User's Guide and Documentation of the Parallel General Benchmark on the IBM SP2*

Masahiro Arakawa, Zhiwei Xu, and Kai Hwang
Parallel Computing Research Laboratory
University of Southern California
Los Angeles, CA 90089

June 19, 1995

CENG Technical Report 95-12

* This work was supported by a research subcontract from MIT Lincoln Laboratories to the University of Southern California as part of the ARPA Mountaintop Program. All rights are reserved by MIT/LL and by USC. Distribution is limited upon approval from the Mountaintop program at MIT/LL.

Abstract

The General benchmark is one of the three programs in the parallel Space-Time Adaptive Processing (STAP) benchmark suite, jointly developed by the University of Southern California and MIT Lincoln Laboratory. This report provides an overview of the parallelization of the General benchmark for the IBM SP2 massively parallel processor. A user's guide, which lists the files and procedures making up the parallel General program as well as directions regarding the compilation and execution of the parallel General benchmark, is provided.

This project was conducted between August 1994 and May 1995 by the benchmark parallelization team at the University of Southern California: Kai Hwang, Zhiwei Xu, and Masahiro Arakawa. Hwang served as the principal investigator for the entire project. Xu developed the parallelization strategies for all the STAP benchmarks. Xu and Arakawa jointly developed the parallel code and collected performance measurements. Arakawa documented the parallel benchmark programs as reported here. This project was funded by the ARPA Mountaintop Program as a subcontract from MIT Lincoln Laboratory.

The STAP benchmarks allow us to evaluate the performance of a particular parallel computer for real-time digital radar signal processing by simulating the computations necessary for STAP. Digital adaptive signal processing is the one technology which has the potential to achieve thermal noise limited performance in the presence of jamming (electronic counter-measures) and clutter (signal returns from land) [Titi94]. A good description of the original General C source code can be found in [LL94].

We used a *single-program, multiple data stream* (SPMD) approach to parallelizing the benchmarks, directing each task to perform the same algorithm on independent portions of the data set. We added message-passing operations, such as broadcast and total exchange, to redistribute data. Because of the high message-passing overhead associated with the SP2, we adopted a coarse-grain strategy, minimizing the number of internode communication operations in our parallel programs.

The General benchmark is designed to stress general signal-processing computations, as well as test the data communication capabilities of the machine. This benchmark performs the following steps:

1. Search and sort processing
2. FFT processing across each dimension of the data cube
3. Point-by-point vector multiplication along each dimension of the data cube.
4. Miscellaneous linear algebra processing, including Cholesky factorization and back substitution to solve a set of linear equations, and subsequent beamforming (inner product).

Each of these steps is independent. We parallelized each of these steps in a separate subprogram, and will therefore analyze each one separately.

Section 1 describes the computations performed in the General benchmark and its implementation in the sequential version of the program. Section 2 describes the parallel General benchmark on the SP2. Sections 3 through 6 contain the user's guide to the parallel program, listing the files and procedures making up the General program, as well as directions regarding the compilation and execution of the parallel benchmark.

An overview of the Mountaintop Program can be found in [Titi94]. Descriptions of SP2 message-passing operations can be found in [IBM94b]. More information about the message-passing performance of the SP2 can be found in [Xu95a]. Details of parallel application code development can be found in [Xu95b]. Additional algorithm analysis and performance results from this project can be found in [Arak95a, Hwan95a, Hwan95b, Hwan95c]. The paper [Hwan95c] presents a comprehensive report on the STAP benchmark performance results on the SP2. User's guides for the parallel APT (Adaptive Processing Testbed) and HO-PD (Higher-Order Post-Doppler) benchmarks can be found in [Arak95b, Arak95c].

Contents

1 Sequential General Program

1.1 Sorting Subprogram

While analyzing the original sequential program, we developed a higher-level program skeleton to help us visualize the structure of the benchmark. Program skeletons are much shorter and simpler to understand than the complete code, while still displaying all the control and data dependence information needed for parallelization. The key words **`in`**, **`out`**, and **`inout`** are similar to those used in the Ada programming language, and indicate that a subroutine parameter is read only, write only, or read/write, respectively. The array index notation `[.]` is used to indicate that all array elements along that dimension are to be used. The program skeletons were adapted from [Hwan95c]. Below, Figure 1.1 shows the program skeleton for the sequential sorting subprogram.

```
COMPLEX data_cube[DIM1][DIM2][DIM3], max_element;
float total[DIM1], average[DIM1];

/* the SRCH substep */
for (i = 0; i < DIM1; i++)
  for (j = 0; j < DIM2; j++)
    for (k = 0; k < DIM3; k++)
      find_max(inout max_element, in data_cube[i][j][k]);

/* the SORT substep */
for (j = 0; j < DIM2; j++)
  for (k = 0; k < DIM3 / 3; k++)
    sort_and_total(inout total[.], in data_cube[.][j][k]);
for (i = 0; i < DIM1; i++)
  average[i] = total[i] / (DIM2 * DIM3 / 3);
```

Figure 1.1: Sequential sorting subprogram skeleton

Not included in this program skeleton, but implicit to the program, is the data file access step. In this step, the program reads the input data cube and the steering vectors from disk. The input data cube is stored in a packed binary format; prior to use by the program, it must be converted into floating-point format (see Figure 1.2). The steering vectors consist of both primary and auxiliary steering vectors, and are stored on disk in ASCII format.

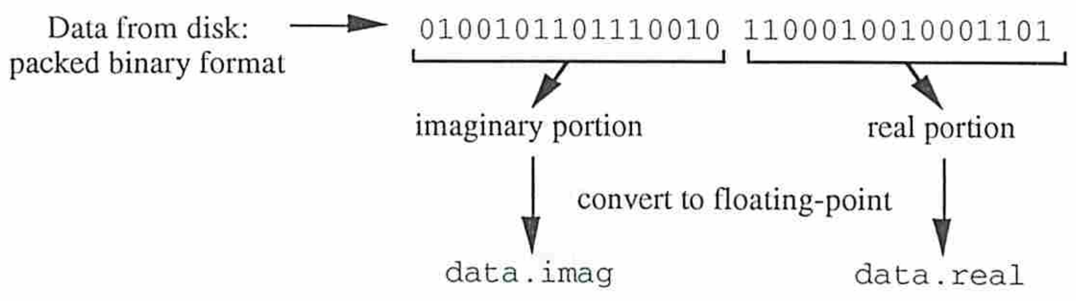

Figure 1.2: Packed binary format to floating-point format conversion

The subprogram first searches the entire data cube for the element with the largest power magnitude. After this searching step, the subprogram performs DIM2$\times$DIM3$\div$3 DIM1-element bubble sorts. Also during this step, the subprogram calculates DIM1 averages.

1.2 FFT Subprogram

We developed a program skeleton of the sequential subprogram while analyzing the original FFT benchmark, shown below in Figure 1.3.

```
COMPLEX data_cube[DIM1][DIM2][DIM3], doppler_cube[DIM1][DIM2][DIM3],
        twiddle;
float max_element;

compute_twiddle_factor(out twiddle, in DOP_GEN);

/* the FFT1 substep */
for (j = 0; j < DIM2; j++)
  for (k = 0; k < DIM3; k++)
    fft (in data_cube[.][j][k], in twiddle, out doppler_cube[.][j][k]);

/* the FFT2 substep */
for (i = 0; i < DIM1; i++)
  for (k = 0; k < DIM3; k++)
    fft (inout doppler_cube[i][.][k], in twiddle);

/* the FFT3 substep */
for (i = 0; i < DIM1; i++)
  for (j = 0; j < DIM2; j++)
    fft (inout doppler_cube[i][j][.], in twiddle);

/* the SRCH substep */
for (i = 0; i < DIM1; i++)
  for (j = 0; j < DIM2; j++)
    for (k = 0; k < DOP_GEN; k++)
      find_max(inout max_element, in data_cube[i][j][k]);
```

Figure 1.3: Sequential FFT subprogram skeleton

Not included in this program skeleton, but implicit to the program, is the input data file access step. The program reads the input data cube and the vectors from disk before any computation is done.

This subprogram performs FFTs along each of the three dimensions of the data cube: DIM2×DIM3 DIM1-point FFTs, then DIM1×DIM3 DIM2-point FFTs, then DIM1×DIM2 DIM3-point FFTs. After the FFTs, the subprogram searches the entire data cube for the element with the largest power magnitude.

1.3 Vector Multiply Subprogram

We developed a program skeleton of the sequential subprogram while analyzing the original vector multiply benchmark, shown below in Figure 1.4.

```
COMPLEX data_cube[DIM1][DIM2][DIM3], v1[DIM1], v2[DIM2], v3[DIM3];
float max_element1, max_element2, max_element3;

/* the VEC1 substep */
for (j = 0; j < DIM2; j++)
  for (k = 0; k < DIM3; k++)
    vec_multiply_and_max(in data_cube[.][j][k], in V1,
                         inout max_element1);

/* the VEC2 substep */
for (i = 0; i < DIM1; i++)
  for (k = 0; k < DIM3; k++)
    vec_multiply_and_max(in data_cube[i][.][k], in V2,
                         inout max_element2);

/* the VEC3 substep */
for (i = 0; i < DIM1; i++)
  for (j = 0; j < DIM2; j++)
    vec_multiply_and_max(in data_cube[i][j][.], in V3,
                         inout max_element3);
```

Figure 1.4: Sequential vector multiply subprogram skeleton

Not included in this program skeleton, but implicit to the program, is the input data file access step. The program reads the input data cube and the vectors from disk before any computation is done.

The subprogram performs three sets of vector multiplications. First, the subprogram multiplies vectors along the DIM1 dimension of the input data cube by the input vector V1. Then, the subprogram multiplies vectors along the DIM2 dimension of the input data cube by the input vector V2. Finally, the subprogram multiplies vectors along the DIM3 dimension of the input data cube by the input vector V3. During each set of multiplications, the subprogram searches for the element with the largest power magnitude.

1.4 Linear Algebra Subprogram

We developed a program skeleton of the sequential subprogram while analyzing the original linear algebra benchmark, shown below in Figure 1.5.

```
COMPLEX data_cube[DIM1][DIM2][DIM3], temp_mat[V4][COLS], weight_vec[V4],
        beams[NUM_MAT][DIM3];
float max_element;

for (j = 0; j < NUM_MAT; j++)
  {
    CHOL (in data_cube[.][j][.], out temp_mat);
    SUB (in temp_mat, out weight_vec);
    BF (in weight_vec, in data_cube[.][j][.], out beams[.][j][.]);
    SRCH (in beams[.][j][.], out max_element);
  }
```

Figure 1.5: Sequential linear algebra subprogram skeleton

Not included in this program skeleton, but implicit to the program, is the input data file access step. The program reads the input data cube and the vectors from disk before any computation is done.

The subprogram first computes NUM_MAT number of Cholesky factors using the first NUM_MAT number of $DIM1 \times DIM3$ matrices from the input data set. Then, the subprogram forward and back solves these Cholesky factors using the vector V4 as a steering vector to generate NUM_MAT number of weight vectors. These weight vectors are applied to their corresponding $DIM1 \times DIM3$ matrices from the input data cube to generate NUM_MAT number of output beams of length DIM3. The subprogram also searches for the element with the largest magnitude in each output beam.

2 Parallel General Program Development

In parallelizing the General program, we adopted the method described in [Xu95b]. This method (see Figure 2.1), which entails an early MPP performance prediction scheme, significantly reduced the parallel software development cost

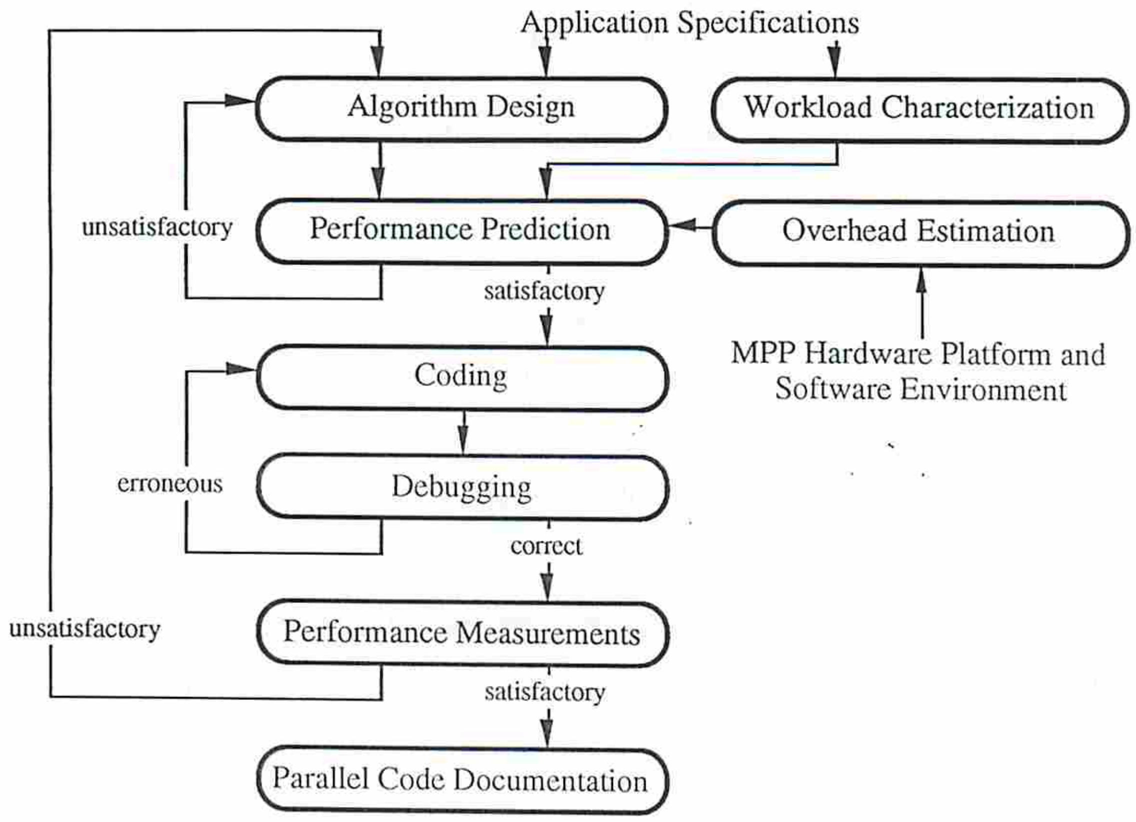

Figure 2.1: Early performance prediction based on workload and overhead characterization

During the algorithm design phase of the parallel program development, we considered several parallelization strategies [Hwan95c]. The nature of the algorithm in the General program made the simple compute-interact paradigm sufficient. In this paradigm, the parallel program is written as a sequence of alternating computation and interaction steps. During a computation step, each task performs computations independently. During an interaction step, the tasks exchange data or synchronize. Each of the four subprograms in the General benchmark can be described as a sequence of compute-interact steps.

2.1 Sorting Subprogram

The sorting subprogram can be described as the following sequence of compute-interact steps:

Compute:	Search data cube slice for local maximum
Interact:	Reduce to find global maximum
Compute:	Bubble sort data cube slice Search data cube slice for local maximum
Interact:	Reduce to find global maximum

The mapping of the parallel algorithm and the data set onto the SP2 is shown in Figure 2.2.

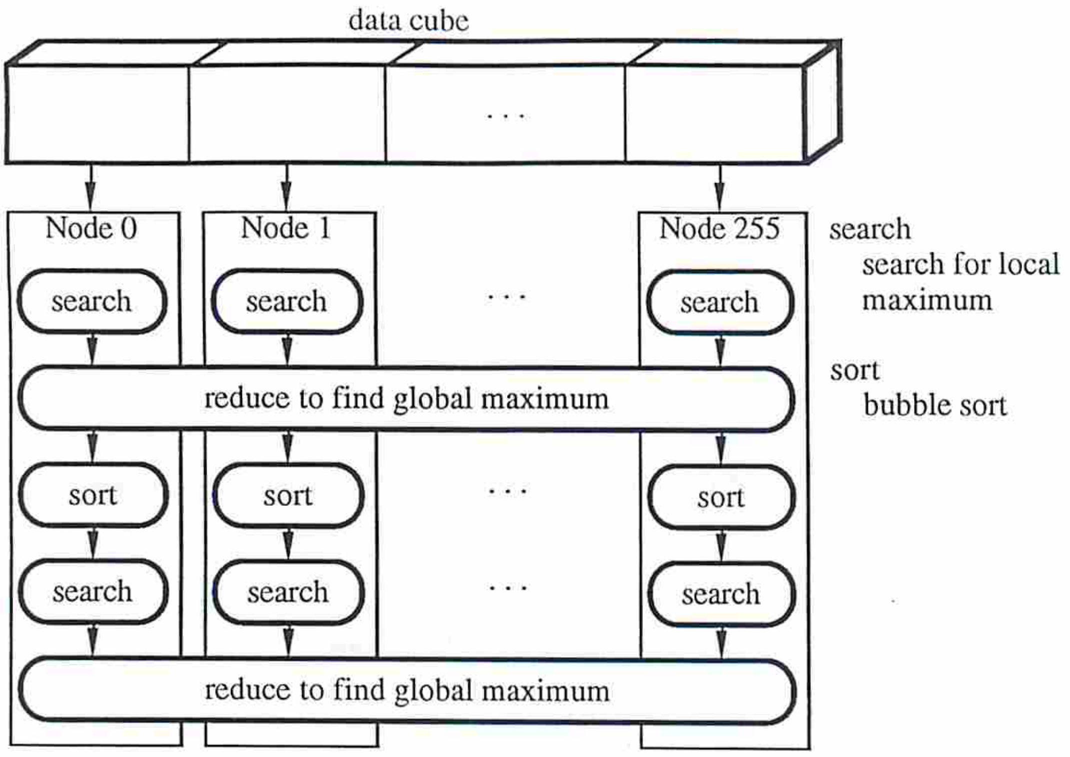

Figure 2.2: Mapping of the parallel sorting subprogram and data set onto the SP2

To help illustrate the structure of the parallel sorting subprogram, I have included the program skeleton in Figure 2.3.

```
COMPLEX data_cube[DIM1][DIM2][DIM3];
float total[DIM1], average[DIM1];
struct { float power; int loc1, loc2, loc3; }
  max_element;

parfor (p = 0; p < n; p++)
  {
    COMPLEX local_max;
    float local_total[DIM1];

    /* the SRCH substep */
    for (i = 0; i < DIM1; i++)
      for (j = 0; j < DIM2; j++)
        for (k = p; k < DIM3; k = k + n)
          find_max (inout local_max, in data_cube[i][j][k]);

    reduce local_maxs to max_element;

    /* the SORT substep */
    for (j = 0; j < DIM2; j++)
      for (k = p; k < DIM3 / 3; k = k + n)
        sort_and_total (inout local_total[.], in data_cube[.][j][k]);
  }

reduce local_totals to total;
```

Figure 2.3: Parallel sorting subprogram skeleton

At the beginning of the parallel subprogram, each task reads its portion of the data cube from disk in parallel. In order to improve disk access performance, we modified the `read_input_SORT_FFT` procedure (see Appendix) so each task reads its entire slice of the data cube from disk before converting the data from a packed binary format (in which the data is stored on disk) to floating-point format (in which computations are done). In the sequential version, each complex number is read then converted immediately. Because the execution time of this step is not included in the total execution time of the parallel program, the performance of this step is not critical: the timer is started after the data is loaded and converted.

First, each task searches its slice of the data cube to find the element with the largest power magnitude. To find the global maximum, we perform the MPL call mpc_reduce. Because we need to keep track of the location of the element along with its power, we created the INDEXED_MAX data structure. By placing the coordinates of the maximum element in a single data structure along with the power, we can let the mpc_reduce call handle the entire global maximum search operation, thus improving program performance.

Each task then bubble sorts its slice of the data cube along the DIM1 dimension. The subprogram also accumulates the local total during the bubble sorts. To find the global totals, we again use the MPL mpc_reduce call.

In our implementation, we combined the sorting and FFT subprograms into one subprogram.

2.2 FFT Subprogram

The FFT subprogram can be described as the following sequence of compute-interact steps:

Compute:	First and second set of FFTs
Interact:	Total Exchange
Compute:	Third set of FFTs Search data cube slice for local maximum
Interact:	Reduce to find global maximum

The mapping of the parallel algorithm and the data set onto the SP2 is shown in Figure 2.4.

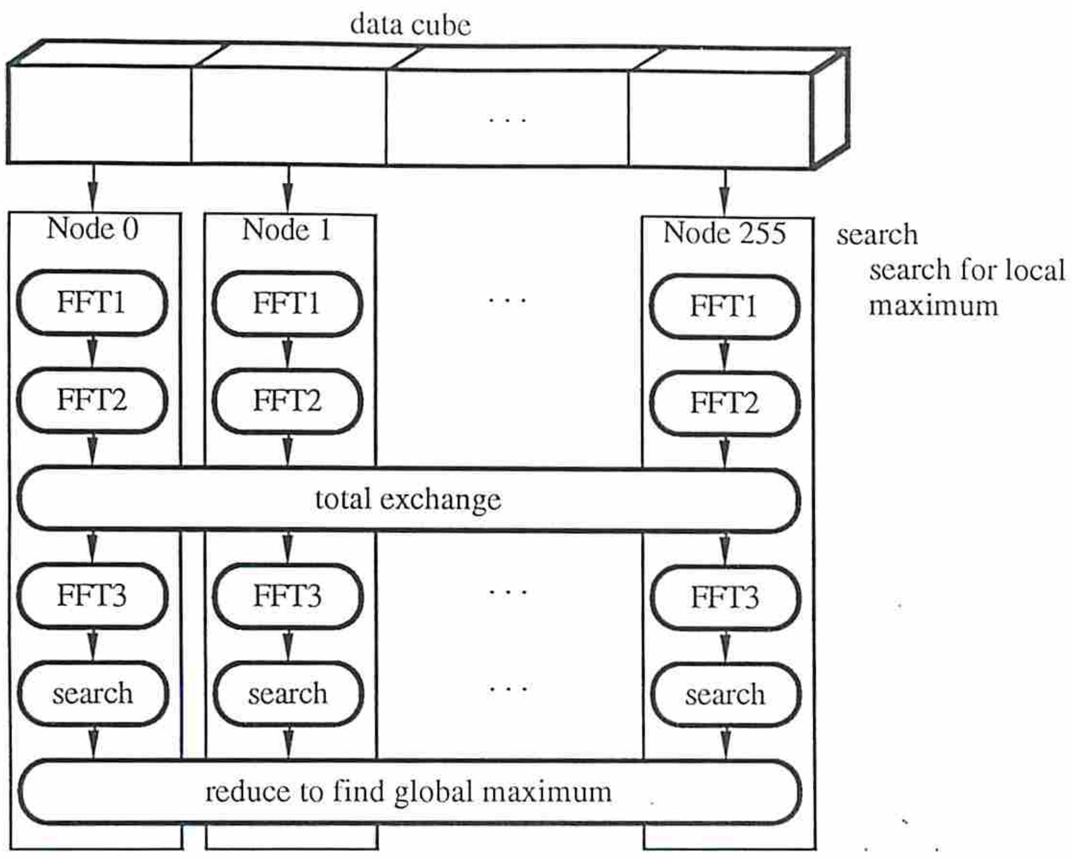

Figure 2.4: Mapping of the parallel FFT subprogram and data set onto the SP2

To help illustrate the structure of the parallel FFT subprogram, I have included the program skeleton in Figure 2.5.

```
COMPLEX data_cube[DIM1][DIM2][DIM3], doppler_cube[DIM1][DIM2][DIM3],
        twiddle[DOP_GEN];
float local_max, max_element;

parfor (k = 0; k < DIM3; k++)
  {
    compute_twiddle_factor (out twiddle, in DOP_GEN);

    /* the FFT1 substep */
    for (j = 0; j < DIM2; j++)
      fft (in data_cube[.][j][k], in twiddle,
           out doppler_cube[.][j][k]);

    /* the FFT2 substep */
    for (i = 0; i < DIM1; i++)
      fft (inout doppler_cube[i][.][k], in twiddle);
  }

total_exchange doppler_cube[.][.][k] to doppler_cube[.][j][.];

parfor (j = 0; j < DIM2; j++)
  {

    /* the FFT3 substep */
    for (i = 0; i < DIM1; i++)
      fft (inout doppler_cube[i][j][.], in twiddle);
  }

/* the SRCH substep */
parfor (k = 0; k < DOP_GEN; k++)
  {
    COMPLEX local_max;
    for (i = 0; i < DIM1; i++)
      for (j = 0; j < DIM2; j++)
        find_max (inout local_max, in doppler_cube[i][j][k]);
  }

reduce local_maxs to max_element;
```

Figure 2.5: Parallel FFT subprogram skeleton

Like the sorting subprogram, the FFT subprogram loads the data cube from disk in parallel prior to any computation.

