MetaMP: A Higher Level Abstraction for Message-Passing Programming

Steve W. Otto

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 91-003

March, 1991

MetaMP: A Higher Level Abstraction for Message-Passing Programming

Steve W. Otto
Dept of Computer Science and Engineering
Oregon Graduate Institute of Science and Technology
19600 NW von Neumann Dr, Beaverton, OR, 97006-1999 USA
otto@cse.ogi.edu
503-690-1486

January 15, 1991

Abstract

The potential performance of distributed-memory parallel computers is very high, but their programming has proven to be difficult. The only successful approach so far has been to program them directly in the message-passing system of the machine. To a large extent this forms the "assembly language" of the computer.

Higher level programming abstractions are available, such as versions of parallel Fortran, but it has proven difficult to compile these to efficient distributed-memory code. Here, we propose a slightly more modest approach, whereby useful abstractions are supported by a compiler and run-time system (MetaMP), but these constructs are within a message-passing framework. The user still writes a message-passing program, but the MetaMP compiler understands distributed data structures and is therefore able to help in powerful ways.

A preliminary version of MetaMP has been written which supports simple multi-dimensional arrays. Extensions to more complex data structures (e.g., unstructured meshes, dynamically changing arrays) are planned. MetaMP programs have proven to be succinct and more understandable than their "assembly language" counterparts. The performance of MetaMP programs is close to that obtainable by manual programming.

Currently, MetaMP compiles down to Express, a commercial message-passing system developed at Caltech and available on many parallel computers.

Objectives and Relation to Other Work

Parallel computers such as the Intel Touchstone, the Ncube II, and the Meiko Computing Surface form a class of MIMD machines which can be termed "message-passing." The processors inside these systems are, to a first approximation, conventional microprocessors with a "large" amount of memory (.5 to 16 Mbytes in 1990) and an interface to a hardware message passing system. Though the programming of these machines remains problematical, they have been successfully used in many specific cases. The potential performance is very high, since this architecture can be easily scaled to large numbers of processors.

To a great extent, these machines have been manually programmed, using the message-passing calls provided by the system directly. A fundamental property of message-passing machines is that message passing times are one to three orders of magnitude slower than fundamental floating point operation times. This necessitates a style of programming in which communications are carefully scheduled so that: the correctness of the program is preserved; the communications occur infrequently; and many data items are transferred per message. Message-passing programming has often been compared to assembly language programming. Intricate details of distributed data structures must be managed by the programmer.

The question naturally arises: "Can message-passing programming be abstracted to a more understandable form without losing much of the performance of custom programming?" Many parallel languages and compilers have been proposed and implemented on MIMD computers [1–16]. These systems often allow the programmer an extremely clean and simple model of the parallel computation. Typically, all elements of an array are accessible by any processor (shared memory), and synchronization is provided automatically by the compiler (e.g., the programmer just writes `doall`). Unfortunately, it appears to be difficult to compile from a parallel language such as this to a message-passing computer, with the restriction that the resultant code be efficient.

The research proposed here concerns a set of language extensions and a compiler called MetaMP. In contrast to the systems mentioned previously, MetaMP has a somewhat less ambitious goal. MetaMP does not attempt to completely hide the message passing nature of the underlying hardware. This makes the compiler implementable while preserving the efficiency of the resultant code. The programmer is still given a message-passing view of the hardware, but it is an abstract, minimalist one. The user still writes a mes-

sage passing program, but the MetaMP compiler understands distributed data structures and is therefore able to help in powerful ways. Programs written in this language have proven to be more compact and understandable than those written directly in the underlying message passing system.

Scientific computing focuses on programs which construct and manipulate large, multi-dimensional arrays. Our first version of MetaMP provides support for these types of programs. The MetaMP compiler introduces auxiliary data structures which describe the shapes, sizes, offsets, etc., of distributed multi-dimensional arrays. Different arrays can be distributed (or "decomposed") in different ways and MetaMP keeps track of each decomposition. Abstract loop constructs similar to `doall` are available and release the programmer from having to remember the decomposition details of each array. Communications are more easily expressed since the compiler understands the shapes of arrays and *spread* and *reduction* operations can be succinctly written. Array sections of different sizes from one processor to the next are completely supported by MetaMP. This means that the problem of "odd sizes" (array dimensions not exactly conforming to the machine size) can be removed. As we will demonstrate in our examples below, programs which handle *any* size problem on *any* size machine can be written, yet they are still succinct.

Locality often plays a large role in scientific computing. In solving a set of partial differential equations for example, arrays (or meshes) representing spatial locations are distributed across the parallel computer. The locality of the differential operator reflects itself in the fact that the required communications are of the "nearest neighbor" type. Such algorithms require array elements from a narrow boundary strip (or face in three dimensions) in the array sections of neighboring processors. MetaMP provides full support for this. Guard strips, that is, extra array elements which map to neighboring array sections, can be specified within MetaMP. It turns out that there is an elegant way to do this which makes the extra, guard elements transparent to the programmer. This will be discussed later in the context of a two dimensional elliptical PDE solver.

The MetaMP language consists of two components:

- normal, sequential C (or Fortran) containing `for` loops that run over the multi-dimensional arrays,
- MetaMP directives which modify the meaning of the sequential `for` loops to their parallel, distributed-memory counterparts.

The directives always appear between "%" delimiters, that is, they look like this: % *directive* %. Compile time checking is done to ensure that the directives to distribute `for` loops make sense. Loop indices have associated ranges and these are compared with the allowable ranges of the distributed arrays. This checking catches most simple types of programming error, such as mixing up array subscripts or combining distributed arrays in an incompatible way. A dependency analysis can also be done to check if the semantics of the loops has been altered by the parallel directives. This is planned, but is not done in this first version of MetaMP. Currently, if the user inserts a directive to distribute a `for` loop, MetaMP does it, even if the meaning of the program is altered.

A first version of MetaMP has been developed and non-trivial programs have been written in the language. The current version compiles down to a commercially available parallel message passing system, Express. The programs can be executed on actual parallel hardware. There is good reason to believe that efficiencies close to that obtained by manual programming are being achieved, though these have not yet been measured. The syntax and semantics of the MetaMP directives seem to be clean; the programs succinctly state what is happening in the parallel machine. We will discuss our plan of development for MetaMP in a later section. First we will describe a bit more thoroughly what MetaMP is through the use of a few examples.

Examples of MetaMP

A few, relevant example programs are discussed below. A detailed explanation of these can be found in the user's guide [17].

