User's Guide and Documentation of the Parallel APT 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 95-11

* 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.

Abstract

The Adaptive Processing Testbed (APT) 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 APT benchmark for the IBM SP2 massively parallel processor. A user's guide, which lists the files and procedures making up the parallel APT program as well as directions regarding the compilation and execution of the parallel APT 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 APT 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.

Section 1 describes the computations performed in the APT benchmark and its implementation in the sequential version of the program. Section 2 describes the parallel APT 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 APT 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 General and HO-PD (Higher-Order Post-Doppler) benchmarks can be found in [Arak95b, Arak95c].

Contents

1 Sequential APT Program

We generated the parallel APT program by modifying the original sequential version. Below, Figure 1.1 shows the data flow of the APT benchmark.

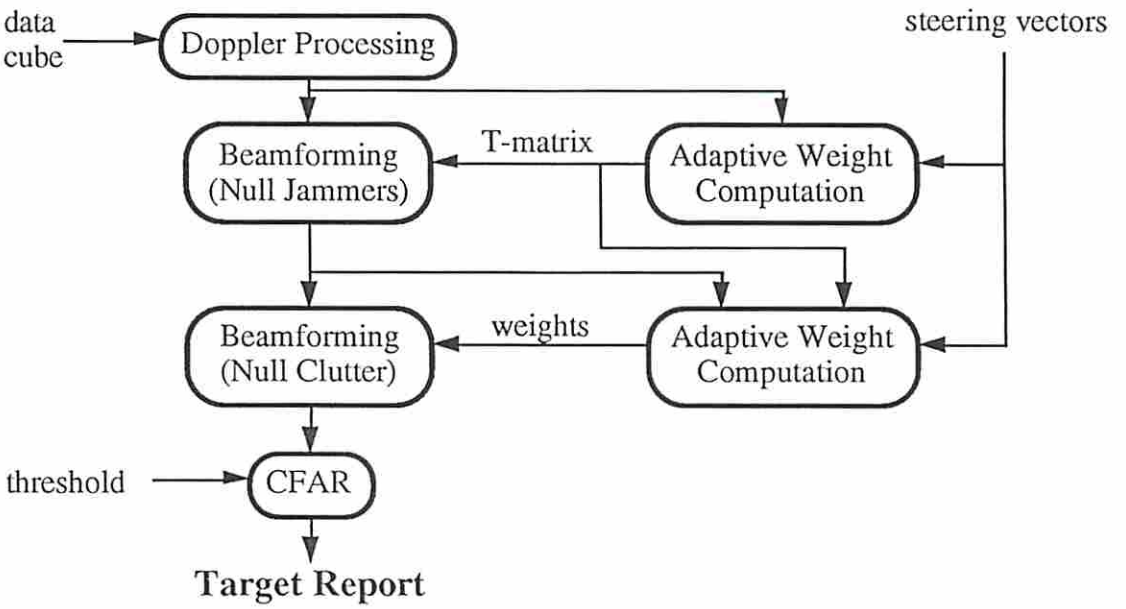


Figure 1.1: APT benchmark data flow diagram

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 1.2 shows the program skeleton for the sequential APT program.

```
COMPLEX data_cube[PRI][RNG][EL], detect_cube[PRI][BEAM][RNG],
        twiddle[PRI];

/* Doppler Processing */
compute_twiddle_factor (out twiddle, in PRI);
for (j = 0; j < RNG; j++)
  for (k = 0; k < EL; k++)
    fft (inout data_cube[.][j][k], in twiddle, in PRI);

/* Beamforming */
for (i = 0; i < PRI; i++)
  beam_form (in data_cube[i][.][.], in data_cube[i-1 % PRI][.][.],
             in data_cube[i+1 % PRI][.][.], out detect_cube[i][.][.]);

/* Target Detection/Cell Averaging CFAR */
for (i = 0; i < PRI; i++)
  for (j = 0; j < BEAM; j++)
    compute_target (inout detect_cube[i][j][.]);

/* Target Reporting */
m = 0;
for (k = 0; k < RNG; k++)
  for (i = 0; i < PRI; i++)
    for (j = 0; j < BEAM; j++)
      if (IsTarget (in detect_cube[i][j][k]))
        {
          target_report[m].pri = i;
          target_report[m].beam = j;
          target_report[m].rng = k;
          target_report[m].power = detect_cube[i][j][k].real;
          m = m + 1;
          if (m > 25) goto finished;
        }
finished:
```

Figure 1.2: Sequential APT program 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 1.3). The steering vectors consist of both primary and auxiliary steering vectors, and are stored on disk in ASCII format.

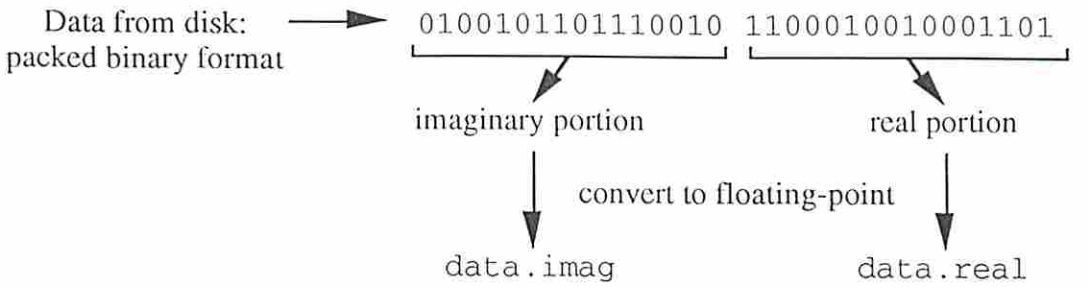


Figure 1.3: Packed binary format to floating-point format conversion

The program first transforms the input data cube into the Doppler domain by performing fast Fourier transforms (FFTs) on the data. In this Doppler processing step, an FFT is performed along the PRI dimension on each column of constant EL and RNG (see Figure 1.4).

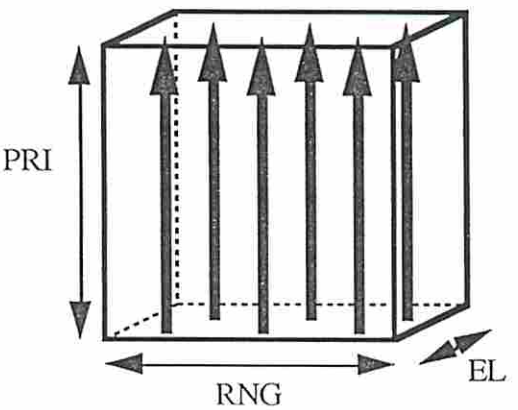


Figure 1.4: Dimension along which the FFTs are performed in the APT benchmark

The beamforming step consists of two adaptive nulling substeps. In the first substep, the program computes a set of adaptive weight vectors (the T matrix) designed to optimally null jamming interference while maintaining gain in the directions specified by the steering vectors. The principal steering vectors specify the directions of interest, while the auxiliary steering vectors provide additional degrees of freedom to assist in the adaptive nulling. The jamming training set is computed from data extracted from the

highest and lowest Doppler bins, under the assumption that these bins contain jamming interference but no signals (targets) or clutter. The weight vectors are then applied to the Doppler data cube to compute a set of jammer-nulled beams.

In the second beamforming substep, the program computes a set of adaptive weight vectors designed to optimally null clutter while maintaining gain in the directions specified by the principal steering vectors. The range extent is divided into a series of range segments, and weight vectors are computed for each range segment. The training sets are computed for each steering vector and for each Doppler and range segment by extracting samples from the nine most powerful auxiliary beams (i.e. beams corresponding to the auxiliary steering vectors) and the principal beams under consideration. The principal steering vectors must be transformed by the same transformation that produced the set of jammer-nulled beams from which the training set is chosen. This entails transforming the steering vector using an appropriate subset of the T matrix. Finally, the weight vectors are applied to the data cube to compute a set of clutter (and jammer) nulled beams.

In the target detection step, a cell averaging, constant false alarm rate (CFAR) detection process is performed to find those signals whose power exceeds a given threshold. The target report generated at the end of the program consists of targets ordered by range. If more than 25 targets were found, only the 25 closest are reported.