The FFT subprogram begins by performing DIM2×DIM3 DIM1-point FFTs. Each task performs these FFTs along the DIM1 dimension on its slice of the data cube in parallel. Next, each task performs FFTs along the DIM2 dimension in parallel.

Before the third set of FFTs can be performed, the subprogram must perform a total exchange operation. This data redistribution is necessary because each task loads a slice of data from the disk which has all the elements along the DIM1 and DIM2 dimensions but only a fraction of the elements along the DIM3 dimension, and this last set of FFTs is performed along the DIM3 dimension.

After the last set of FFTs are performed, each task searches its data cube slice for the element with the largest magnitude. As was done in the sorting subprogram, an `mpc_reduce` call is used to find the global maximum.

2.3 Vector Multiply Subprogram

The vector multiply subprogram can be described as the following sequence of compute-interact steps:

Compute:	First set of vector multiplications Search data cube slice for local maximum
Interact:	Reduce to find global maximum
Compute:	Second set of vector multiplications Search data cube slice for local maximum
Interact:	Reduce to find global maximum Total exchange
Compute:	Third set of vector multiplications Search data cube slice for local maximum
Interact:	Reduce to find global maximum

The mapping of the parallel algorithm and the data set onto the SP2 is shown in Figure 2.6.

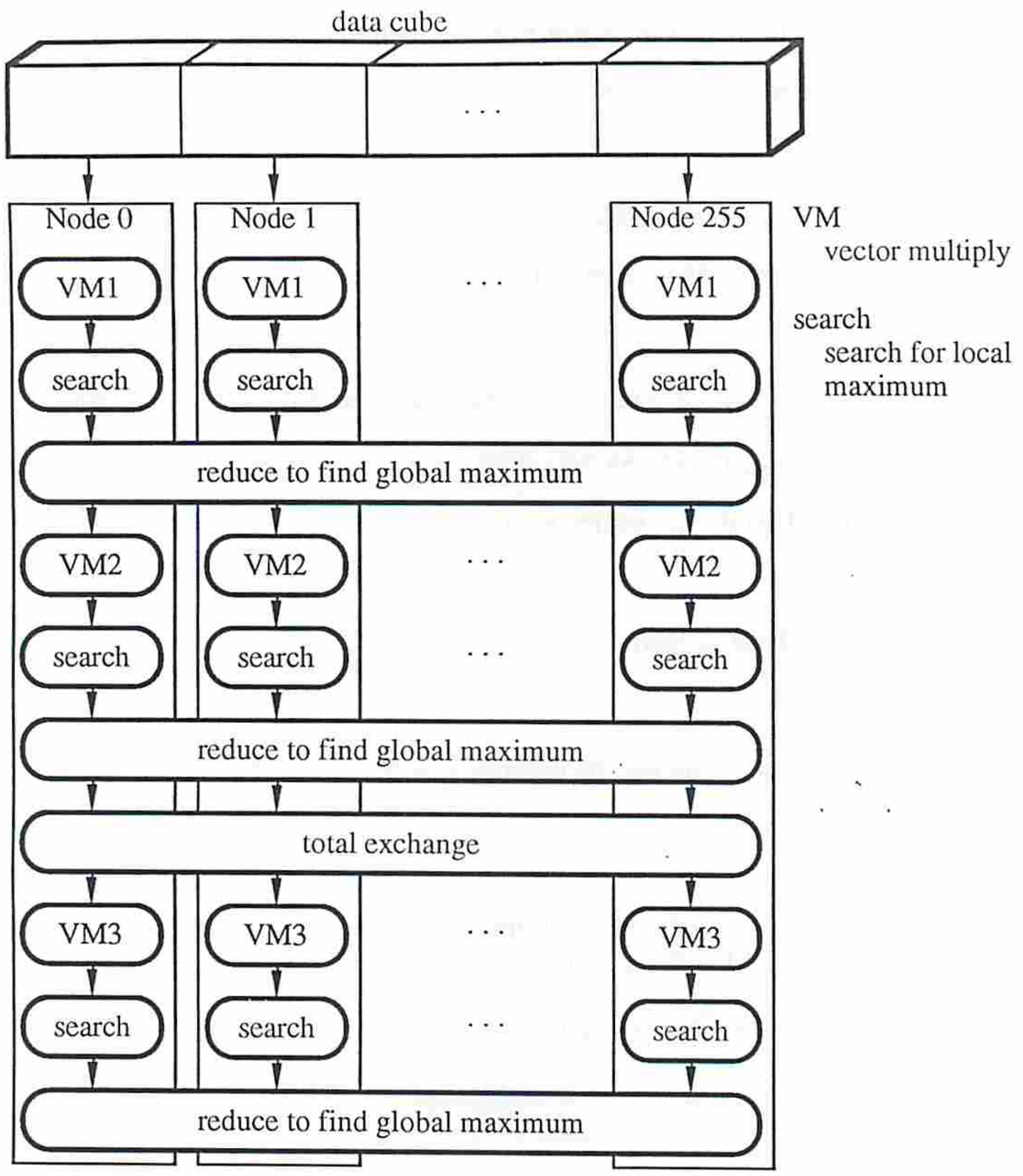

Figure 2.6: Mapping of the parallel vector multiply subprogram and data set onto the SP2

To help illustrate the structure of the parallel vector multiply subprogram, I have included the program skeleton in Figure 2.7.

```
parfor (k = 0; k < DIM3; k++)
  {

    /* the VEC1 substep */
    for (j = 0; j < DIM2; j++)
      vec_multiply_and_max(in data_cube[.][j][k], in V1,
                           inout local_max);
    reduce local_maxs to max_element1;

    /* the VEC2 substep */
    for (i = 0; i < DIM1; i++)
      vec_multiply_and_max(in data_cube[i][.][k], in V2,
                           inout local_max);
    reduce local_maxs to max_element2;
  }

total_exchange data_cube[.][.][k] to doppler_cube[.][j][.];

parfor (j = 0; j < DIM2; j++)
  {

    /* the VEC3 substep */
    for (i = 0; i < DIM1; i++)
      vec_multiply_and_max(in data_cube[i][j][.], in V3,
                           inout local_max);
    reduce local_maxs to max_element3;
  }
```

Figure 2.7: Parallel vector multiply subprogram skeleton

Like the sorting and FFT subprograms, the vector multiply subprogram loads the data cube from disk in parallel prior to any computation.

First, the subprogram performs DIM2×DIM3 DIM1-element vector multiplications. After these multiplications, each task searches its data cube slice for the element with the largest magnitude, then aggregates the local maximums into a global maximum using the `mpc_reduce` operation. Next, the subprogram performs vector multiplications along the DIM2 dimension, and finds the global maximum again.

Before vector multiplications can be done along the DIM3 dimension, the subprogram must perform a total exchange operation to redistribute the data cube. This operation is necessary because, prior to the total exchange, each task only has part of the data cube along the DIM3 dimension. After this data redistribution, each task can perform the vector multiplications on its portion of the data cube along the DIM3 dimension, and find the global maximum.

2.4 Linear Algebra Subprogram

The linear algebra subprogram consists of only one computation step, and no interaction steps. As such, the only difference between the sequential and parallel subprograms is the fact that each task in the parallel subprogram performs the linear algebra steps on only a slice of the data cube.

3 The Parallel General Code

The parallel program has been divided into the following files, listed below by subprogram:

Sorting & FFT	
`bench_mark_SORT.c`	`main ()` for sorting only
`bench_mark_SORT_FFT.c`	`main ()` for sorting and FFT
`bubble_sort.c`	`bubble_sort ()`
`cmd_line.c`	`cmd_line ()`
`fft.c`	`fft (), bit_reverse ()`
`read_input_SORT_FFT.c`	`read_input_SORT_FFT ()`
Vector Multiply	
`bench_mark_VEC.c`	`main ()` for vector multiply
`cmd_line.c`	`cmd_line ()`
`read_input_VEC.c`	`read_input_VEC ()`

Linear Algebra

`bench_mark_LIN.c`	`main ()` for linear algebra
`cmd_line.c`	`cmd_line ()`
`forback.c`	`forback ()`
`house.c`	`house ()`
`read_input_LIN.c`	`read_input_LIN ()`

The purpose of each procedure is briefly described below:

Sorting & FFT

`bench_mark_SORT ()`	This procedure represents the body of the parallel sorting subprogram.
`bench_mark_SORT_FFT ()`	This procedure represents the body of the parallel sorting and FFT subprograms.
`bubble_sort ()`	This procedure performs a bubble sort, and also calculates the average value of the sorts. The data is sorted by the magnitude of the power.
`cmd_line ()`	This procedure extracts the names of the data file and the vector file from the command line
`fft ()`	This procedure performs a single n-point in-place decimation-in-time complex FFT using n/2 complex twiddle factors. The implementation used in this procedure is a hybrid between the implementation in the original sequential version of this program and the implemented suggested by MHPCC. This modification was made to improve the performance of the FFT on the SP2.
`bit_reverse ()`	This procedure performs a simple but somewhat inefficient bit reversal. This procedure is used by `fft ()`.
`read_input_SORT_FFT ()`	This procedure reads the node's slice of the data cube from disk, then converts the data from the packed binary integer form in which it is stored on disk into floating-point format.

Vector Multiply

`bench_mark_VEC ()`	This procedure represents the body of parallel vector multiply subprogram.
`cmd_line ()`	This procedure extracts the names of the data file and the vector file from the command line.

`read_input_VEC ()`	This procedure reads the node's slice of the data cube from disk, then converts the data from the packed binary integer form in which it is stored on disk into floating-point format.

Linear Algebra

`bench_mark_LIN ()`	This procedure represents the body of parallel linear algebra subprogram.
`cmd_line ()`	This procedure extracts the names of the data file and the vectors file from the command line.
`forback ()`	This procedure performs a forward and back substitution on an input array using the steering vectors, and normalizes the returned solution vector.
`house ()`	This procedure performs an in-place Householder transformation on a complex $N \times M$ input array, where $M \geq N$. The results are returned in the same matrix as the input data.
`read_input_LIN ()`	This procedure reads the node's slice of the data cube from disk, then converts the data from the packed binary integer form in which it is stored on disk into floating-point format.

Figures 3.1 through 3.4 show the calling relationship between the procedures for the subprograms.

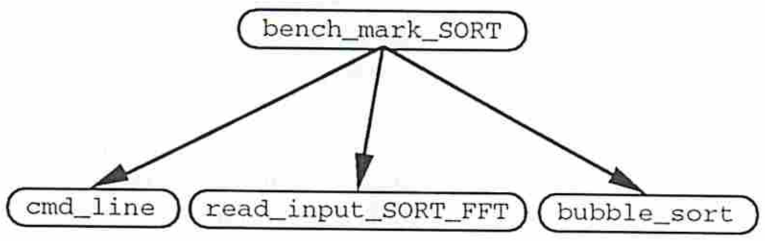

Figure 3.1: Calling relationship between the procedures in the parallel sorting subprogram

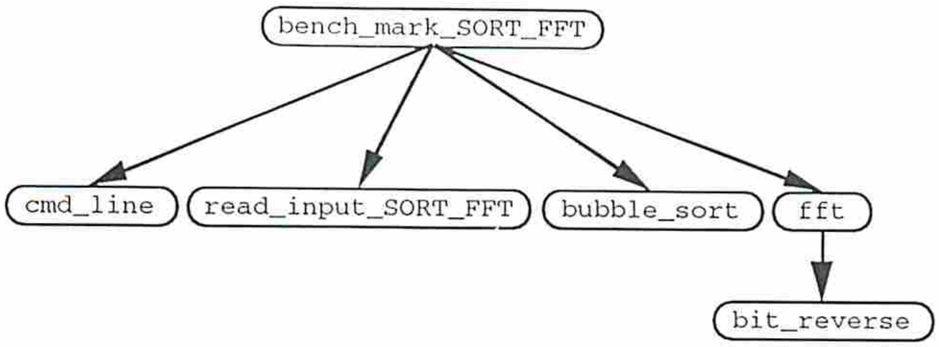

Figure 3.2: Calling relationship between the procedures in the parallel sorting and FFT subprogram

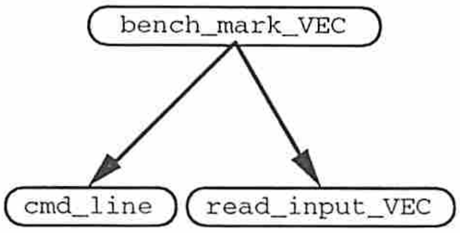

Figure 3.3: Calling relationship between the procedures in the parallel vector multiply subprogram

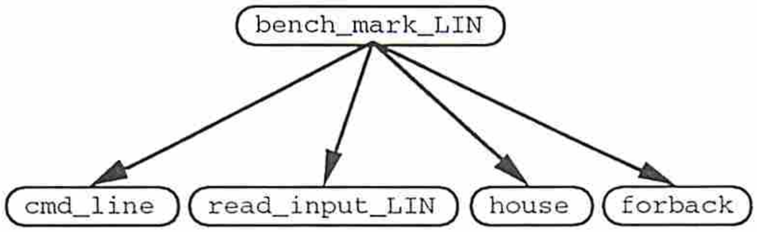

Figure 3.4: Calling relationship between the procedures in the parallel linear algebra subprogram

The complete code listings for the IBM SP2 can be found in the Appendix.

4 The General Data Files

The data set for the General benchmark is a DIM1 ×DIM2×DIM3 cube. Nominally, DIM1 = 64, DIM2 = 128, and DIM3 = 1536. On disk, this data file is 96 MB.

The original input data cube file gen_data.dat has been reordered and saved in several versions. Each of these reorderings are designed to optimize the process of reading the data cube from disk in parallel. The parallel vectors file is identical to the original vectors file gen_data.str.

5 Compiling the Parallel General Program

The individual subprograms which comprise the parallel General benchmark: sorting and FFT, vector multiply, and linear algebra, are compiled by calling the script files compile_SORT_FFT, compile_VEC, and compile_LIN, respectively. The script files execute the following shell commands:

```
compile_SORT_FFT:
      mpcc -O3 -qarch=pwr2 -DGEN -DIBM -o gen bench_mark_SORT_FFT.c
        bubble_sort.c cmd_line.c fft.c read_input_SORT_FFT.c  -lm

compile_VEC:
      mpcc -O3 -qarch=pwr2 -DGEN -DIBM -o vec bench_mark_VEC.c
        cmd_line.c read_input_VEC.c  -lm

compile_LIN:
      mpcc -O3 -qarch=pwr2 -DGEN -DIBM -o lin bench_mark_LIN.c
        cmd_line.c forback.c house.c read_input_LIN.c  -lm
```

The -qarch=pwr2 option directs the compiler to generate POWER2-efficient code. The -O3 option selects the highest level of compiler optimization available on the mpcc parallel C code compiler. The define flag -DGEN must be included so that the compiler selects the proper data cube dimensions in the defs.h header file. The define flag -DIBM must be included so that the compiler can set the number of clock ticks per second (100 ticks per second on IBM machines, 60 ticks per second on SUN machines). To compile the program to use double-precision floating-point numbers instead of the default single-precision, add the define flag -DDBLE.

Prior to compiling the program, the user must set the number of nodes on which the program will be run. This number is defined as NN in the header file defs.h. This step is necessary because it is not possible to statically allocate arrays with their sizes defined by a variable in C. We have provided a set of header files with NN set to powers of two from 1 to 256. These header files are called defs.001 through defs.256.

6 Running the Parallel General Program

The parallel General benchmark is invoked by calling the script file run.###, where ### is the number of nodes on which the program will be run (and is equal to NN in defs.h). This script file executes the following shell command:

```
poe program_name data_filename -procs ### -us
```

The argument program_name is the name of the subprogram to be run: gen (for the sorting and FFT subprogram), vec (for the vector multiply subprogram), and lin (for the linear algebra subprogram). The command poe distributes the parallel executable file to all nodes and invokes the parallel program. The command line argument data_filename is the name of the files containing the input data cube and the vectors. The input data cube file must have the extension .dat, while the vectors file must have the extension `.str`. The command line arguments `-procs` and `-us` are flags to `poe`. The flag `-procs` sets the number of nodes on which the program will be run. The flag `-us` directs `poe` to use the User Space communication library. This library gives the best communication performance. We have provided a set of script files to run the General benchmark, with the number of nodes equal to powers of two from 1 to 256. These files are called `run.001` through `run.256`.

The subprograms in the General benchmark do not have any real outputs; however, to confirm the correctness of the parallel algorithms, the program displays intermediate results on the screen. In addition, timing reports are displayed on the screen. Timing reports from the sort and FFT subprogram, the vector multiply subprogram, and the linear algebra subprogram are shown in Figures 6.1 through 6.3, respectively.

In Figures 6.1 through 6.3, the timing entries ending with `_user_max` represent the largest amount of user CPU time spent by a node while performing that step, while the timing entries ending with `_sys_max` represent the largest amount of system CPU time spent by a node while performing that step. The `Wall clock timing` entries indicate the amount of wall clock time spent on a given step, regardless of whether the time was spent by the program or by other programs.

```
0:*** CPU Timing information - numtask = 128
0:
0:   all_user_max       = 2.38 s, all_sys_max      = 0.06 s
0:   disk_user_max      = 0.10 s, disk_sys_max     = 0.02 s
0:   max1_user_max      = 0.02 s, max1_sys_max     = 0.00 s
0:   before_user_max     = 0.01 s, before_sys_max     = 0.00 s
0:   sort_user_max      = 0.75 s, sort_sys_max     = 0.00 s
0:   sort_total_user_max      = 0.02 s, sort_total_sys_max      = 0.00 s
0:   fft1_user_max      = 0.10 s, fft1_sys_max     = 0.00 s
0:   fft2_user_max      = 0.19 s, fft2_sys_max     = 0.01 s
0:   fft3_user_max      = 0.19 s, fft3_sys_max     = 0.01 s
0:   index_user_max      = 0.17 s, index_sys_max     = 0.00 s
0:   max2_user_max      = 0.02 s, max2_sys_max     = 0.01 s
0:   after_user_max      = 0.01 s, after_sys_max     = 0.00 s
0:
0:*** Wall Clock Timing information - numtask = 128
0:
0:   all_wall          = 2529295 us
0:   task_wall          = 502 us
0:   disk_wall          = 1631067 us
0:   max1_wall          = 32853 us
0:   before_wall        = 3375 us
0:   sort_wall          = 165641 us
0:   sort_total_wall    = 12129 us
0:   fft1_wall          = 110256 us
0:   fft2_wall          = 199256 us
0:   fft3_wall          = 215477 us
0:   index_wall         = 171785 us
0:   max2_wall          = 32516 us
0:   after_wall         = 10493 us
```

Figure 6.1: Timing report from the parallel sort and FFT subprogram

```
0:*** CPU Timing information - numtask = 128
0:
0:   all_user_max       = 1.96 s, all_sys_max      = 0.05 s
0:   disk_user_max      = 0.29 s, disk_sys_max     = 0.05 s
0:   vec1_user_max      = 8084644.00 s, vec1_sys_max     = 8084547.00 s
0:   vec2_user_max      = 0.77 s, vec2_sys_max     = 0.00 s
0:   vec3_user_max      = 0.06 s, vec3_sys_max     = 0.00 s
0:   index_user_max      = 0.17 s, index_sys_max     = 0.00 s
0:
0:*** Wall Clock Timing information - numtask = 128
0:
0:   all_wall          = 2156784 us
0:   task_wall          = 507 us
0:   disk_wall          = 1876482 us
0:   vec1_wall          = 1450517 us
0:   vec2_wall          = 58877 us
0:   vec3_wall          = 50551 us
0:   index_wall         = 182361 us
```

Figure 6.2: Timing report from the parallel vector multiply program

```
0:*** CPU Timing information - numtask = 16
0:
0:   all_user_max      = 7.13 s, all_sys_max      = 0.13 s
0:   disk_user_max     = 0.36 s, disk_sys_max     = 0.13 s
0:   lin_user_max     = 6.82 s, lin_sys_max     = 0.01 s
0:
0:*** Wall Clock Timing information - numtask = 16
0:
0:   all_wall         = 7440370 us
0:   task_wall         = 30867 us
0:   disk_wall         = 1766285 us
0:   lin_wall         = 5566332 us
```

Figure 6.3: Timing report from the parallel linear algebra program

The steps we timed are listed below:

sort and FFT subprogram:

`all`	This component is the end-to-end execution time of the entire subprogram. This time is not equal to the sum of the times of the other steps, because this `all` time includes time spent waiting for synchronization not included in the individual steps.
`disk`	This component is the time to read the node's portion of the data cube from disk.
`max1`	This component is the time spent searching the local portion of the data cube for the largest element.
`before`	This component is the time spent performing a reduction operation to find the global maximum.
`sort`	This component is the time spent performing bubble sorts on the local portion of the data cube.
`sort_total`	This component is the time spent performing a reduction operation to collect the totals from all the nodes.
`fft1`	This component is the time spent performing the first FFT step.
`fft2`	This component is the time spent performing the second FFT step.
`fft3`	This component is the time spent performing the third FFT step.
`index`	This component is the time spent performing the total exchange operation.
`max2`	This component is the time spent searching the local portion of the data cube for the largest element after the FFTs.
`after`	This component is the time spent performing a reduction operation to find the global maximum.

vector multiply subprogram:

`all`	This component is the end-to-end execution time of the entire subprogram. This time is not equal to the sum of the times of the other steps, because this `all` time includes time spent waiting for synchronization not included in the individual steps.
`disk`	This component is the time to read the node's portion of the data cube from disk.
`vec1`	This component is the time spent performing the first vector multiply step.
`vec2`	This component is the time spent performing the second vector multiply step.
`vec3`	This component is the time spent performing the third vector multiply step.
`index`	This component is the time spent performing the total exchange operation.

linear algebra subprogram:

`all`	This component is the end-to-end execution time of the entire subprogram. This time is not equal to the sum of the times of the other steps, because this `all` time includes time spent waiting for synchronization not included in the individual steps.
`disk`	This component is the time to read the node's portion of the data cube from disk.
`lin`	This component is the time spent performing all the linear algebra computation steps.

The time spent in these steps (T_{max1}, T_{before}, etc.) are combined to determine the total execution time of the individual parallel General subprograms:

$$T_{sort\_execution} = T_{max1} + T_{before} + T_{sort} + T_{sort\_total} \tag{6.1}$$

$$T_{fft\_execution} = T_{fft1} + T_{fft2} + T_{fft3} + T_{index} + T_{max2} + T_{after} \tag{6.2}$$

$$T_{vec\_execution} = T_{vec1} + T_{vec2} + T_{vect3} + T_{index} \tag{6.3}$$

$$T_{lin\_execution} = T_{lin} \tag{6.4}$$

The sort subprogram's workload consists of 1.18 billion floating-point operations. The FFT subprogram's workload consists of 1.91 billion floating-point operations. The vector multiply subprogram's workload consists of 0.60 billion floating-point operations. The linear algebra subprogram's workload consists of 1.60 billion floating-point operations. By dividing the individual workloads by the corresponding total execution time, we can calculate the sustained floating-point processing rate:

$$\text{Sustained Processing Rate} = \frac{\text{Floating-point workload}}{T_{execution}} \tag{6.2}$$

We define the system efficiency as the ratio between the sustained processing rate and the peak processing rate:

$$\text{System Efficiency} = \frac{\text{Sustained Processing Rate}}{\text{Peak Processing Rate}} \tag{6.3}$$

As was the case in the APT and HO-PD programs, we used both the CPU time and the wall clock time to more accurately measure the execution time of each step. To write these reports to a file, use standard UNIX output redirection:

```
run.### > out_filename
```

Both the parallel C source code and the data files have been stored on tape. Specifically, we archived them using the following UNIX command:

```
tar cvf /dev/rst8 commented
```

where `commented` is the name of the directory in which all the files are stored. The files can be extracted from the tape using the following UNIX command:

```
tar xvf /dev/rst8
```

The device specification (`/dev/rst8`) is machine specific.