Global Combine

A common operation needed for parallel programs is the global combine. In a global combine, a datum from each processor is combined to form a single, global quantity whose value appears in all processors. An example of this is a global sum. Each processor has a value and we would like to compute the global sum of these and distribute the result to all processors. MetaMP has a compact syntax for this operation. Suppose `val` is a variable containing the processor's local value which we wish to globally combine. Let `gval` be the variable which will contain the result of the global combine.

```
#include <stdio.h>
int N;
int vec[N:4];    % distribute %

main()
{   int i,sum,gsum;
    printf("Enter the number of elements to be summed\n");
    scanf("%d",&N);
    % Alloc %
    printf("Enter %d ints\n",N);
    scanli("%d", vec);

    sum = 0;
    for (i=0; i<N; ++i) %{    % splitFor on vec[*] %
        sum += vec[i];
    %}
    gsum %int +=% sum;

    printf("Result:\n");
    fmulti(stdout);
    printf("%d %d\n",gsum,sum);
    exit(0);
}
```

Figure 1: `sum.mmp`: sum an input list of integers.

The MetaMP syntax for global combine is:

$$\texttt{gval}\ \%\textit{type}\ \textit{op} = \%\ \texttt{val};$$

The *type* field refers to the data type of the quantities to be combined. For example, if `val` is an `int`, then *type* should be `int`. The *op* field refers to what combining operator is desired. This binary operator must be associative so that the global result does not depend upon the order in which local values are combined. MetaMP 1.0 supports the following operators: `+` (add), `max` (take maximum of two operands), `min` (take minimum of two operands), and `|`, `&` (logical OR, AND of two operands). The user can also add his or her own combining functions.

Fig 1 is a simple example program demonstrating global combine usage. This program reads in a list of integers which is distributed by treating the items as elements of a distributed vector. The goal of the program is to compute the sum of the input list. This is accomplished by first computing the local sum in each processor, and then globally combining these partial results with the operator `+`.

MetaMP is told to distribute the vector across 4 processors by the declaration syntax:

```
int vec[N:4]; % distribute %
```

The directive, `% splitFor on vec[*] %`, tells MetaMP to distribute the `for` loop across the parallel machine. Each processor loops over only those members of `vec` which it actually holds.

The I/O routines `printf()` and `scanf()` are those taken from the underlying Cubix system of Express. The routine `scan1i()` is a MetaMP library routine which is used to scan a 1 dimensional array of `int`s. We will not discuss the details of I/O in this document. The reader is referred to [17] for further details.

It is worth emphasizing at this point that the program shown in Fig 1 is a complete program. It is compiled and run on a four processor parallel machine as follows.

```
iliamna% make sum
MMP sum.mmp > sum.c
netcc -c sum.c
netcc -o sum sum.o -lMMP -kplotix
iliamna% cubix -n4 sum
Allocated 4 nodes, origin at 0, process id 0.
Loading file sum to nodes 0-3 ....
Enter the number of elements to be summed
13
Enter 13 ints
1 2 3 4 5 6 7 8 9 10 11 12 13
Result:
91 10
91 18
91 36
91 27
System 0:14   User 0:36
CUBIX: exit status 0
```

Matrix Multiplication

In this section we discuss a program which reads in two matrices, `A` and `B`, computes their product and prints out the result. `B` is an $L \times N$ matrix, `A` is $M \times L$, and the result, `C`, is $M \times N$. There are *no* restrictions on M, N, or L. In particular, the "odd size" case, where these dimensions are not exactly divided by the number of processors, is allowed.

We derive the parallel algorithm in two stages. First, we begin with a simple sequential program, such as that shown in Fig 2. The addition of a few MetaMP directives gives us the parallel program of Fig 3. The arrays `C` and `A` have been distributed across the four processors along their row directions. That is, each processor holds elements of `C` and `A` which correspond to *all* of the columns of each of these matrices, but only *some*

```
#include <stdio.h>
#define MAX 64
int A[MAX][MAX];
int B[MAX][MAX];
int C[MAX][MAX];

main()
{   int L,M,N;
    int i,j,k,l;
    printf("This multiplies an MxL matrix into an LxN matrix\n");
    printf("For a resultant MxN matrix.\n");
    printf("Enter M, L, N\n");
    scanf("%d %d %d",&M,&L,&N);
    printf("Enter a %dx%d matrix (A) (ints)\n",M,L);
    for (i=0;i<M;++i)
        for (j=0;j<L;++j)
            scanf("%d",&A[i][j]);
    printf("Enter a %dx%d matrix (B) (ints)\n",L,N);
    for (i=0;i<L;++i)
        for (j=0;j<N;++j)
            scanf("%d",&B[i][j]);
    for (i=0;i<M;++i) {
        for (j=0;j<N;++j) {
            C[i][j] = 0;
            for (k=0;k<L;++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    printf("The result matrix (C) is:\n");
    for (i=0;i<M;++i) {
        for (j=0;j<N;++j)
            printf("%4d ",C[i][j]);
        printf("\n");
    }
    printf("\n");
    exit(0);
}
```

Figure 2: `matS.c`: sequential matrix multiplication.

```
#include <stdio.h>
int L,M,N;
int C[M:4][N];    % distribute %
int A[M:4][L];    % distribute %
int B[L][N];    % replicate %

main()
{   int i,j,k,l;
    int startk;
    printf("This multiplies an MxL matrix into an LxN matrix\n");
    printf("For a resultant MxN matrix.\n");
    printf("Enter M, L, N\n");
    scanf("%d %d %d",&M,&L,&N);
    % Alloc %
    printf("Enter a %dx%d matrix (A) (ints)\n",M,L);
    scan2i("%d",A);
    print2i("%4d ",A);
    printf("Enter a %dx%d matrix (B) (ints)\n",L,N);
    scan2i("%d",B);
    print2i("%4d ",B);
    for (i=0; i<M; ++i) %{    % splitFor on C[*][] %
        for (j=0; j<N; ++j) {
            C[i][j] = 0;
            for (k=0; k<L; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    %}
    printf("The result matrix (C) is:\n");
    print2i("%4d ",C);
    exit(0);
}
```

Figure 3: `matmult1.mmp`: parallel matrix multiplication.

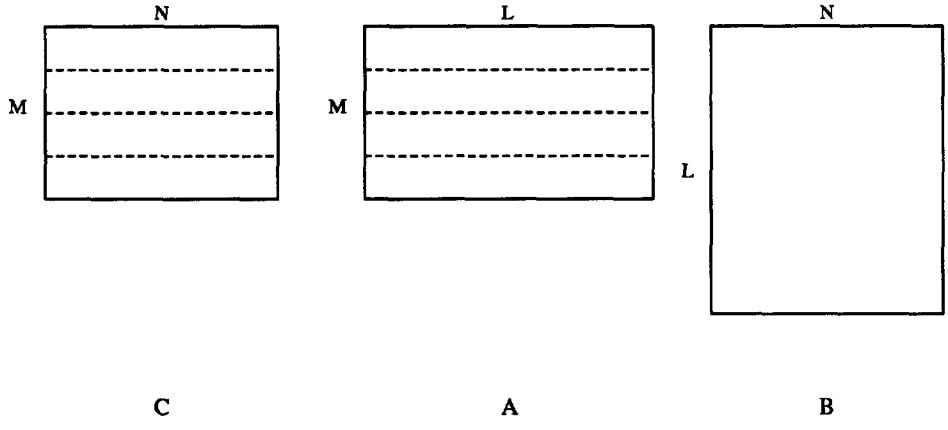

Figure 4: Decomposition for `matmult1.mmp`. `C` and `A` are distributed along rows, while `B` is replicated.

of the rows. The matrix `B` is replicated on the processors, that is, each processor holds a complete copy of `B`. The decompositions for `matmult1.mmp` are illustrated in Fig 4.