2 Parallel APT Program Development

In parallelizing the APT 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.

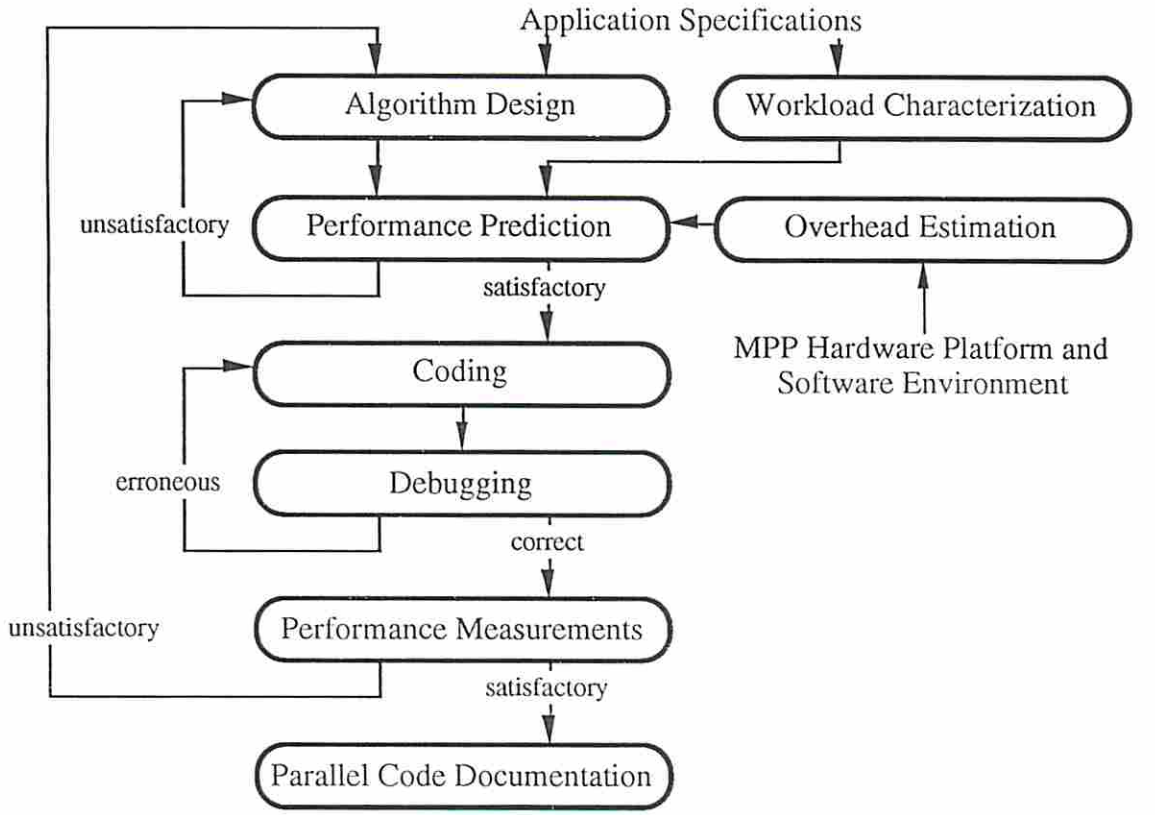


Figure 2.1: Early performance prediction based on workload and overhead characterization

During the algorithm design phase of the parallel program development, we considered several parallelization strategies [Hwan95c]. The nature of the algorithm in the APT 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. The parallel APT program can be described as the following sequence of steps:

Compute:	Doppler Processing
Interact:	Total Exchange Broadcast
Compute:	Beamforming Cell Averaging CFAR
Interact:	Target List Reduction

Because the same algorithm is applied to the entire data cube, we were able to use the SPMD (single program, multiple data stream) paradigm. We exploited the parallelism inherent in the input data set: in this data parallel approach, each task performed the same algorithm on its portion of the input data set. The mapping of the parallel algorithm and the data set onto the SP2 is shown in Figure 2.2.

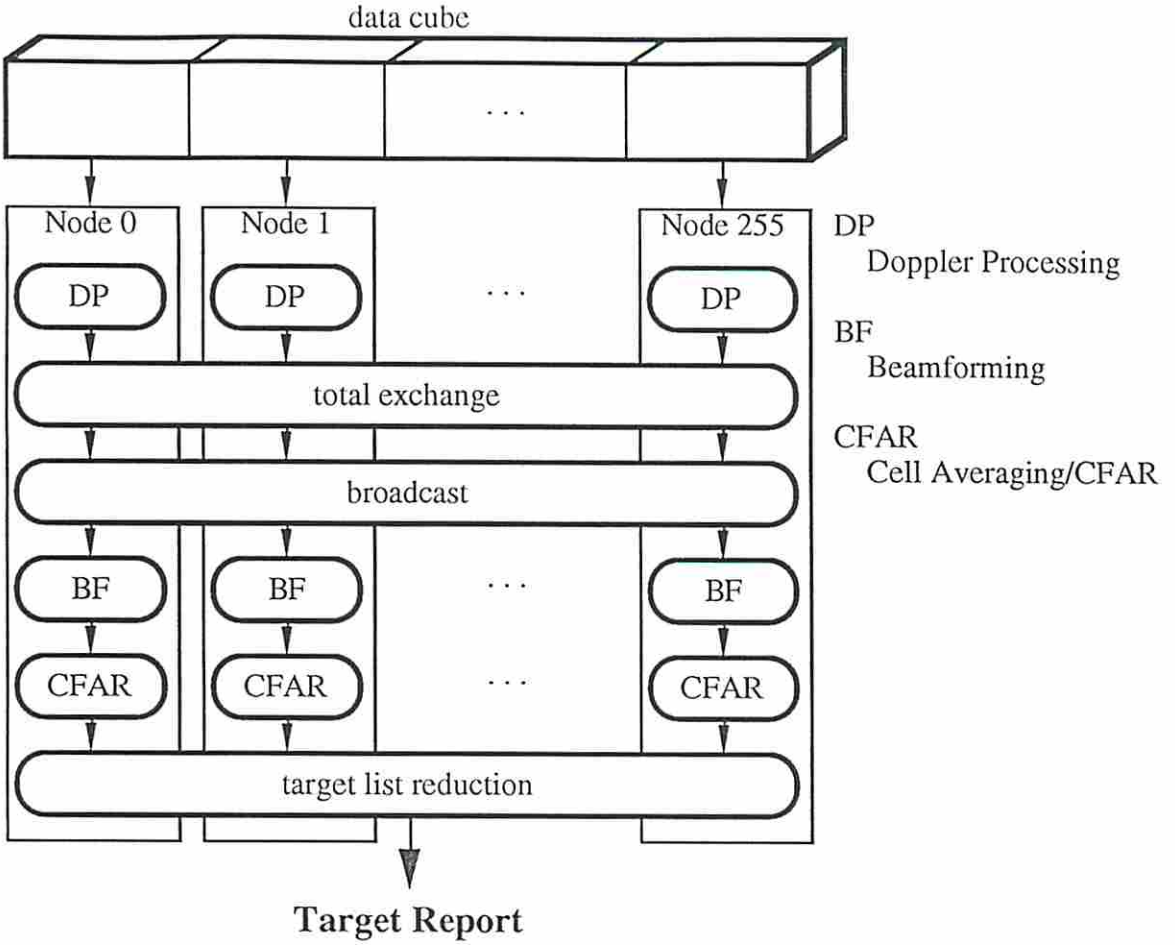


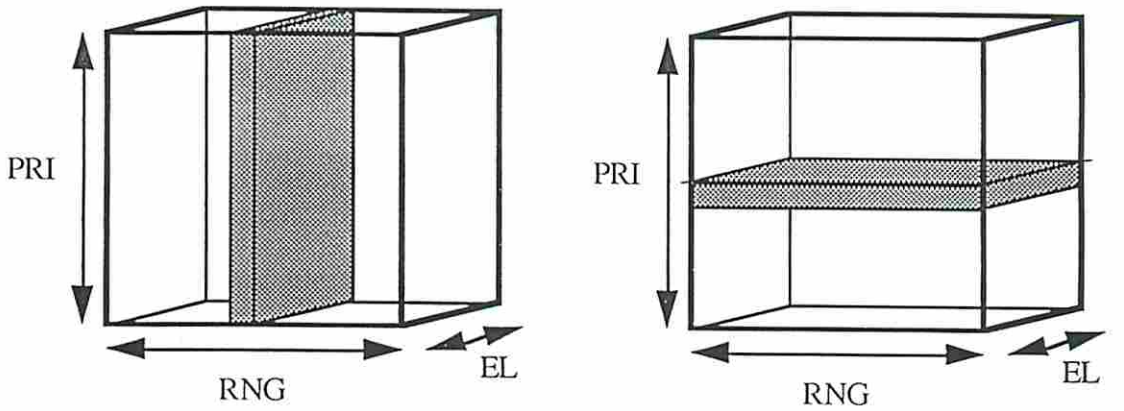
Figure 2.2: Mapping of the parallel algorithm and data set onto the SP2