Bibliography

[Arak95a] M. Arakawa, "Parallel STAP Benchmarks and Their Performance on the IBM SP2", Master's thesis, School of Engineering at the University of Southern California, August 1995

[Arak95b] M. Arakawa, Z. Xu, K. Hwang, "User's Guide and Documentation of the Parallel APT Benchmark on the IBM SP2", *Technical Report*, University of Southern California, June 1995

[Arak95c] M. Arakawa, Z. Xu, K. Hwang, "User's Guide and Documentation of the Parallel HO-PD Benchmark on the IBM SP2", *Technical Report*, University of Southern California, June 1995

[Hwan95a] K. Hwang, Z. Xu, M. Arakawa, "STAP Benchmark Performance on the IBM SP2 Massively Parallel Processor", *Proceedings of the Adaptive Sensor Array Processing Workshop 1995*, MIT Lincoln Laboratory, March 15-17, 1995, p. 75-91

[Hwan95b] K. Hwang, Z. Xu, M. Arakawa, "STAP Benchmark Performance of the IBM SP2 for Real-Time Signal Processing", submitted to *ACM/IEEE Supercomputing Conference '95, San Diego*, April 1, 1995

[Hwan95c] K. Hwang, Z. Xu, M. Arakawa, "Benchmark Evaluation of the IBM SP2 for Parallel Signal Processing", submitted to *IEEE Transactions on Parallel and Distributed Systems*, May 11, 1995

[IBM94a] IBM Corp., *AIX Parallel Environment: Programing Primer*, Release 2.0, Pub. No. SH26-7223, IBM Corp., June 1994

[IBM94b] IBM Corp., *IBM AIX Parallel Environment: Parallel Programming Subroutine Reference*, Release 2.0, Pub. No. SH26-7228-01, IBM Corp., June 1994

[LL94] MIT Lincoln Laboratory, *Commercial Programmable Processor Benchmarks*, MIT Lincoln Laboratory, February 28, 1994

[Stun94] C. B. Stunkel, D. G. Shea, B. Abali, M. Atkins, C. A. Bender, D. G. Grice, P. H. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stucke, M. Tsao, P. R. Varker, "The SP2 Communication Subsystem", Technical Report, IBM Thomas J. Watson Research Center & IBM Highly Parallel Supercomputing Systems Laboratory, August 22, 1994

[Titi94] G. W. Titi, "An Overview of the ARPA/NAVY Mountaintop Program", *IEEE Adaptive Antenna Systems Symposium*, November 7-8, 1994

[Xu95a] Z. Xu, K. Hwang, "Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2 Multicomputer", submitted to *IEEE Parallel & Distributed Technology*, January 10, 1995

[Xu95b] Z. Xu, K. Hwang, "Early Prediction of MPP Performance by Workload and Overhead Quantification: A Case Study of the IBM SP2 System", submitted to *Parallel Computing*, April 1995

Appendix Parallel General Code

The parallel code for the General benchmark, along with the script to compile and run the parallel General program, are given in this appendix.

A.1 Sorting and FFT Subprogram

A.1.1 bench_mark_SORT.c

```
/*
 * bench_mark_SORT.c
 */

/*
 * Parallel General Benchmark Program for the IBM SP2
 * ------------------------------------------------
 *
 * This parallel General benchmark program was written for the IBM SP2 by the
 * STAP benchmark parallelization team at the University of Southern
 * California (Prof. Kai Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as
 * part of the ARPA Mountaintop program.
 *
 * The sequential General benchmark program was originally written by Tony
 * Adams on 8/30/93.
 *
 *
 * This file contains the procedure main (), and represents the body of the
 * General Sorting subprogram.
 *
 * This subprogram's overall structure is as follows:
 *    1. Load data in parallel.
 *    2. Each task searches its slice of the data cube for the largest element.
 *    3. Perform a reduction operation to find the largest element in the
 *       entire data cube.
 *    4. Each task sorts its slice of the data cube along the DIM1 dimension.
 *    5. Collect the totals calculated in the bubble sort operation using a
 *       reduction operation.
 *
 * This program can be compiled by calling the shell script compile_sort.
 * Because of the nature of the program, before the program is compiled, the
 * header file defs.h must be adjusted so that NN is set to the number of
 * nodes on which the program will be run. We have provided a defs.h file for
 * each power of 2 node size from 1 to 256, called defs.001 to defs.256.
 *
 * This program can be run by calling the shell script run.###, where ### is
 * the number of nodes on which the program is being run.
 *
 * Unlike the original sequential version of this program, the parallel General
 * program does not support command-line arguments specifying the number of
 * times the program body should be executed, nor whether or not timing
 * information should be displayed. The program body will be executed once,
 * and the timing information will be displayed at the end of execution.
 *
 * The input data file on disk is stored in a packed format: Each complex
 * number is stored as a 32-bit binary integer; the 16 LSBs are the real
 * half of the number, and the 16 MSBs are the imaginary half of the number.
 * These numbers are converted into floating point numbers usable by this
 * benchmark program as they are loaded from disk.
 *
 * The steering vectors file has the data stored in a different fashion. All
 * data in this file is stored in ASCII format. The first two numbers in this
 * file are the number of PRIs in the data set and the threshold level,
 * respectively. Then, the remaining data are the steering vectors, with
 * alternating real and imaginary numbers.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"
#include <sys/types.h>
#include <sys/times.h>
#include <mpproto.h>
#include <sys/time.h>
#include <time.h>



/*
 * main ()
 *    inputs: argc, argv
 *    outputs: none
 */

main (argc, argv)
   int argc;
   char *argv[];
{

/*
 * Variables
 *
 * data_cube: input data cube
 * t1, t2: holder for the time function return in seconds
 * dim1, dim2, dim3: dimensions 1, 2, and 3 in the input data cube
 * loc1, loc2, loc3: holders for the location of the maximum entry
 * temp: buffer for binary input integerdata
 * f_temp1, f_temp2: holders for float data
 * f_pwr: holder for float power data
 * i, j, k, n, offset, r: loop counters
 * max: float variable to hold the maximum of complex data
 * index_max, g_index_max: data structures with the location and power of the
 *    largest element (locally and globally, respectively)
 * data_name, str_name: the names of the files containing the input data cube
 *    and the vectors, respectively
 * totals: holders for DIM1 totals
 * global_totals: holders for global DIM1 totals
 * logpoints: log base 2 of the number of points in the FFT
 * sum: a COMPLEX variable to hold the sum of complex data
 * points: number of points in the FFT
 * pix2: pi x 2 (2 x 3.14159)
 * pi: pi (3.14159)
 * x: storage for temporary FFT vector
 * w: storage for twiddle factors table
 * blklen: block length
 * rc: return code from an MPL call
 * taskid: identifier for this task
 * numtask: the number of tasks running this program
 * nbuf: a buffer to hold data returned by the mpc_task_query call
 * allgrp: an identifier for the group encompassing all tasks running this
 *    program
 * source, dest, type, nbytes: used by point-to-point communications
 */

  COMPLEX data_cube[DIM2][DIM1][DIM3];
```

```
    struct tms t1,t2;
    int dim1 = DIM1;
    int dim2 = DIM2;
    int dim3 = DIM3;
    int loc1,loc2,loc3;
    unsigned int temp[1];
    float f_temp1, f_temp2;
    float f_pwr;
    int i, j, k, n, offset, r;
    float max;
    INDEXED_MAX index_max[1], g_index_max[1];
    char data_name[LINE_MAX];
    char str_name[LINE_MAX];
    FILE *fopen();
    FILE *f_dop;
    float totals[DIM1];
    float global_totals[DIM1];
    int logpoints;
    COMPLEX sum;
    int points = DOP_GEN;
    float pix2, pi;
    static COMPLEX x[DOP_GEN];
    static COMPLEX w[DOP_GEN];
    int blklen;
    int rc;
    int taskid;
    int numtask;
    int nbuf[4];
    int allgrp;
    int source, dest, type, nbytes;

/*
 * Timing variables
 */

    struct timeval tv0, tv1, tv2;
    struct tms all_start, all_end;
    float all_user, all_sys;
    float all_user_max, all_sys_max;
    float all_wall, task_wall;

    struct tms disk_start, disk_end;
    float disk_user, disk_sys;
    float disk_user_max, disk_sys_max;
    float disk_wall, disk_wall_max;

    struct tms max1_start, max1_end;
    float max1_user, max1_sys;
    float max1_user_max, max1_sys_max;
    float max1_wall, max1_wall_max;

    struct tms max2_start, max2_end;
    float max2_user, max2_sys;
    float max2_user_max, max2_sys_max;
    float max2_wall, max2_wall_max;

    struct tms before_start, before_end;
    float before_user, before_sys;
    float before_user_max, before_sys_max;
    float before_wall, before_wall_max;

    struct tms sort_start, sort_end;
    float sort_user, sort_sys;
    float sort_user_max, sort_sys_max;
    float sort_wall, sort_wall_max;

    struct tms sort_total_start, sort_total_end;
    float sort_total_user, sort_total_sys;
    float sort_total_user_max, sort_total_sys_max;
```

```
  float sort_total_wall, sort_total_wall_max;

  struct tms index_start, index_end;
  float index_user, index_sys;
  float index_user_max, index_sys_max;
  float index_wall, index_wall_max;

  struct tms fft1_start, fft1_end;
  float fft1_user, fft1_sys;
  float fft1_user_max, fft1_sys_max;
  float fft1_wall, fft1_wall_max;

  struct tms fft2_start, fft2_end;
  float fft2_user, fft2_sys;
  float fft2_user_max, fft2_sys_max;
  float fft2_wall, fft2_wall_max;

  struct tms fft3_start, fft3_end;
  float fft3_user, fft3_sys;
  float fft3_user_max, fft3_sys_max;
  float fft3_wall, fft3_wall_max;

  struct tms after_start, after_end;
  float after_user, after_sys;
  float after_user_max, after_sys_max;
  float after_wall, after_wall_max;


/*
 * Externally defined functions.
 */

  extern void cmd_line();
  extern void read_input_SORT_FFT();
  extern void bubble_sort();
  extern void fft();

/*
 * Begin function body: main ()
 *
 * Initialize for paralle processing: Here, each task or node determines its
 * task number (taskid) and the total number of tasks or nodes running this
 * program (numtask) by using the MPL call mpc_environ. Then, each task
 * determines the identifier for the group which encompasses all tasks or
 * nodes running this program. This identifier (allgrp) is used in collective
 * communication or aggregated computation operations, such as mpc_index.
 */

  gettimeofday(&tv0, (struct timeval*)0);          /* before time */

  rc = mpc_environ (&numtask, &taskid);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_environ.\n");
      exit (-1);
    }

  if (numtask != NN)
    {
      if (taskid == 0)
        {
          printf ("Error - task number mismatch... check defs.h.\n");
          exit (-1);
        }
    }

  rc = mpc_task_query (nbuf, 4, 3);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_task_query.\n");
      exit (-1);
    }
  allgrp = nbuf[3];

  if (taskid == 0)
    {
      printf ("Running...\n");
    }
  gettimeofday(&tv2, (struct timeval*)0);           /* before time */
  task_wall = (float) (tv2.tv_sec - tv0.tv_sec) * 1000000 + tv2.tv_usec
    - tv0.tv_usec;

  times (&all_start);

/*
 * Get arguments from command line. In the sequential version of the program,
 * the following procedure was used to extract the number of times the main
 * computational body was to be repeated, and flags regarding the amount of
 * reporting to be done during and after the program was run. In this paralle
 * program, there are no command line arguments to be extracted except for the
 * name of the file containing the data cube.
 */

   cmd_line (argc, argv, str_name, data_name);

/*
 * Read input files. In this section, each task loads its portion of the data
 * cube from the data file.
 */

  if (taskid == 0)
    {
      printf ("  loading data...\n");
    }

  mpc_sync (allgrp);
  times (&disk_start);
  gettimeofday(&tv1, (struct timeval*)0);           /* before time */
  read_input_SORT_FFT (data_name, data_cube);
  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  disk_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;

  times (&disk_end);


/*
 * Start SORT Steps. Find the element with the largest magnitude in this
 * task's slice of the data cube.
 */

  if (taskid == 0)
    printf("Finding maximum COMPLEX entry in data_cube before FFT \n");

  times (&max1_start);
  gettimeofday(&tv1, (struct timeval*)0);           /* before time */
  max = 0.0;

  for (i = 0; i < dim1; i++)
    for (j = 0; j < dim2; j++)
      for (k = 0; k < dim3; k++)
       {
         f_temp1 = data_cube[j][i][k].real;
         f_temp2 = data_cube[j][i][k].imag;
         f_pwr = f_temp1*f_temp1 + f_temp2*f_temp2;
         if (f_pwr > max)
           {
             max = f_pwr;
```

```
              loc1=i;
              loc2=j;
              loc3=k;
            }
        }

/*
 *  printf("task %d finds max %f at %d %d %d\n",
 *       taskid, max, loc1, loc2, loc3);
 */

  index_max[0].value = max;
  index_max[0].loc1 = loc1;
  index_max[0].loc2 = loc2;
  index_max[0].loc3 = loc3 + taskid * dim3;

  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  max1_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;

  times(&max1_end);


/*
 * Now aggregate to get the global maximum (the largest element in the entire
 * data cube).
 */

  mpc_sync (allgrp);
  times(&before_start);
  gettimeofday(&tv1, (struct timeval*)0);          /* before time */
  rc = mpc_reduce (index_max, g_index_max, sizeof (INDEXED_MAX), 0,
                xu_index_max, allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  before_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;
  times(&before_end);

  if (taskid == 0 )
    {
      max = g_index_max[0].value ;
      loc1 = g_index_max[0].loc1 ;
      loc2 = g_index_max[0].loc2 ;
      loc3 = g_index_max[0].loc3 ;

      printf("TIME: finding max before FFT = %f user secs and %f sys secs\n",
            USERTIME(t1,t2), SYSTIME(t1,t2) );

      printf("POWER of max COMPLEX entry before FFT = %f \n", max);
      printf("LOCATION of max entry before FFT = %d %d %d \n", loc1,loc2,loc3);
    }


/*
 * SORT DIM1 of data_cube DIM2*DIM3/3 times and output DIM1 average values.
 * DIM3 must be divisible by 3 to give a whole number dimension.
 */

  if (taskid == 0)
    printf("Sort %d element vectors %d*%d times. Output %d averages\n",
                                        DIM1, DIM2, DIM3/3, DIM1 );
  times (&sort_start);
  gettimeofday(&tv1, (struct timeval*)0);          /* before time */
  bubble_sort(data_cube, totals);

  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  sort_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;

  times (&sort_end);

/*
 * Get global totals of bubble sort.
 */

  mpc_sync (allgrp);
  times (&sort_total_start);

  gettimeofday(&tv1, (struct timeval*)0);          /* before time */

  rc = mpc_reduce (totals, global_totals, dim1*sizeof(float), 0,
                s_vadd, allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  sort_total_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;
  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  all_wall = (float) (tv2.tv_sec - tv0.tv_sec) * 1000000 + tv2.tv_usec
    - tv0.tv_usec;

  times (&sort_total_end);
  times(&all_end);
  if (taskid==0)
    {
      for (i = 0; i < DIM1; i++)
       printf(" Average value # %d = %f\n",
              i, global_totals[i]/(dim2*dim3*NN/3));
      printf("TIME: Sorting %d elements vectors = %f user secs and %f sys
secs\n", DIM1,USERTIME(t1,t2), SYSTIME(t1,t2) );
    }



/*
 * Compute all times.
 */

  all_user = (float) (all_end.tms_utime - all_start.tms_utime)/100.0;
  all_sys = (float)(all_end.tms_stime - all_start.tms_stime)/100.0;
  rc = mpc_reduce (&all_user, &all_user_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&all_sys, &all_sys_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }

/*
 * Compute disk times.
 */

  rc = mpc_reduce (&disk_wall, &disk_wall_max, sizeof(float), 0, s_vmax,
               allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  disk_user = (float) (disk_end.tms_utime - disk_start.tms_utime)/100.0;
  disk_sys = (float)(disk_end.tms_stime - disk_start.tms_stime)/100.0;
  rc = mpc_reduce (&disk_user, &disk_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&disk_sys, &disk_sys_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }

/*
 * Compute max1 times.
 */

  rc = mpc_reduce (&max1_wall, &max1_wall_max, sizeof(float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  max1_user = (float) (max1_end.tms_utime - max1_start.tms_utime)/100.0;
  max1_sys = (float)(max1_end.tms_stime - max1_start.tms_stime)/100.0;
  rc = mpc_reduce (&max1_user, &max1_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&max1_sys, &max1_sys_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }

/*
 * Compute before times.
 */

  rc = mpc_reduce (&before_wall, &before_wall_max, sizeof(float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  before_user = (float) (before_end.tms_utime - before_start.tms_utime)/100.0;
\
  before_sys = (float)(before_end.tms_stime - before_start.tms_stime)/100.0;
  rc = mpc_reduce (&before_user, &before_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&before_sys, &before_sys_max, sizeof (float), 0, s_vmax,
                  allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }

/*
 * Compute sort times.
 */

  rc = mpc_reduce (&sort_wall, &sort_wall_max, sizeof(float), 0, s_vmax,
                  allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  sort_user = (float) (sort_end.tms_utime - sort_start.tms_utime)/100.0;
  sort_sys = (float)(sort_end.tms_stime - sort_start.tms_stime)/100.0;
  rc = mpc_reduce (&sort_user, &sort_user_max, sizeof (float), 0, s_vmax,
                  allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&sort_sys, &sort_sys_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }

/*
 * Compute sort_total times.
 */

  rc = mpc_reduce (&sort_total_wall, &sort_total_wall_max, sizeof(float), 0,
                  s_vmax, allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  sort_total_user = (float) (sort_total_end.tms_utime
                           - sort_total_start.tms_utime)/100.0;
  sort_total_sys = (float)(sort_total_end.tms_stime
                         - sort_total_start.tms_stime)/100.0;
  rc = mpc_reduce (&sort_total_user, &sort_total_user_max, sizeof (float), 0,
                  s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&sort_total_sys, &sort_total_sys_max, sizeof (float), 0,
                  s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }

/*
 * Display timing information.
 */
```

```
  if (taskid == 0)
    {
      printf ("\n\n*** CPU Timing information - numtask = %d\n\n", NN);
      printf ("  all_user_max      = %.2f s, all_sys_max      = %.2f s\n",
              all_user_max, all_sys_max);
      printf ("  disk_user_max     = %.2f s, disk_sys_max     = %.2f s\n",
              disk_user_max, disk_sys_max);
      printf ("  max1_user_max     = %.2f s, max1_sys_max     = %.2f s\n",
              max1_user_max, max1_sys_max);
      printf ("  before_user_max     = %.2f s, before_sys_max     = %.2f s\n",
              before_user_max, before_sys_max);
      printf ("  sort_user_max     = %.2f s, sort_sys_max     = %.2f s\n",
              sort_user_max, sort_sys_max);
      printf ("  sort_total_user_max     = %.2f s, sort_total_sys_max     = %.2f
s\n", sort_total_user_max, sort_total_sys_max);

      printf ("\n*** Wall Clock Timing information - numtask = %d\n\n", NN);
      printf ("  all_wall          = %.0f us\n", all_wall);
      printf ("  task_wall          = %.0f us\n", task_wall);
      printf ("  disk_wall          = %.0f us\n", disk_wall_max);
      printf ("  max1_wall          = %.0f us\n", max1_wall_max);
      printf ("  before_wall        = %.0f us\n", before_wall_max);
      printf ("  sort_wall          = %.0f us\n", sort_wall_max);
      printf ("  sort_total_wall    = %.0f us\n", sort_total_wall_max);
    }
  exit(0);
}
```