A simple `splitFor` directive on the outermost loop has produced an effective parallel program. Each processor computes some number of rows of `C`. The processor has all the data it needs to do this; no inter-processor communications are necessary. For some choices of matrix sizes and machine memory, this is the optimal parallel program.

There is one problem with this algorithm, however. The difficulty concerns the replication of `B`. If the matrices become large (which they are likely to do in a parallel context), then the storage of all of `B` in each processor forms a severe memory bottleneck, one which does not scale with the rest of the algorithm. To remove this bottleneck, we wish to distribute `B` also [18]. This decomposition is shown in Fig 5, while the program is given in Fig 6.

As before, each processor holds all of the elements of `A` which it requires to compute it's portion of `C`. This implies that no communication of `A` is required. As for `B`, each processor holds only some of the elements it needs. In fact, each processor must eventually hold every element of `B` to complete the matrix multiplication. The technique employed is to roll or shift `B` around the processors (treating them as a ring). At each step of the shifting, the newly arrived elements of `B` are combined with the correct elements of `A`

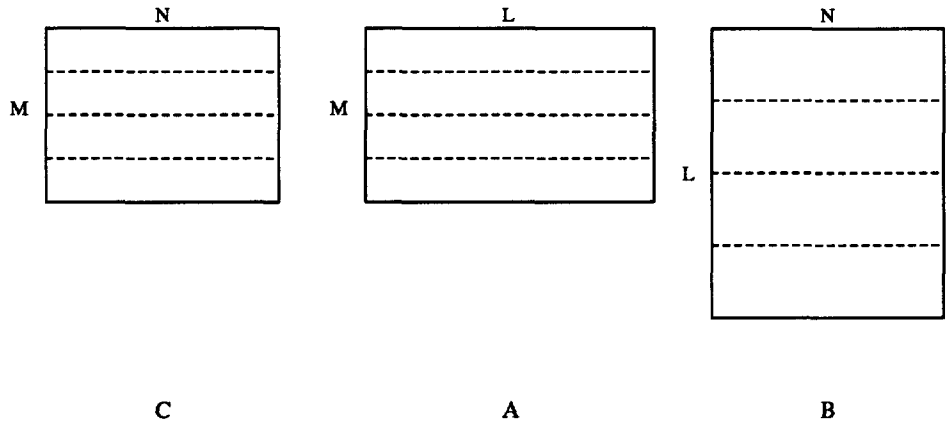

Figure 5: Decomposition for `matmult2.mmp`. All three matrices are distributed.

to contribute to `C`. The algorithm is complete when `B` has rolled completely around the processors and has landed back in it's original configuration. The four roll cycles for the four processor case are diagrammed in Fig 7. Shown there is the dataflow of `B` and, for each step, the portion of `A` which is to be multiplied into it.

The directive which causes the correct communications to happen is: `% roll on B[*][] %`. The notation `[*][]` specifies to MetaMP the direction of the roll.

This discussion of `matmult2.mmp` has been quite brief; again, more detail may be found in [17]. Though `matmult2.mmp` may appear complex, it is quite an improvement over versions written directly in the underlying message passing system [18, 19]. `matmult2.mmp` is an effective parallel matrix multiplier. There are no bottlenecks within the algorithm and it will scale correctly as the machine and the matrices are scaled. Finally, this program is correct for *any* choice of M, N, or L. The way in which this is accomplished is discussed in a later section.

```
#include <stdio.h>
int L,M,N;
int C[M:4][N];    % distribute %
int A[M:4][L];    % distribute %
int B[L:4][N];    % distribute %

main()
{   int i,j,k,l;
    int startk;
    printf("This multiplies an MxL matrix into an LxN matrix,\n");
    printf("for a resultant MxN matrix.\n");
    printf("Enter M, L, N\n");
    scanf("%d %d %d",&M,&L,&N);
    % Alloc %
    printf("Enter a %dx%d matrix (A) (ints)\n",M,L);
    scan2i("%d",A);
    print2i("%4d ",A);
    printf("Enter a %dx%d matrix (B) (ints)\n",L,N);
    scan2i("%d",B);
    print2i("%4d ",B);
    for (l=0; l<env.nprocs; ++l) {  // the four roll cycles
        for (i=0; i<M; ++i) %{      % splitFor on C[*][] %
            for (j=0; j<N; ++j) {
                if (l == 0) C[i][j] = 0;    // set only in 1st cycle
                for (k=0; k<L; ++k) %{    % splitFor on B[*][] %
                    C[i][j] += A[i][k] * B[k][j];
                %}
            }
        %}
        % roll on B[*][] %
    }
    printf("The result matrix (C) is:\n");
    print2i("%4d ",C);
    exit(0);
}
```

Figure 6: `matmult2.mmp`: parallel matrix multiplication, no memory bottleneck.

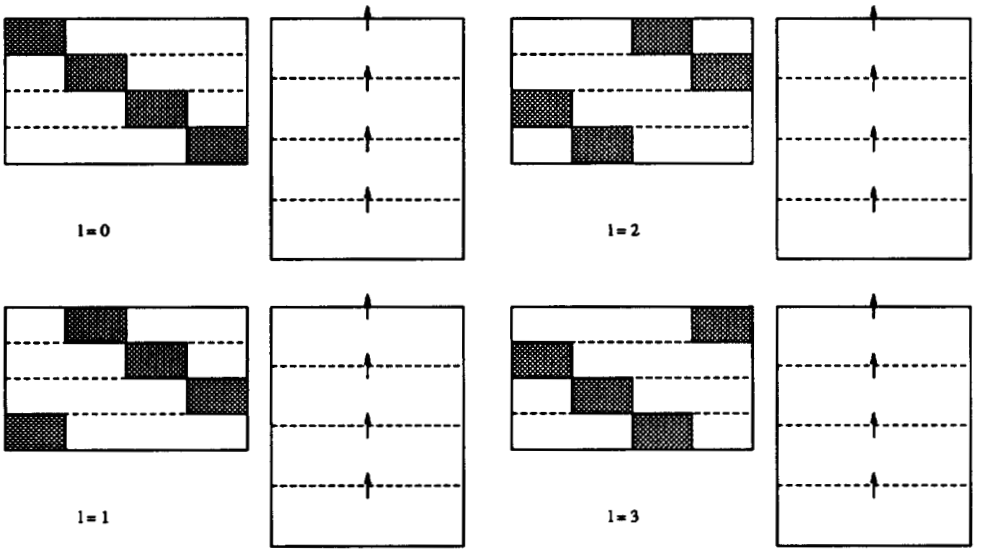

Figure 7: The four shift cycles for matrix multiplication on four processors. Arrows denote the direction of dataflow of `B`, the shading represents the currently active parts of `A`. `l` refers to the index in the program, `matmult2.mmp`.