During the coding step, we started with a program skeleton to help us visualize the structure of the benchmark. The program skeleton is included below in Figure 2.3.

```
COMPLEX data_cube[PRI][RNG][EL], detect_cube[PRI][BEAM][RNG],
        house_matrix[EL][320], t_matrix[EL][EL];

/* Doppler Processing (DP) */
parfor (j = 0; j < RNG; j++)
  {
    COMPLEX twiddle[PRI];
    compute_twiddle_factor (out twiddle, in PRI);
    for (k = 0; k < EL; k++)
      fft (inout data_cube[.][j][k], in twiddle, in PRI);
  }

total_exchange data_cube[.][j][.] to data_cube[i][.][.];

house_matrix[.][0..159] = data_cube[0..1][RNG-80..RNG-1][.];
Task LAST sends data_cube[PRI-2..PRI-1][RNG-80..RNG-1][.], and
  task 0 receives into house_matrix[.][160..319]

broadcast house_matrix to all tasks;

/* Beamforming Step 1 (BF1), performed sequentially by all tasks */
bf1 (in house_matrix, in str_vecs, out t_matrix);

parfor (i = 0 ; i < PRI; i++)
  {
    /* Beamforming Step 2 (BF2) */
    bf2 (in data_cube[i][.][.], in t_matrix, out detect_cube[i][.][.]);

    Target Detection and Local Target Report Generation
  }

reduce local_reports to target_report in task 0;
```

Figure 2.3: Parallel APT program skeleton

At the beginning of the parallel program, each task reads its portion of the data cube from disk in parallel. Because each FFT performed in the Doppler Processing step requires all the data along the PRI dimension, but is independent of the other FFTs along the RNG (as well as the EL) dimension (see Figure 1.4), we divide the data cube equally along the RNG dimension (see Figure 2.4a).



(a) before total exchange (b) after total exchange

Figure 2.4: One task's slice of the data cube

In order to improve disk access performance, we modified the `read_input_APT ()` 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 (this step would not exist in the final implementation as part of an airborne radar processor), the performance of this step is not critical: the timer is started after the data is loaded and converted.

Each task performs FFTs on its slice of the data cube in parallel. In order to improve the performance of the FFTs, we recoded the FFT routine. Our version, a hybrid between the one in the original APT program and a routine suggested by MHPCC, is a more efficient implementation of the same in-place FFT algorithm with bit reversal. By replacing two separate arrays of floating-point numbers accessed by pointers with a single array of `COMPLEX` (a user-defined data structure) numbers, we were able to improve the

processing rate on this routine from 15 MFLOPS per node to as high as 24 MFLOPS per node.

The remaining computational steps of the APT benchmark require all the data along the RNG dimension, but have no data dependencies along the PRI and EL dimensions. Therefore, before continuing with the computation, the APT program executes a total exchange operation to redistribute the data cube so that it is resliced along the RNG dimension (see Figure 2.4b). Because the MPL command `mpc_index` (which implements the total exchange operation) did not work quite to our specification, we needed to adjust the output of this operation. The output of the `mpc_index` operation is a 1-D vector, not the 3-D data cube slice the program needs. Therefore, we added the rewind step, which rewinds the 1-D vector back into a 3-D data cube slice.

So that we can give each task an equal slice of the data cube after the total exchange operation, we reduced the number of range gates in the data set from its nominal value of 280 to 256. With 256 range gates, the data can be divided along this dimension evenly by any power of two from 1 to 256. This adjustment affects the power levels of the targets slightly, but has no impact on the correctness of the parallel program. We confirmed this correctness by comparing the output of the sequential program using a data set with 256 range gates to the output of the parallel program using the same data set.

Following the total exchange step, the APT program forms the Householder matrix. In the sequential APT program, this step was part of the first beamforming step. In the parallel program, the portions of the data cube needed to form the Householder matrix are located in several different tasks. These tasks copy the data into sections of the

Householder matrix (see Figure 2.5), then broadcast these sections to all tasks. At the end of the broadcasts, each task has a complete copy of the Householder matrix.

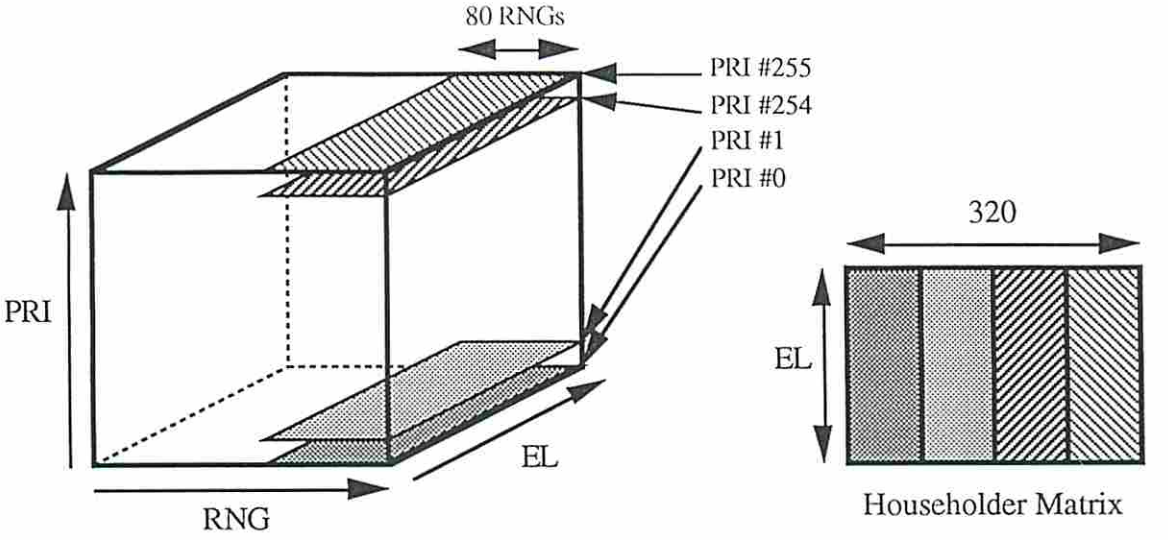


Figure 2.5: Forming the Householder matrix from the data cube

The first beamforming substep cannot be efficiently parallelized, and is therefore executed sequentially. Instead of performing this step on one task and broadcasting the results to the remaining tasks, each task performs this step in its entirety.

Because the algorithm in the second beamforming substep displays data independence along the PRI and EL dimensions, each task can perform this step on its own portion of the data cube in parallel. The cell averaging CFAR step has been parallelized in a similar fashion. Each task generates its own target report, consisting of the closest targets found in its data cube slice, up to 25 targets.

In the target report reduction step, the parallel APT program reduces the individual target reports into one final target report, consisting of the closest targets in the entire data cube, up to 25 targets. Several algorithms were studied, but only one was

time-efficient [Hwan95c]. In this algorithm, the tasks pair off, then merge the two target lists, sorting by range and keeping up to 25 targets. The tasks holding the merged target lists then pair off, repeating this process until one task holds the target list for the entire data cube (see Figure 2.6).