A.1.2 bench_mark_SORT_FFT.c

```
/*
 * bench_mark_SORT_FFT.c
 */

/*
 * Parallel General Benchmark Program for the IBM SP2
 * -------------------------------------------------
 *
 * This parallel General benchmark program was written for the IBM SP2 by the
 * STAP benchmark parallelization team at the University of Southern
 * California (Prof. Kai Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as
 * part of the ARPA Mountaintop program.
 *
 * The sequential General benchmark program was originally written by Tony
 * Adams on 8/30/93.
 *
 *
 * This file contains the procedure main (), and represents the body of the
 * General Sorting and FFT subprograms.
 *
 * This subprogram's overall structure is as follows:
 *    1. Load data in parallel.
 *    2. Each task searches its slice of the data cube for the largest element.
 *    3. Perform a reduction operation to find the largest element in the
 *       entire data cube.
 *    4. Each task sorts its slice of the data cube along the DIM1 dimension.
 *    5. Collect the totals calculated in the bubble sort operation using a
 *       reduction operation.
 *    6. Perform FFTs along the DIM1 and DIM2 dimensions.
 *    7. Redistribute the data cube using a total exchange operation.
 *    8. Perform FFTs along the DIM3 dimension.
 *    9. Each task searches its slice of the data cube for the largest element
 *       after the FFTs.
 *    10. Perform a reduction operation to find the largest element in the
 *       entire data cube.
 *
 * This program can be compiled by calling the shell script compile_sort_fft.
 * Because of the nature of the program, before the program is compiled, the
 * header file defs.h must be adjusted so that NN is set to the number of
 * nodes on which the program will be run. We have provided a defs.h file for
 * each power of 2 node size from 1 to 256, called defs.001 to defs.256.
 *
 * This program can be run by calling the shell script run.###, where ### is
 * the number of nodes on which the program is being run.
 *
 * Unlike the original sequential version of this program, the parallel General
 * program does not support command-line arguments specifying the number of
 * times the program body should be executed, nor whether or not timing
 * information should be displayed. The program body will be executed once,
 * and the timing information will be displayed at the end of execution.
 *
 * The input data file on disk is stored in a packed format: Each complex
 * number is stored as a 32-bit binary integer; the 16 LSBs are the real
 * half of the number, and the 16 MSBs are the imaginary half of the number.
 * These numbers are converted into floating point numbers usable by this
 * benchmark program as they are loaded from disk.
 *
 * The steering vectors file has the data stored in a different fashion. All
 * data in this file is stored in ASCII format. The first two numbers in this
 * file are the number of PRIs in the data set and the threshold level,
 * respectively. Then, the remaining data are the steering vectors, with
 * alternating real and imaginary numbers.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"
#include <sys/types.h>
#include <sys/times.h>
#include <mpproto.h>
#include <sys/time.h>
#include <time.h>

/*
 * main ()
 *   inputs: argc, argv
 *   outputs: none
 */

main (argc, argv)
   int argc;
   char *argv[];
{

/*
 * Variables
 *
 * data_cube: input data cube
 * doppler_cube: Doppler data cube
 * data_vector: used in the index step
 * t1, t2: holders for the time function return in seconds
 * dim1, dim2, dim3: dimensions 1, 2, and 3 for the input data cube.
 * loc1, loc2, loc3: holders for the location of the largest element
 * temp: buffer for binary input integer data
 * f_temp1, f_temp2: holders for float data
 * f_pwr: holder for float power data
 * i, j, k, n, offset, r: loop counters
 * max: float variable to hold the maximum of complex data
 * index_max, g_index_max: data structures holding the location and power of
 *   the largest elements (local and global, repsectively)
 * data_name, str_name: names of the files containing the input data file and
 *   the vectors, respectively
 * totals: holders for DIM1 totals
 * global_totals: holders for global DIM1 totals
 * logpoints: log base 2 of the number of points in the FFT
 * sum: a COMPLEX variable to hold the sum of complex data
 * points: the number of points in the FFT
 * pix2: pi times 2 (2 x 3.14159)
 * pi: pi (3.14159)
 * x: storage for temporary FFT vector
 * w: storage for the twiddle factors table
 * blklen: block length; used for MPL calls
 * rc: return code from an MPL call
 * taskid: the identifier for this task
 * numask: the number of tasks running this program
 * nbuf: buffer to hold data returned by the mpc_task_query call
 * allgrp: the identifier for the group encompassing all the tasks running
 *    this program
 * source, dest, type, nbytes: used by MPL calls
 */

  COMPLEX data_cube[DIM2][DIM1][DIM3];
  COMPLEX doppler_cube[DIM1][DIM2/NN][DOP_GEN];
  COMPLEX data_vector[DIM2*DIM1*DIM3];
  struct tms t1,t2;
  int dim1 = DIM1;
  int dim2 = DIM2;
  int dim3 = DIM3;
  int loc1,loc2,loc3;
  unsigned int temp[1];
  float f_temp1, f_temp2;
  float f_pwr;
  int i, j, k, n, offset, r;
  float max;
  INDEXED_MAX index_max[1], g_index_max[1] ;
  char data_name[LINE_MAX];
  char str_name[LINE_MAX];
  FILE *fopen();
  FILE *f_dop;
  float totals[DIM1];
  float global_totals[DIM1];
  int logpoints;
  COMPLEX sum;
  int points = DOP_GEN;
  float pix2, pi;
  static COMPLEX x[DOP_GEN];
  static COMPLEX w[DOP_GEN];
  int blklen;
  int rc;
  int taskid;
  int numtask;
  int nbuf[4];
  int allgrp;
  int source, dest, type, nbytes;

/*
 * Timing variables.
 */

  struct timeval tv0, tv1, tv2;
  struct tms all_start, all_end;
  float all_user, all_sys;
  float all_user_max, all_sys_max;
  float all_wall, task_wall;

  struct tms disk_start, disk_end;
  float disk_user, disk_sys;
  float disk_user_max, disk_sys_max;
  float disk_wall, disk_wall_max;

  struct tms max1_start, max1_end;
  float max1_user, max1_sys;
  float max1_user_max, max1_sys_max;
  float max1_wall, max1_wall_max;

  struct tms max2_start, max2_end;
  float max2_user, max2_sys;
  float max2_user_max, max2_sys_max;
  float max2_wall, max2_wall_max;

  struct tms before_start, before_end;
  float before_user, before_sys;
  float before_user_max, before_sys_max;
  float before_wall, before_wall_max;

  struct tms sort_start, sort_end;
  float sort_user, sort_sys;
  float sort_user_max, sort_sys_max;
  float sort_wall, sort_wall_max;

  struct tms sort_total_start, sort_total_end;
  float sort_total_user, sort_total_sys;
  float sort_total_user_max, sort_total_sys_max;
  float sort_total_wall, sort_total_wall_max;

  struct tms index_start, index_end;
  float index_user, index_sys;
  float index_user_max, index_sys_max;
  float index_wall, index_wall_max;

  struct tms fft1_start, fft1_end;
  float fft1_user, fft1_sys;
  float fft1_user_max, fft1_sys_max;
  float fft1_wall, fft1_wall_max;

  struct tms fft2_start, fft2_end;
  float fft2_user, fft2_sys;
  float fft2_user_max, fft2_sys_max;
  float fft2_wall, fft2_wall_max;

  struct tms fft3_start, fft3_end;
  float fft3_user, fft3_sys;
  float fft3_user_max, fft3_sys_max;
  float fft3_wall, fft3_wall_max;

  struct tms after_start, after_end;
  float after_user, after_sys;
  float after_user_max, after_sys_max;
  float after_wall, after_wall_max;


/*
 * Externally defined functions.
 */

  extern void cmd_line();
  extern void read_input_SORT_FFT();
  extern void bubble_sort();
  extern void fft();

/*
 * Begin function body: main ()
 *
 * Initialize for parallel processing: Here, each task or node determines its
 * task number (taskid) and the total number of tasks or nodes running this
 * program (numtask) by using the MPL call mpc_environ. Then, each task
 * determines the identifier for the group which encompasses all tasks or
 * nodes running this program. This identifier (allgrp) is used in collective
 * communication or aggregated computation operations, such as mpc_index.
 */
```

```
  gettimeofday(&tv0, (struct timeval*)0);          /* before time */

  rc = mpc_environ (&numtask, &taskid);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_environ.\n");
      exit (-1);
    }

  if (numtask != NN)
    {
      if (taskid == 0)
        {
          printf ("Error - task number mismatch... check defs.h.\n");
          exit (-1);
        }
    }

  rc = mpc_task_query (nbuf, 4, 3);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_task_query.\n");
      exit (-1);
    }
  allgrp = nbuf[3];

  if (taskid == 0)
    {
      printf ("Running...\n");
    }
  gettimeofday(&tv2, (struct timeval*)0);          /* before time */
  task_wall = (float) (tv2.tv_sec - tv0.tv_sec) * 1000000 + tv2.tv_usec
    - tv0.tv_usec;

  times (&all_start);


/*
 * Get arguments from command line. In the sequential version of the program,
 * the following procedure was used to extract the number of times the main
 * computational body was to be repeated, and flags regarding the amount of
 * reporting to be done during and after the program was run. In this paralle
 * program, there are no command line arguments to be extracted except for the
 * name of the file containing the data cube.
 */

   cmd_line (argc, argv, str_name, data_name);

/*
 * Read input files. In this section, each task loads its portion of the data
 * cube from the data file.
 */

  if (taskid == 0)
    {
      printf ("  loading data...\n");
    }

  mpc_sync (allgrp);
  times (&disk_start);
  gettimeofday(&tv1, (struct timeval*)0);          /* before time */

  read_input_SORT_FFT (data_name, data_cube);
  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  disk_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;

  times (&disk_end);
```

```
/*
 * Start SORT Steps. Find the element with the largest magnitude in this task's
 * slice of the data cube.
 */

/*
 * if (taskid == 0)
 *   printf("Finding maximum COMPLEX entry in data_cube before FFT \n");
 */

  times (&max1_start);
  gettimeofday(&tv1, (struct timeval*)0);          /* before time */
  max = 0.0;

  for (i = 0; i < dim1; i++)
    for (j = 0; j < dim2; j++)
      for (k = 0; k < dim3; k++)
       {
         f_temp1 = data_cube[j][i][k].real;
         f_temp2 = data_cube[j][i][k].imag;
         f_pwr = f_temp1*f_temp1 + f_temp2*f_temp2;
         if (f_pwr > max)
           {
             max = f_pwr;
             loc1=i;
             loc2=j;
             loc3=k;
           }
       }

/*
 * printf("task %d finds max %f at %d %d %d\n", taskid, max, loc1, loc2, loc3);
 */
  index_max[0].value = max ;
  index_max[0].loc1 = loc1 ;
  index_max[0].loc2 = loc2 ;
  index_max[0].loc3 = loc3 + taskid * dim3 ;

  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  max1_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;

  times(&max1_end);

/*
 * Now aggregate to get the global maximum (the largest element in the entire
 * data cube).
 */

  mpc_sync (allgrp);
  times(&before_start);
  gettimeofday(&tv1, (struct timeval*)0);          /* before time */
  rc = mpc_reduce (index_max, g_index_max, sizeof (INDEXED_MAX), 0,
                   xu_index_max, allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  before_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;
  times(&before_end);


  if (taskid == 0 )
    {
      max = g_index_max[0].value ;
      loc1 = g_index_max[0].loc1 ;
      loc2 = g_index_max[0].loc2 ;
      loc3 = g_index_max[0].loc3 ;

      printf("TIME: finding max before FFT = %f user secs and %f sys secs\n",
            USERTIME(t1,t2), SYSTIME(t1,t2) );

      printf("POWER of max COMPLEX entry before FFT = %f \n", max);
      printf("LOCATION of max entry before FFT = %d %d %d \n", loc1,loc2,loc3);
    }


/*
 * SORT DIM1 of data_cube DIM2*DIM3/3 times and output DIM1 average values.
 * DIM3 must be divisible by 3 to give whole number dimension.
 */

  if (taskid == 0)
    printf("Sort %d element vectors %d*%d times. Output %d averages\n",
          DIM1, DIM2, DIM3/3, DIM1 );
  times (&sort_start);
  gettimeofday(&tv1, (struct timeval*)0);          /* before time */

  bubble_sort(data_cube, totals);

  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  sort_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;

  times (&sort_end);

/*
 * Get global totals of bubble sort.
 */

  mpc_sync (allgrp);
  times (&sort_total_start);

  gettimeofday(&tv1, (struct timeval*)0);          /* before time */

  rc = mpc_reduce (totals, global_totals, dim1*sizeof(float), 0,
                s_vadd, allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  sort_total_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;

  times (&sort_total_end);

  if (taskid==0)
    {
      for (i = 0; i < DIM1; i++)
       printf(" Average value # %d = %f\n", i,
              global_totals[i]/(dim2*dim3*NN/3));
      printf("TIME: Sorting %d elements vectors = %f user secs and %f sys
secs\n", DIM1,USERTIME(t1,t2), SYSTIME(t1,t2) );
    }

/*
 * Start FFT steps.
 *
 * Generate twiddle factors table w.
 */

  pi = 3.14159265358979;
  pix2 = 2.0 * pi;
  for (i=0; i < points; i++)
    {
      w[i].imag = - sin(pix2 * (float)i / (float)points);
      w[i].real = cos(pix2 * (float)i / (float)points);
    }

/*
 * 1st FFT: Perform dim2*dim3 number of dim1 points FFTs.
 */

  logpoints = log2((float) dim1 ) + 0.1;

  if (taskid==0)
    printf("Computing %d*%d    %d point FFTs \n", dim2, dim3*NN, dim1);

  times(&fft1_start);
  gettimeofday(&tv1, (struct timeval*)0);          /* before time */
  for (j = 0; j < dim2; j++)
    for (k = 0; k < dim3; k++)
      {
        for (i = 0; i < dim1; i++)
          {
            x[i].real = data_cube[j][i][k].real;
            x[i].imag = data_cube[j][i][k].imag;
          }
        fft (x, w, dim1, logpoints);
        for (i = 0; i < dim1; i++)
          {
            data_cube[j][i][k].real =  x[i].real ;
            data_cube[j][i][k].imag =  x[i].imag ;
          }
      }
  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  fft1_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;

  times(&fft1_end);

/*
 * 2nd FFT: Perform dim1*dim3 number of dim2 points FFTs.
 */

  logpoints = log2((float) dim2 ) + 0.1;

  if (taskid==0)
    printf("Computing %d*%d    %d point FFTs \n", dim1, dim3*NN, dim2);

  times(&fft2_start);
  gettimeofday(&tv1, (struct timeval*)0);          /* before time */

  for (i = 0; i < dim1; i++)
    for (k = 0; k < dim3; k++)
      {
       for (j = 0; j < dim2; j++)
         {
            x[j].real = data_cube[j][i][k].real;
            x[j].imag = data_cube[j][i][k].imag;
          }
       fft (x, w, dim2, logpoints);
       for (j = 0; j < dim2; j++)
         {
            data_cube[j][i][k].real =  x[j].real ;
            data_cube[j][i][k].imag =  x[j].imag ;
          }
      }
  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  fft2_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;
  times(&fft2_end);

/*
 * Index data_cube to doppler_cube. This total exchange operation is needed
 * so that each task has all the elements along the DIM3 dimension (the
 * dimension along which the last set of FFTs is performed).
 */

  if (taskid == 0)
    {
      printf ("  indexing data cube...\n");
    }

  mpc_sync (allgrp);
  times (&index_start);
  gettimeofday(&tv1, (struct timeval*)0);           /* before time */
  rc = mpc_index (data_cube, data_vector,
                  DIM1 * DIM2 * DIM3 * sizeof (COMPLEX) / NN, allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_index.\n");
      exit (-1);
    }

/*
 *  if (taskid == 0)
 *    {
 *      printf ("  rewinding data cube...\n");
 *    }
 */

  offset = 0;
  for (n = 0; n < NN; n++)
    for (j= 0 ; j < DIM2/NN; j++)
      for (i = 0; i < DIM1 ; i++)
       for (k = n * DIM3; k < (n + 1) * DIM3 ; k++)
         {
             doppler_cube[i][j][k].real = data_vector[offset].real;
             doppler_cube[i][j][k].imag = data_vector[offset].imag;
             offset++;
          }

  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  index_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;

  times (&index_end);

/*
 * 3rd FFT: Perform dim1*dim2 number of DOP_GEN points FFTs.
 */

  dim2 = dim2/NN;
  dim3 = dim3*NN ;
  logpoints = log2((float) DOP_GEN ) + 0.1;

  if (taskid==0)
    printf("Computing %d*%d    %d point FFTs \n", dim1, dim2*NN, DOP_GEN);
  times(&fft3_start);
  gettimeofday(&tv1, (struct timeval*)0);           /* before time */


  for (i = 0; i < dim1; i++)
    for (j = 0; j < dim2; j++)
      {
       for (k = 0; k < dim3; k++)
         {
           x[k].real = doppler_cube[i][j][k].real;
```

```
            x[k].imag = doppler_cube[i][j][k].imag;
          }
        for (k = dim3; k < DOP_GEN; k++)
          {
            x[k].real = 0.0;
            x[k].imag = 0.0;
          }
        fft (x, w, DOP_GEN, logpoints);
        for (k = 0; k < DOP_GEN; k++)
          {
            doppler_cube[i][j][k].real =  x[k].real ;
            doppler_cube[i][j][k].imag =  x[k].imag ;
          }
      }
  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  fft3_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;

  times(&fft3_end);

  if (taskid == 0)
    printf("Finding maximum COMPLEX entry in data_cube after FFT \n");

  times (&max2_start);
  gettimeofday(&tv1, (struct timeval*)0);          /* before time */

  max = 0.0;

/*
 * Find the element with the largest magnitude in each task's slice of the
 * data cube after the FFTs.
 */

  for (i = 0; i < dim1; i++)
    for (j = 0; j < dim2; j++)
      for (k = 0; k < DOP_GEN; k++)
       {
         f_temp1 = doppler_cube[i][j][k].real;
         f_temp2 = doppler_cube[i][j][k].imag;
         f_pwr = f_temp1*f_temp1 + f_temp2*f_temp2;
         if (f_pwr > max)
           {
             max = f_pwr;
             loc1=i;
             loc2=j;
             loc3=k;
           }
       }
  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  max2_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;
  times(&max2_end);

/*
 *  printf("task %d finds max %f at %d %d %d\n",
 *         taskid, max, loc1, loc2, loc3);
 */

  index_max[0].value = max;
  index_max[0].loc1 = loc1;
  index_max[0].loc2 = loc2 + taskid * dim2;
  index_max[0].loc3 = loc3;

/*
 * Now aggregate to get the global maximum (the element with the largest
 * magnitude in the entire data cube after the FFTs).
 */

  mpc_sync (allgrp);
  times (&after_start);
  gettimeofday(&tv1, (struct timeval*)0);          /* before time */

  rc = mpc_reduce (index_max, g_index_max, sizeof (INDEXED_MAX), 0,
                   xu_index_max, allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  after_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;
  all_wall = (float) (tv2.tv_sec - tv0.tv_sec) * 1000000 + tv2.tv_usec
    - tv0.tv_usec;
  times(&after_end);
  times(&all_end);

/*
 * Compute all times.
 */

  all_user = (float) (all_end.tms_utime - all_start.tms_utime)/100.0;
  all_sys = (float)(all_end.tms_stime - all_start.tms_stime)/100.0;
  rc = mpc_reduce (&all_user, &all_user_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&all_sys, &all_sys_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }


/*
 * Compute disk times.
 */

  rc = mpc_reduce (&disk_wall, &disk_wall_max, sizeof(float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  disk_user = (float) (disk_end.tms_utime - disk_start.tms_utime)/100.0; \
  disk_sys = (float)(disk_end.tms_stime - disk_start.tms_stime)/100.0; \
  rc = mpc_reduce (&disk_user, &disk_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&disk_sys, &disk_sys_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }

/*
 * Compute max1 times.
 */
```

```
  rc = mpc_reduce (&max1_wall, &max1_wall_max, sizeof(float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  max1_user = (float) (max1_end.tms_utime - max1_start.tms_utime)/100.0;
  max1_sys = (float)(max1_end.tms_stime - max1_start.tms_stime)/100.0;
  rc = mpc_reduce (&max1_user, &max1_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&max1_sys, &max1_sys_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }

/*
 * Compute before times.
 */

  rc = mpc_reduce (&before_wall, &before_wall_max, sizeof(float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  before_user = (float) (before_end.tms_utime - before_start.tms_utime)/100.0;
  before_sys = (float)(before_end.tms_stime - before_start.tms_stime)/100.0;
  rc = mpc_reduce (&before_user, &before_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&before_sys, &before_sys_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }

/*
 * Compute sort times.
 */

  rc = mpc_reduce (&sort_wall, &sort_wall_max, sizeof(float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  sort_user = (float) (sort_end.tms_utime - sort_start.tms_utime)/100.0;
  sort_sys = (float)(sort_end.tms_stime - sort_start.tms_stime)/100.0;
  rc = mpc_reduce (&sort_user, &sort_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
        printf ("Error mpc_reduce.\n");
        exit (-1);
      }
    rc = mpc_reduce (&sort_sys, &sort_sys_max, sizeof (float), 0, s_vmax,allgrp);
    if (rc == -1)
      {
        printf ("Error mpc_reduce.\n");
        exit (-1);
      }

/*
 * Compute sort_total times.
 */

    rc = mpc_reduce (&sort_total_wall, &sort_total_wall_max, sizeof(float), 0,
                     s_vmax, allgrp);
    if (rc == -1)
      {
        printf ("Error mpc_reduce.\n");
        exit (-1);
      }
    sort_total_user = (float) (sort_total_end.tms_utime -
                               sort_total_start.tms_utime)/100.0;
    sort_total_sys = (float)(sort_total_end.tms_stime -
                             sort_total_start.tms_stime)/100.0;
    rc = mpc_reduce (&sort_total_user, &sort_total_user_max, sizeof (float), 0,
                     s_vmax,allgrp);
    if (rc == -1)
      {
        printf ("Error mpc_reduce.\n");
        exit (-1);
      }
    rc = mpc_reduce (&sort_total_sys, &sort_total_sys_max, sizeof (float), 0,
                     s_vmax,allgrp);
    if (rc == -1)
      {
        printf ("Error mpc_reduce.\n");
        exit (-1);
      }

/*
 * Compute fft1 times.
 */
    rc = mpc_reduce (&fft1_wall, &fft1_wall_max, sizeof(float), 0, s_vmax,
                     allgrp);
    if (rc == -1)
      {
        printf ("Error mpc_reduce.\n");
        exit (-1);
      }
    fft1_user = (float) (fft1_end.tms_utime - fft1_start.tms_utime)/100.0;
    fft1_sys = (float)(fft1_end.tms_stime - fft1_start.tms_stime)/100.0;
    rc = mpc_reduce (&fft1_user, &fft1_user_max, sizeof (float), 0, s_vmax,
                     allgrp);
    if (rc == -1)
      {
        printf ("Error mpc_reduce.\n");
        exit (-1);
      }
    rc = mpc_reduce (&fft1_sys, &fft1_sys_max, sizeof (float), 0, s_vmax,allgrp);
    if (rc == -1)
      {
        printf ("Error mpc_reduce.\n");
        exit (-1);
      }

/*
 * Compute fft2 times.
 */
```

```
  rc = mpc_reduce (&fft2_wall, &fft2_wall_max, sizeof(float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  fft2_user = (float) (fft2_end.tms_utime - fft2_start.tms_utime)/100.0;
  fft2_sys = (float)(fft2_end.tms_stime - fft2_start.tms_stime)/100.0;
  rc = mpc_reduce (&fft2_user, &fft2_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&fft2_sys, &fft2_sys_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }

/*
 * Compute fft3 times.
 */

  rc = mpc_reduce (&fft3_wall, &fft3_wall_max, sizeof(float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  fft3_user = (float) (fft3_end.tms_utime - fft3_start.tms_utime)/100.0;
  fft3_sys = (float)(fft3_end.tms_stime - fft3_start.tms_stime)/100.0;
  rc = mpc_reduce (&fft3_user, &fft3_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&fft3_sys, &fft3_sys_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }

/*
 * Compute index times.
 */

  rc = mpc_reduce (&index_wall, &index_wall_max, sizeof(float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  index_user = (float) (index_end.tms_utime - index_start.tms_utime)/100.0;
  index_sys = (float)(index_end.tms_stime - index_start.tms_stime)/100.0;
  rc = mpc_reduce (&index_user, &index_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
```

```
        exit (-1);
      }
    rc = mpc_reduce (&index_sys, &index_sys_max, sizeof (float), 0, s_vmax,
                     allgrp);
    if (rc == -1)
      {
        printf ("Error mpc_reduce.\n");
        exit (-1);
      }

/*
 * Compute max2 times.
 */

    rc = mpc_reduce (&max2_wall, &max2_wall_max, sizeof(float), 0, s_vmax,
                     allgrp);
    if (rc == -1)
      {
        printf ("Error mpc_reduce.\n");
        exit (-1);
      }
    max2_user = (float) (max2_end.tms_utime - max2_start.tms_utime)/100.0;
    max2_sys = (float)(max2_end.tms_stime - max2_start.tms_stime)/100.0;
    rc = mpc_reduce (&max2_user, &max2_user_max, sizeof (float), 0, s_vmax,
                     allgrp);
    if (rc == -1)
      {
        printf ("Error mpc_reduce.\n");
        exit (-1);
      }
    rc = mpc_reduce (&max2_sys, &max2_sys_max, sizeof (float), 0, s_vmax,allgrp);
    if (rc == -1)
      {
        printf ("Error mpc_reduce.\n");
        exit (-1);
      }

/*
 * Compute after times.
 */

    rc = mpc_reduce (&after_wall, &after_wall_max, sizeof(float), 0, s_vmax,
                     allgrp);
    if (rc == -1)
      {
        printf ("Error mpc_reduce.\n");
        exit (-1);
      }
    after_user = (float) (after_end.tms_utime - after_start.tms_utime)/100.0;
    after_sys = (float)(after_end.tms_stime - after_start.tms_stime)/100.0;
    rc = mpc_reduce (&after_user, &after_user_max, sizeof (float), 0, s_vmax,
                     allgrp);
    if (rc == -1)
      {
        printf ("Error mpc_reduce.\n");
        exit (-1);
      }
    rc = mpc_reduce (&after_sys, &after_sys_max, sizeof (float), 0, s_vmax,
                     allgrp);
    if (rc == -1)
      {
        printf ("Error mpc_reduce.\n");
        exit (-1);
      }

/*
 * Report results.
 */
```

```
  if (taskid == 0 )
    {
      max = g_index_max[0].value;
      loc1 = g_index_max[0].loc1;
      loc2 = g_index_max[0].loc2;
      loc3 = g_index_max[0].loc3;

      printf("TIME: finding max after FFT = %f user secs and %f sys secs\n",
            USERTIME(t1,t2), SYSTIME(t1,t2) );
      printf("POWER of max COMPLEX entry after FFT = %f \n", max);
      printf("LOCATION of max entry after FFT = %d %d %d \n", loc1,loc2,loc3);
    }

/*
 * Display timing information.
 */

  if (taskid == 0)
    {
      printf ("\n\n*** CPU Timing information - numtask = %d\n\n", NN);
      printf ("  all_user_max      = %.2f s, all_sys_max      = %.2f s\n",
              all_user_max, all_sys_max);
      printf ("  disk_user_max     = %.2f s, disk_sys_max     = %.2f s\n",
              disk_user_max, disk_sys_max);
      printf ("  max1_user_max     = %.2f s, max1_sys_max     = %.2f s\n",
              max1_user_max, max1_sys_max);
      printf ("  before_user_max     = %.2f s, before_sys_max     = %.2f s\n",
              before_user_max, before_sys_max);
      printf ("  sort_user_max     = %.2f s, sort_sys_max     = %.2f s\n",
              sort_user_max, sort_sys_max);
      printf ("  sort_total_user_max     = %.2f s, sort_total_sys_max     = %.2f
s\n", sort_total_user_max, sort_total_sys_max);
      printf ("  fft1_user_max     = %.2f s, fft1_sys_max     = %.2f s\n",
              fft1_user_max, fft1_sys_max);
      printf ("  fft2_user_max     = %.2f s, fft2_sys_max     = %.2f s\n",
              fft2_user_max, fft2_sys_max);
      printf ("  fft3_user_max     = %.2f s, fft3_sys_max     = %.2f s\n",
              fft3_user_max, fft3_sys_max);
      printf ("  index_user_max     = %.2f s, index_sys_max     = %.2f s\n",
              index_user_max, index_sys_max);
      printf ("  max2_user_max     = %.2f s, max2_sys_max     = %.2f s\n",
              max2_user_max, max2_sys_max);
      printf ("  after_user_max     = %.2f s, after_sys_max     = %.2f s\n",
              after_user_max, after_sys_max);

      printf ("\n*** Wall Clock Timing information - numtask = %d\n\n", NN);
      printf ("  all_wall         = %.0f us\n", all_wall);
      printf ("  task_wall         = %.0f us\n", task_wall);
      printf ("  disk_wall         = %.0f us\n", disk_wall_max);
      printf ("  max1_wall         = %.0f us\n", max1_wall_max);
      printf ("  before_wall       = %.0f us\n", before_wall_max);
      printf ("  sort_wall         = %.0f us\n", sort_wall_max);
      printf ("  sort_total_wall   = %.0f us\n", sort_total_wall_max);
      printf ("  fft1_wall         = %.0f us\n", fft1_wall_max);
      printf ("  fft2_wall         = %.0f us\n", fft2_wall_max);
      printf ("  fft3_wall         = %.0f us\n", fft3_wall_max);
      printf ("  index_wall        = %.0f us\n", index_wall_max);
      printf ("  max2_wall         = %.0f us\n", max2_wall_max);
      printf ("  after_wall        = %.0f us\n", after_wall_max);
    }
  exit(0);
}
```