Locality and Guard Strips, Laplace Solver

In this section we will discuss a problem in which array locality is extreme. The problem is that of solving the elliptical system,

$$\nabla^2 \phi = 0,$$

in two dimensions. We will do this via Jacobi relaxation. The strategy of the Jacobi algorithm is extremely simple (and of course, non-optimal). Discretize the problem with a homogeneous, rectangular mesh. One begins with some initial guess for ϕ which satisfies the boundary conditions. This could be random numbers; it can simply be `0.0` at all grid points. Set ϕ^{old} to this initial value. Then cycle through all the mesh points (a "sweep") setting a new field, ϕ, as:

$$\phi_{i,j} = \frac{1}{4} \cdot (\phi^{old}_{i-1,j} + \phi^{old}_{i+1,j} + \phi^{old}_{i,j-1} + \phi^{old}_{i,j+1}).$$

At the end of each sweep, re-enforce the given boundary conditions. Now set ϕ^{old} to this new value. Continue sweeping until the fields stop moving. At this point the algorithm has converged and the system is solved.

```
#include <stdio.h>
#define M    11
#define N    11
float phi[M][N];
float phio[M][N];
int bndy[M][N];
main()
{   int i,j,num,l;
    float tmp;
    FILE *fp;
    for (i=0; i<M; ++i) {
    for (j=0; j<N; ++j) {
        bndy[i][j] = 0;
        phi[i][j] = phio[i][j] = 0.0;
    }
    }
    printf("Reading in bndy data from bdata...\n");
    fp = fopen("bdata","r");
    fscanf(fp,"%d",&num);
    for (l=0; l<num; ++l) {
        fscanf(fp,"%d %d %f",&i,&j,&tmp);
        bndy[i][j] = 1;
        phio[i][j] = tmp;
    }
    fclose(fp);
    while (1) {
        printf("Enter number of iterations (neg entry ends run)\n");
        scanf("%d",&num);
        if (num <= 0) break;
        for (l=0; l<num; ++l) {
            for (i=1; i<M-1; ++i) {
            for (j=1; j<N-1; ++j) {
                phi[i][j] = 0.25 * ( phio[i-1][j] + phio[i+1][j] +
                                     phio[i][j-1] + phio[i][j+1] );
            }
            }
            for (i=0; i<M; ++i) {
            for (j=0; j<N; ++j) {
                if ( !bndy[i][j] )
                    phio[i][j] = phi[i][j];
            }
            }
        }
        for (i=0; i<M; ++i) {    /* Print out */
            for (j=0; j<N; ++j)
                printf("%5.3f ",phio[i][j]);
            printf("\n");
        }
        printf("\n");
    }
    exit(0);
}
```

Figure 8: `laplaceS.c`: laplace solver, sequential version.

We start off with a sequential program, `laplaceS.c`, shown in Fig 8. `phi` is the field ϕ, `phio` is ϕ^{old}. The parallel version of this, `laplace.mmp`, is given in Fig 9. This program is very similar in structure to `laplaceS.c`. For the most part, the arrays are decomposed and C `for` loops are distributed with `splitFor` directives. One new feature of this program is the use of a "guard strip" for the `phio` array. The idea is this. A look at the inner computation loop of `laplaceS.c` shows that the only need for inter-processor communication occurs when `phio` is accessed one step outside of the processor's portion of `phio`. This "nearest neighbor" communications pattern is a common problem in parallel programming and MetaMP provides a solution. The first thing to do is to add a guard strip to `phio`. The statement,

```
guardStrip=1
```

in the `phio` declaration directive tells MetaMP to allocate an extra strip of storage, of width 1, all around the edges of the processor's portion of `phio`. This insures that memory locations such as `phio[i-1][j]` actually exist for all legal values of `i` and `j`. This is illustrated in Fig 10. The guard strip values are meant to map across to the values of `phio` at the corresponding location in the neighboring processor. How are these set? The answer is the statement,

```
% updateGuard for phio %
```

The meaning of this is to perform all the necessary communications so as to update all of the guard strips associated with the array `phio`. The mapping of the guard strips and the operation of `updateGuard` is shown in Fig 11.

It is important to note that the guard strips of `phio` are transparent to the programmer. That is, `phio` still behaves as an array of the same size and shape as `phi`, even though `phi` has no guard strip. The guard values are located at "illegal" locations of `phio`, such as `phio[-1][j]`, `phio[i][-1]`, and `phio[i][% size of phio[][*] %]`. For more details of how the guard strips are accomplished in this transparent manner, the reader is referred to [17].

The final decision is: "where to put the `updateGuard`?" The correct location is at the top of the `while` as shown in Fig 9. This insures that the `phio` guard values will always have their most current values before being used.

Again, there are no restrictions on the problem sizes that can be solved by this parallel program. Boundary values can be placed anywhere in the

```
#include <stdio.h>
int M,N;
float phi[M:2][N:2];    % distribute %
float phio[M:2][N:2];    % distribute, guardStrip=1 %
int bndy[M:2][N:2];    % distribute %
main()
{   int i,j,num,l;
    float tmp;
    FILE *fp;
    printf("Laplace Solver for an MxN region.  Enter M,N\n");
    scanf("%d %d",&M,&N);
    % Alloc %
    for (i=0; i<M; ++i) %{        % splitFor on phi[*][] %
    for (j=0; j<N; ++j) %{        % splitFor on phi[][*] %
        bndy[i][j] = 0;
        phi[i][j] = phio[i][j] = 0.0;
    %}
    %}
    printf("Reading in bndy data from bdata...\n");
    fp = fopen("bdata","r");
    fscanf(fp,"%d",&num);
    for (l=0; l<num; ++l) {
        fscanf(fp,"%d %d %f",&i,&j,&tmp);
        bndy[i][j] = 1;        % mineOnly %
        phio[i][j] = tmp;        % mineOnly %
    }
    fclose(fp);
    while (1) {
        printf("Enter number of iterations (neg entry ends run)\n");
        scanf("%d",&num);
        if (num < 0) break;
        for (l=0; l<num; ++l) {
            % updateGuard for phio %
            for (i=1; i<M-1; ++i) %{    % splitFor on phi[*][] %
            for (j=1; j<N-1; ++j) %{    % splitFor on phi[][*] %
                phi[i][j] = 0.25 * ( phio[i-1][j] + phio[i+1][j] +
                                     phio[i][j-1] + phio[i][j+1] );
            %}
            %}
            for (i=0; i<M; ++i) %{        % splitFor on phi[*][] %
            for (j=0; j<N; ++j) %{        % splitFor on phi[][*] %
                if ( !bndy[i][j] )
                    phio[i][j] = phi[i][j];
            %}
            %}
        }
        print2f("%5.3f ", phio);
    }
    exit(0);
}
```

Figure 9: `laplace.mmp`: laplace solver, parallel version.