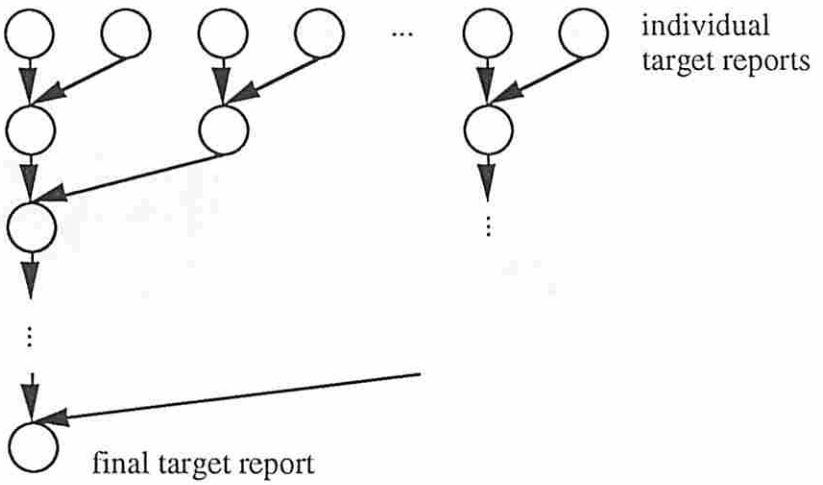


Figure 2.6: Target report merging process

Once the final target list is generated, the timer is stopped. The remainder of the program displays the final target list and collects execution time information. In order to improve the resolution and accuracy of the execution time measurements, we used two different timing calls: `times` and `gettimeofday`. The timing call `times` measures only CPU time used by the program, but has a resolution of only 10 ms. The timing call `gettimeofday` measures wall clock time (which may include time not spent on the APT program), but has a resolution of 1 μs.

3 The Parallel APT Code

The parallel program has been divided into the following files:

`bench_mark_APT.c`	`main ()`
`cell_avg_cfar.c`	`cell_avg_cfar ()`
`cmd_line.c`	`cmd_line ()`
`fft.c`	`fft (), bit_reverse ()`
`fft_APT.c`	`fft_APT ()`
`forback.c`	`forback ()`
`house.c`	`house ()`
`read_input_APT.c`	`read_input_APT ()`
`step1_beams.c`	`step1_beams ()`
`step2_beams.c`	`step2_beams ()`

The purpose of each procedure is briefly described below:

`main ()`	This procedure represents the main body of the parallel APT program.
`cell_avg_cfar ()`	This procedure performs the cell averaging CFAR target detection on the slice of the detect cube local to a node.
`cmd_line ()`	This procedure extracts the names of the data file and the steering 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 implementation 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 ()`.
`fft_APT ()`	This procedure performs an FFT along the PRI dimension for each column of data by calling `fft ()` (there are a total of RNG×EL such columns).
`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×M input array, where M ≥ N. The results are returned in the same matrix as the input data.
`read_input_APT ()`	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.
`step1_beams ()`	This procedure performs the first beamforming substep on the local slice of the data cube.
`step2_beams ()`	This procedure performs the second beamforming substep on the local slice of the data cube.

Figure 3.1 shows the calling relationship between the procedures.

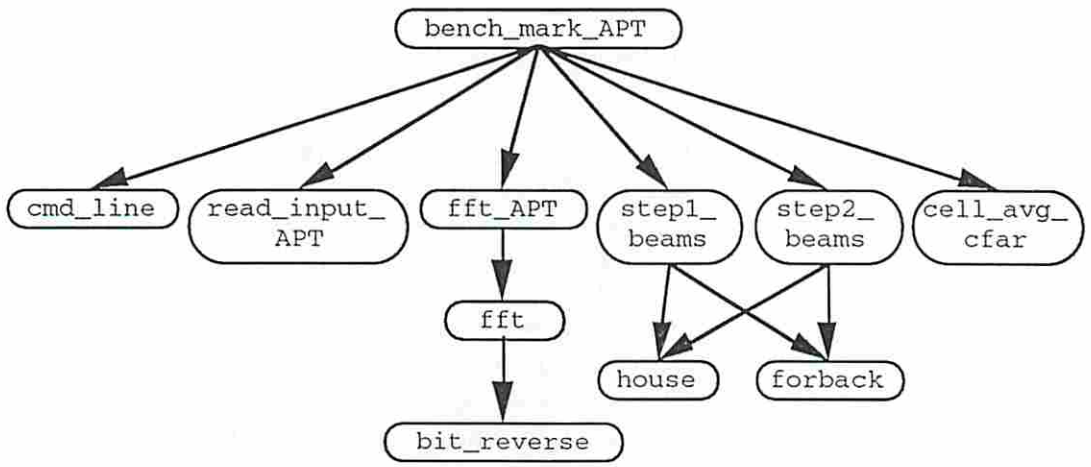


Figure 3.1: Calling relationship between the procedures in the parallel APT program

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

4 The APT Data Files

The input data cube file `new_data.dat` is a modified version of the original data cube file `apt_data.dat`. We reordered the data in `new_data.dat` so that each task can read its slice of the data cube from one contiguous location in the file. The modified data cube file still has PRI×RNG×EL complex numbers stored in 32-bit binary integer packed format. In this nominal data set, PRI = 256, RNG = 256, and EL = 32. On disk, this data file is 8 MB.

The steering vectors file `new_data.str` is identical to the original steering vector file `apt_data.str`. The steering vectors file contains the number of PRIs in the data cube, the power threshold used in target detection, and the complex values for the steering vectors.

5 Compiling the Parallel APT Program

The parallel APT benchmark is compiled by calling the script file `compile_apt`. This script file executes the following shell command:

```
mpcc -qarch=pwr2 -O3 -DAPT -DIBM -o apt bench_mark_APT.c \
  cell_avg_cfar.c fft.c fft_APT.c forback.c cmd_line.c house.c \
  read_input_APT.c step1_beams.c step2_beams.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 `-DAPT` 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 APT Program

The parallel APT 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 apt data_filename -procs ### -us
```

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 steering vectors. The input data cube file must have the extension .dat, while the steering 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 APT benchmark, with the

number of nodes equal to powers of two from 1 to 256. These files are called `run.001` through `run.256`.

Once the parallel program has begun and the timer has been started, the program will generate a "`Running...`" message. Once the program has finished and the timer has been stopped, it will generate a "`... done.`" message. The output of the program has the following format, shown in Figures 6.1 and 6.2. Figure 6.1 shows the target list, consisting of the targets found in the data cube, up to 25 targets. This number of targets was set as a parameter to the APT benchmark. Figure 6.2 shows the timing report from the parallel APT benchmark.

```
0:ENTRY     RANGE    BEAM    DOPPLER      POWER
0:   00       019      00       151      0.705780
0:   01       019      01       151      0.713037
0:   02       039      02       138      0.747974
0:   03       059      04       182      0.729151
0:   04       099      06       130      0.763610
0:   05       119      08       075      0.847957
0:   06       119      07       075      0.785197
0:   07       139      10       147      0.745536
0:   08       139      09       147      0.840084
0:   09       159      11       116      0.792221
0:   10       000      00       000      0.000000
0:   11       000      00       000      0.000000
0:   12       000      00       000      0.000000
0:   13       000      00       000      0.000000
0:   14       000      00       000      0.000000
0:   15       000      00       000      0.000000
0:   16       000      00       000      0.000000
0:   17       000      00       000      0.000000
0:   18       000      00       000      0.000000
0:   19       000      00       000      0.000000
0:   20       000      00       000      0.000000
0:   21       000      00       000      0.000000
0:   22       000      00       000      0.000000
0:   23       000      00       000      0.000000
0:   24       000      00       000      0.000000
```

Figure 6.1: Target list generated by the parallel APT program

In Figure 6.1, the ENTRY column counts the number of targets detected, starting with target #0. The RANGE column indicates the range gate in which the target was found, and is proportional to the distance to the target. The BEAM column indicates the direction to the target. The DOPPLER column provides information regarding the speed of the target. The POWER column represents the power of the signal returned by the target.

