# Distribution Independent Programming and the Saxpy

*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 91-004

March, 1991

# Distribution Independent Programming and the Saxpy

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

February 28, 1991

**Abstract**

We show how to use MetaMP's object-oriented features to write distribution independent programs. This facilitates the construction of a distributed-memory, MIMD, software library such as Linpack or the BLAS.

## Introduction

It is generally agreed that distributed-memory parallel computers have the potential for very high performance. In specific applications this performance has been realized. In many cases, however, the parallel programs are far from matching the functionality of their sequential counterparts. Though there doesn't seem to be fundamental obstacles to using distributed memory, MIMD machines for large-scale scientific and engineering computations, the machines and their associated software environments have proven to be idiosyncratic and problematical.

To achieve widespread usage, a natural approach to consider is to build large software libraries of highly functional, optimized components. In some domains at least, this approach seems to be workable.

In another paper [1], we introduced MetaMP, a set of compile time directives and a run time system which supports multi-dimensional arrays distributed on a MIMD machine. MetaMP compiles down to C and Express, and so is portable across many parallel machines. Here, we wish to concentrate on some of MetaMP's object-oriented features and how these may be applied toward writing *distribution independent* programs. By this we mean programs which, at their high level description, remain the same (and remain correct) as different choices are made for the distribution strategy of the arrays involved. Distribution independent programs and subroutines are much more suitable for the construction of a software library.

In this paper we will show how to write MetaMP functions and their calling programs so that they are distribution independent. As an example, we illustrate the method with the **saxpy()** routine, one of the members of the BLAS library [2]. We show how this routine can function correctly in a variety of distribution environments and how this can be embedded within a distribution independent Gaussian elimination program.

## The Saxpy

The **saxpy()** is a member of the BLAS library used to add to a vector a constant times another vector. That is, let **y** and **x** be vectors and $\alpha$ some scalar. Then **saxpy(alpha, y, x)** computes the assignment:

$$y_i \leftarrow \alpha x_i + y_i, \ \forall i.$$

Figure 1 shows a simple sequential program which reads in two vectors, prompts the user for $\alpha$, applies **saxpy()**, and prints out the result. Some of the options of the real **saxpy()** have been left out for clarity (e.g., non-unit strides), but the essential operations remain.

Figure 2 gives the MetaMP equivalent of figure 1. A brief description of each of the MetaMP directives appearing in the program follows.

- The vectors are distributed across all of the nodes of the machine (this program is written with the number of processors, **4**, wired into the code). The vectors themselves have length **M**, and the declaration and distribution of the vectors is accomplished with:

```
float X[M:4]; % distribute %
float Y[M:4]; % distribute %
```

- The vectors are allocated at the MetaMP statement:

```
#include <stdio.h>
#define MAX    64
int M;
float X[MAX],Y[MAX];

main()
{   int i,tmp;
    float alpha;
    FILE *fp;
    printf("s-ax-py demo program.  Enter alpha\n");
    scanf("%f",&alpha);
    fp = fopen("testVecs2","r");
    fscanf(fp,"%d",&M);
    for (i=0; i<M; ++i)
        fscanf(fp,"%f",&X[i]);
    fscanf(fp,"%d",&tmp);
    if (tmp != M) perror("vector size mismatch");
    for (i=0; i<M; ++i)
        fscanf(fp,"%f",&Y[i]);
    fclose(fp);

    saxpy( alpha, Y, X, M);

    printf("Result vector is:\n");
    for (i=0; i<M; ++i)
        printf("%8.3f  ",Y[i]);
    printf("\n");
    exit(0);
}

saxpy( alph, y, x, size )
float alph,*y,*x;
int size;
{   int i;
    for (i=0; i<size; ++i) {
        y[i] += alph * x[i];
    }
}
```

Figure 1: saxpyS.c: sequential saxpy().

```
#include <stdio.h>
int M;
float X[M:4];    % distribute %
float Y[M:4];    % distribute %

main()
{   int tmp;
    float alpha;
    FILE *fp;
    printf("s-ax-py demo program.  Enter alpha\n");
    scanf("%f",&alpha);
    fp = fopen("testVecs2","r");
    fscanf(fp,"%d",&M);
    % Alloc %
    fscanif(fp,"%f", X);
    fscanf(fp,"%d",&tmp);
    if (tmp != M) perror("vector size mismatch");
    fscanif(fp,"%f", Y);
    fclose(fp);

    saxpy(alpha, Y, X);

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

saxpy( alph, y, x)
float alph;
float *y,*x;
%{{   % setDcmp of y,x %
    int i;
    for (i=0; i<% gsize of y[*] %; ++i) %{        % splitFor on y[*] %
        y[i] += alph * x[i];
    %}
%}}
```

Figure 2: saxpy.mmp: parallel saxpy(), first version.