A.1.3 bubble_sort.c

```
/*
 * bubble_sort.c
 */

/*
 * This file contains the procedure bubble_sort (), and is part of the parallel
 * General benchmark program written for the IBM SP2 by the STAP benchmark
 * parallelization team at the University of Southern California (Prof. Kai
 * Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA Mountaintop
 * program.
 *
 * The sequential General benchmark program was originally written by Tony
 * Adams on 8/30/93.
 *
 * The bubble_sort () procedure contains the sort routine to do DIM2*DIM3/3
 * number of DIM1 element bubble sorts and to output DIM1 average values of the
 * DIM2*DIM3/3 sorts. DIM3 must be divisible by 3 to give whole number
 * dimensions. The power of the COMPLEX entries will be used to sort the data.
 * The procedure sorts the COMPLEX input data_cube[DIM1][DIM2][DIM3].
 * Nominally, DIM1, DIM2, and DIM3 = 64, 128, and 1500, respectively, and
 * thus the program sorts 128 * 500 = 64,000 64-element vectors.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"

/*
 * bubble_sort ()
 *    inputs: data_cube
 *    outputs: totals
 *
 *    data_cube: input data cube
 *    totals: holders for DIM1 totals
 */

bubble_sort (data_cube, totals)
     COMPLEX data_cube[][DIM1][DIM3];
     float totals[];

{

/*
 * Variables
 *
 * dim1, dim2, dim3: dimensions 1, 2, and 3 in the input data cube
 * sort_num: number of the sort number
 * f_pwr: holder for float power data
 * vec_dim1: holder for DIM1 element SORT vector
 * this_ptr, next_ptr: pointers to float data
 * i, j, k, n: loop counters
 */

  int dim1 = DIM1;
  int dim2 = DIM2;
  int dim3 = DIM3;
  int sort_num;
  float f_pwr;
  float vec_dim1[DIM1];
  float *this_ptr, *next_ptr;
  int i, j, k, n;

/*
 * Begin function body: bubble_sort ().
 *
 * Perform DIM2*DIM3/3 number of DIM1 element sorts.
 *
 * Zero out DIM1 totals.
 */

  for(n = 0; n < DIM1; n++) totals[n] = 0.0;

/*
 * Sort by the magnitude of the complex number (sum of the squares of the real
 * and imaginary parts of the number).
 *
 * Select dimensions so we always do dim2*dim3/3 number of dim1 element sorts.
 */

  for (j = 0; j < dim2; j++)
    for (k = 0; k < dim3/3; k++)
      {

/*
 * Initialize pointer to start of holding vector.
 */

      this_ptr = vec_dim1;
      for (i = 0; i < dim1; i++)
        {
          *this_ptr++ = data_cube[j][i][k].real * data_cube[j][i][k].real +
            data_cube[j][i][k].imag * data_cube[j][i][k].imag;
        }

/*
 * Bubble sort the contents of the holding vector. 1st initialize pointers to
 * start of holding vector.
 */

      this_ptr = vec_dim1;
      next_ptr = vec_dim1 + 1;

/*
 * Initialize the number of bubbles.
 */
      sort_num = dim1 - 1;

/*
 * Get 1st number.
 */

      f_pwr = *this_ptr;
      while (sort_num > 0)
        {
          for (n = 0; n < sort_num; n++)
            {
             if (*next_ptr < f_pwr)
               {
                 *this_ptr++ = *next_ptr++;
               }
             else
               {
                 *this_ptr++ = f_pwr;
                 f_pwr = *next_ptr++;
               }
            }
          *this_ptr = f_pwr;
          sort_num -= 1;

/*
 * Reinitialize pointers to start of holding vector.
 */

          this_ptr = vec_dim1;
```

```
            next_ptr = vec_dim1 + 1;

/*
 * Get 1st number.
 */
            f_pwr = *this_ptr;
          }
        for(n = 0; n < dim1; n++) totals[n] += vec_dim1[n];
       }
   return;
}
```

A.1.4 cmd_line.c

```
/*
 * cmd_line.c
 */

/*
 * This file contains the procedure cmd_line (), and is part of the parallel
 * General benchmark program written for the IBM SP2 by the STAP benchmark
 * parallelization team at the University of Southern California (Prof. Kai
 * Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA Mountaintop
 * program.
 *
 * The sequential General benchmark program was originally written by Tony
 * Adams on 8/30/93.
 *
 * The procedure cmd_line () extracts the name of the files from which the
 * input data cube and the steering vector data should be loaded. The
 * function of this parallel version of cmd_line () is different from that
 * of the sequential version, because the sequential version also extracted
 * the number of iterations the program should run and some reporting
 * options.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"
#include <sys/types.h>
#include <sys/time.h>

/*
 * cmd_line ()
 *    inputs: argc, argv
 *    outputs: str_name, data_name
 *
 *    argc, argv: these are used to get data from the command line arguments
 *    str_name: a holder for the name of the input steering vectors file
 *    data_name: a holder for the name of the input data file
 */

cmd_line (argc, argv, str_name, data_name)
     int argc;
     char *argv[];
     char str_name[LINE_MAX];
     char data_name[LINE_MAX];

{

/*
 * Begin function body: cmd_line ()
 */
```

```
  strcpy (str_name, argv[1]);
  strcat (str_name, ".str");
  strcpy (data_name, argv[1]);
  strcat (data_name, ".dat");
  return;
}
```

A.1.5 fft.c

```
/*
 * fft.c
 */

/*
 * This file contains the procedures fft () and bit_reverse (), and is part
 * of the parallel General benchmark program written for the IBM SP2 by the
 * STAP benchmark parallelization team at the University of Southern California
 * (Prof. Kai Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA
 * Mountaintop program.
 *
 * The sequential General benchmark program was originally written by Tony
 * Adams on 8/30/93.
 *
 * The procedure fft () implements an n-point in-place decimation-in-time
 * FFT of complex vector "data" using the n/2 complex twiddle factors in
 * "w_common". The implementation used in this procedure is a hybrid between
 * the implementation in the original sequential version of this program and
 * the implementation suggested by MHPCA. This modification was made to
 * improve the performance of the FFT on the SP2.
 *
 * The procedure bit_reverse () implements a simple (but somewhat inefficient)
 * bit reversal.
 */

#include "defs.h"

/*
 * fft ()
 *    inputs: data, w_common, n, logn
 *    outputs: data
 */

void fft (data, w_common, n, logn)
     COMPLEX *data, *w_common;
     int n, logn;

{  /* fft */
  int incrvec, i0, i1, i2, nx, t1, t2, t3;
  float f0, f1;
  void bit_reverse();

/*
 * Begin function body: fft ()
 *
 * Bit-reverse the input vector.
 */

  (void) bit_reverse (data, n);

/*
 * Do the first log n - 1 stages of the FFT.
 */

  i2 = logn;
  for (incrvec = 2; incrvec < n; incrvec <<= 1)
    {  /* for (incrvec...) */
      i2--;
      for (i0 = 0; i0 < incrvec >> 1; i0++)
       {  /* for (i0...) */
         for (i1 = 0; i1 < n; i1 += incrvec)
           {  /* for (i1...) */
             t1 = i0 + i1 + incrvec / 2;
             t2 = i0 << i2;
             t3 = i0 + i1;
             f0 = data[t1].real * w_common[t2].real -
              data[t1].imag * w_common[t2].imag;
             f1 = data[t1].real * w_common[t2].imag +
              data[t1].imag * w_common[t2].real;
             data[t1].real = data[t3].real - f0;
             data[t1].imag = data[t3].imag - f1;
             data[t3].real = data[t3].real + f0;
             data[t3].imag = data[t3].imag + f1;
           }  /* for (i1...) */
       }  /* for (i0...) */
    }  /* for (incrvec...) */

/*
 * Do the last stage of the FFT.
 */

  for (i0 = 0; i0 < n / 2; i0++)
    {  /* for (i0...) */
      t1 = i0 + n / 2;
      f0 = data[t1].real * w_common[i0].real -
       data[t1].imag * w_common[i0].imag;
      f1 = data[t1].real * w_common[i0].imag +
       data[t1].imag * w_common[i0].real;
      data[t1].real = data[i0].real - f0;
      data[t1].imag = data[i0].imag - f1;
      data[i0].real = data[i0].real + f0;
      data[i0].imag = data[i0].imag + f1;
    }  /* for (i0...) */
}  /* fft */

/*
 * bit_reverse ()
 *    inputs: a, n
 *    outputs: a
 */

void bit_reverse (a, n)
     COMPLEX *a;
     int n;

{
  int i, j, k;

/*
 * Begin function body: bit_reverse ()
 */

  j = 0;
  for (i = 0; i < n - 2; i++)
    {
      if (i < j)
       SWAP(a[j], a[i]);
      k = n >> 1;
      while (k <= j)
       {
         j -= k;
         k >>= 1;
       }
      j += k;
    }
}
```

A.1.6 read_input_SORT_FFT.c

```
/*
 * read_input_SORT_FFT.c
 */

/*
 * This file contains the procedure read_input_SORT_FFT (), and is part of
 * the parallel General benchmark program written for the IBM SP2 by the
 * STAP benchmark parallelization team at the University of Southern California
 * (Prof. Kai Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA
 * Mountaintop program.
 *
 * The sequential General benchmark program was originally written by Tony
 * Adams on 8/30/93.
 *
 * The procedure read_input_SORT_FFT () reads the input data files (containing
 * the data cube and the steering vectors).
 *
 * In this parallel version of read_input_SORT_FFT (), each task reads its
 * portion of the data cube in from the data file simulatenously. In order to
 * improve disk performance, the entire data cube slice is read from disk, then
 * converted from the packed binary integer format to the floating point
 * number format.
 *
 * Each complex number is stored on disk as a packed 32-bit binary integer.
 * The 16 LSBs are the real portion of the number, and the 16 MSBs are the
 * imaginary portion of the number.
 *
 * The steering vector file contains the number of PRIs in the input data,
 * the target power threshold, and the steering vectors. The data is stored
 * in ASCII format, and the complex steering vector numbers are stored as
 * alternating real and imaginary numbers.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"
#include <sys/types.h>
#include <sys/time.h>
#include <mpproto.h>

#ifdef DBLE
char *fmt = "%lf";
#else
char *fmt = "%f";
#endif

extern int taskid, numtask, allgrp;


 /*
  * read_input_SORT_FFT ()
  *    inputs: data_name, str_name
  *    outputs: v4, data_cube
  *
  *    data_name: the name of the file containing the data cube
  *    str_name: the name of the file containing the input vectors
  *    v4: storage for the v4 vector
  *    data_cube: input data cube
 */

read_input_SORT_FFT (data_name, data_cube)
     char data_name[];
     COMPLEX data_cube[][DIM1][DIM3];

{

/*
 * Variables
 *
 * dim1, dim2, dim3: dimension 1, 2, and 3 in the input cube, respectively
 * temp: buffer for binary input integer data
 * temp1, temp2: holders for integer data
 * f_temp1, f_temp2: holders for float data
 * junk: a temporary variable
 * f_power: holder for float power data
 * i, j, k: loop counters
 * local_int_cube: the integer version of the local portion of the data cube
 * blklen, rc: variables used for MPL calls
 */

  int dim1 = DIM1;
  int dim2 = DIM2;
  int dim3 = DIM3;
  unsigned int temp[1];
  long int temp1, temp2;
  float f_temp1, f_temp2;
  float f_pwr;
  int i, j, k;
  FILE *fopen();
  FILE *f_dat;
  unsigned int local_int_cube[DIM3][DIM2][DIM1];
  long blklen, rc;

/*
 * Begin function body: read_input_SORT_FFT ().
 *
 * Read in the data_cube file in parallel.
 */

  if ((f_dat = fopen (data_name, "r")) == NULL)
    {
      printf ("Error - task %d unable to open data file.\n", taskid);
      exit (-1);
    }

  fseek (f_dat, taskid * dim1 * dim2 * dim3 * sizeof (unsigned int), 0);

  fread (local_int_cube, sizeof (unsigned int), dim1 * dim2 * dim3, f_dat);
  fclose (f_dat);

/*
 * Convert data from unsigned int format to floating point format.
 */

  for (k = 0; k < dim3; k++)
      for (j = 0; j < dim2; j++)
          for (i = 0; i < dim1; i++)
            {
              temp[0] = local_int_cube[k][j][i];
              temp1 = 0x0000FFFF & temp[0];
              temp1 = (temp1 & 0x00008000) ? temp1 | 0xffff0000 : temp1;
              temp2 = (temp[0] >> 16) & 0x0000FFFF;
              temp2 = (temp2 & 0x00008000) ? temp2 | 0xffff0000 : temp2;
              data_cube[j][i][k].real = (float) temp1;
              data_cube[j][i][k].imag = (float) temp2;
      }
  return;
}
```

A.1.7 defs.h

```
/* defs.h */

/* 94 Aug 25 - We added a #define macro to compensate for the lack of a    */
/*             log2 function in the IBM version of math.h                  */

/* 94 Sep 13 - We added a definition for number of clock ticks per         */
/*             second, because this varies between the Sun OS and the IBM */
/*             AIX OS.                                                     */

/* Sun OS version - 60 clock ticks per second                              */

#ifdef SUN
#define CLK_TICK 60.0
#endif


/* IBM AIX version - 100 clock ticks per second                            */

#ifdef IBM
#define log2(x) ((log(x))/(M_LN2))
#define CLK_TICK 100.0
#endif


#ifdef DBLE

#define float double

#endif

typedef struct {
   float real;
   float imag;
} COMPLEX;

/* used in new fft */
#define SWAP(a,b) {float
swap_temp=(a).real;(a).real=(b).real;(b).real=swap_temp;\

swap_temp=(a).imag;(a).imag=(b).imag;(b).imag=swap_temp;}


#define NN 64                                  /* XXXUUUUUU */
#define PRI_CHUNK (PRI/NN)                         /* XXXXXUU */


#define AT_LEAST_ARG 2
#define AT_MOST_ARG 4
#define ITERATIONS_ARG 3
#define REPORTS_ARG 4

#define USERTIME(T1,T2)  ((t2.tms_utime-t1.tms_utime)/CLK_TICK)
#define SYSTIME(T1,T2)  ((t2.tms_stime-t1.tms_stime)/CLK_TICK)
#define USERTIME1(T1,T2)  ((time_end.tms_utime-time_start.tms_utime)/CLK_TICK)
#define SYSTIME1(T1,T2)  ((time_end.tms_stime-time_start.tms_stime)/CLK_TICK)
#define USERTIME2(T1,T2)  ((end_time.tms_utime-start_time.tms_utime)/CLK_TICK)
#define SYSTIME2(T1,T2)  ((end_time.tms_stime-start_time.tms_stime)/CLK_TICK)

#ifndef LINE_MAX
#define LINE_MAX 256
```

```
#endif
#define TRUE 1
#define FALSE 0

#define COLS 1536   /* Maximum number of columns in holding vector "vec" */
                    /* in house.c for max columns in Householder multiply */
#define MBEAM 12    /* Number of main beams */
#define ABEAM 20    /* Number of auxiliary beams */
#define PWR_BM 9    /* Number of max power beams */
#define V1 64
#define V2 128
#define V3 COLS
#define V4 V1
#define DIM1 V1     /* Number for dimension1 in input data cube */
#define DIM2 128    /* Number for dimension2 in input data cube */
#define DIM3 V3/NN  /* Number for dimension3 in input data cube */
#define DOP_GEN 2048 /* MAX Number of dopplers after FFT */
#define PRI_GEN 2048 /* MAX Number of points for 1500 data points FFT */
                    /*  zero filled after 1500 up to 2048 points */
#define NUM_MAT 32 /* Number of matrices to do householde on */
                    /* NUM_MAT must be less than DIM2 above */

#ifdef APT

#define PRI 256  /* Number of pris in input data cube */
#define DOP 256   /* Number of dopplers after FFT */
/* #define RNG 280 */    /* Number of range gates in input data cube */
#define RNG 256     /* Number of range gates in input data cube */
#define EL 32       /* number of elements in input data cube */
#define BEAM 32     /* Number of beams */
#define NUMSEG 7    /* Number of range gate segments */
#define RNGSEG RNG/NUMSEG  /* Number of range gates per segment */
#define RNG_S 320   /* Number sample range gates for 1st step beam forming */
#define DOF EL      /* Number of degrees of freedom */
extern COMPLEX t_matrix[BEAM][EL];   /* T matrix */
#endif


#ifdef STAP

#define PRI 128   /* Number of pris in input data cube */
#define DOP 128   /* Number of dopplers after FFT */
#define RNG 1250  /* Number of range gates in input data cube */
#define EL 48      /* number of elements in input data cube */
#define BEAM 2     /* Number of beams */
#define NUMSEG 2   /* Number of range gate segments */
#define RNGSEG RNG/NUMSEG /* Number of range gates per segment */
#define RNG_S 288  /* Number of sample range gates for beam forming */
#define DOF 3*EL   /* Number of degrees of freedom */

#endif

#ifdef GEN
#define EL V4
#define DOF EL

#endif

extern int output_time;    /* Flag if set TRUE, output execution times */
extern int output_report; /* Flag if set TRUE, output data report files */
extern int repetitions;    /* number of times program has executed */
extern int iterations;     /* number of times to execute program */

typedef struct max_index
{
  float value ;
  int loc1, loc2, loc3 ;
} INDEXED_MAX ;
```

```
extern void xu_index_max ( in1, in2, out, len )
INDEXED_MAX in1[], in2[], out[];
int *len ;
{
  int i, n ;
  n = *len/sizeof(int) ;
  for (i=0; i<n; i++)
    {
      if (in1[i].value > in2[i].value)
       {
         out[i].value = in1[i].value ;
         out[i].loc1 = in1[i].loc1;
         out[i].loc2 = in1[i].loc2;
         out[i].loc3 = in1[i].loc3;
       }
      else
       {
         out[i].value = in2[i].value ;
         out[i].loc1 = in2[i].loc1;
         out[i].loc2 = in2[i].loc2;
         out[i].loc3 = in2[i].loc3;
       }
    }
}
```

A.1.8 compile_sort

```
mpcc -O3 -qarch=pwr2 -DGEN -DIBM -o gen bench_mark_SORT_FFT.c bubble_sort.c
  cmd_line.c fft.c read_input_SORT_FFT.c  -lm
```