The targets listed in Figure 6.1 are those targets placed in the input data cube when the data cube was generated. We confirmed the correctness of our parallel APT program by comparing this target list to the target list generated by the original sequential APT benchmark (see [LL94]).

```
0:*** Timing information - numtask = 128
0:
0:  all_user_max      = 2.17 s, all_sys_max      = 0.03 s
0:  disk_user_max     = 0.14 s, disk_sys_max     = 0.02 s
0:  fft_user_max      = 0.04 s, fft_sys_max      = 0.00 s
0:  index_user_max    = 0.06 s, index_sys_max    = 0.00 s
0:  rewind_user_max   = 0.01 s, rewind_sys_max   = 0.00 s
0:  partial_user_max = 0.00 s, partial_sys_max = 0.00 s
0:  bcast_user_max    = 0.05 s, bcast_sys_max    = 0.00 s
0:  step1_user_max    = 0.04 s, step1_sys_max    = 0.00 s
0:  step2_user_max    = 0.09 s, step2_sys_max    = 0.00 s
0:  cfar_user_max     = 0.01 s, cfar_sys_max     = 0.00 s
0:  report_user_max   = 0.08 s, report_sys_max   = 0.00 s
0:
0:Wall clock timing -
0:  all_clock_time      = 2.339890
0:  disk_clock_time     = 0.346691
0:  fft_clock_time      = 0.018968
0:  index_clock_time    = 0.048371
0:  rewind_clock_time   = 0.001918
0:  partial_clock_time = 0.000653
0:  bcast_clock_time    = 0.023326
0:  step1_clock_time    = 0.037597
0:  step2_clock_time    = 0.076094
0:  cfar_clock_time     = 0.003508
0:  report_clock_time   = 0.075713
```

Figure 6.2: Timing report generated by the parallel APT program

In Figure 6.2, 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.

The steps we timed are listed below:

`all`	This component is the end-to-end execution time of the entire program. 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.
`fft`	This component is the time spent performing FFTs on this node's portion of the data cube.
`index`	This component is the time spent performing the total exchange operation.
`rewind`	This component is the time spent rewinding the 1-D output of the total exchange operation back into a 3-D data cube slice.
`partial`	This component is the time spent forming the `partial_temp` matrix.
`bcast`	This component is the time spent broadcasting the `partial_temp` matrix to all nodes.
`step1`	This component is the time spent performing the first beamforming substep.
`step2`	This component is the time spent performing the second beamforming substep.
`cfar`	This component is the time spent performing the cell averaging CFAR step.
`report`	This component is the time spent combining the individual target reports from all nodes into one final target report.

The time spent in these steps (T_{fft}, T_{index}, etc.) are combined to determine the total execution time of the parallel APT benchmark:

$$T_{execution} = T_{fft} + T_{index} + T_{rewind} + T_{partial} + T_{bcast} + T_{step1} + T_{step2} + T_{cfar} + T_{report} \quad (6.1)$$

The APT program's workload consists of 1.45 billion floating-point operations. By dividing this workload by the total execution time, we can calculate the sustained floating-point processing rate:

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

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

$$System\ Efficiency = \frac{Sustained\ Processing\ Rate}{Peak\ Processing\ Rate} \quad (6.3)$$

When determining the time spent for a particular step, we compared the sum of the user and system CPU times to the wall clock time, and took the smaller of the two. This time most accurately represents the time spent on the step. If the wall clock time is smaller than the sum of the user and system CPU times, the wall clock time includes only time spent on the benchmark, and we use this time, which has greater resolution. If the wall clock time is greater than the sum of the user and system CPU times, the wall clock time includes time spent on non-benchmark activities, and is not representative of the execution time for that step.

The timing results vary with machine size and from run to run, but the target list should be identical between runs and independent of machine size. Both the target list

and timing report are printed to the screen. 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 General 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 APT Code

