(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD,mynode,ierr)
```

When computing inner products for the BCG algorithm, first the local inner products are computed. While doing this the appropriate scaling must be used for those edges shared across domains. Then, to compute the global sum of the inner product at node 0, one would use the call statement:

```
call MPI_REDUCE(InnerProduct,InnerProduct,1,MPI_DOUBLE_PRECISION,
                MPI_SUM,0,MPI_COMM_WORLD,ierr)
```

Then, node 0 will broadcast the result to all processors:

```
if ( mynode .eq. 0) then call MPI_Bcast(InnerProduct,1,MPI_DOUBLE_PRECISION,
                                        0,MPI_COMM_WORLD,ierr)
```

For the matrix-vector product, each processor first computes the product of the local sparse block of K with the local vector. For shared edges, the results must be superimposed with those computed by local processors sharing the same edge. A list of processors with which *mynode* shares edges is stored in array *AddressofInterface*. Which edges are shared with each processor must also be determined *a priori.* A call to a user-defined subroutine then fills a buffer (which is a vector) with the data to be sent to processor j. This vector is then communicated with the MPI library as:

```
do  j = 1,NumberOfInterfaces
    call fill_buffer(BUF,j)
    call MPI_ISSEND(BUF,NumberOfBytes(j),MPI_CHARACTER,
                    AddressofInterface(j),Tag,MPI_COMM_WORLD,ireq,ierr)
```

```
    call MPI_REQUEST_FREE(ireq,ierr)
enddo
```

Once the data is transmitted to all neighboring processors, the processor must receive its data. Using the MPI call below, the data from the *j*-th processor is received into the buffer BUFR, and a user-defined subroutine *sup_res* then superimposes the result of the partial matrix-vector product:

```
do  j = 1,NumberOfInterfaces
    call MPI_RECV(BUFR,NumberOfBytes, MPI_ANY_SOURCE,
                  Tag,MPI_COMM_WORLD,status,ierr)
    call sup_res(BUFR,j)
enddo
```

Finally, after the algorithm has completed we need to terminate the environment with