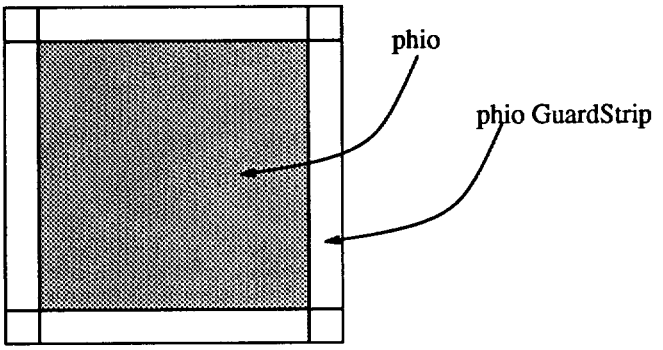

Figure 10: The guard strip for `phio`. Shown here is one processor's portion of `phio`.

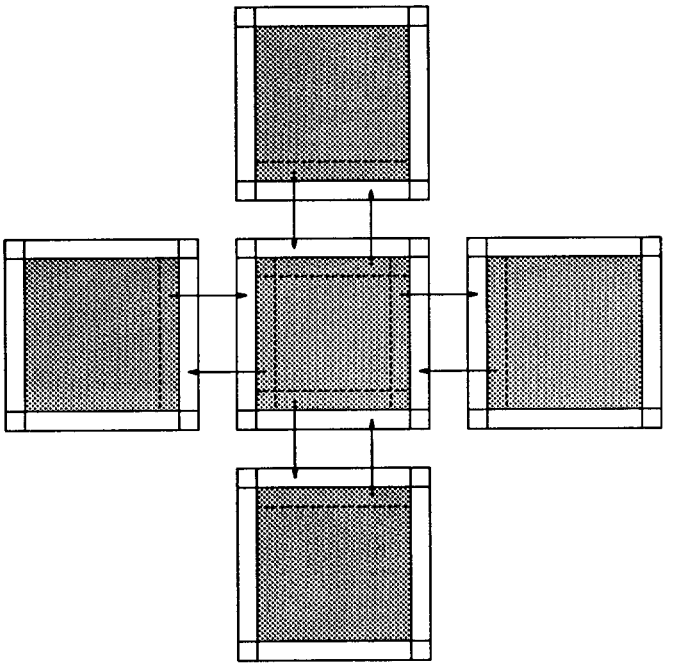

Figure 11: The guard strip mapping and the actions of `updateGuard`.

two dimensional mesh, leading to the possibility of solving highly irregular problems with this program. It is striking how much the MetaMP version of the laplace solver maps directly to the sequential C version. This isn't a general feature of all MetaMP programs, but it is reassuring that it's true for this conceptually straightforward parallel algorithm. What we *do* hope is a general feature of MetaMP is it's compactness. It is instructive to compare Fig 9 with the direct message passing versions given in Chapter 7 of [18] and in [19].

A BLAS-2 Routine

In this section we will present the parallelization of one of the routines from the BLAS-2 library [20, 21, 22]. Figure 12 shows the sequential program `mvS.c` which will be our template. This program performs the operations of `s-ge-mv()`, the BLAS-2 matrix-vector multiplier. Some of the options of `s-ge-mv()` have been left out for clarity (e.g., non-unit strides), but the essential operations remain. `s-ge-mv()` takes two vectors, **x** and **y**, and a matrix, **A**, as arguments and performs the assignment,

$$y_i \leftarrow \alpha A_{ij} x_j + \beta y_i, \ \forall i.$$

`mvS.c` reads in a matrix and two vectors, prompts for α and β, loops over **y** computing the assignment, and prints out **y** before exiting. **A** is an $M \times N$ matrix, **x** is of length N, and **y** is of length M. The only slight complication arises from the cases: $\beta = 0.0$, or $\beta = 1.0$. In order to avoid needless floating point operations, these cases are treated specially by the program.

The parallel, MetaMP version of this, `mv.mmp`, is given in figure 13. The fundamental decision to make is: "How should the data be distributed?" A simple choice is shown in figure 14. The matrix, `A`, is distributed along it's rows, that is, each processor contains all the columns of `A`. Since there is a direct correspondence between **y**'s subscript and the row subscript of `A`, we decompose **y** in the same way as the rows of `A`. This means that each processor will be entirely responsible for the computation and assignment to it's elements of **y**.

The final choice is the distribution of **x**. We decide to simply replicate **x** in each processor. This is the correct choice in terms of time, since no inter-processor communication of sections of **x** will be necessary. As for space considerations, this really doesn't form a memory bottleneck, in contrast to the situation with `matmult1.mmp`. The reason is that **x** is not the leading

```
#include <stdio.h>
#define MAX    64
int M,N;
float X[MAX],Y[MAX];
float A[MAX][MAX];
main()
{   int i,j,tmp;
    float tmpf,alpha,beta;
    FILE *fp;
    printf("s-ge-mv demo program.  Enter alpha, beta\n");
    scanf("%f %f",&alpha,&beta);
    fp = fopen("testMat","r");
    fscanf(fp,"%d %d",&M,&N);
    for (i=0; i<M; ++i)
        for (j=0; j<N; ++j)
            fscanf(fp,"%f",&A[i][j]);
    fclose(fp);
    fp = fopen("testVecs","r");
    fscanf(fp,"%d",&tmp);
    if (tmp != N)
        perror("vector size <-> matrix size mismatch");
    for (j=0; j<N; ++j)
        fscanf(fp,"%f",&X[j]);
    fscanf(fp,"%d",&tmp);
    if (tmp != M)
        perror("vector size <-> matrix size mismatch");
    for (i=0; i<M; ++i)
        fscanf(fp,"%f",&Y[i]);
    fclose(fp);

    if (beta != 1.0)
        if (beta == 0.0)
            for (i=0; i<M; ++i) {
                Y[i] = 0.0;
            }
        else
            for (i=0; i<M; ++i) {
                Y[i] = beta*Y[i];
            }
    for (i=0; i<M; ++i) {
        tmpf = 0.0;
        for (j=0; j<N; ++j)
            tmpf += A[i][j] * X[j];
        Y[i] += alpha * tmpf;
    }
    printf("Result vector is:\n");
    for (i=0; i<M; ++i) {
        printf("%8.3f  ",Y[i]);
    }
    printf("\n");
    exit(0);
}
```