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

A.1 bench_mark_APT.c

```
/*
 * bench_mark_APT.c
 */

/*
 * Parallel APT Benchmark Program for the IBM SP2
 * ----------------------------------------------
 *
 * This parallel APT 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 APT benchmark program was originally written by Tony Adams
 * on 7/12/93.
 *
 *
 * This file contains the procedure main (), and represents the body of the
 * APT benchmark.
 *
 * The program's overall structure is as follows:
 *    1.  Load data in parallel.
 *    2.  Perform FFTs in parallel.
 *    3.  Redistribute the data using the total exchange operation.
 *    4.  Form the Householder matrix.
 *    5.  Broadcast the Householder matrix to all nodes.
 *    6.  Perform beamforming 1 sequentially.
 *    7.  Perform beamforming 2 in parallel.
 *    8.  Perform CFAR/cell averaging in parallel.
 *    9.  Gather individual target reports from each node.
 *    10. Sort the target reports, and display the closest targets (up to
 *        25).
 *
 * This program can be compiled by calling the shell script compile_apt.
 * 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 APT
 * 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 "math.h"
#include <sys/types.h>
#include <sys/times.h>
#include <mpproto.h>
#include <sys/time.h>
#include <time.h>



/*
 * Global variables
 *
 * output_time: a flag; 1 = output timing information, 0 = don't output timing
 *   information
 * output_report: a flag; 1 = write data to disk, 0 = don't write data to disk
 * pricnt: the number of points in the FFT
 * threshold: minimum power return before a signal is considered a target
 * t_matrix: the T matrix
 */

int output_time = FALSE;
int output_report = FALSE;
int pricnt = PRI;
float threshold;
COMPLEX t_matrix[BEAM][EL];



/*
 * main ()
 *   inputs: int argc; char *argv[];
 *   outputs: none
 *
 * This procedure is the body of the program.
 */

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

{  /* main */

/*
 * Original variables (variables which were in the sequential version of
 * the APT program)
 *
 * target_report: an array holding data for the closest targets
 * targets: a loop counter
 * r: a loop counter
 * el: the maximum number of elements in each PRI
 * beams: the maximum number of beams
 * det_bms: the number of beams in the detection data
 * rng: the maximum number of range gates in each PRI
 * chol_rng: the number of range gates in each Cholesky segment
 * appl_rng: the number of range gates in each weight apply segment
 * f_temp: a buffer for the binary floating point output data
```

```
* i, j, k: loop counters
* row: a loop counter
* col: a loop counter
* cube_row: a loop counter for the row of in the input data cube
* dopplers: the number of dopplers after the FFT
* xrealptr: a pointer to the real part of the FFT vector
* ximagptr: a pointer to the imaginary part of the FFT vector
* str_vecs: steering vectors with EL COMPLEX elements each
* weight_vec: storage for the weight solution vectors
* make_t: a flag; 1 = generate the T matrix from the solution weight vectors,
*   0 = don't generate the T matrix but instead pass back the weight vector
* num: a holder for the EL complex input data elements
* tmp_ptr: a pointer to the next piece of data
* real_ptr: a pointer to the next piece of real data
* imag_ptr: a pointer to the next piece of imaginary data
* sum: a variable to hold the sum of complex data
* str_name: the name of the input steering vectors file
* data_name: the name of the input data file
* fopen: a pointer to the file open procedure
* f_tmatrix, f_dop: pointers to the output T matrix and Doppler files
*/

 static float target_report[25][4];
 int targets;
 int r;
 int el = EL;
 int beams = BEAM;
 int det_bms = MBEAM;
 int rng = RNG;
 int chol_rng = RNGSEG;
 int appl_rng = RNGSEG;
 float f_temp[1];
 int i, j, k;
 int row;
 int col;
 int cube_row;
 int dopplers;
 int logpoints;
 float *xrealptr;
 float *ximagptr;
 COMPLEX str_vecs[BEAM][DOF];
 COMPLEX weight_vec[DOF];
 int make_t;
 float num[2 * EL];
 float *tmp_ptr;
 float *real_ptr;
 float *imag_ptr;
 COMPLEX sum;
 char str_name[LINE_MAX];
 char data_name[LINE_MAX];
 FILE *fopen();
 FILE *f_tmatrix, *f_dop;

/*
 * Variables we added or modified to parallelize the code
 *
 * fft_out: the local portion of the data cube
 * local_cube: the local portion of the data cube after the total exchange
 *   operation
 * detect_cube: the local portion of the detect cube
 * partial_temp: the local portion of the temp_matrix
 * total_report: a target list containing all the targets found by all
 *   nodes
 * blklen: the length of the message being sent
 * rc: the return code from an MPL call
 * taskid: this task's unique identifier
 * numtask: the total number of tasks or nodes currently running this program
 * nbuf: an array used as part of the MPL call mpc_task_query; used to get
 *   the value of allgrp
```

```
 * source, dest, type, nbytes: variables used for point-to-point messages
 */

 static COMPLEX fft_out[PRI][RNG / NN][EL];
 static COMPLEX local_cube[PRI_CHUNK][RNG][EL];
 static COMPLEX detect_cube[MBEAM][PRI_CHUNK][RNG];
 static COMPLEX partial_temp[320][EL];
 static float total_report[25 * NN][4];
 int blklen;
 int rc;
 int taskid;
 int numtask;
 int nbuf[4];
 int allgrp;
 int source, dest, type, nbytes;

/*
 * Timing variables
 *
 * *_start and *_end: the start and end CPU times for...
 * *_user and *_sys: the user and system time breakdowns for the time spent
 *   for...
 * *_user_max and *_sys_max: the largest user and system times for...
 * *_clock_start and *_clock_end: the start and end wall clock times for...
 * *_clock_time: the net wall clock time spent for...
 *
 *   all: the entire program
 *   disk: reading the data from disk
 *   fft: the FFTs
 *   index: the MPL command mpc_index
 *   rewind: rewinding the 1-D output of the mpc_index operation into a 3-D
 *     data cube
 *   gather: the gather operation; the mpc_gather operation is no longer
 *     used to redistribute the data cube, as it is replaced by the
 *     mpc_index operation followed by the rewind operation
 *   reorder: the reorder operation, which is performed between the gather
 *     and scatter sequence to redistribute the data cube; this reorder
 *     operation is no longer used as this whole step is replaced by the
 *     mpc_index operation followed by the rewind operation
 *   scatter: the scatter operation; the mpc_scatter operation is no longer
 *     used to redistribute the data cube, as it is replaced by the mpc_index
 *     operation followed by the rewind operation
 *   partial: copying the appropriate portions of the data cube into the
 *     partial_temp matrix
 *   bcast: the broadcast operation
 *   step1: the first beamforming step
 *   step2: the second beamforming step
 *   cfar: the start and end time of the CFAR/cell averaging step
 *   report: the target reporting step
 */

 struct tms all_start, all_end;
 struct tms disk_start, disk_end;
 struct tms fft_start, fft_end;
 struct tms index_start, index_end;
 struct tms rewind_start, rewind_end;
 struct tms gather_start, gather_end;
 struct tms reorder_start, reorder_end;
 struct tms scatter_start, scatter_end;
 struct tms partial_start, partial_end;
 struct tms bcast_start, bcast_end;
 struct tms step1_start, step1_end;
 struct tms step2_start, step2_end;
 struct tms cfar_start, cfar_end;
 struct tms report_start, report_end;

 float all_user, all_sys;
 float disk_user, disk_sys;
 float fft_user, fft_sys;
```

```
  float index_user, index_sys;
  float rewind_user, rewind_sys;
  float partial_user, partial_sys;
  float bcast_user, bcast_sys;
  float step1_user, step1_sys;
  float step2_user, step2_sys;
  float cfar_user, cfar_sys;
  float report_user, report_sys;

  float all_user_max, all_sys_max;
  float disk_user_max, disk_sys_max;
  float fft_user_max, fft_sys_max;
  float index_user_max, index_sys_max;
  float rewind_user_max, rewind_sys_max;
  float partial_user_max, partial_sys_max;
  float bcast_user_max, bcast_sys_max;
  float step1_user_max, step1_sys_max;
  float step2_user_max, step2_sys_max;
  float cfar_user_max, cfar_sys_max;
  float report_user_max, report_sys_max;

  struct timeval all_clock_start, all_clock_end;
  struct timeval disk_clock_start, disk_clock_end;
  struct timeval fft_clock_start, fft_clock_end;
  struct timeval index_clock_start, index_clock_end;
  struct timeval rewind_clock_start, rewind_clock_end;
  struct timeval partial_clock_start, partial_clock_end;
  struct timeval bcast_clock_start, bcast_clock_end;
  struct timeval step1_clock_start, step1_clock_end;
  struct timeval step2_clock_start, step2_clock_end;
  struct timeval cfar_clock_start, cfar_clock_end;
  struct timeval report_clock_start, report_clock_end;

  float all_clock_time;
  float disk_clock_time;
  float fft_clock_time;
  float index_clock_time;
  float rewind_clock_time;
  float partial_clock_time;
  float bcast_clock_time;
  float step1_clock_time;
  float step2_clock_time;
  float cfar_clock_time;
  float report_clock_time;

/*
 * Variables added for gather/scatter (to replace index)
 *
 * The gather and scatter operation is no longer used to redistribute the
 * data cube. However, these variables are still used by the index and
 * rewind steps of this program, and therefore should not be removed.
 */

  static COMPLEX fft_vector[PRI * RNG * EL / NN];