```
call MPI_FINALIZE(ierr)
```

VI. Microstrip Example

To demonstrate the principles discussed in the previous sections, an example problem of a printed microstrip circuit is studied. To this end, a simple microstrip circuit printed on a 2.1 mm thick substrate with $e_r = 3.2$ excited by a 4.0 GHz source was modeled using 86,943 first-order tetrahedral edge elements or Whitney elements [2][4]. This gave an average edge length of less than a 20^{th} of a wavelength in the dielectric. A PML (Perfectly Matched Layer) [13] based on the anisotropic PML formulation [14] was chosen to terminate the computational domain. The tetrahedral mesh was decomposed using METIS for parallel simulations.

To study the efficiency of the parallel BCG algorithm, the FEM solution previously outlined was implemented on a 32-processor subcomplex of an HP SPP2200 [15] computing system. The problem was solved using processor configurations in powers of two from one to thirty-two.

Table I shows the element time (time to formulate equation (3)), solution time (time to solve equation (3) using the BCG method) and the total time (element time plus solution time) for each processor configuration. The total execution time continually decreases as the number of processors increases.

A metric used to evaluate parallel algorithms is *speedup*. Speedup is defined to be the execution time need for one processor divided by the execution time need for *n* processors. Mathematically this concept is defined as:

$$Speedup_{nprocessors} = \frac{Execution\ \ Time\ \ for\ \ 1\ \ processor}{Execution\ \ Time\ \ for\ \ n\ \ processors} \tag{6}$$

TABLE I
Comparison of Execution Times on an HP 2200 using the MPI library.

Domains	Unknowns	Element Time	Solution Time	Total Time
1	86943	424	6405	6829
2	86943	220	3239	3460
4	86943	123	1590	1714
8	86943	69	771	841
16	86943	36	515	531
32	86943	19	345	364

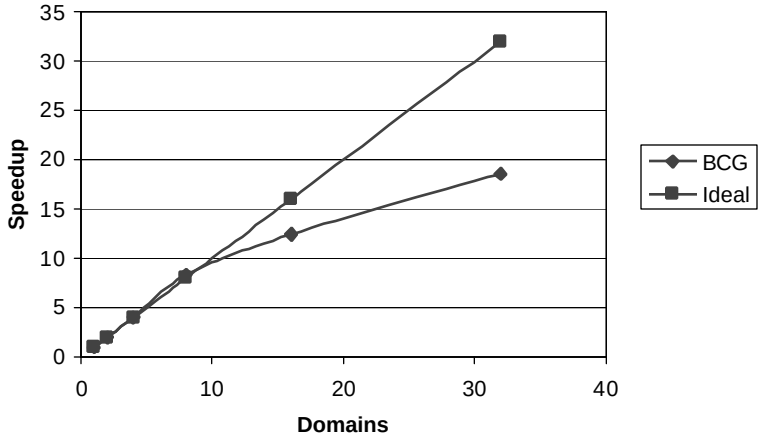

Fig 2. Speedup for the BCG algorithm versus the number of processors as compared to an Ideal (or linear) speedup on an HP SPP2200 using the MPI library.

Ideal speedup is linear with the number of processors *n*. This is rarely attainable since this would insinuate that there are no inefficiencies in the algorithm. Fig. 2 shows the speedup for the parallel BCG solution. For a small number of processors, close to linear speedup is realized. As more processors are added inefficiencies start to accumulate. The inefficiencies are mainly due to an imbalance of computation and communication time across processors. An alternative metric to speedup is efficiency, which is defined as the speedup for n processors divided by n .

$$Efficiency_{nprocessors} = \frac{Speedup \quad for \quad n \quad processors}{n} \tag{7}$$

The efficiency of the parallel BCG solution is illustrated in Fig. 3. Again, for a small number of processors, the algorithm maintains nearly perfect efficiency of 100 %. As the number of processors increases, the efficiency begins to decrease. With few processing elements the geometry of the domains are very similar. As the number of processors increase above eight, the number of unknowns and neighbors per domain start to become unbalanced due to the geometry of the problem. Even though efficiency drops off with an increased number of processors, overall execution time (the most important metric) continues to decrease as Fig. 4 indicates (using a logarithmic scale on the y-axis). The reference line is the execution time of one processor while the other data series shows the execution time for the BCG algorithm for different processor configurations. The gap between the reference line and the data set continually widens as processors are added to the system.