A.1.9 run.128

```
poe gen /scratch1/masa/vec_data -procs 128 -us
```

A.2 Vector Multiply Subprogram

A.2.1 bench_mark_VEC.c

```
/*
 * bench_mark_VEC.c
 */

/*
 * Parallel General Benchmark Program for the IBM SP2
 * -------------------------------------------------
 *
 * This parallel General benchmark program was written for the IBM SP2 by the
 * STAP benchmark parallelization team at the University of Southern
 * California (Prof. Kai Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as
 * part of the ARPA Mountaintop program.
 *
 * The sequential General benchmark program was originally written by Tony
 * Adams on 8/30/93.
 *
 *
 * This file contains the procedure main (), and represents the body of the
 * General Vector Multiply subprogram.
 *
 * This subprogram's overall structure is as follows:
 *    1. Load data in parallel.
 *    2. Perform DIM1*DIM3 DIM2-element vector multiplications using vector v2.
 *    3. Each task searches its slice of the data cube for the largest element.
 *    4. Perform a reduction operation to find the largest element in the entire
 *       data cube.
 *    5. Perform DIM2*DIM3 DIM1-element vector multiplications using vector v1.
 *    6. Each task searches its slice of the data cube for the largest element.
 *    7. Perform a reduction operation to find the largest element in the entire
 *       data cube.
 *    8. Redistribute the data cube using a total exchange operation.
 *    9. Perform DIM1*DIM2 DIM3-element vector multiplications using vector v3.
 *    10. Each task searches its slice of the data cube for the largest element.
 *    11. Perform a reduction operation to find the largest element in the
 *        entire data cube.
 *
 * This program can be compiled by calling the shell script compile_vec.
 * Because of the nature of the program, before the program is compiled, the
 * header file defs.h must be adjusted so that NN is set to the number of
 * nodes on which the program will be run. We have provided a defs.h file for
 * each power of 2 node size from 1 to 256, called defs.001 to defs.256.
 *
 * This program can be run by calling the shell script run.###, where ### is
 * the number of nodes on which the program is being run.
 *
 * Unlike the original sequential version of this program, the parallel General
 * program does not support command-line arguments specifying the number of
 * times the program body should be executed, nor whether or not timing
 * information should be displayed. The program body will be executed once,
 * and the timing information will be displayed at the end of execution.
 *
 * The input data file on disk is stored in a packed format: Each complex
 * number is stored as a 32-bit binary integer; the 16 LSBs are the real
 * half of the number, and the 16 MSBs are the imaginary half of the number.
 * These numbers are converted into floating point numbers usable by this
 * benchmark program as they are loaded from disk.
 *
 * The steering vectors file has the data stored in a different fashion. All
 * data in this file is stored in ASCII format. The first two numbers in this
 * file are the number of PRIs in the data set and the threshold level,
 * respectively. Then, the remaining data are the steering vectors, with
 * alternating real and imaginary numbers.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"
#include <sys/types.h>
#include <sys/times.h>
#include <mpproto.h>
#include <sys/time.h>
#include <time.h>


/*
 * main ()
 *    inputs: argc, argv
 *    outputs: none
 */
```

```
main (argc, argv)
   int argc;
   char *argv[];

{

/*
 * Variables
 *
 * data_cube: input data cube
 * new_cube: indexed data cube
 * data_vector: used in the inex step
 * v1, v2, v3: vectors used during the vector multiplications
 * t1, t2: holders for the time function return in seconds
 * dim1, dim2, dim3: dimensions 1, 2, and 3 of the input data cube
 * loc1, loc2, loc3: holders for the location of the maximum entry
 * temp: buffer for binary input integer data
 * f_temp1, f_temp2: holders for float data
 * f_pwr: holder for float power data
 * i, j, k, n, offset, r: loop counters
 * max: float variable to hold the maximum of complex data
 * max_pwr: holder for max float power data
 * real_prt, imag_prt: pointers to float data
 * xreal, ximag: holding vectors for vector multiply
 * vreal, vimag: holding vectors for float vectors v1, v2, and v3
 * index_max, g_index_max: data structures with the location and power of the
 *    largest element (locally and globally, repectively)
 * data_name, str_name: holders for the names of the files containing the
 *    input data cube and the vectors, respectively
 * totals: holders for DIM1 totals
 * global_totals: holders for the global DIM1 totals
 * sum: COMPLEX variable to hold the sum of complex data
 * blklen: block length
 * rc: return code from an MPL call
 * taskid: identifier for this task
 * numtask: total number of tasks running this program
 * nbuf: a buffer to hold data returned by mpc_task_query
 * allgrp: the identifier for the group encompassing all the tasks running
 *    this program
 * source, dest, type, nbytes: used by point-to-point communications
 */

  COMPLEX data_cube[DIM2][DIM1][DIM3];
  COMPLEX new_cube[DIM1][DIM2/NN][DIM3*NN];
  COMPLEX data_vector[DIM2*DIM1*DIM3];
  COMPLEX v1[V1];
  COMPLEX v2[V2];
  COMPLEX v3[V3];
  struct tms t1,t2;
  int dim1 = DIM1;
  int dim2 = DIM2;
  int dim3 = DIM3;
  int loc1,loc2,loc3;
  unsigned int temp[1];
  float f_temp1, f_temp2;
  float f_pwr;
  int i, j, k, n, offset, r;
  float max;
  float max_pwr;
  float *real_ptr, *imag_ptr;
  float xreal[COLS], ximag[COLS];
  float vreal[COLS], vimag[COLS];
  float *vreal_ptr, *vimag_ptr;
  INDEXED_MAX index_max[1], g_index_max[1] ;
  char data_name[LINE_MAX];
  char str_name[LINE_MAX];
  FILE *fopen();
  float totals[DIM1];
  float global_totals[DIM1];
  COMPLEX sum;
  int blklen;
  int rc;
  int taskid;
  int numtask;
  int nbuf[4];
  int allgrp;
  int source, dest, type, nbytes;

/*
 * Timing variables.
 */

  struct timeval tv0, tv1, tv2;
  struct tms all_start, all_end;
  float all_user, all_sys;
  float all_user_max, all_sys_max;
  float all_wall, task_wall;

  struct tms disk_start, disk_end;
  float disk_user, disk_sys;
  float disk_user_max, disk_sys_max;
  float disk_wall, disk_wall_max;

  struct tms index_start, index_end;
  float index_user, index_sys;
  float index_user_max, index_sys_max;
  float index_wall, index_wall_max;

  struct tms vec1_start, vec1_end;
  float vec1_user, vec1_sys;
  float vec1_user_max, vec1_sys_max;
  float vec1_wall, vec1_wall_max;

  struct tms vec2_start, vec2_end;
  float vec2_user, vec2_sys;
  float vec2_user_max, vec2_sys_max;
  float vec2_wall, vec2_wall_max;

  struct tms vec3_start, vec3_end;
  float vec3_user, vec3_sys;
  float vec3_user_max, vec3_sys_max;
  float vec3_wall, vec3_wall_max;


/*
 * Externally defined functions.
 */

  extern void cmd_line();
  extern void read_input_VEC();

/*
 * Begin function body: main ()
 *
 * Initialize for parallel processing. Here, each task or node determines its
 * task number (taskid) and the total number of tasks or nodes running this
 * program (numtask) by using the MPL call mpc_environ. Then, each task
 * determines the identifier for the group which encompasses all tasks or
 * nodes running this program. This identifier (allgrp) is used in collective
 * communication or aggregated computation operations, such as mpc_index.
 */

  gettimeofday(&tv0, (struct timeval*)0);           /* before time */

  rc = mpc_environ (&numtask, &taskid);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_environ.\n");
```

```
        exit (-1);
      }

  if (numtask != NN)
      {
        if (taskid == 0)
          {
            printf ("Error - task number mismatch... check defs.h.\n");
            exit (-1);
          }
      }

  rc = mpc_task_query (nbuf, 4, 3);
  if (rc == -1)
      {
        printf ("Error - unable to call mpc_task_query.\n");
        exit (-1);
      }
  allgrp = nbuf[3];

  if (taskid == 0)
      {
        printf ("Running...\n");
      }
  gettimeofday(&tv2, (struct timeval*)0);          /* before time */
  task_wall = (float) (tv2.tv_sec - tv0.tv_sec) * 1000000 + tv2.tv_usec
      - tv0.tv_usec;

  times (&all_start);

/*
 * Get arguments from command line. In the sequential version of the program,
 * the following procedure was used to extract the number of times the main
 * computational body was to be repeated, and flags regarding the amount of
 * reporting to be done during and after the program was run. In this paralle
 * program, there are no command line arguments to be extracted except for the
 * name of the file containing the data cube.
 */

    cmd_line (argc, argv, str_name, data_name);

/*
 * Read input files. In this section, each task loads its portion of the data
 * cube from the data file.
 */

  if (taskid == 0)
      {
        printf ("  loading data...\n");
      }

  mpc_sync (allgrp);
  times (&disk_start);
  gettimeofday(&tv1, (struct timeval*)0);          /* before time */
  read_input_VEC (data_name, str_name, v1, v2, v3, data_cube) ;
  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  disk_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
      - tv1.tv_usec;

  times (&disk_end);


/*
 * Start Vector Multiplication steps.
 */

  if (taskid==0)
    printf("VECTOR multiply data_cube by v1,v2,v3 and output largest values\n");
```

```
/*
 * Perform DIM1*DIM3 DIM2 element vector multiplies using vector v2.
 */

  if (taskid == 0)
    printf ("Perform %d*%d number of %d element multiplies by vector v2\n",
            DIM1, DIM3*NN, DIM2 );

  times (&vec2_start);
  gettimeofday(&tv1, (struct timeval*)0);          /* before time */

/*
 * Initialize max_pwr = 0.0 and location = 0,0,0.
 */

  max_pwr = 0.0;
  loc1 = 0;
  loc2 = 0;
  loc3 = 0;

/*
 * Move vector v2 into holding vectors for the use of pointers.
 */

  vreal_ptr = vreal;
  vimag_ptr = vimag;
  for (n = 0; n < V2; n++)
    {
      *vreal_ptr++ = v2[n].real;
      *vimag_ptr++ = v2[n].imag;
    }
  for (i = 0; i < dim1; i++)
    for (k = 0; k < dim3; k++)
      {
        real_ptr = xreal;
        imag_ptr = ximag;
        vreal_ptr = vreal;
        vimag_ptr = vimag;
        for (j = 0; j < dim2; j++)
          {
            *real_ptr++ = data_cube[j][i][k].real * *vreal_ptr -
              data_cube[j][i][k].imag * *vimag_ptr;
            *imag_ptr++ = data_cube[j][i][k].real * *vimag_ptr++ +
              data_cube[j][i][k].imag * *vreal_ptr++;
          }

/*
 * Find the element with the largest magnitude in this task's slice of the
 * data cube.
 */

        real_ptr = xreal;
        imag_ptr = ximag;
        for (j = 0; j < dim2; j++)
         {
            f_pwr = *real_ptr * *real_ptr + *imag_ptr * *imag_ptr;
            *real_ptr++;
            *imag_ptr++;
            if (max_pwr < f_pwr)
              {
                max_pwr = f_pwr;
                loc1 = i;
                loc2 = j;
                loc3 = k;
              }
          }
      }

/*
```

```
 *  printf("task %d finds max %f at %d %d %d\n",
 *      taskid, max_pwr, loc1, loc2, loc3);
 */

  index_max[0].value = max_pwr;
  index_max[0].loc1 = loc1;
  index_max[0].loc2 = loc2;
  index_max[0].loc3 = loc3 + taskid * dim3;

/*
 * Now aggregate to get the global maximum (the element with the largest
 * magnitude in the entire data cube).
 */

  rc = mpc_reduce (index_max, g_index_max, sizeof (INDEXED_MAX), 0,
                   xu_index_max, allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  vec2_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;

  times (&vec2_end);

  if (taskid == 0 )
    {
      max_pwr = g_index_max[0].value ;
      loc1 = g_index_max[0].loc1 ;
      loc2 = g_index_max[0].loc2 ;
      loc3 = g_index_max[0].loc3 ;
      printf("Max value for v2 vector multiply = %f\n",  max_pwr);
      printf("Location = %d %d %d\n", loc1, loc2, loc3);
    }

/*
 * Perform DIM2*DIM3 DIM1 element vector multiplies using vector v1.
 */

  if (taskid==0)
    printf ("Perform %d*%d number of %d element multiplies by vector v1\n",
            DIM1, DIM3*NN, DIM2 );

  times (&vec1_start);
  gettimeofday(&tv1, (struct timeval*)0);          /* before time */

/*
 * Initialize max_pwr = 0.0 and location = 0,0,0.
 */

  max_pwr = 0.0;
  loc1 = 0;
  loc2 = 0;
  loc3 = 0;

/*
 * Move vector v1 into holding vectors for the use of pointers.
 */

  vreal_ptr = vreal;
  vimag_ptr = vimag;
  for (n = 0; n < V1; n++)
    {
      *vreal_ptr++ = v1[n].real;
      *vimag_ptr++ = v1[n].imag;
    }
  for (j = 0; j < dim2; j++)
```

```
    for (k = 0; k < dim3; k++)
      {
        real_ptr = xreal;
        imag_ptr = ximag;
        vreal_ptr = vreal;
        vimag_ptr = vimag;
        for (i = 0; i < dim1; i++)
          {
            *real_ptr++ = data_cube[j][i][k].real * *vreal_ptr -
              data_cube[j][i][k].imag * *vimag_ptr;
            *imag_ptr++ = data_cube[j][i][k].real * *vimag_ptr++ +
              data_cube[j][i][k].imag * *vreal_ptr++;
          }

/*
 * Find the element with the largest magnitude in this tasks's slice of the
 * data cube.
 */

        real_ptr = xreal;
        imag_ptr = ximag;
        for (i = 0; i < dim1; i++)
         {
            f_pwr = *real_ptr * *real_ptr + *imag_ptr * *imag_ptr;
            *real_ptr++;
            *imag_ptr++;
            if (max_pwr < f_pwr)
              {
                max_pwr = f_pwr;
                loc1 = i;
                loc2 = j;
                loc3 = k;
              }
          }
      }

/*
 *  printf("task %d finds max %f at %d %d %d\n",
 *      taskid, max_pwr, loc1, loc2, loc3);
 */

  index_max[0].value = max_pwr;
  index_max[0].loc1 = loc1;
  index_max[0].loc2 = loc2;
  index_max[0].loc3 = loc3 + taskid * dim3;

/*
 * Now aggregate to get the global maximum (the element with the largest
 * magnitude in the entire data cube).
 */

  rc = mpc_reduce (index_max, g_index_max, sizeof (INDEXED_MAX), 0,
                xu_index_max, allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  vec1_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;

  times (&vec1_end);

  if (taskid == 0)
    {
      max_pwr = g_index_max[0].value ;
      loc1 = g_index_max[0].loc1 ;
      loc2 = g_index_max[0].loc2 ;
      loc3 = g_index_max[0].loc3 ;
      printf("Max value for v2 vector multiply   %f\n",  max_pwr);
      printf("Location = %d %d %d\n", loc1, loc2, loc3);
    }

/*
 * Index data_cube[DIM2][DIM1][DIM3] to new_cube[DIM1][DIM2/NN][DIM3*NN].
 * This total exchange operation is needed so that each task has all the data
 * along the DIM3 dimension (the dimension along which the last set of vector
 * multiplications is performed).
 */

  if (taskid == 0)
    {
      printf ("  indexing data cube...\n");
    }

  mpc_sync (allgrp);
  times (&index_start);
  gettimeofday(&tv1, (struct timeval*)0);          /* before time */

  rc = mpc_index (data_cube, data_vector,
                DIM1 * DIM2 * DIM3 * sizeof (COMPLEX) / NN, allgrp);

  if (rc == -1)
    {
      printf ("Error - unable to call mpc_index.\n");
      exit (-1);
    }

  if (taskid == 0)
    {
      printf ("  rewinding data cube...\n");
    }
  mpc_sync (allgrp);

  offset = 0;
  for (n = 0; n < NN; n++)
    for (j= 0 ; j < DIM2/NN; j++)
      for (i = 0; i < DIM1 ; i++)
        for (k = n * DIM3; k < (n + 1) * DIM3 ; k++)
          {
            new_cube[i][j][k].real = data_vector[offset].real;
            new_cube[i][j][k].imag = data_vector[offset].imag;
            offset++;
          }
  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  index_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;

  times (&index_end);

/*
 * Perform DIM1*DIM2 DIM3 element vector multiplies using vector v3.
 */

  if (taskid==0)
    printf ("Perform %d*%d number of %d element multiplies by vector v3\n",
            DIM1, DIM2, DIM3*NN );
  times (&vec3_start);
  gettimeofday(&tv1, (struct timeval*)0);          /* before time */

/*
 * Initialize max_pwr = 0.0 and location = 0,0,0.
 */

  max_pwr = 0.0;
  loc1 = 0;
  loc2 = 0;
  loc3 = 0;

/*
 * Move vector v3 into holding vectors for the use of pointers.
 */

  vreal_ptr = vreal;
  vimag_ptr = vimag;
  for (n = 0; n < V3; n++)
    {
      *vreal_ptr++ = v3[n].real;
      *vimag_ptr++ = v3[n].imag;
    }
  for (i = 0; i < dim1; i++)
    for (j = 0; j < dim2/NN; j++)
      {
       real_ptr = xreal;
       imag_ptr = ximag;
       vreal_ptr = vreal;
       vimag_ptr = vimag;
       for (k = 0; k < dim3*NN; k++)
         {
           *real_ptr++ = new_cube[i][j][k].real * *vreal_ptr -
             new_cube[i][j][k].imag * *vimag_ptr;
           *imag_ptr++ = new_cube[i][j][k].real * *vimag_ptr++ +
             new_cube[i][j][k].imag * *vreal_ptr++;
         }

/*
 * Finding the element with the largest magnitude in this task's slice of the
 * data cube.
 */

       real_ptr = xreal;
       imag_ptr = ximag;
       for (k = 0; k < dim3*NN; k++)
         {
           f_pwr = *real_ptr * *real_ptr + *imag_ptr * *imag_ptr;
           *real_ptr++;
           *imag_ptr++;
           if (max_pwr < f_pwr)
             {
              max_pwr = f_pwr;
              loc1 = i;
              loc2 = j;
              loc3 = k;
             }
         }
      }

/*
 *  printf("task %d finds max %f at %d %d %d\n",
 *      taskid, max, loc1, loc2, loc3);
 */

  index_max[0].value = max_pwr;
  index_max[0].loc1 = loc1;
  index_max[0].loc2 = loc2 + taskid * dim2/NN;
  index_max[0].loc3 = loc3;

/*
 * Now aggregate to get the global maximum (the element with the largest
 * magnitude in the entire data cube).
 */

  rc = mpc_reduce (index_max, g_index_max, sizeof (INDEXED_MAX), 0,
                   xu_index_max, allgrp);
  if (rc == -1)
    {
        printf ("Error - unable to call mpc_reduce.\n");
        exit (-1);
      }

  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  vec3_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;

  times (&vec3_end);
  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  all_wall = (float) (tv2.tv_sec - tv0.tv_sec) * 1000000 + tv2.tv_usec
    - tv0.tv_usec;
  times(&all_end);

  if (taskid == 0)
    {
      max_pwr = g_index_max[0].value;
      loc1 = g_index_max[0].loc1;
      loc2 = g_index_max[0].loc2;
      loc3 = g_index_max[0].loc3;
      printf("Max value for v1 vector multiply = %f\n", max_pwr);
      printf("Location = %d %d %d\n", loc1, loc2, loc3);
    }

/*
 * Compute all times.
 */

  all_user = (float) (all_end.tms_utime - all_start.tms_utime)/100.0;
  all_sys = (float)(all_end.tms_stime - all_start.tms_stime)/100.0;
  rc = mpc_reduce (&all_user, &all_user_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&all_sys, &all_sys_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }

/*
 * Compute disk times.
 */

  rc = mpc_reduce (&disk_wall, &disk_wall_max, sizeof(float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  disk_user = (float) (disk_end.tms_utime - disk_start.tms_utime)/100.0;
  disk_sys = (float)(disk_end.tms_stime - disk_start.tms_stime)/100.0;
  rc = mpc_reduce (&disk_user, &disk_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&disk_sys, &disk_sys_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
```

```
/*
 * Compute vec1 times.
 */
  rc = mpc_reduce (&vec1_wall, &vec1_wall_max, sizeof(float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  vec1_user = (float) (vec1_end.tms_utime - vec1_start.tms_utime)/100.0;
  vec1_sys = (float)(vec1_end.tms_stime - vec1_start.tms_stime)/100.0;

  if (taskid==0)
    printf("vec1 sys = %f  user = %f\n", vec1_sys,vec1_user);
  rc = mpc_reduce (&vec1_user, &vec1_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&vec1_sys, &vec1_sys_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }

/*
 * Compute vec2 times.
 */

  rc = mpc_reduce (&vec2_wall, &vec2_wall_max, sizeof(float); 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  vec2_user = (float) (vec2_end.tms_utime - vec2_start.tms_utime)/100.0;
  vec2_sys = (float)(vec2_end.tms_stime - vec2_start.tms_stime)/100.0;
  rc = mpc_reduce (&vec2_user, &vec2_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&vec2_sys, &vec2_sys_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }

/*
 * Compute vec3 times.
 */
  rc = mpc_reduce (&vec3_wall, &vec3_wall_max, sizeof(float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  vec3_user = (float) (vec3_end.tms_utime - vec3_start.tms_utime)/100.0;
  vec3_sys = (float)(vec3_end.tms_stime - vec3_start.tms_stime)/100.0;
```

```
   rc = mpc_reduce (&vec3_user, &vec3_user_max, sizeof (float), 0, s_vmax,
                    allgrp);
   if (rc == -1)
     {
       printf ("Error mpc_reduce.\n");
       exit (-1);
     }
   rc = mpc_reduce (&vec3_sys, &vec3_sys_max, sizeof (float), 0, s_vmax,allgrp);
   if (rc == -1)
     {
       printf ("Error mpc_reduce.\n");
       exit (-1);
     }

/*
 * Compute index times.
 */

   rc = mpc_reduce (&index_wall, &index_wall_max, sizeof(float), 0, s_vmax,
                    allgrp);
   if (rc == -1)
     {
       printf ("Error mpc_reduce.\n");
       exit (-1);
     }
   index_user = (float) (index_end.tms_utime - index_start.tms_utime)/100.0;
   index_sys = (float)(index_end.tms_stime - index_start.tms_stime)/100.0;
   rc = mpc_reduce (&index_user, &index_user_max, sizeof (float), 0, s_vmax,
                    allgrp);
   if (rc == -1)
     {
       printf ("Error mpc_reduce.\n");
       exit (-1);
     }
   rc = mpc_reduce (&index_sys, &index_sys_max, sizeof (float), 0, s_vmax,
                    allgrp);
   if (rc == -1)
     {
       printf ("Error mpc_reduce.\n");
       exit (-1);
     }

/*
 * Display timing information.
 */

   if (taskid == 0)
     {
       printf ("\n\n*** CPU Timing information - numtask = %d\n\n", NN);
       printf ("  all_user_max      = %.2f s, all_sys_max      = %.2f s\n",
               all_user_max, all_sys_max);
       printf ("  disk_user_max     = %.2f s, disk_sys_max     = %.2f s\n",
               disk_user_max, disk_sys_max);
       printf ("  vec1_user_max     = %.2f s, vec1_sys_max     = %.2f s\n",
               vec1_user_max, vec1_sys_max);
       printf ("  vec2_user_max     = %.2f s, vec2_sys_max     = %.2f s\n",
               vec2_user_max, vec2_sys_max);
       printf ("  vec3_user_max     = %.2f s, vec3_sys_max     = %.2f s\n",
               vec3_user_max, vec3_sys_max);
       printf ("  index_user_max     = %.2f s, index_sys_max     = %.2f s\n",
               index_user_max, index_sys_max);

       printf ("\n*** Wall Clock Timing information - numtask = %d\n\n", NN);
       printf ("  all_wall          = %.0f us\n", all_wall);
       printf ("  task_wall          = %.0f us\n", task_wall);
       printf ("  disk_wall          = %.0f us\n", disk_wall_max);
       printf ("  vec1_wall          = %.0f us\n", vec1_wall_max);
       printf ("  vec2_wall          = %.0f us\n", vec2_wall_max);
       printf ("  vec3_wall          = %.0f us\n", vec3_wall_max);
```

```
      printf ("  index_wall        = %.0f us\n", index_wall_max);
    }
  exit(0);
}
```

A.2.2 cmd_line.c

```
/*
 * cmd_line.c
 */

/*
 * This file contains the procedure cmd_line (), and is part of the parallel
 * General benchmark program written for the IBM SP2 by the STAP benchmark
 * parallelization team at the University of Southern California (Prof. Kai
 * Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA Mountaintop
 * program.
 *
 * The sequential General benchmark program was originally written by Tony
 * Adams on 8/30/93.
 *
 * The procedure cmd_line () extracts the name of the files from which the
 * input data cube and the steering vector data should be loaded. The
 * function of this parallel version of cmd_line () is different from that
 * of the sequential version, because the sequential version also extracted
 * the number of iterations the program should run and some reporting
 * options.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"
#include <sys/types.h>
#include <sys/time.h>

/*
 * cmd_line ()
 *   inputs: argc, argv
 *   outputs: str_name, data_name
 *
 *   argc, argv: these are used to get data from the command line arguments
 *   str_name: a holder for the name of the input steering vectors file
 *   data_name: a holder for the name of the input data file
 */

cmd_line (argc, argv, str_name, data_name)
     int argc;
     char *argv[];
     char str_name[LINE_MAX];
     char data_name[LINE_MAX];

{

/*
 * Begin function body: cmd_line ()
 */

  strcpy (str_name, argv[1]);
  strcat (str_name, ".str");
  strcpy (data_name, argv[1]);
  strcat (data_name, ".dat");
  return;
}
```

A.2.3 read_input_VEC.c

```
/*
 * read_input_VEC.c
 */

/*
 * This file contains the procedure read_input_VEC (), and is part of the
 * parallel General benchmark program written for the IBM SP2 by the STAP
 * benchmark parallelization team at the University of Southern Calfornia
 * (Prof. Kai Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA
 * Mountaintop program.
 *
 * The sequential General benchmark program was originally written by Tony
 * Adams on 8/30/93.
 *
 * The procedure read_input_VEC () reads the input data files (containing the
 * data cube and the steering vectors).
 *
 * In this parallel version of read_input_VEC (), each task reads its portion
 * of the data cube in from the data file simulatenously. In order to improve
 * disk performance, the entire data cube slice is read from disk, then
 * converted from the packed binary integer format to the floating point
 * number format.
 *
 * Each complex number is stored on disk as a packed 32-bit binary integer.
 * The 16 LSBs are the real portion of the number, and the 16 MSBs are the
 * imaginary portion of the number.
 *
 * The steering vector file contains the number of PRIs in the input data,
 * the target power threshold, and the steering vectors. The data is stored
 * in ASCII format, and the complex steering vector numbers are stored as
 * alternating real and imaginary numbers.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"
#include <sys/types.h>
#include <sys/time.h>
#include <mpproto.h>

#ifdef DBLE
char *fmt = "%lf";
#else
char *fmt = "%f";
#endif

extern int taskid, numtask, allgrp;



/*
 * read_input_VEC ()
 *   inputs: data_name, str_name
 *   outputs: v1, v2, v3, data_cube
 *
 *   data_name: the name of the file containing the data cube
 *   str_name: the name of the file containing the input vectors
 *   v1, v2, v3: storage for the v1, v2, and v3 vectors, respectively
 *   data_cube: input data cube
 */

read_input_VEC (data_name, str_name, v1, v2, v3, data_cube)
  char data_name[];
  char str_name[];
  COMPLEX v1[];
```

```
    COMPLEX v2[];
    COMPLEX v3[];
    COMPLEX data_cube[][DIM1][DIM3];

  {

  /*
   * Variables
   *
   * dim1, dim2, dim3: dimension 1, 2, and 3 in the input cube, respectively
   * temp: buffer for binary input integer data
   * temp1, temp2: holders for integer data
   * f_temp1, f_temp2: holders for float data
   * junk: a temporary variable
   * f_power: holder for float power data
   * i, j, k: loop counters
   * local_int_cube: the integer version of the local portion of the data cube
   * blklen, rc: variables used for MPL calls
   */

    int dim1 = DIM1;
    int dim2 = DIM2;
    int dim3 = DIM3;
    unsigned int temp[1];
    long int temp1, temp2;
    float f_temp1, f_temp2;
    float junk;
    float f_pwr;
    int i, j, k;
    FILE *fopen();
    FILE *f_dat, *f_vec;
    unsigned int local_int_cube[DIM3][DIM2][DIM1];
    long blklen, rc;

  /*
   * Begin function body: read_input_VEC ().
   *
   * Read in the data_cube file in parallel.
   */

    if ((f_dat = fopen (data_name, "r")) == NULL)
      {
        printf ("Error - task %d unable to open data file.\n", taskid);
        exit (-1);
      }

    fseek (f_dat, taskid * dim1 * dim2 * dim3 * sizeof (unsigned int), 0);
    fread (local_int_cube, sizeof (unsigned int), dim1 * dim2 * dim3, f_dat);
    fclose (f_dat);

  /*
   * Convert the data from the unsigned integer format to floating point
   * format.
   */

    for (k = 0; k < dim3; k++)
        for (j = 0; j < dim2; j++)
           for (i = 0; i < dim1; i++)
             {
               temp[0] = local_int_cube[k][j][i];
               temp1 = 0x0000FFFF & temp[0];
               temp1 = (temp1 & 0x00008000) ? temp1 | 0xffff0000 : temp1;
               temp2 = (temp[0] >> 16) & 0x0000FFFF;
               temp2 = (temp2 & 0x00008000) ? temp2 | 0xffff0000 : temp2;
               data_cube[j][i][k].real = (float) temp1;
               data_cube[j][i][k].imag = (float) temp2;
        }

/*
```

```
 * Open the vector file using the file pointer f_vec.
 */

  if ((f_vec = fopen (str_name, "r")) == NULL)
    {
      printf (" Cant open input steering vector file %s\n ", str_name);
      exit (-1);
    }

/*
 * Read the VECTORS from input file with vectors in v1,v2,v3 order.
 */

   for (i = 0; i < V1; i++)
     {
      fscanf (f_vec, fmt, &v1[i].real);
      fscanf (f_vec, fmt, &v1[i].imag);
    }

   for (i = 0; i < V2; i++)
     {
      fscanf (f_vec, fmt, &v2[i].real);
      fscanf (f_vec, fmt, &v2[i].imag);
    }

   for (i = 0 ; i < V3; i++)
     {
      fscanf (f_vec, fmt, &v3[i].real);
      fscanf (f_vec, fmt, &v3[i].imag);
    }

  fclose(f_vec);
  return;
}
```