  int offset;
  int n;

/*
 * Temporary variables for diagnostic purposes
 *
 * We used these variables in debugging this parallel version of the
 * APT program.
 */

  FILE *fp1, *fp2;
  int p, e, c, b;

/*
 * Externally defined functions
```

```
*/

 extern void cmd_line();
 extern void fft_APT();
 extern void step1_beams();
 extern void step2_beams();
 extern void cell_avg_cfar();

/*
 * 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.
 */

 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 (&all_clock_start, (struct timeval*) 0);
 times (&all_start);

/*
 * Get arguments from the command line. In the sequential version of the
 * program, the following procedure was used to extract the number of times
 * the main computational body (after the FFT) was to be repeated, and flags
 * regarding the amount of reporting to be done during and after the program
 * was run. In this parallel 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);
  gettimeofday (&disk_clock_start, (struct timeval*) 0);
  times (&disk_start);
  read_input_APT (data_name, str_name, str_vecs, fft_out);
  times (&disk_end);
  gettimeofday (&disk_clock_end, (struct timeval*) 0);

/*
 * FFT: In this section, each task performs FFTs along the PRI dimension
 * on its portion of the data cube. The FFT implementation used in this
 * program is a hybrid between the original implementation found in the
 * sequential version of this program and a suggestion given to us by MHPCC.
 * This change in implementation was done to improve the performance of the
 * FFT on the SP2.
 */

/*
 *  if (taskid == 0)
 *    {
 *      printf ("  running parallel FFT...\n");
 *    }
 */

  mpc_sync (allgrp);
  gettimeofday (&fft_clock_start, (struct timeval*) 0);
  times (&fft_start);
  fft_APT (fft_out);
  times (&fft_end);
  gettimeofday (&fft_clock_end, (struct timeval*) 0);

/*
 * Perform index operation to redistribute the data cube. Before the index
 * operation, the data cube was sliced along the RNG dimension, so each
 * task got all the PRIs. After the index operation, the data cube is sliced
 * along the PRI dimension, so each task gets all the RNGs.
 *
 * Because the MPL command mpc_index doesn't work to our specifications,
 * we need to rewind the 1-D matrix which is the output of the mpc_index
 * operation back into a 3-D data cube slice.
 */

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

  mpc_sync (allgrp);
  gettimeofday (&index_clock_start, (struct timeval*) 0);
  times (&index_start);
  rc = mpc_index (fft_out, fft_vector,
                  PRI * RNG * EL * sizeof (COMPLEX) / (NN * NN), allgrp);
  times (&index_end);
  gettimeofday (&index_clock_end, (struct timeval*) 0);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_index.\n");
      exit (-1);
    }

/*
 *  if (taskid == 0)
```

```
*     {
*       printf ("  rewinding data cube...\n");
*     }
*/

  mpc_sync (allgrp);
  gettimeofday (&rewind_clock_start, (struct timeval*) 0);
  times (&rewind_start);
  offset = 0;
  for (n = 0; n < NN; n++)
    {
      for (p = 0; p < PRI / NN; p++)
       {
         for (r = n * RNG / NN; r < (n + 1) * RNG / NN; r++)
           {
             for (e = 0; e < EL; e++)
              {
                local_cube[p][r][e].real = fft_vector[offset].real;
                local_cube[p][r][e].imag = fft_vector[offset].imag;
                offset++;
              }
           }
       }
    }

  times (&rewind_end);
  gettimeofday (&rewind_clock_end, (struct timeval*) 0);

/*
 * In this step, the partial_temp matrix is generated. If there are less than
 * 256 nodes, then the data which goes into the partial_temp matrix can be
 * found in tasks 0 and (NN - 1), where NN is the total number of tasks running
 * this program. If there are 256 nodes, then the data which goes into the
 * partial_temp matrix can be found in tasks 0, 1, 254, and 255.
 *
 * In the sequential version of the program, the partial_temp matrix was
 * formed in the first beamforming step (step1_beams ()).
 *
 * The Householder matrix which is being formed here is generated by taking
 * range gates 201-280 from dopplers 1, 2, doppler-1, and doppler into an
 * EL by RNG_S temp matrix. Nominally, EL = 32 elements and RNG_S = 320
 * samples. In the parallel version, because there are only 256 range gates,
 * the Householder matrix is generated by taking range gates 177-256.
 */

  if (NN < 256)
    {  /* if (NN < 256) */

/*
 *       if (taskid == 0)
 *     {
 *       printf ("  generating partial_temp...\n");
 *     }
 */

      mpc_sync (allgrp);
      gettimeofday (&partial_clock_start, (struct timeval*) 0);
      times (&partial_start);
      if (taskid == 0)
       {
         for (j = RNG - 80, i = 0; j < RNG; j++, i++)
           {
             for (k = 0; k < el; k++)
              {
                partial_temp[i][k].real = local_cube[0][j][k].real;
                partial_temp[i][k].imag = local_cube[0][j][k].imag;
                partial_temp[i + 80][k].real = local_cube[1][j][k].real;
                partial_temp[i + 80][k].imag = local_cube[1][j][k].imag;
              }
```

```
        }
      }

      if (taskid == (NN - 1))
      {
        for (j = RNG - 80, i = 0; j < RNG; j++, i++)
          {
            for (k = 0; k < el; k++)
            {
              partial_temp[i + 160][k].real
                = local_cube[PRI_CHUNK - 2][j][k].real;
              partial_temp[i + 160][k].imag
                = local_cube[PRI_CHUNK - 2][j][k].imag;
              partial_temp[i + 240][k].real
                = local_cube[PRI_CHUNK - 1][j][k].real;
              partial_temp[i + 240][k].imag
                = local_cube[PRI_CHUNK - 1][j][k].imag;
            }
          }
      }
      times (&partial_end);
      gettimeofday (&partial_clock_end, (struct timeval*) 0);
    }  /* if (NN < 256) */

  else
    {  /* if (NN == 256) */

/*
 *      if (taskid == 0)
 *    {
 *      printf ("  generating partial_temp...\n");
 *    }
 */

      mpc_sync (allgrp);
      gettimeofday (&partial_clock_start, (struct timeval*) 0);
      times (&partial_start);
      if (taskid == 0)
      {
        for (j = RNG - 80, i = 0; j < RNG; j++, i++)
          {
            for (k = 0; k < el; k++)
            {
              partial_temp[i][k].real = local_cube[0][j][k].real;
              partial_temp[i][k].imag = local_cube[0][j][k].imag;
            }
          }
      }

      if (taskid == 1)
      {
        for (j = RNG - 80, i = 0; j < RNG; j++, i++)
          {
            for (k = 0; k < el; k++)
            {
              partial_temp[i + 80][k].real = local_cube[1][j][k].real;
              partial_temp[i + 80][k].imag = local_cube[1][j][k].imag;
            }
          }
      }

      if (taskid == 254)
      {
        for (j = RNG - 80, i = 0; j < RNG; j++, i++)
          {
            for (k = 0; k < el; k++)
            {
              partial_temp[i + 160][k].real
                = local_cube[PRI_CHUNK - 2][j][k].real;
```

```
                partial_temp[i + 160][k].imag
                  = local_cube[PRI_CHUNK - 2][j][k].imag;
              }
            }
        }

        if (taskid == 255)
        {
          for (j = RNG - 80, i = 0; j < RNG; j++, i++)
            {
              for (k = 0; k < el; k++)
              {
                partial_temp[i + 240][k].real
                  = local_cube[PRI_CHUNK - 1][j][k].real;
                partial_temp[i + 240][k].imag
                  = local_cube[PRI_CHUNK - 1][j][k].imag;
              }
            }
        }
        gettimeofday (&partial_clock_end, (struct timeval*) 0);
        times (&partial_end);
      }  /* if (NN == 256) */

/*
 * Once the appropriate tasks copy portions of their data cube slices into
 * sections of partial_temp, these tasks then broadcast these sections to
 * all the nodes. After these broadcasts are done, each task will have
 * identical copies of the complete partial_temp matrix.
 */

    if (NN < 256)
      {  /* if (NN < 256) */
        blklen = 160 * EL * sizeof (COMPLEX);

        mpc_sync (allgrp);
        gettimeofday (&bcast_clock_start, (struct timeval*) 0);
        times (&bcast_start);
        rc = mpc_bcast (partial_temp, blklen, 0, allgrp);
        times (&bcast_end);
        if (rc == -1)
        {
          printf ("Error - unable to call mpc_bcast.\n");
          exit(-1);
        }

        bcast_user = bcast_end.tms_utime - bcast_start.tms_utime;
        bcast_sys = bcast_end.tms_stime - bcast_start.tms_stime;

        times (&bcast_start);
        rc = mpc_bcast (partial_temp + 160, blklen, NN - 1, allgrp);
        times (&bcast_end);
        gettimeofday (&bcast_clock_end, (struct timeval*) 0);
        if (rc == -1)
        {
          printf ("Error - unable to call mpc_bcast.\n");
          exit(-1);
        }

        bcast_user += bcast_end.tms_utime - bcast_start.tms_utime;
        bcast_sys += bcast_end.tms_stime - bcast_start.tms_stime;
      }  /* if (NN < 256) */

    else
      {  /* if (NN == 256) */
        blklen = 80 * EL * sizeof (COMPLEX);

        mpc_sync (allgrp);
        gettimeofday (&bcast_clock_start, (struct timeval*) 0);
        times (&bcast_start);
```

```
      rc = mpc_bcast (partial_temp, blklen, 0, allgrp);
      times (&bcast_end);
      if (rc == -1)
       {
         printf ("Error - unable to call mpc_bcast.\n");
         exit(-1);
       }