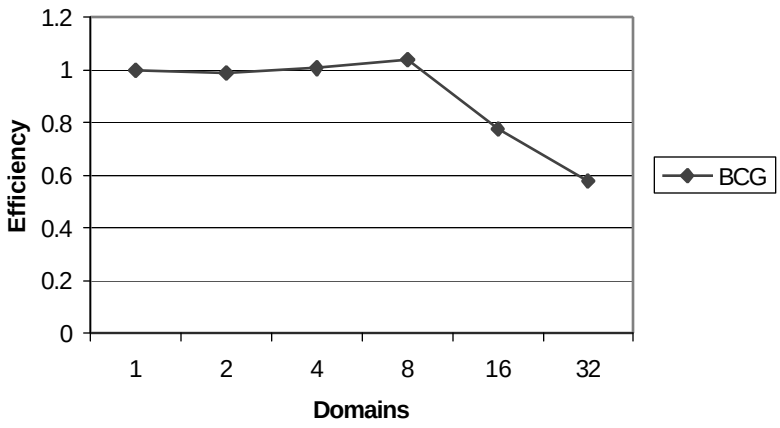

Fig. 3. Efficiency of the BCG algorithm versus the number of processors on an HP SPP2200.

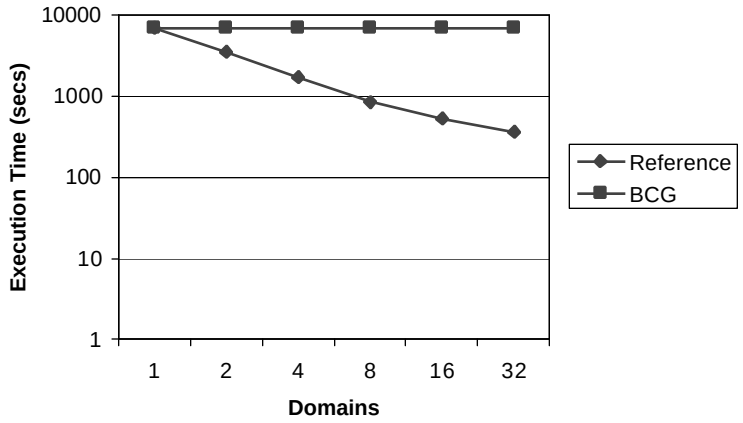

Fig. 4. Total execution time for the BCG algorithm versus the number of processors as compared to the execution time of 1 processor (Reference data set) on an HP SPP2200.

VII. Conclusions

This paper detailed the transition from algorithm to implementation of a parallel solver using the MPI communication library. We started with a FEM problem description, detailed the solving algorithm, discussed the communication aspects of the algorithm, showed the implementation of the communication calls using the MPI library, and presented the results of a microstrip problem on a parallel computing system. We were able to show that for small number of processors the algorithm was very efficient. This also implies that the communication overhead was very low. For large number of processors, the algorithm was less efficient. This is due to the fact that as the local problem size becomes smaller, small imbalances in the workload and interprocessor communication become more significant in the local compute time. Generally, the most efficient implementation is to scale the problem size with the number of processors.

Acknowledgements

The authors would like to thank Tim Tillotson of Lexmark International, Inc. for his comments regarding the development of this manuscript and the University of Kentucky Computing Center for system support of the HP SPP2200.

References

[1] Message Passing Interface Forum. MPI: A Message-Passing Interface standard (version 1.1). Technical report, 1995. http://www.mpi-forum.org.

[2] J. Jin, The Finite Element Method in Electromagnetics, John Wiley & Sons, Inc., 1993.

[3] P.P. Silvester and R.L. Ferrari, Finite elements for electrical engineers, Cambridge University Press, 3rd Edition, 1996.

[4] J.L.Volakis, A. Chatterjee, L.C. Kempel, Finite Element Method for Electromagnetics, Institute of Electrical and Electronics Engineers, Inc., 1998.

[5] I.S. Duff, A.M. Erisman, and J.K. Reid, Direct Methods for Sparse Matrices, Oxford University Press, NY, 1986.

[6] W. Hackbusch, Iterative Solution of Large Sparse Systems of Equations, Springer Verlag, NY, 1994.

[7] Y. Saad, Iterative Methods for Sparse Linear Systems, PWS Publishing Company, 1996.

[8] B. Nour-Omid, A. Raefsky, and G. Lyzenga, "Solving finite element equations on concurrent computers," in *Symposium on Parallel Computation and their Impact on Mechanics,* Boston, MA, 1987.

[9] S.-H. Hsieh, G. Paulino, and J. Abel, "Recursive spectral algorithms for automatic domain partitioning in parallel finite element analysis," *Comput. Method. Appl. Mech. Eng.*, vol. 121, pp. 137-162, 1995.

[10] R. Van Driessche and D. Roose, "An improved spectral bisection algorithm and its application to dynamic load balancing," *Parallel Computing*, vol. 32, pp. 29-48, Jan. 1995.

[11] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," Department of Computer Science, University of Minnesota, Tech. Rep. TR 95-035, 1995.

[12] G. Karypis and V. Kumar, METIS 2.0: Unstructured graph parititioning and sparse matrix ordering system, Technical Report 97-061, Department of Computer Science, University of Minnesota, 1997.

[13] J.P. Berenger, "A perfectly matcher layer for the absorption of electromagnetic waves," J. Computational Phys., vol. 114, no. 2, pp. 185-200, Oct. 1994.

[14] Z.S. Sacks, D.M. Kingsland, R. Lee and J.F. Lee, "A perfectly matched anistropic absorber for use as an absorbing boundary condition," IEEE Transactions on Antennas and Propagation, vol. 43, pp. 1460-1463, December 1995.

[15] HP MIP User's Guide, Third Edition, Hewlett Packard, June 1998.