`% Alloc %`

MetaMP creates, along with a distributed array, an associated data structure (the `Dcmp` structure) which gives the run time attributes of the array. Each processor has a copy of this structure and it is computed at allocation time. The attributes are such things as: the starting and ending points of this processor's portion of the array, for each dimension of the array; the global sizes of the array; whether or not the array has associated guard strips; and so on. Some of the attributes vary from processor to processor.

- The data for the vectors are read in by the MetaMP library routine `fscanlf()`. We do not wish to discuss I/O here; it is covered in detail in the user's guide [3].
- The actual call to `saxpy()` is made much like that in the sequential case. The one difference is that the size of the vectors is not supplied as an argument. Since the vectors are MetaMP objects, their sizes are available from the `Dcmp` structure. But which `Dcmp` structure? After all, in the specification of `saxpy()`, `y` and `x` are merely dummy arguments. The MetaMP directive, `% setDcmp of y,x %`, answers this question. When `saxpy()` is called, `setDcmp` associates the correct `Dcmp` structure with `y` and `x`. This means that, within `saxpy()`, one can query the `Dcmp` structure for attributes of the argument arrays.
- The statement, `% gsize of y[*] %`, is precisely such a query. This one says to return the global size of the array `y` in it's 1st dimension. That is, writing this is equivalent to writing `M`. By the way, calling this a "query" may cause one to suspect that this is a slow operation, but this isn't true. MetaMP inline expands such queries to a simple access of memory.
- At this point, the `for` within `saxpy()` is a loop over the entire vector, `y`. The final directive, `splitFor on y[*]`, turns this into a parallel loop. It causes each processor to loop over only those members of `y` which are stored in this processor. Again, this is done efficiently – the upper limit of the loop is modified to this processor's `% size of y[*] %` attribute.

This complete our discussion of `saxpy.mmp`. As specified, the `saxpy()` routine can be used on any vector object. In many linear algebra contexts, however, we wish to run the routine on rows or columns of a distributed matrix.

## Using the Saxpy in Other Contexts

In a real linear algebra application, for instance, Gaussian elimination, we need to run the `saxpy()` on two vectors, where the vectors are rows of a matrix. The sequential program in figure 3 shows the sort of operation we need to perform. `X` and `Y` are set to point at the beginning of the rows of `A`, they are fed into `saxpy()`, and it works.

Now let's look at how we would do this in MetaMP. We begin by distributing the matrix, `A`, in the row direction. We will treat the other cases (column-wise and two-dimensional distribution) later. Figure 4 gives the correct MetaMP analog of `saxpyS2.c`. Let us discuss the new MetaMP directives in order:

- The declarations,

  ```
  float X[N];          % replicate %
  float Y[N];          % replicate %
  float A[M:4][N];      % distribute %
  ```

  distribute `A` row-wise across the machine, and give each processor a copy of `X` and `Y`.

- The directives,

  ```
  X = A[5];     % set to subarray A[5][*], copy to all %
  Y = A[2];     % set to subarray A[2][*] %
  ```

  cause the following to happen. In the processor which contains `A[5][*]` (row 5 of `A`), the pointer `X` is set to point at it, and the `set?` attribute of `X` is assigned `TRUE`. In other processors, the `set?` attribute of `X` is assigned `FALSE`. In the processor which contains `A[2][*]` (row 2 of `A`), the pointer `Y` is set to point at it, and the `set?` attribute of `Y` is assigned `TRUE`. In other processors, the `set?` attribute of `Y` is assigned `FALSE`.

- The second directive on `X` in the above, `copy to all`, causes the set `X` to be copied to all other replicas of `X`, that is, a broadcast occurs to

```
#include <stdio.h>
#define MAX    64
int M,N;
float *X,*Y;
float A[MAX][MAX];
main()
{   int i,j,tmp;
    float alpha;
    FILE *fp;
    printf("s-ax-py demo program.  Enter alpha\n");
    scanf("%f",&alpha);
    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);

    Y = A[2];     /* Do a saxpy of row 5 with row 2 */
    X = A[5];
    saxpy( alpha, Y, X, N);
    printf("Result vector is:\n");
    for (i=0; i<N; ++i)
        printf("%8.3f  ",Y[i]);
    printf("\n");
    exit(0);
}

saxpy( alph, y, x, len )
float alph;
float *y,*x;
int len;
{   int i;
    for (i=0; i<len; ++i) {
        y[i] += alph * x[i];
    }
}
```