A.2.4 defs.h

```
/* defs.h */

/* 94 Aug 25 - We added a #define macro to compensate for the lack of a   */
/*             log2 function in the IBM version of math.h                 */

/* 94 Sep 13 - We added a definition for number of clock ticks per        */
/*             second, because this varies between the Sun OS and the IBM */
/*             AIX OS.                                                    */

/* Sun OS version - 60 clock ticks per second                             */

#ifdef SUN
#define CLK_TICK 60.0
#endif


/* IBM AIX version - 100 clock ticks per second                           */

#ifdef IBM
#define log2(x) ((log(x))/(M_LN2))
#define CLK_TICK 100.0
#endif


#ifdef DBLE
```

```
#define float double

#endif

typedef struct {
   float real;
   float imag;
} COMPLEX;

/* used in new fft */
#define SWAP(a,b) {float
swap_temp=(a).real;(a).real=(b).real;(b).real=swap_temp;\

swap_temp=(a).imag;(a).imag=(b).imag;(b).imag=swap_temp;}


#define NN 2                                     /* XXXUUUUUU */
#define PRI_CHUNK (PRI/NN)                           /* XXXXXUU */


#define AT_LEAST_ARG 2
#define AT_MOST_ARG 4
#define ITERATIONS_ARG 3
#define REPORTS_ARG 4

#define USERTIME(T1,T2)   ((t2.tms_utime-t1.tms_utime)/CLK_TICK)
#define SYSTIME(T1,T2)   ((t2.tms_stime-t1.tms_stime)/CLK_TICK)
#define USERTIME1(T1,T2)   ((time_end.tms_utime-time_start.tms_utime)/CLK_TICK)
#define SYSTIME1(T1,T2)   ((time_end.tms_stime-time_start.tms_stime)/CLK_TICK)
#define USERTIME2(T1,T2)   ((end_time.tms_utime-start_time.tms_utime)/CLK_TICK)
#define SYSTIME2(T1,T2)   ((end_time.tms_stime-start_time.tms_stime)/CLK_TICK)

#ifndef LINE_MAX
#define LINE_MAX 256
#endif
#define TRUE 1
#define FALSE 0

#define COLS 1536   /* Maximum number of columns in holding vector "vec" */
                    /* in house.c for max columns in Householder multiply */
#define MBEAM 12    /* Number of main beams */
#define ABEAM 20    /* Number of auxiliary beams */
#define PWR_BM 9    /* Number of max power beams */
#define V1 64
#define V2 128
#define V3 COLS
#define V4 V1
#define DIM1 V1     /* Number for dimension1 in input data cube */
#define DIM2 128    /* Number for dimension2 in input data cube */
#define DIM3 V3/NN   /* Number for dimension3 in input data cube */
#define DOP_GEN 2048 /* MAX Number of dopplers after FFT */
#define PRI_GEN 2048 /* MAX Number of points for 1500 data points FFT */
                    /*  zero filled after 1500 up to 2048 points */
#define NUM_MAT 32 /* Number of matrices to do householde on */
                    /* NUM_MAT must be less than DIM2 above */

#ifdef APT

#define PRI 256   /* Number of pris in input data cube */
#define DOP 256    /* Number of dopplers after FFT */
/* #define RNG 280 */     /* Number of range gates in input data cube */
#define RNG 256     /* Number of range gates in input data cube */
#define EL 32       /* number of elements in input data cube */
#define BEAM 32     /* Number of beams */
#define NUMSEG 7    /* Number of range gate segments */
#define RNGSEG RNG/NUMSEG  /* Number of range gates per segment */
#define RNG_S 320   /* Number sample range gates for 1st step beam forming */
#define DOF EL      /* Number of degrees of freedom */
extern COMPLEX t_matrix[BEAM][EL];    /* T matrix */
```

```
#endif


#ifdef STAP

#define PRI 128     /* Number of pris in input data cube */
#define DOP 128     /* Number of dopplers after FFT */
#define RNG 1250    /* Number of range gates in input data cube */
#define EL 48        /* number of elements in input data cube */
#define BEAM 2       /* Number of beams */
#define NUMSEG 2     /* Number of range gate segments */
#define RNGSEG RNG/NUMSEG /* Number of range gates per segment */
#define RNG_S 288    /* Number of sample range gates for beam forming */
#define DOF 3*EL     /* Number of degrees of freedom */

#endif

#ifdef GEN
#define EL V4
#define DOF EL

#endif

extern int output_time;    /* Flag if set TRUE, output execution times */
extern int output_report; /* Flag if set TRUE, output data report files */
extern int repetitions;    /* number of times program has executed */
extern int iterations;     /* number of times to execute program */

typedef struct max_index
{
  float value ;
  int loc1, loc2, loc3 ;
} INDEXED_MAX ;

extern void xu_index_max ( in1, in2, out, len )
INDEXED_MAX in1[], in2[], out[];
int *len ;
{
  int i, n ;
  n = *len/sizeof(int) ;
  for (i=0; i<n; i++)
    {
      if (in1[i].value > in2[i].value)
       {
         out[i].value = in1[i].value ;
         out[i].loc1 = in1[i].loc1;
         out[i].loc2 = in1[i].loc2;
         out[i].loc3 = in1[i].loc3;
       }
      else
       {
         out[i].value = in2[i].value ;
         out[i].loc1 = in2[i].loc1;
         out[i].loc2 = in2[i].loc2;
         out[i].loc3 = in2[i].loc3;
       }
    }
}
```

A.2.5 compile_vec

```
mpcc -O3 -qarch=pwr2 -DGEN -DIBM -o vec bench_mark_VEC.c cmd_line.c
  read_input_VEC.c  -lm
```

A.2.6 run.128

```
poe vec /scratch1/masa/vec_data -procs 128 -us
```

A.3 Linear Algebra Subprogram

A.3.1 bench_mark_LIN.c

```
/*
 * bench_mark_LIN.c
 */

/*
 * Parallel General Benchmark Program for the IBM SP2
 * -------------------------------------------------
 *
 * This parallel General benchmark program was written for the IBM SP2 by the
 * STAP benchmark parallelization team at the University of Southern
 * California (Prof. Kai Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as
 * part of the ARPA Mountaintop program.
 *
 * The sequential General benchmark program was originally written by Tony
 * Adams on 8/30/93.
 *
 *
 * This file contains the procedure main (), and represents the body of the
 * General Linear Algebra subprogram.
 *
 * This subprogram's overall structure is as follows:
 *   1. Load data in parallel.
 *   2. Perform the Householder transform on the data in parallel.
 *   3. Perform forward and back substitution on the data in parallel.
 *   4. Apply weight vectors to the data in parallel.
 *
 * This program can be compiled by calling the shell script compile_lin.
 * Because of the nature of the program, before the program is compiled, the
 * header file defs.h must be adjusted so that NN is set to the number of
 * nodes on which the program will be run. We have provided a defs.h file for
 * each power of 2 node size from 1 to 32, called defs.001 to defs.032.
 *
 * This program can be run by calling the shell script run.###, where ### is
 * the number of nodes on which the program is being run.
 *
 * Unlike the original sequential version of this program, the parallel General
 * program does not support command-line arguments specifying the number of
 * times the program body should be executed, nor whether or not timing
 * information should be displayed. The program body will be executed once,
 * and the timing information will be displayed at the end of execution.
 *
 * The input data file on disk is stored in a packed format: Each complex
 * number is stored as a 32-bit binary integer; the 16 LSBs are the real
 * half of the number, and the 16 MSBs are the imaginary half of the number.
 * These numbers are converted into floating point numbers usable by this
 * benchmark program as they are loaded from disk.
 *
 * The steering vectors file has the data stored in a different fashion. All
 * data in this file is stored in ASCII format. The first two numbers in this
 * file are the number of PRIs in the data set and the threshold level,
 * respectively. Then, the remaining data are the steering vectors, with
 * alternating real and imaginary numbers.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"
#include <sys/types.h>
#include <sys/times.h>
#include <mpproto.h>
#include <sys/time.h>
#include <time.h>



/*
 * main ()
 *    inputs: argc, argv
 *    outputs: none
 */

main (argc, argv)
   int argc;
   char *argv[];
{

/*
 * Variables
 *
 * data_cube: input data cube
 * v4: vector of complex numbers
 * temp_mat: temporary matrix
 * beams_mat: temporary matrix for holding output beam data
 * dim1, dim2, dim3: dimensions 1, 2, and 3 in the input data cube,
 *    respectively
 * loc1, loc2, loc3: holders for the location of the maximum entry
 * start_row: holder for the row on which to start the Householder
 * lower_triangular_rows: holder for the number of rows to lower triangularize
 * num_rows: holder for the total number of rows
 * num_cols: holder for the total number of columns
 * i, j, k: loop counters
 * dopplers: number of dopplers after the FFT
 * weight_vec: pointer to the start of the COMPLEX weight vector
 * xreal, ximag: pointer to the start of the real and imaginary parts of the
 *    float data vector
 * max: float variable to hold the maximum of the complex data
 * sum: variable to hold the sum of the complex data
 * f_pwr, f_temp1, f_temp2: float variables
 * real_ptr, imag_ptr: pointer to float variables
 * str_vecs: pointer to the start of the steering vector array
 * make_t: a flag; 1 = make T matrix, 0 = don't ma eT matrix but pass back the
 *    weight vector
 * index_max: a data structure holding the location and value of the maximum
 * g_index_max: a data structure holding the location and value of the maximum
 * data_name, str_name: the names of the files containing the input data cube
 *    and the steering vectors, respectively
 * blklen: block length
 * rc: return code from an MPL call
 * taskid: the identifier for this task
 * numtask: the total number of tasks running this program
 * nbuf: a buffer to hold the data from the MPL call mpc_task_query
 * allgrp: the identifier representing all the tasks running this program
 * source, dest, type, nbytes: parameters to point-to-point MPL calls
 */

  COMPLEX data_cube[DIM1][DIM2][DIM3];
  COMPLEX v4[V4];
  static COMPLEX temp_mat[V4][COLS];
  COMPLEX **beams_mat;
  int dim1 = DIM1;
  int dim2 = DIM2;
  int dim3 = DIM3;
  int loc1,loc2,loc3;
  int start_row;
  int lower_triangular_rows;
  int num_rows;
  int num_cols = V3;
  int i, j, k;
  int dopplers;
  COMPLEX weight_vec[V4];
  float xreal[V4];
  float ximag[V4];
  float max;
  COMPLEX sum;
  float f_pwr;
  float f_temp1;
  float f_temp2;
  float *real_ptr;
  float *imag_ptr;
  COMPLEX str_vecs[1][DOF];
  int make_t;
  INDEXED_MAX index_max[NUM_MAT];
  INDEXED_MAX g_index_max[NUM_MAT*NN];
  char data_name[LINE_MAX];
  char str_name[LINE_MAX];

  int blklen;
  int rc;
  int taskid;
  int numtask;
  int nbuf[4];
  int allgrp;
  int source, dest, type, nbytes;

/*
 * Timing variables
 *
 * all_start, all_end: start and end CPU times for the entire program
 * disk_start, disk_end: start and end CPU times for the disk access step
 * lin_start, lin_end: start and end CPU times for the linear algebra step
 * t1, t2, tv0, tv1, tv2: temporary variables
 * all_user, all_sys: user and system CPU times for the entire program
 * all_user_max, all_sys_max: maximum user and system CPU times for the entire
 *    program
 * disk_user, disk_sys: user and system CPU times for the disk access setp
 * disk_user_max, disk_sys_max: maximum user and system CPU times for the
 *    entire program
 * all_wall, task_wall: wall clock time and maximum wall clock time for the
 *    entire program
 * disk_wall, disk_wall_max: wall clock time and maximum wall clock time for
 *    the disk access step
 * lin_user, lin_sys: user and system CPU times for the linear algebra step
 * lin_user_max, lin_sys_max: maximum user and system CPU times for the
 *    linear algebra step
 * lin_wall, lin_wall_max: wall clock time and maximum wall clock time for the
 *    linear algebra step
 */

  struct tms all_start, all_end;
  struct tms disk_start, disk_end;
  struct tms lin_start, lin_end;
  struct tms t1,t2;
  struct timeval tv0, tv1, tv2;

  float all_user, all_sys;
  float all_user_max, all_sys_max;
  float disk_user, disk_sys;
  float disk_user_max, disk_sys_max;
  float all_wall, task_wall;
  float disk_wall, disk_wall_max;

  float lin_user, lin_sys;
  float lin_user_max, lin_sys_max;
  float lin_wall, lin_wall_max;

/*
 * Externally defined functions
 */

  extern void cmd_line();
  extern void read_input_LIN();
  extern void house();
  extern void forback();

/*
 * Begin function body: main ()
 *
 * Initialize for parallel processing. Here, each task or node determines its
 * task number (taskid) and the total number of tasks or nodes running this
 * program (numtask) by using the MPL call mpc_environ. Then, each task
 * determines the identifier for the group which encompasses all tasks or
 * nodes running this program. This identifier (allgrp) is used in collective
 * communication or aggregated computation operations, such as mpc_index.
 */

  gettimeofday(&tv0, (struct timeval*)0);          /* before time */

  rc = mpc_environ (&numtask, &taskid);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_environ.\n");
      exit (-1);
    }

  if (numtask != NN)
    {
      if (taskid == 0)
        {
          printf ("Error - task number mismatch... check defs.h.\n");
          exit (-1);
        }
    }

  rc = mpc_task_query (nbuf, 4, 3);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_task_query.\n");
      exit (-1);
    }
  allgrp = nbuf[3];

  gettimeofday(&tv2, (struct timeval*)0);          /* before time */
  task_wall = (float) (tv2.tv_sec - tv0.tv_sec) * 1000000 + tv2.tv_usec
    - tv0.tv_usec;

  if (taskid == 0)
    {
      printf ("Running...\n");
    }

  times (&all_start);

/*
 * Get arguments from command line. In the sequential version of the program,
 * the following procedure was used to extract the number of times the main
 * computational body was to be repeated, and flags regarding the amount of
 * reporting to be done during and after the program was run. In this paralle
 * program, there are no command line arguments to be extracted except for the
 * name of the file containing the data cube.
 */

  cmd_line (argc, argv, str_name, data_name);

/*
 * Read input files. In this section, each task loads its portion of the data
 * cube from the data file.
 */

  if (taskid == 0)
    {
      printf ("  loading data...\n");
    }

  mpc_sync (allgrp);
  times (&disk_start);
  gettimeofday(&tv1, (struct timeval*)0);          /* before time */
  read_input_LIN (data_name, str_name, v4, data_cube);
  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  disk_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;
  times (&disk_end);

/*
 * Start Linear Algebra steps.
 */

  make_t = 0;                  /* Dont make T matrix. Return a weight vector */
  times(&lin_start);
  gettimeofday(&tv1, (struct timeval*)0);          /* before time */

/*
 * Move steering vector V4 into the str_vecs matrix.
 */

  for (i = 0; i < V4; i++)
    {
      str_vecs[0][i].real = v4[i].real;
      str_vecs[0][i].imag = v4[i].imag;
    }

/*
 * Perform the Householder transform only on NUM_MAT of theV2 dopplers (DIM1
 * by DIM3). Allocate space for the beams_mat matrix = NUM_MAT*V3 COMPLEX
 * matrix.
 */

  if ((beams_mat = (COMPLEX **) malloc (NUM_MAT * sizeof (COMPLEX *))) == NULL)
    {
      printf (" Cant allocate space for output beams_mat holding matrix\n ");
      free (temp_mat);
      exit (-1);
    }

  for (i = 0; i < NUM_MAT; i++)
    {
      if ((beams_mat[i] = (COMPLEX *) malloc (V3 * sizeof (COMPLEX))) == NULL)
       {
         printf (" Cant allocate space for output beams_mat holding matrix ");
         free (temp_mat);
         free (beams_mat);
         exit (-1);
       }
    }

/*
 * Move 1 of NUM_MAT dopplers to temp_mat from data_cube, perform a Householder
 * transform, then forward and back solve for the weights. Apply the weights
 * to the data and put the beamformed dopplers into the beams_mat matrix.
 *
 * Select 1st NUM_MAT matrices with DIM1 by DIM3 data elements.
 */

  for (i = 0; i < NUM_MAT; i++)
    {
      for (j = 0; j < DIM1; j++)
       for (k = 0; k < DIM3; k++)
         {           /* Select dimensions to use 64*1500 data elements  */
           temp_mat[j][k].real = data_cube[j][i][k].real;
           temp_mat[j][k].imag = data_cube[j][i][k].imag;
         }

/*
 * Call the Householder transform routine house ().
 */

      start_row = 0;                /* Start Householder on 1st row */
      num_rows = DIM1;              /* Do Householder on all rows  */
      num_cols = DIM3;              /* Do Householder on all columns  */
      lower_triangular_rows = DIM1; /* Lower triangularize all rows */

      house (num_rows, num_cols, lower_triangular_rows, start_row, temp_mat);

/*
 * temp_mat matrix now has a lower triangular matrix as 1st MxM rows. M
 * nominally is 64. Call forback () to solve once with vector str_vecs = v4.
 * Return the solution weight vector in weight_vec.
 */

      forback (lower_triangular_rows, str_vecs, weight_vec, make_t, temp_mat);

/*
 * Multiply the conjugate of the weight vector with the application data, DIM3
 * columns of DIM1 rows, and store the results in output beams_mat. Use
 * temporary storage for the weight vector with pointers to speed multiply.
 */

     real_ptr = xreal;
     imag_ptr = ximag;
     for (j = 0; j < V4; j++)
       {

/*
 * Store the conjugate of the weight vector.
 */

        *real_ptr++ = weight_vec[j].real;
        *imag_ptr++ = - weight_vec[j].imag;
       }

/*
 * Multiply the weight vector by all columns of the application data set.
 */

      for (k = 0; k < DIM3; k++)
       {
         sum.real = 0.0;
         sum.imag = 0.0;
         real_ptr = xreal;
         imag_ptr = ximag;
          for (j = 0; j < DIM1; j++)
            {

/*
 * Select dimensions to use DIM1 by DIM3 data elements.
 */
```

```
              sum.real += (data_cube[j][i][k].real * *real_ptr -
                         data_cube[j][i][k].imag * *imag_ptr);
              sum.imag += (data_cube[j][i][k].imag * *real_ptr++ +
                         data_cube[j][i][k].real * *imag_ptr++);
            }

/*
 * Store the result into the output beams_mat matrix.
 */

          beams_mat[i][k].real = sum.real;
          beams_mat[i][k].imag = sum.imag;
        }  /* end k for_loop */

      max = 0.0;

      for (j = i; j < i+1; j++)
       for (k = 0; k < DIM3; k++)
         {
           f_temp1 = beams_mat[j][k].real;
           f_temp2 = beams_mat[j][k].imag;
           f_pwr = f_temp1*f_temp1 + f_temp2*f_temp2;
           if (f_pwr > max)
             {
              max = f_pwr;
              loc1=0;          /* Always beam 0 */
              loc2=j;
              loc3=k;
             }
         }

/*
 *       printf("task %d finds max %f at %d %d %d\n",
 *           taskid, max, loc1, loc2, loc3);
 */

      index_max[i].value = max;
      index_max[i].loc1 = loc1;
      index_max[i].loc2 = loc2 + taskid * dim2;
      index_max[i].loc3 = loc3;

    }  /* end i for_loop */

/*
 * Gather local max entries into node 0.
 */

  rc = mpc_gather (index_max, g_index_max, NUM_MAT*sizeof(INDEXED_MAX), 0,
                 allgrp);
      if (rc == -1)
       {
         printf ("Error - unable to call mpc_gather.\n");
         exit (-1);
       }

  gettimeofday(&tv2, (struct timeval*)0);  /* after time */
  lin_wall = (float) (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec
    - tv1.tv_usec;
  all_wall = (float) (tv2.tv_sec - tv0.tv_sec) * 1000000 + tv2.tv_usec
    - tv0.tv_usec;
  times(&lin_end);
  times(&all_end);

/*
 * Compute all times.
 */

  all_user = (float) (all_end.tms_utime - all_start.tms_utime)/100.0;
  all_sys = (float)(all_end.tms_stime - all_start.tms_stime)/100.0;
  rc = mpc_reduce (&all_user, &all_user_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&all_sys, &all_sys_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }

/*
 * Compute disk times.
 */

  rc = mpc_reduce (&disk_wall, &disk_wall_max, sizeof(float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  disk_user = (float) (disk_end.tms_utime - disk_start.tms_utime)/100.0;
  disk_sys = (float)(disk_end.tms_stime - disk_start.tms_stime)/100.0;
  rc = mpc_reduce (&disk_user, &disk_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&disk_sys, &disk_sys_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }

/*
 * Compute lin times.
 */

  rc = mpc_reduce (&lin_wall, &lin_wall_max, sizeof(float), 0, s_vmax, allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  lin_user = (float) (lin_end.tms_utime - lin_start.tms_utime)/100.0;
  lin_sys = (float)(lin_end.tms_stime - lin_start.tms_stime)/100.0;
  rc = mpc_reduce (&lin_user, &lin_user_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&lin_sys, &lin_sys_max, sizeof (float), 0, s_vmax,allgrp);
  if (rc == -1)
    {
      printf ("Error mpc_reduce.\n");
      exit (-1);
    }

/*
 * Report results.
 */
```

```
   if (taskid == 0 )
      for (i = 0; i < NUM_MAT*NN; i++)
        {
          max = g_index_max[i].value;
          loc1 = g_index_max[i].loc1;
          loc2 = g_index_max[i].loc2;
          loc3 = g_index_max[i].loc3;
          printf("#%d POWER of max COMPLEX output beams data = %f \n",i, max);
          printf("  LOCATION of max COMPLEX output beams data = %d %d %d \n",
                 loc1,loc2,loc3);
        }

 /*
  * Display timing information.
  */

   if (taskid == 0)
     {
       printf ("\n\n*** CPU Timing information - numtask = %d\n\n", NN);
       printf ("  all_user_max     = %.2f s, all_sys_max     = %.2f s\n",
               all_user_max, all_sys_max);
       printf ("  disk_user_max    = %.2f s, disk_sys_max    = %.2f s\n",
               disk_user_max, disk_sys_max);
       printf ("  lin_user_max     = %.2f s, lin_sys_max     = %.2f s\n",
               lin_user_max, lin_sys_max);

       printf ("\n*** Wall Clock Timing information - numtask = %d\n\n", NN);
       printf ("  all_wall          = %.0f us\n", all_wall);
       printf ("  task_wall          = %.0f us\n", task_wall);
       printf ("  disk_wall          = %.0f us\n", disk_wall);
       printf ("  lin_wall          = %.0f us\n", lin_wall);
     }
   free (beams_mat);
}
```

A.3.2 cmd_line.c

```
/*
 * cmd_line.c
 */

/*
 * This file contains the procedure cmd_line (), and is part of the parallel
 * General benchmark program written for the IBM SP2 by the STAP benchmark
 * parallelization team at the Univerisity of Southern California (Prof. Kai
 * Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA Mountaintop
 * program.
 *
 * The sequential General benchmark program was originally written by Tony
 * Adams on 8/30/93.
 *
 * The procedure cmd_line () extracts the name of the files from which the
 * input data cube and the steering vector data should be loaded. The
 * function of this parallel version of cmd_line () is different from that
 * of the sequential version, because the sequential version also extracted
 * the number of iterations the program should run and some reporting
 * options.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"
#include <sys/types.h>
```

```
#include <sys/time.h>

/*
 * cmd_line ()
 *    inputs: argc, argv
 *    outputs: str_name, data_name
 *
 *    argc, argv: these are used to get data from the command line arguments
 *    str_name: a holder for the name of the input steering vectors file
 *    data_name: a holder for the name of the input data file
 */

cmd_line (argc, argv, str_name, data_name)
      int argc;
      char *argv[];
      char str_name[LINE_MAX];
      char data_name[LINE_MAX];

{

/*
 * Begin function body: cmd_line ()
 */

  strcpy (str_name, argv[1]);
  strcat (str_name, ".str");
  strcpy (data_name, argv[1]);
  strcat (data_name, ".dat");
  return;
}
```