Figure 12: mvS.c: BLAS-2 matrix-vector multiplier, sequential version.

```
#include <stdio.h>
int M,N;
float X[N];         % replicate %
float Y[M:4];         % distribute %
float A[M:4][N];     % distribute %
main()
{   int i,j,tmp;
    float tmpf,alpha,beta;
    FILE *fp;

    printf("s-ge-mv demo program.  Enter alpha, beta\n");
    scanf("%f %f",&alpha,&beta);
    fp = fopen("testMat","r");
    fscanf(fp,"%d %d",&M,&N);
    % Alloc %
    fscan2f(fp,"%f", A);
    fclose(fp);
    fp = fopen("testVecs","r");
    fscanf(fp,"%d",&tmp);
    if (tmp != N)
        perror("vector size <-> matrix size mismatch");
    fscan1f(fp,"%f", X);
    fscanf(fp,"%d",&tmp);
    if (tmp != M)
        perror("vector size <-> matrix size mismatch");
    fscan1f(fp,"%f", Y);
    fclose(fp);

    if (beta != 1.0)
        if (beta == 0.0)
            for (i=0; i<M; ++i) %{    % splitFor on Y[*] %
                Y[i] = 0.0;
            %}
        else
            for (i=0; i<M; ++i) %{    % splitFor on Y[*] %
                Y[i] = beta*Y[i];
            %}
    for (i=0; i<M; ++i) %{    % splitFor on Y[*] %
        tmpf = 0.0;
        for (j=0; j<N; ++j)
            tmpf += A[i][j] * X[j];
        Y[i] += alpha * tmpf;
    %}

    printf("Result vector is:\n");
    print1f("%8.3f  ", Y);
    exit(0);
}
```

Figure 13: `mv.mmp`: BLAS-2 matrix-vector multiplier, parallel version.

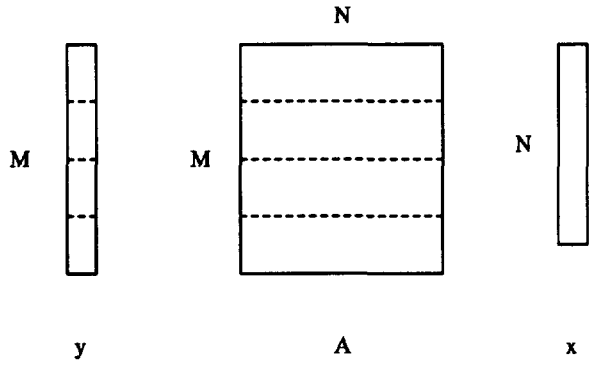

Figure 14: The decomposition for `mv.mmp`.

term in the expression for the space complexity of the algorithm (`A` is), and the storage cost of replicated `x` is of the same order as the storage cost of `A`.

Once the data distribution has been decided (and specified in the top few lines of `mv.mmp`), the rest of the parallelization is almost trivial. Conventional I/O primitives are replaced by their MetaMP analogs, and the outer loop over the members of `y` is "split" or made parallel by the usual `splitFor` directive.

The transformation of `mvS.c` to `mv.mmp` literally took only 20 minutes and the program functioned correctly the first time.

That is the good news. Now here is a little bad news. The above discussion might lead one to conclude that we have completely succeeded in providing a parallel version of the BLAS-2 routine `s-ge-mv()`. The truth is, for realistic cases, `mv.mmp` as given in figure 13 is inadequate. The root of the problem is the choice of decompositions which were made. If we imagine `mv.mmp` to be converted into a subroutine and embedded within some larger linear algebra program, we quickly realize that `A`, `x`, and `y`, will be coming from other parts of the program and may not be distributed appropriately.

The key need is to write *distribution independent* programs and libraries. This is possible to do (at least to a large extent) by exploiting some of MetaMP's object-oriented features. We leave this discussion and refer the reader to our companion paper on the subject [23].

MetaMP as an Object-Oriented System

MetaMP has some of the attributes of an object-oriented system. When a programmer declares a MetaMP data type (i.e., a distributed or replicated array), the compiler creates auxiliary variables associated with the array and it also emits code that sets the variables at run time. These variables specify various attributes of the arrays, such as the global and local sizes of the array in each dimension, how to map global subscripts to local subscripts (and the reverse), whether or not the array is decomposed (for each dimension), and so on. Each processor gets a copy of the associated variables. Some of the attributes are the same in each processor (such as global sizes), while others are distinct from processor to processor (an example is the mapping of global to local subscripts).

Attributes are passed into functions using the `setDcmp` directive discussed in [23, 17]. Attributes are also moved along with array sections as they are communicated from processor to processor. For instance, at the `% roll .. %` statement of `matmult2.mmp`, the attributes of `B` are communicated along with the actual contents of `B`. This is how the program functions correctly for any choice of array sizes. The loops recieve their limits from the array attributes; these are changing as the array is moved. This allows the loops to dynamically adjust to the current array section size.

The attributes associated with a MetaMP associated array allows it to function much like a `class` in C++. MetaMP directives such as `splitFor` and `roll`, act on instances of the `class` and have access to the `private` data of the `class`. They are analagous to the mthods or member functions of the `class`.

Instead of implementing the MetaMP directives as a set of classes and associated member functions, we have made a (simple) compiler. Why do this? Why not implement MetaMP as a set of C++ classes? One answer is that the current MetaMP seems to be easier to learn. One can often start with a sequential program (as was done for the programs appearing in this paper), decide upon an array distribution strategy, and then add MetaMP directives which accomplish the distribution and parallelization. MetaMP programs still have the look of their sequential specifications. Because of this, their exists the intriguing possibility of a semantics checker (for example, dependency analysis in loops) which could tell the user whether or not the meaning of the sequential program has been altered by the MetaMP additions.

There is a bad aspect to MetaMP as a compiler, however. If a user wants

to change the way MetaMP operates or add a new attribute to a MetaMP type, he or she cannot. MetaMP is, currently, a "hard-wired" compiler. For this reason, we wish to make future versions of MetaMP re-configurable. Users will be able to augment the default MetaMP classes and methods with their own.

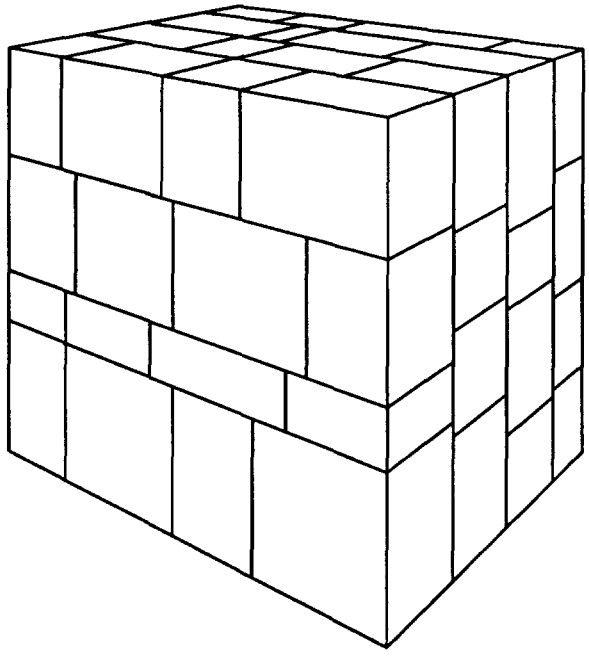

Figure 15: A 3D, hierarchical decomposition needed for PIC codes, as discussed in [24].