Figure 3: `saxpyS2.c`: run `saxpy()` on two rows of a matrix (sequential).

```
#include <stdio.h>
int M,N;
float X[N];         % replicate %
float Y[N];         % replicate %
float A[M:4][N];     % distribute %

main()
{   int i,tmp,origin;
    float alpha;
    FILE *fp;
    printf("s-ax-py demo program.  Enter alpha\n");
    scanf("%f",&alpha);
    fp = fopen("testMat","r");
    fscanf(fp,"%d %d",&M,&N);
    % Alloc %
    fscan2f(fp,"%f", A);
    fclose(fp);

    X = A[5];    % set to subarray A[5][*], copy to all %
    Y = A[2];    % set to subarray A[2][*] %
    saxpy(alpha, Y, X);

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

saxpy( alph, y, x)
float alph;
float *y,*x;
%{{    % setDcmp of y,x %
    int i;

    if (% y set? % && % x set? %) {
        for (i=0; i<% gsize of y[*] %; ++i) %{    % splitFor on y[*] %
            y[i] += alph * x[i];
        %}
    }
%}}
```

Figure 4: saxpy2.mmp: MetaMP version of saxpyS2.c.

```
#include <stdio.h>

int M,N;
float X[N:4];         % distribute %
float Y[N:4];         % distribute %
float A[M][N:4];    % distribute %

 ...  The rest is the same
```

Figure 5: saxpy3.mmp: column-wise version of saxpy2.mmp.

all the replicas so that each processor has row 5 of A. This is done so that some processor actually contains both X and Y. Doing this as a broadcast is appropriate – in Gaussian elimination, for example, one is running a saxpy() of one row of the matrix with all the rows of the matrix below it. The copy also has the effect of setting the **set?** attribute of X to TRUE in all processors.

- Finally, we wish to run the saxpy() only in those processors in which both X and Y have been set, that is, where they are both valid. This is accomplished inside the routine by the statement:

```
if (% y set? % && % x set? %)
```

## Other Data Distribution Choices

The methods employed in saxpy2.mmp have given us *distribution independence*. We can now vary the distribution of the matrix and still have correct behavior.

### Column-wise Distribution

To change saxpy2.mmp to a column-wise distribution, the array declarations need to be modified to that shown in figure 5. With these declarations, the distribution is column-wise and the program is still correct. Here are a few comments about this case.

- Processors no longer contain entire rows, and so, each processor has only a part of X and Y.

```
#include <stdio.h>

int M,N;
float X[N:2];          % distribute on [][*], replicate on [*][] %
float Y[N:2];          % distribute on [][*], replicate on [*][] %
float A[M:2][N:2];      % distribute %

  ...  The rest is the same
```

Figure 6: saxpy4.mmp: two-dimensional version of saxpy2.mmp.

- There is no need to replicate the vectors, and there is no problem in getting the valid X and Y together in the same processor. This, in turn, implies that a broadcast (the copy to all directive) isn't necessary. MetaMP knows this, however, since it knows how the arrays are distributed. Therefore, even though the copy to all directive remains in the program, no code to actually do a broadcast is emitted by the MetaMP compiler.

- The saxpy() routine continues to function correctly.

### Two-Dimensional Distribution

Now we wish to distribute the matrix in both the row direction and the column direction. Figure 6 gives the declaration syntax for this case. Notable features of this case are listed below.

- In this case, the vectors are distributed in the column direction but need to be replicated in the row direction. A mixed directive like this is available in MetaMP and shown in figure 6. The directions are specified with the usual [][*], [*][], notation.

- The copy to all directive now causes a broadcast to occur only to the set of replicas of X. That is, a broadcast in the row direction is emitted by the MetaMP compiler.

- Again, the saxpy() routine functions correctly.

## Actual Runs

To show that these are real programs, we give the output of them operating on some test data. Here is the test data and the result of running the sequential program, saxpyS2.c, on it:

```
iliamna% cat testMat
9  7
-5.726 -3.622 4.902 -4.336 -7.347 -4.286 -0.158
-0.797 4.693 -6.129 1.932 0.714 4.449 1.977
0.320 -6.667 -5.266 -4.886 3.900 4.171 2.433
3.902 6.757 6.638 3.588 0.322 -0.101 2.179
3.090 3.075 1.011 5.363 7.453 -2.087 -6.973
-7.894 1.627 0.869 -0.691 -1.680 2.740 -6.759
7.034 -0.811 3.218 -0.646 0.522 5.952 2.467
-3.578 2.123 -3.100 -7.676 0.880 -4.462 3.912
-6.797 3.438 -1.910 4.292 -1.487 7.102 1.656
iliamna% saxpyS2
s-ax-py demo program.  Enter alpha
.56
Result vector is:
  -4.101    -5.756    -4.779    -5.273     2.959     5.705    -1.352
```

Here is the run of saxpy2.mmp (cubix is the Express interface to the parallel machine):

```
iliamna% cubix -n4 saxpy2
Allocated 4 nodes, origin at 0, process id 0.
Loading file saxpy2 to nodes 0-3 ....
s-ax-py demo program.  Enter alpha
.56
Result vector is:
Processor 0 has:
  -4.101    -5.756    -4.779    -5.273     2.959     5.705    -1.352
Processor 1 has:
   0.000     0.000     0.000     0.000     0.000     0.000     0.000
Processor 2 has:
   0.000     0.000     0.000     0.000     0.000     0.000     0.000
Processor 3 has:
   0.000     0.000     0.000     0.000     0.000     0.000     0.000
System 0:15   User 0:3
CUBIX: exit status 0
```

Note that the saxpy was computed only in processor 0, which is where Y was located.