A.3.3 forback.c

```
/*
 * forback.c
 */

/*
 * This file contains the procedure forback (), and is part of the parallel
 * General benchmark program written for the IBM SP2 by the STAP benchmark
 * parallelization team at the University of Southern California (Prof. Kai
 * Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA Mountaintop
 * program.
 *
 * The sequential General benchmark program was originally written by Tony
 * Adams on 8/30/93.
 *
 * The procedure forback () performs a forward and back substitution on an
 * input temp_mat array, using the steering vectors, str_vecs, and
 * normalizes the solution returned in the vector "weight_vec".
 *
 * If the input variable "make_t" = 1, then the T matrix is generated in
 * this routine by doing BEAM forward and back solution vectors. Nominally,
 * BEAM = 32.
 *
 * The conjugate transpose of each weight vector, to get the Hermitian, is
 * put into the T matrix as row vectors. The input, temp_mat, is a square
 * lower triangular array, with the number of rows and columns = num_rows.
 *
 * This routine requires 5 input parameters as follows:
 *   num_rows: the number of COMPLEX elements, M, in temp_mat
 *   str_vecs: a pointer to the start of the steering vector array
 *   weight_vec: a pointer to the start of the COMPLEX weight vector
 *   make_t: 1 = make T matrix; 0 = don't make T matrix but pass back the
 *      weight vector
 *    temp_mat[][]: a temporary matrix holding various data
 *
 * This procedure was untouched during our parallelization effort, and
 * therefore is virtually identical to the sequential version. Also, most,
 * if not all, of the comments were taken verbatim from the sequential
 * version.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"

/*
 * forback ()
 *    inputs: num_rows, str_vecs, weight_vec, make_t, temp_mat
 *    outputs: weight_vec
 *
 *    num_rows: the number of elements, M, in the input data
 *    str_vecs: a pointer to the start of the steering vector array
 *    weight_vec: a pointer to the start of the COMPLEX weight vector
 *    make_t: 1 = make T matrix; 0 = don't make T matrix but pass back the
 *      weight vector
 *    temp_mat: MAX temprary matrix holding various data
 */

forback (num_rows, str_vecs, weight_vec, make_t, temp_mat)
      int num_rows;
      COMPLEX str_vecs[][DOF];
      COMPLEX weight_vec[];
      int make_t;
      COMPLEX temp_mat[][COLS];

{

/*
 * Variables
 *
 * i, j, k: loop counters
 * last: the last row or element in the matrices or vectors
 * beams: loop counter
 * num_beams: loop counter
 * sum_sq: a holder for the sum squared of the elements in a row
 * sum: a holder for the sum of the COMPLEX elements in a row
 * temp: a holder for temporary storage of a COMPLEX element
 * abs_mag: the absolute magnitude of a complex element
 * wt_factor: a holder for the weight normalization factor, according to
 *    Adaptive Matched Filter Normalization
 * steer_vec: one steering vector with DOF COMPLEX elements
 * vec: a holder for the complex solution intermediate vector
 * tmp_mat: a pointer to the start of the COMPLEX temp matrix
 */

 int i, j, k;
 int last;
 int beams;
 int num_beams;
 float sum_sq;
 COMPLEX sum;
 COMPLEX temp;
 float abs_mag;
 float wt_factor;
 COMPLEX steer_vec[DOF];
 COMPLEX vec[DOF];
 COMPLEX tmp_mat[DOF][DOF];

/*
 * Begin function body: forback ()
 *
```

```
  * The temp_mat matrix contains a lower triangular COMPLEX matrix as the
  * first MxM rows. Do a forward back solution BEAM times with a different
  * steering vector. If make_t = 1, then make the T matrix.
  */

   if (make_t)
     {
       num_beams = num_rows;   /* Do M forward back solution vectors. Put */
     }                         /* them else into T matrix. */

  else
    {
      num_beams = 1;  /* Do only 1 solution weight vector */
    }

  for (beams = 0; beams < num_beams; beams++)
    {  /* for (beams...) */
      for (j = 0; j < num_rows; j++)
       {
         steer_vec[j].real = str_vecs[beams][j].real;
         steer_vec[j].imag = str_vecs[beams][j].imag;
       }

/*
 * Step 1: Do forward elimination. Also, get the weight factor = square root
 * of the sum squared of the solution vector. Used to divide back substitution
 * solution to get the weight vector. Divide the first element of the COMPLEX
 * steering vector by the first COMPLEX diagonal to get the first element of
 * the COMPLEX solution vector. First, get the absolute magnitude of the first
 * lower triangular diagonal.
 */

      abs_mag = temp_mat[0][0].real * temp_mat[0][0].real
                + temp_mat[0][0].imag * temp_mat[0][0].imag;
/*
 * Solve for the first element of the solution vector.
 */

      vec[0].real = (temp_mat[0][0].real * steer_vec[0].real +
                     temp_mat[0][0].imag * steer_vec[0].imag) / abs_mag;
      vec[0].imag = (temp_mat[0][0].real * steer_vec[0].imag -
                     temp_mat[0][0].imag * steer_vec[0].real) / abs_mag;
/*
 * Start summing the square of the solution vector.
 */

      sum_sq = vec[0].real * vec[0].real + vec[0].imag * vec[0].imag;

/*
 * Now solve for the remaining elements of the solution vector.
 */

      for (i = 1; i < num_rows; i++)
        {  /* for (i...) */
         sum.real = 0.0;
         sum.imag = 0.0;
         for (k = 0; k < i; k++)
           {  /* for (k...) */
             sum.real += (temp_mat[i][k].real * vec[k].real -
                        temp_mat[i][k].imag * vec[k].imag);
             sum.imag += (temp_mat[i][k].imag * vec[k].real +
                        temp_mat[i][k].real * vec[k].imag);
           }  /* for (k...) */

/*
 * Now subtract the sum from the next element of the steering vector.
 */

         temp.real = steer_vec[i].real - sum.real;
         temp.imag = steer_vec[i].imag - sum.imag;

/*
 * Get the absolute magnitude of the next diagonal.
 */

         abs_mag = temp_mat[i][i].real * temp_mat[i][i].real
                      + temp_mat[i][i].imag * temp_mat[i][i].imag;

/*
 * Solve for the next element of the solution vector.
 */

         vec[i].real = (temp_mat[i][i].real * temp.real +
                       temp_mat[i][i].imag * temp.imag) / abs_mag;
         vec[i].imag = (temp_mat[i][i].real * temp.imag -
                       temp_mat[i][i].imag * temp.real) / abs_mag;

/*
 * Sum the square of the solution vector.
 */

         sum_sq += (vec[i].real * vec[i].real + vec[i].imag * vec[i].imag);
        }  /* for (i...) */
      wt_factor = sqrt ((double) sum_sq);

/*
 * Step 2: Take the conjugate transpose of the lower triangular matrix to
 * form an upper triangular matrix.
 */

      for (i = 0; i < num_rows; i++)
       {  /* for (i...) */
         for (j = 0; j < num_rows; j++)
            {  /* for (j...) */
             tmp_mat[i][j].real = temp_mat[j][i].real;
             tmp_mat[i][j].imag = - temp_mat[j][i].imag;
            }  /* for (j...) */
       }   /* for (i...) */

/*
 * Step 3: Do a back substitution.
 */

      last = num_rows - 1;

/*
 * Get the absolute magnitude of the last upper triangular diagonal.
 */

      abs_mag = tmp_mat[last][last].real * tmp_mat[last][last].real
                 + tmp_mat[last][last].imag * tmp_mat[last][last].imag;

/*
 * Solve for the last element of the weight solution vector.
 */

      weight_vec[last].real = (tmp_mat[last][last].real * vec[last].real
                             + tmp_mat[last][last].imag * vec[last].imag)
                               / abs_mag;
      weight_vec[last].imag = (tmp_mat[last][last].real * vec[last].imag
                             - tmp_mat[last][last].imag * vec[last].real)
                               / abs_mag;

/*
 * Now solve for the remaining elements of the weight solution vector from
 * the next to last element up to the first element.
 */

       for (i = last - 1; i >= 0; i--)
         {  /* for (i...) */
          sum.real = 0.0;
          sum.imag = 0.0;
          for (k = i + 1; k <= last; k++)
            {  /* for (k...) */
             sum.real += (tmp_mat[i][k].real * weight_vec[k].real -
                         tmp_mat[i][k].imag * weight_vec[k].imag);
             sum.imag += (tmp_mat[i][k].imag * weight_vec[k].real +
                         tmp_mat[i][k].real * weight_vec[k].imag);
            }  /* for (k...) */

/*
 * Subtract the sum from the next element up of the forward solution vector.
 */

          temp.real = vec[i].real - sum.real;
          temp.imag = vec[i].imag - sum.imag;

/*
 * Get the absolute magnitude of the next diagonal up.
 */

          abs_mag = tmp_mat[i][i].real * tmp_mat[i][i].real
                  + tmp_mat[i][i].imag * tmp_mat[i][i].imag;

/*
 * Solve for the next element up of the weight solution vector.
 */

          weight_vec[i].real = (tmp_mat[i][i].real * temp.real +
                               tmp_mat[i][i].imag * temp.imag) / abs_mag;
          weight_vec[i].imag = (tmp_mat[i][i].real * temp.imag -
                               tmp_mat[i][i].imag * temp.real) / abs_mag;
         }  /* for (i...) */

/*
 * Step 4: Divide the solution weight_vector by the weight factor.
 */

       for (i = 0; i < num_rows; i++)
         {
          weight_vec[i].real /= wt_factor;
          weight_vec[i].imag /= wt_factor;
         }

#ifdef APT

/*
 * If make_t = 1, make the T matrix.
 */

       if (make_t)
         {  /* if (make_t) */

/*
 * Conjugate transpose the weight vector to get the Hermitian. Put each
 * weight vector into the T matrix as row vectors.
 */

          for (j = 0; j < num_rows; j++)
            {
             t_matrix[beams][j].real = weight_vec[j].real;
             t_matrix[beams][j].imag = - weight_vec[j].imag;
            }
         }  /* if (make_t) */
```

```
#endif

    }  /* for (beams...) */
  return;
}
```

A.3.4 house.c

```
/*
 * house.c
 */

/*
 * This file contains the procedure house (), and is part of the parallel
 * General benchmark program written for the IBM SP2 by the STAP benchmark
 * parallelization team at the University of Southern California (Prof. Kai
 * Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA Mountaintop
 * program.
 *
 * The sequential General benchmark program was originally written by Tony
 * Adams on 8/30/93.
 *
 * The procedure house () performs the Householder transform, in place, on
 * an N by M complex imput matrix, where M >= N. It returns the results in
 * the same location as the input data.
 *
 * This routine requries 5 input parameters as follows:
 *   num_rows: number of elements in the temp_mat
 *   num_cols: number of range gates in the temp_mat
 *   lower_triangular_rows: the number of rows in the output temp_mat that
 *     have been lower triangularized
 *   start_row: the number of the row on which to start the Householder
 *   temp_mat[][]: a temporary matrix holding various data
 *
 * This procedure was untouched during our parallelization effort, and
 * therefore is virtually identical to the sequential version. Also, most,
 * if not all, of the comments were taken verbatim from the sequential
 * version.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"

/*
 * house ()
 *   inputs: num_rows, num_cols, lower_triangular_rows, start_row, temp_mat
 *   outputs: temp_mat
 *
 *   num_rows: number of elements, N, in the input data
 *   num_cols: number of range gates, M, in the input data
 *   lower_triangular_rows: the number of rows to be lower triangularized
 *   start_row: the row number of which to start the Householder
 *   temp_mat: temporary matrix holding various data
 */

house (num_rows, num_cols, lower_triangular_rows, start_row, temp_mat)
     int num_rows;
     int num_cols;
     int lower_triangular_rows;
     int start_row;
     COMPLEX temp_mat[][COLS];

{
```

```
/*
 * Variables
 *
 * i, j, k: loop counters
 * rtemp: a holder for temporary scalar data
 * x_square: a holder for the absolute square of complex variables
 * xmax_sq: a holder for the maximum of the complex absolute of variables
 * vec: a holder for the maximum complex vector 2 * num_cols max
 * sigma: a holder for a complex variable used in the Householder
 * gscal: a holder for a complex variable used in the Householder
 * alpha: a holder for a scalar variable used in the Householder
 * beta: a holder for a scalar variable used in the Householder
 */

  int i, j, k;
  float rtemp;
  float x_square;
  float xmax_sq;
  COMPLEX vec[2*COLS];
  COMPLEX sigma;
  COMPLEX gscal;
  float alpha;
  float beta;

/*
 * Begin function body: house ()
 *
 * Loop through temp_mat for number of rows = lower_triangular_rows. Start at
 * the row number indicated by the start_row input variable.
 */

  for (i = start_row; i < lower_triangular_rows; i++)
    {  /* for (i...) */

/*
 * Step 1: Find the maximum absolute element for each row of temp_mat,
 * starting at the diagonal element of each row.
 */

      xmax_sq = 0.0;
      for (j = i; j < num_cols; j++)
       {  /* for (j...) */
         x_square = temp_mat[i][j].real * temp_mat[i][j].real
                    + temp_mat[i][j].imag * temp_mat[i][j].imag;
         if (xmax_sq < x_square)
           {
             xmax_sq = x_square;
           }
       }  /* for (j...) */

/*
 * Step 2: Normalize the row by the maximum value and generate the complex
 * transpose vector of the row in order to calculate alpha = square root of
 * the sum square of all the elements in the row.
 */

      xmax_sq = (float) sqrt ((double) xmax_sq);
      alpha = 0.0;
      for (j = i; j < num_cols; j++)
       {  /* for (j...) */
         vec[j].real = temp_mat[i][j].real / xmax_sq;
         vec[j].imag = - temp_mat[i][j].imag / xmax_sq;
         alpha += (vec[j].real * vec[j].real + vec[j].imag * vec[j].imag);
       }  /* for (j...) */

      alpha = (float) sqrt ((double) alpha);

/*
```

```
 * Step 3: Find beta = 2 / (b (transpose) * b). Find sigma of the relevant
 * element = x(i) / |x(i)|.
 */

      rtemp = vec[i].real * vec[i].real + vec[i].imag * vec[i].imag;
      rtemp = (float) sqrt ((double) rtemp);
      beta = 1.0 / (alpha * (alpha + rtemp));
      if (rtemp >= 1.0E-16)
       {
         sigma.real = vec[i].real / rtemp;
         sigma.imag = vec[i].imag / rtemp;
       }
      else
       {
         sigma.real = 1.0;
         sigma.imag = 0.0;
       }

/*
 * Step 4: Calculate the vector operator for the relevent element.
 */

      vec[i].real += sigma.real * alpha;
      vec[i].imag += sigma.imag * alpha;

/*
 * Step 5: Apply the Householder vector to all the rows of temp_mat.
 */

      for (k = i; k < num_rows; k++)
       {  /* for (k...) */

/*
 * Find the scalar for finding g.
 */

         gscal.real = 0.0;
         gscal.imag = 0.0;

         for (j = i; j < num_cols; j++)
           {  /* for (j...) */
             gscal.real += (temp_mat[k][j].real * vec[j].real -
                            temp_mat[k][j].imag * vec[j].imag);
             gscal.imag += (temp_mat[k][j].real * vec[j].imag +
                            temp_mat[k][j].imag * vec[j].real);
            }  /* for (j...) */
         gscal.real *= beta;
         gscal.imag *= beta;

/*
 * Modify only the necessary elements of the temp_mat, subtracting gscal
 * * conjg (vec) from temp_mat elements.
 */

         for (j = i; j < num_cols; j++)
           {  /* for (j...) */
              temp_mat[k][j].real -= (gscal.real * vec[j].real +
                                      gscal.imag * vec[j].imag);
              temp_mat[k][j].imag -= (gscal.imag * vec[j].real -
                                      gscal.real * vec[j].imag);
            }  /*  for (j...) */
       }  /* for (k...) */
    }  /* for (i...) */
  return;
}
```

A.3.5 read_input_LIN.c

```
/*
 * read_input_LIN.c
 */

/*
 * This file contains the procedure read_input_LIN (), and is part of the
 * parallel General benchmark program written for the IBM by the STAP benchmark
 * parallelization team at the University of Southern California (Prof. Kai
 * Hwang, Dr. Zhiwei Xu, and Masahiro Arakawa), as part of the ARPA Mountaintop
 * program.
 *
 * The sequential General benchmark program was originally written by Tony
 * Adams on 8/30/93.
 *
 * The procedure read_input_LIN () reads the input data files (containing the
 * data cube and the steering vectors).
 *
 * In this parallel version of read_input_LIN (), each task reads its portion
 * of the data cube in from the data file simulatenously. In order to improve
 * disk performance, the entire data cube slice is read from disk, then
 * converted from the packed binary integer format to the floating point
 * number format.
 *
 * Each complex number is stored on disk as a packed 32-bit binary integer.
 * The 16 LSBs are the real portion of the number, and the 16 MSBs are the
 * imaginary portion of the number.
 *
 * The steering vector file contains the number of PRIs in the input data,
 * the target power threshold, and the steering vectors. The data is stored
 * in ASCII format, and the complex steering vector numbers are stored as
 * alternating real and imaginary numbers.
 */

#include "stdio.h"
#include "math.h"
#include "defs.h"
#include <sys/types.h>
#include <sys/time.h>
#include <mpproto.h>

#ifdef DBLE
char *fmt = "%lf";
#else
char *fmt = "%f";
#endif

extern int taskid, numtask, allgrp;


/*
 * read_input_LIN ()
 *   inputs: data_name, str_name
 *   outputs: v4, data_cube
 *
 *   data_name: the name of the file containing the data cube
 *   str_name: the name of the file containing the input vectors
 *   v4: storage for the v4 vector
 *   data_cube: input data cube
 */

read_input_LIN (data_name, str_name, v4, data_cube)
  char data_name[];
  char str_name[];
  COMPLEX v4[];
  COMPLEX data_cube[][DIM2][DIM3];
{

/*
 * Variables
 *
 * dim1, dim2, dim3: dimension 1, 2, and 3 in the input cube, respectively
 * temp: buffer for binary input integer data
 * temp1, temp2: holders for integer data
 * f_temp1, f_temp2: holders for float data
 * junk: a temporary variable
 * f_power: holder for float power data
 * i, j, k: loop counters
 * local_int_cube: the integer version of the local portion of the data cube
 * blklen, rc: variables used for MPL calls
 */

  int dim1 = DIM1;
  int dim2 = DIM2;
  int dim3 = DIM3;
  unsigned int temp[1];
  long int temp1, temp2;
  float f_temp1, f_temp2;
  float junk;
  float f_pwr;
  int i, j, k;
  FILE *fopen();
  FILE *f_dat, *f_vec;
  unsigned int local_int_cube[DIM2][DIM1][DIM3];
  long blklen, rc;

/*
 * Begin function body: read_input_LIN ().
 *
 * Read in the data_cube file in parallel.
 */

  if ((f_dat = fopen (data_name, "r")) == NULL)
    {
      printf ("Error - task %d unable to open data file.\n", taskid);
      exit (-1);
    }
  fseek (f_dat, taskid * dim1 * dim2 * dim3 * sizeof (unsigned int), 0);
  fread (local_int_cube, sizeof (unsigned int), dim1 * dim2 * dim3, f_dat);
  fclose (f_dat);

/*
 * Convert the data from the unsigned integer format to floating point
 * format.
 */

  for (i = 0; i < dim1; i++)
    for (j = 0; j < dim2; j++)
      for (k = 0; k < dim3; k++)
        {
          temp[0] = local_int_cube[j][i][k];
          temp1 = 0x0000FFFF & temp[0];
          temp1 = (temp1 & 0x00008000) ? temp1 | 0xffff0000 : temp1;
          temp2 = (temp[0] >> 16) & 0x0000FFFF;
          temp2 = (temp2 & 0x00008000) ? temp2 | 0xffff0000 : temp2;
          data_cube[i][j][k].real = (float) temp1;
          data_cube[i][j][k].imag = (float) temp2;
        }

/*
 * Open the vector file using the file pointer f_vec.
 */
```

```
   if ((f_vec = fopen (str_name, "r")) == NULL)
      {
        printf (" Cant open input steering vector file %s\n ", str_name);
        exit (-1);
      }

/*
 * Read the vectors from the input file with vectors v1, v2, v3, and v4 in
 * order.
 */

    for (i = 0; i < V4; i++)
      {
       fscanf (f_vec, fmt, &v4[i].real);
       fscanf (f_vec, fmt, &v4[i].imag);
     }

  fclose(f_vec);

  return;
}
```

A.3.6 defs.h

```
/* defs.h */

/* 94 Aug 25 - We added a #define macro to compensate for the lack of a   */
/*             log2 function in the IBM version of math.h                 */

/* 94 Sep 13 - We added a definition for number of clock ticks per        */
/*             second, because this varies between the Sun OS and the IBM */
/*             AIX OS.                                                    */

/* Sun OS version - 60 clock ticks per second                             */

#ifdef SUN
#define CLK_TICK 60.0
#endif


/* IBM AIX version - 100 clock ticks per second                           */

#ifdef IBM
#define log2(x)  ((log(x))/(M_LN2))
#define CLK_TICK 100.0
#endif


#ifdef DBLE

#define float double

#endif

typedef struct {
   float real;
   float imag;
} COMPLEX;

/* used in new fft */
#define SWAP(a,b) {float
swap_temp=(a).real;(a).real=(b).real;(b).real=swap_temp;\

swap_temp=(a).imag;(a).imag=(b).imag;(b).imag=swap_temp;}
```

```
#define NN 32
#define PRI_CHUNK (PRI/NN)                          /* XXXUUUUUU */
                                                        /* XXXXXUU */

#define AT_LEAST_ARG 2
#define AT_MOST_ARG 4
#define ITERATIONS_ARG 3
#define REPORTS_ARG 4

#define USERTIME(T1,T2)   ((t2.tms_utime-t1.tms_utime)/CLK_TICK)
#define SYSTIME(T1,T2)   ((t2.tms_stime-t1.tms_stime)/CLK_TICK)
#define USERTIME1(T1,T2)   ((time_end.tms_utime-time_start.tms_utime)/CLK_TICK)
#define SYSTIME1(T1,T2)   ((time_end.tms_stime-time_start.tms_stime)/CLK_TICK)
#define USERTIME2(T1,T2)   ((end_time.tms_utime-start_time.tms_utime)/CLK_TICK)
#define SYSTIME2(T1,T2)   ((end_time.tms_stime-start_time.tms_stime)/CLK_TICK)
#ifndef LINE_MAX
#define LINE_MAX 256
#endif
#define TRUE 1
#define FALSE 0

#define COLS 1536   /* Maximum number of columns in holding vector "vec" */
                    /* in house.c for max columns in Householder multiply */
#define MBEAM 12    /* Number of main beams */
#define ABEAM 20    /* Number of auxiliary beams */
#define PWR_BM 9    /* Number of max power beams */
#define V1 64
#define V2 32/NN
#define V3 1536
#define V4 V1
#define DIM1 V1     /* Number for dimension1 in input data cube */
#define DIM2 V2   /* Number for dimension2 in input data cube */
#define DIM3 V3  /* Number for dimension3 in input data cube */
#define DOP_GEN 2048 /* MAX Number of dopplers after FFT */
#define PRI_GEN 2048 /* MAX Number of points for 1500 data points FFT */
                  /*  zero filled after 1500 up to 2048 points */
#define NUM_MAT V2 /* Number of matrices to do householde on */
                  /* NUM_MAT must be less than DIM2 above */

#ifdef APT

#define PRI 256   /* Number of pris in input data cube */
#define DOP 256    /* Number of dopplers after FFT */
/* #define RNG 280 */      /* Number of range gates in input data cube */
#define RNG 256      /* Number of range gates in input data cube */
#define EL 32        /* number of elements in input data cube */
#define BEAM 32      /* Number of beams */
#define NUMSEG 7     /* Number of range gate segments */
#define RNGSEG RNG/NUMSEG  /* Number of range gates per segment */
#define RNG_S 320   /* Number sample range gates for 1st step beam forming */
#define DOF EL       /* Number of degrees of freedom */
extern COMPLEX t_matrix[BEAM][EL];    /* T matrix */
#endif


#ifdef STAP

#define PRI 128     /* Number of pris in input data cube */
#define DOP 128     /* Number of dopplers after FFT */
#define RNG 1250    /* Number of range gates in input data cube */
#define EL 48         /* number of elements in input data cube */
#define BEAM 2        /* Number of beams */
#define NUMSEG 2      /* Number of range gate segments */
#define RNGSEG RNG/NUMSEG /* Number of range gates per segment */
#define RNG_S 288     /* Number of sample range gates for beam forming */
```

```
#define DOF 3*EL    /* Number of degrees of freedom */

#endif

#ifdef GEN
#define EL V4
#define DOF EL

#endif

extern int output_time;    /* Flag if set TRUE, output execution times */
extern int output_report;  /* Flag if set TRUE, output data report files */
extern int repetitions;    /* number of times program has executed */
extern int iterations;     /* number of times to execute program */

typedef struct max_index
{
  float value ;
  int loc1, loc2, loc3 ;
} INDEXED_MAX ;

extern void xu_index_max ( in1, in2, out, len )
INDEXED_MAX in1[], in2[], out[];
int *len ;
{
  int i, n ;
  n = *len/sizeof(int) ;
  for (i=0; i<n; i++)
    {
      if (in1[i].value > in2[i].value)
       {
         out[i].value = in1[i].value ;
         out[i].loc1 = in1[i].loc1;
         out[i].loc2 = in1[i].loc2;
         out[i].loc3 = in1[i].loc3;
       }
      else
       {
         out[i].value = in2[i].value ;
         out[i].loc1 = in2[i].loc1;
         out[i].loc2 = in2[i].loc2;
         out[i].loc3 = in2[i].loc3;
       }
    }
}
```

A.3.7 compile_lin

```
mpcc -O3 -qarch=pwr2 -DGEN -DIBM -o lin bench_mark_LIN.c cmd_line.c forback.c
  house.c read_input_LIN.c  -lm
```

A.3.8 run.032

```
poe lin /scratch1/masa/lin_data -procs 32 -us
```