Plan of Development

A preliminary version of MetaMP offering support for simple, multi-dimensional arrays is functional and compiles down to a message passing system available on a wide variety of parallel computers. The development plan for MetaMP includes the following:

- Extending MetaMP to include unstructured meshes, hierarchical decompositions such as that shown in Fig 15, scattered decompositions, and dynamically changing meshes. This is essential for the support of scientific computation. The plan is to motivate the development of MetaMP by implementing large, realistic scientific programs in it. Currently, the largest program written in MetaMP is a Jacobi Eigensolver taken from *Numerical Recipes* [25].

- Investigation of other message-passing "back ends." Compiling down to a shared memory machine is interesting, since even shared memory computers achieve higher performance when locality is exploited [26].

- A Fortran version of MetaMP is straightforward to implement (MetaMP is not a full compiler so this change is less drastic than it sounds). A

study of the relation between CM Fortran and MetaMP is planned.

- Currently, MetaMP produces a log file telling the user what transformations have been done by the compiler. This information can be presented to the user in a more digestible form. An X/Motif-based browser tool is being developed which will present the transformations clearly.
- Displaying the meaning of distributed loops graphically, such as in Fig 7, can be automated and built into MetaMP. This is planned to combine with the code browser mentioned above.
- MetaMP already has some object-oriented features. We need to expose these features to the user so that he or she can customize or enhance them.
- Understanding the performance of MetaMP programs versus custom programmed codes is essential. It is thought that near-optimal performance is achievable, since the programmer still has control over the scheduling of communications.

Acknowledgements

I would like to thank both Rik Littlefield of Batelle Pacific Northwest Laboratories and David Walker of Oak Ridge Nat'l Lab for making many useful suggestions helpful to this research.

References

[1] D. Callahan and K. Kennedy. Compiling programs for distributed memory multiprocessors. *The Journal of Supercomputing*, 2:151–69, 1988.

[2] A.H. Karp and R.G. Babb II. A comparison of 12 parallel Fortran dialects. *IEEE Software*, 5(5):52–67, 1988.

[3] E.L. Lusk and R.A. Overbeek. A minimalist approach to portable, parallel programming. In L.H. Jamieson, D.B. Gannon, and R.J. Douglas, editors, *The Characteristics of Parallel Algorithms*, pages 351–62. MIT Press, 1987.

[4] L.H. Hamel, P.J. Hatcher, and M.J. Quinn. An optimizing C* compiler for a hypercube multicomputer. In *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines.* Elsevier. To appear.

[5] P. Mehrotra and J. Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5(3):339–61, 1987.

[6] R. Miller and Q.F. Stout. An introduction to the portable parallel programming language Seymour. In *Proceedings of the Thirteenth Annual International Computer Software and Applications Conference.* IEEE Computer Society, 1989.

[7] A.P. Reeves. Parallel Pascal: An extended Pascal for parallel computers. *Journal of Parallel and Distributed Computing*, 1:64–80, 1984.

[8] G.W. Sabot. *The Paralation Model.* MIT Press, 1988.

[9] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 69–80, 1989. SIGPLAN Notices 24, 7.

[10] M. Rosing, R.B. Schnabel, and R. Weaver. Dino: Summary and examples. In *The Third Conference on Hypercube Concurrent Computers and Applications.* ACM Press, 1988.

[11] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In D.W. Walker and Q.F. Stout, editors, *Proceedings of the Fifth Distributed Memory Computing Conference*, pages 1160–70. IEEE Computer Society Press, 1990.

[12] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. Technical Report YALEU/DCS/TR-725, Dept of Computer Science, Yale University, 1989.

[13] H.P. Zima, H.J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic mimd/simd parallelization. *Parallel Computing*, 6:1–18, 1988.

[14] E. Gabber. Developing a portable parallelizing Pascal compiler in Prolog. In L. Sterling, editor, *The Practice of Prolog.* MIT Press, 1990. to be published.

[15] L. Snyder. Parallel programming and the Poker programming environment. *IEEE Comput. Mag.*, 17(7):27–36, July 1984.

[16] E. Felten and S. Otto. Coherent parallel C. In *The Third Conference on Hypercube Concurrent Computers and Applications.* ACM Press, 1988.

[17] S.W. Otto. MetaMP users guide. Technical report, Oregon Graduate Institute of Science and Technology, 1991. document in preparation.

[18] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, Englewood Cliffs, NJ, 1988.

[19] I. Angus, G. Fox, J. Kim, and D. Walker. *Solving Problems on Concurrent Processors*, volume 2. Prentice Hall, Englewood Cliffs, NJ, 1990.

[20] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.*, 5:308–23, 1979.

[21] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An extended set of Fortran basic linear algebra subprograms. Technical Report 41, Mathematics and Computer Science Division, Argonne Nat'l Lab, 1986. Available from `netlib@ornl.gov`.

[22] J. Dongarra and D. Sorensen. Linear algebra on high-performance computers. In U. Schendel, editor, *Parallel Computing 85.* North Holland, 1986.

[23] S. Otto. Distribution independent programming and the saxpy. Technical report 91-004, Dept of Computer Science, Oregon Graduate Institute, 1991.

[24] P.M. Campbell, E.A. Carmona, and D.W. Walker. Hierarchical domain decomposition with unitary load balancing for electromagnetic particle-in-cell codes. In D.W. Walker and Q.F. Stout, editors, *Proceedings of the Fifth Distributed Memory Computing Conference.* IEEE Computer Society Press, 1990.

[25] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vettering. *Numerical Recipes in C.* Cambridge University Press, 1988.

[26] C. Lin and L. Snyder. A comparison of programming models for shared memory multiprocessors. In *1990 International Conference on Parallel Processing.* The Pennsylvania State University Press, 1990.