Here is the run of saxpy3.mmp:

```
iliamna% cubix -n4 saxpy3
Allocated 4 nodes, origin at 0, process id 0.
Loading file saxpy3 to nodes 0-3 ....
s-ax-py demo program.  Enter alpha
.56
Result vector is:
Processor 0 has:
  -4.101    -5.756
Processor 1 has:
  -4.779    -5.273
Processor 3 has:
   2.959     5.705
```

```
Processor 2 has:
  -1.352
System 0:14   User 0:3
CUBIX: exit status 0
```

Now all of the processors have a section of `Y`, and so they all contribute to the computation.

Here is the run of `saxpy4.mmp`:

```
iliamna% cubix -n4 saxpy4
Allocated 4 nodes, origin at 0, process id 0.
Loading file saxpy2 to nodes 0-3 ....
s-ax-py demo program.  Enter alpha
.56
Result vector is:
Processor 0 has:
  -4.101    -5.756    -4.779    -5.273
Processor 1 has:
   2.959     5.705    -1.352
Processor 2 has:
   0.000     0.000     0.000     0.000
Processor 3 has:
   0.000     0.000     0.000
System 0:15   User 0:3
CUBIX: exit status 0
```

This is the mixed case. Two processors participate in the `saxpy()`, while the other two don't.

## A More Realistic Program

Finally, we give a more realistic program using our parallel saxpy(). It is shown in figure 7, and it zeros out all elements below the diagonal of the first column of the matrix by doing the appropriate saxpy() operation. A complete Gaussian elimination program is not far behind. Here it is in operation, on the same test matrix as before:

```
iliamna% cubix -n4 ge
Allocated 4 nodes, origin at 0, process id 0.
Loading file ge to nodes 0-3 ....
ge demo program.
Matrix is:
Processor 0 has:
  -5.726    -3.622     4.902    -4.336    -7.347    -4.286    -0.158
  -0.000     5.197    -6.811     2.536     1.737     5.046     1.999
   0.000    -6.869    -4.992    -5.128     3.489     3.931     2.424

Processor 1 has:
  -0.000     4.289     9.978     0.633    -4.685    -3.022     2.071
   0.000     1.120     3.656     3.023     3.488    -4.400    -7.058

Processor 3 has:
  -0.000     6.620    -5.889     5.287     8.449     8.649    -6.541
   0.000    -5.260     9.240    -5.972    -8.503     0.687     2.273

Processor 2 has:
  -0.000     4.386    -6.163    -4.967     5.471    -1.784     4.011
   0.000     7.737    -7.729     9.439     7.234    12.190     1.844

System 0:15   User 0:1
CUBIX: exit status 0
iliamna%
```

## References

[1] S. Otto. MetaMP: A higher level abstraction for message-passing programming. Technical report 91-003, Dept of Computer Science, Oregon Graduate Institute, 1991.

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

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

```
#include <stdio.h>
int M,N;
float X[N];          % replicate %
float Y[N];          % replicate %
float A[M:4][N];      % distribute %

main()
{   int i,tmp,origin;
    float alpha;
    FILE *fp;
    printf("ge demo program.\n");
    fp = fopen("testMat","r");
    fscanf(fp,"%d %d",&M,&N);
    % Alloc %
    fscan2f(fp,"%f", A);
    fclose(fp);

    X = A[0];    % set to subarray A[0][*], copy to all %

    for (i=1; i<M; ++i) %{           % splitFor on A[*][] %
        alpha = -A[i][0]/X[0];
        Y = A[i];    % set to subarray A[i][*] %
        saxpy(alpha, Y, X);
    %}

    printf("Matrix is:\n");
    print2f("%8.3f  ", A);
    exit(0);
}

saxpy( alph, y, x)
float alph;
float *y,*x;
%{{    % setDcmp of y,x %
    int i;

    if (% y set? % && % x set? %) {
        for (i=0; i<% gsize of y[*] %; ++i) %{    % splitFor on y[*] %
            y[i] += alph * x[i];
        %}
    }
%}}
```

Figure 7: ge.mmp: zero-out first column, using saxpy()