Appendix: A Jacobi Eigensolver

For those who have not yet had enough, here is a listing of a Jacobi eigensolver written in MetaMP. The solver is adapted from *Numerical Recipes in C*, the main, driver program is taken from *Numerical Recipes, Example Book (C).*

```
#include <math.h>
#include <malloc.h>
#include <stdio.h>

void jacobi();
void Interact();

//    This is a complete MetaMP version of jacobiS.c
//    Driver Program, taken from Numerical Recipes Example Book

float r[10];
int NP;
#define NMAT    3
float d[NP:4];          % distribute %
float v[NP][NP:4];      % distribute %    // store complete columns of v
float e[NP:4][NP];      % distribute %    // store complete rows of matrix
float ee[NP:4][NP];     % distribute %

int *workList;
int *readWorkList();
typedef struct _apart {
    float row[10],col[10];
} PARTICLE;
PARTICLE part[NP:4];    % distribute %
PARTICLE space[NP:4];   % distribute %

#define MyPos    partDcmp->me[0]

main()
{
    int i,j,k,l,nrot;
    static int num[4] = {0,3,5,10};
    float tmp;
    FILE *fp;

    fp = fopen("testMat","r");
```

```
for (i=1; i<=NMAT; ++i) {
NP = num[i];
% Alloc %
workList = readWorkList(MyPos,"worklist4");

for (j=0; j< % gsize of e[*][] %; ++j) {
    for (k=0; k< % gsize of e[][*] %; ++k) {
            fscanf(fp,"%f",&tmp);
        e[j][k] = tmp;        % mineOnly %
        ee[j][k] = e[j][k];    % mineOnly %
    }
}
jacobi(ee,num[i],d,v,&nrot);
printf("matrix number %2d\n",i);
printf("number of jacobi rotations: %3d\n",nrot);
printf("eigenvalues:\n");
fmulti(stdout);
for (j=0; j<NP; ++j) %{        % splitFor on d[*] %
    printf("%12.6f",d[j]);
%}
fsingl(stdout);
printf("\neigenvectors:\n");
fmulti(stdout);
for (j=0; j<NP; ++j) %{        % splitFor on v[][*] %
    printf("number %3d \n",j);
    for (k=0; k<NP; ++k) %{    % splitFor on v[*][] %
    printf("%12.6f",v[k][j]);
    %}
    printf("\n");
%}
fsingl(stdout);
    // eigenvector test
fmulti(stdout);
for (j=0; j<NP; ++j) %{        % splitFor on v[][*] %
    for (l=0; l<NP; ++l) %{    % splitFor on e[*][] %
        r[l] = 0.0;
        for (k=0; k<NP; ++k) %{    % splitFor on e[][*] %
            r[l] += e[l][k]*v[k][j];
        %}
    %}
    printf("vector number %3d\n",j);
    printf("      %11s %14s %10s\n","vector","mtrx*vec.", "ratio");
    for (l=0; l<NP; ++l) %{    % splitFor on e[*][] %
    printf("Elem No %2d %12.6f %12.6f %12.6f\n",l,
        v[l][j],r[l],r[l]/v[l][j]);
    %}
%}
fsingl(stdout);
    printf("Press RETURN to continue...\n");
    getchar();
% Free %
free(workList);
}
fclose(fp);
exit(0);
```

```
}

int locNrot,iter;
#define DOTRowCol(sum,m1,i,m2,j)    sum=0.0; \
    for (di=0; di<n; ++di) {    \
        sum += m1[i][di]*m2[di][j];    \
    }

void jacobi(a,n,d,v,nrot)
float **a,d[],**v;
int n,*nrot;
//
//    Cyclic Jacobi, Factored Form
// Taken partially from Numerical Recipes, suggestions
// from R. Littlefield and W. Eggers.
// Computes all eigenvalues and eigenvectors of a real symmetric
// matrix a[0..n-1][0..n-1].  On output, a is destroyed.
// This is in contrast to the routine in Numerical Recipes.
// d[0..n-1] returns the eigenvalues of a.
// v[0..n-1][0..n-1] is a matrix whose columns contain, on
// output, the normalized eigenvectors of a.  nrot returns the
// number of Jacobi rotations that were required.

%{{    % setDcmp of a,v %
    % shape a[NP:4][NP],v[NP][NP:4] %

    int j,iq,ip,i,di,num;
    int l;

                    // Init V to identity
    for (ip=0; ip<NP; ++ip) %{          % splitFor on v[*][] %
    for (iq=0; iq<NP; ++iq) %{    % splitFor on v[][*] %
        if (iq == ip)     // subtle!Need to get MMP to complain!!!
            v[ip][iq] = 1.0;
        else
            v[ip][iq] = 0.0;
    %}
    %}
                // Copy a and v to row and col of particles
                // Part i stores the ith row of a, ith col of v
    for (i=0; i<NP; ++i) %{         % splitFor on part[*] %
    for (j=0; j<NP; ++j) %{         % splitFor on v[*][] %
        part[i].row[j] = a[i][j];
        part[i].col[j] = v[j][i];
    %}
    %}
    locNrot = 0;
    for (iter=1; iter<=10; ++iter) {
                // Normal triang loop within processor
    for (i=0; i<NP; ++i) %{         % splitFor on part[*] %
        for (j=i+1; j<% size of part[*] %; ++j) %{ % range of j is NP:4 %
            Interact(&part[i],&part[j]);
        %}
    %}
                // the P-1 cycles of the LRF-NC Alg
```

```
    for (l=0; l<env.nprocs-1; ++l) {
        computeInterWithProc(MyPos,workList[l]);
    }
    }
                // Copy particles back to a and v
                // Part i stores the ith row of a, ith col of v
    for (i=0; i<NP; ++i) %{        % splitFor on part[*] %
    for (j=0; j<NP; ++j) %{        % splitFor on v[*][] %
        a[i][j] = part[i].row[j];
        v[j][i] = part[i].col[j];
    %}
    %}
                    // set returned eigenvalues
    for (ip=0; ip<NP; ++ip) %{        % splitFor on d[*] %
        DOTRowCol(d[ip],a,ip,v,ip)    % leave alone %
    %}
    locNrot %int +=% locNrot;    // turn into a global
    *nrot = locNrot;
    return;
%}}



#define ROTATErow(p1,j,p2,l) g=p1->row[j]; h=p2->row[l]; \
                    p1->row[j]=g-s*(h+g*tau); \
                    p2->row[l]=h+s*(g-h*tau);

#define ROTATEcol(p1,j,p2,l) g=p1->col[j]; h=p2->col[l]; \
                    p1->col[j]=g-s*(h+g*tau); \
                    p2->col[l]=h+s*(g-h*tau);
#define DOTRowColPart(sum,p1,p2)    sum=0.0; \
    for (di=0; di<NP; ++di) {    \
        sum += p1->row[di]*p2->col[di];    \
    }

void Interact(part1,part2)
PARTICLE *part1,*part2;
{
    float tresh,theta,tau,t,s,h,g,c;
    float a_pq,a_pp,a_qq;
    int di,j;

    tresh = 0.0;
    DOTRowColPart(a_pq,part1,part2)
    DOTRowColPart(a_pp,part1,part1)
    DOTRowColPart(a_qq,part2,part2)
    g = 100.0 * fabs(a_pq);
    if ( (iter>4) && (fabs(a_pp)+g == fabs(a_pp))
           && (fabs(a_qq)+g == fabs(a_qq)) ) {
        ;
    }
    else if (fabs(a_pq) > tresh) {
        h = a_qq-a_pp;
        if (fabs(h)+g == fabs(h)) {    // t = 1/(2*theta)
            t = a_pq/h;
```

```
        }
        else {
            theta = 0.5*h/(a_pq);    // 11.1.10
            t = 1.0/(fabs(theta)+sqrt(1.0+theta*theta));
            if (theta < 0.0) t = -t;
        }
        c = 1.0/sqrt(1+t*t);    // error in 11.1.11
        s = t*c;
        tau = s/(1.0+c);
        for (j=0; j<NP; ++j) {    // update V^tA and V together
            ROTATErow(part1,j,part2,j)
            ROTATEcol(part1,j,part2,j)
        }
        ++locNrot;
    }
}
```