      bcast_user = bcast_end.tms_utime - bcast_start.tms_utime;
      bcast_sys = bcast_end.tms_stime - bcast_start.tms_stime;

      times (&bcast_start);
      rc = mpc_bcast (partial_temp + 80, blklen, 1, allgrp);
      times (&bcast_end);
      if (rc == -1)
       {
         printf ("Error - unable to call mpc_bcast.\n");
         exit(-1);
       }

      bcast_user += bcast_end.tms_utime - bcast_start.tms_utime;
      bcast_sys += bcast_end.tms_stime - bcast_start.tms_stime;

      times (&bcast_start);
      rc = mpc_bcast (partial_temp + 160, blklen, 254, allgrp);
      times (&bcast_end);
      if (rc == -1)
       {
         printf ("Error - unable to call mpc_bcast.\n");
         exit(-1);
       }

      bcast_user += bcast_end.tms_utime - bcast_start.tms_utime;
      bcast_sys += bcast_end.tms_stime - bcast_start.tms_stime;

      times (&bcast_start);
      rc = mpc_bcast (partial_temp + 240, blklen, 255, allgrp);
      times (&bcast_end);
      gettimeofday (&bcast_clock_end, (struct timeval*) 0);
      if (rc == -1)
       {
         printf ("Error - unable to call mpc_bcast.\n");
         exit(-1);
       }

      bcast_user += bcast_end.tms_utime - bcast_start.tms_utime;
      bcast_sys += bcast_end.tms_stime - bcast_start.tms_stime;
    }  /* if (NN == 256) */

/*
 * Step 1 Beamforming: This step cannot be parallelized efficiently, and was
 * left to run sequentially. In order to improve performance, we chose to have
 * each task execute this step. The alternative, having one task execute this
 * step, then broadcast the results, takes much longer.
 */

/*
 *  if (taskid == 0)
 *    {
 *      printf ("  performing step 1 beamforming...\n");
 *    }
 */

  dopplers = PRI_CHUNK;
  mpc_sync (allgrp);
  gettimeofday (&step1_clock_start, (struct timeval*) 0);
  times (&step1_start);
  step1_beams (str_vecs, partial_temp);
  times (&step1_end);
```

```
  gettimeofday (&step1_clock_end, (struct timeval*) 0);

/*
 * Step 2 Beamforming: Each task performs this beamforming step on its slice
 * of the the data cube.
 */

/*
 *  if (taskid == 0)
 *    {
 *      printf ("  performing step 2 beamforming...\n");
 *    }
 */

  mpc_sync (allgrp);
  gettimeofday (&step2_clock_start, (struct timeval*) 0);
  times (&step2_start);
  step2_beams (dopplers, beams, rng, str_vecs, local_cube, detect_cube);
  times (&step2_end);
  gettimeofday (&step2_clock_end, (struct timeval*) 0);

/*
 * Cell averaging CFAR and target detection: Each task performs this step on
 * its slice of the data cube in parallel.
 */

/*
 *  if (taskid == 0)
 *    {
 *      printf ("  performing cell_avg_cfar...\n");
 *    }
 */

  mpc_sync (allgrp);
  gettimeofday (&cfar_clock_start, (struct timeval*) 0);
  times (&cfar_start);
  cell_avg_cfar (threshold, dopplers, det_bms, rng, target_report,
                 detect_cube);
  times (&cfar_end);
  gettimeofday (&cfar_clock_end, (struct timeval*) 0);

/*
 * Gather target reports: In this step, the tasks collect the closest 25
 * targets. This target sorting is performed in the following manner. The tasks
 * pair off. The two tasks in this pair combine their target lists (i.e. the
 * targets found in their own slice of the data cube). Then, one of these
 * two tasks sorts the list and keeps the closest targets (up to 25). Then,
 * the tasks with these new, combined target lists pair off, and this
 * process is repeated again. At the end, one task will have the final target
 * list, which contains the closest targets in the entire data cube (up to
 * 25).
 *
 * The target list combining is done by matched blocking-sends and blocking-
 * receives.
 */

/*
 *  if (taskid == 0)
 *    {
 *      printf ("  gathering target reports...\n");
 *    }
 */

  mpc_sync (allgrp);
  gettimeofday (&report_clock_start, (struct timeval*) 0);
  times (&report_start);

  for (i = 1; i < NN; i = 2 * i)
    {  /* for (i...) */
```

```
      blklen = 25 * 4 * sizeof (float);
      source = taskid + i;
      dest = taskid - i;
      type = i;
      if ((taskid % (2 * i)) == 0 && (NN != 1))
       {
         rc = mpc_brecv (target_report + 25, blklen, &source, &type,
                         &nbytes);
         if (rc == -1)
           {
             printf ("Error - unable to call mpc_brecv.\n");
             exit(-1);
           }
       }

      if ((taskid % (2 * i)) == i && (NN != 1))
       {
         rc = mpc_bsend (target_report, blklen, dest, type);
         if (rc == -1)
           {
             exit(-1);
           }
       }

/*
 * Sort combined target list.
 */

      for (targets = 0; targets < 50; targets++)
       {  /* for (targets...) */
         for (r = targets + 1 ; r < 50 ; r ++)
           {  /* for (r...) */
             if (((target_report[targets][0] > target_report[r][0])
                 && (target_report[r][0] > 0.0))
                || ((target_report[targets][0] < target_report[r][0])
                    && (target_report[targets][0] == 0.0)))
              {  /* if (...) */
                float tmp;

                tmp = target_report[r][0];
                target_report[r][0] = target_report[targets][0];
                target_report[targets][0] = tmp;
                tmp = target_report[r][1];
                target_report[r][1] = target_report[targets][1];
                target_report[targets][1] = tmp;
                tmp = target_report[r][2];
                target_report[r][2] = target_report[targets][2];
                target_report[targets][2] = tmp;
                tmp = target_report[r][3];
                target_report[r][3] = target_report[targets][3];
                target_report[targets][3] = tmp;
              }  /* if (...) */
           }  /* for (r...) */
       }  /* for (targets...) */
    }  /* for (i...) */

  times (&report_end);
  times (&all_end);
  gettimeofday (&report_clock_end, (struct timeval*) 0);
  gettimeofday (&all_clock_end, (struct timeval*) 0);

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

/*
 * Now, the program is done with the computation. All that remains to be done
 * is to report the targets that were found, and to report the amount of time
```

```
 * each step took. The target list should be identical from run to run, since
 * the program started with the same input data cube. This target list and
 * execution time data reporting is performed by task 0.
 */

  if (taskid == 0)
    {
      if (output_report == FALSE)
       {
         printf ("ENTRY    RANGE   BEAM   DOPPLER      POWER\n");
         for (i = 0; i < 25; i++)
           printf ("  %.02d      %.03d      %.02d      %.03d      %f\n", i,
                 (int)target_report[i][0], (int)target_report[i][1],
                 (int)target_report[i][2], target_report[i][3] );
       }
    }

/*
 * Collect timing information. First, calculate the number of seconds of CPU
 * time spent in each step (user and system time separately).
 */

  all_user = (all_end.tms_utime - all_start.tms_utime) / CLK_TICK;
  all_sys = (all_end.tms_stime - all_start.tms_stime) / CLK_TICK;
  disk_user = (disk_end.tms_utime - disk_start.tms_utime) / CLK_TICK;
  disk_sys = (disk_end.tms_stime - disk_start.tms_stime) / CLK_TICK;
  fft_user = (fft_end.tms_utime - fft_start.tms_utime) / CLK_TICK;
  fft_sys = (fft_end.tms_stime - fft_start.tms_stime) / CLK_TICK;
  index_user = (index_end.tms_utime - index_start.tms_utime) / CLK_TICK;
  index_sys = (index_end.tms_stime - index_start.tms_stime) / CLK_TICK;
  rewind_user = (rewind_end.tms_utime - rewind_start.tms_utime) / CLK_TICK;
  rewind_sys = (rewind_end.tms_stime - rewind_start.tms_stime) / CLK_TICK;
  partial_user = (partial_end.tms_utime - partial_start.tms_utime) / CLK_TICK;
  partial_sys = (partial_end.tms_stime - partial_start.tms_stime) / CLK_TICK;
  bcast_user = bcast_user / CLK_TICK;
  bcast_sys = bcast_sys / CLK_TICK;
  step1_user = (step1_end.tms_utime - step1_start.tms_utime) / CLK_TICK;
  step1_sys = (step1_end.tms_stime - step1_start.tms_stime) / CLK_TICK;
  step2_user = (step2_end.tms_utime - step2_start.tms_utime) / CLK_TICK;
  step2_sys = (step2_end.tms_stime - step2_start.tms_stime) / CLK_TICK;
  cfar_user = (cfar_end.tms_utime - cfar_start.tms_utime) / CLK_TICK;
  cfar_sys = (cfar_end.tms_stime - cfar_start.tms_stime) / CLK_TICK;
  report_user = (report_end.tms_utime - report_start.tms_utime) / CLK_TICK;
  report_sys = (report_end.tms_stime - report_start.tms_stime) / CLK_TICK;

/*
 * Calculate the number of wall clock seconds spent on each section.
 */

  all_clock_time = (float) (all_clock_end.tv_sec
                          - all_clock_start.tv_sec)
    + (float) ((all_clock_end.tv_usec
              - all_clock_start.tv_usec) / 1000000.0);

  disk_clock_time = (float) (disk_clock_end.tv_sec
                          - disk_clock_start.tv_sec)
    + (float) ((disk_clock_end.tv_usec
              - disk_clock_start.tv_usec) / 1000000.0);

  fft_clock_time = (float) (fft_clock_end.tv_sec
                          - fft_clock_start.tv_sec)
    + (float) ((fft_clock_end.tv_usec
              - fft_clock_start.tv_usec) / 1000000.0);

  index_clock_time = (float) (index_clock_end.tv_sec
                          - index_clock_start.tv_sec)
    + (float) ((index_clock_end.tv_usec
              - index_clock_start.tv_usec) / 1000000.0);
```

```
  rewind_clock_time = (float) (rewind_clock_end.tv_sec
                        - rewind_clock_start.tv_sec)
    + (float) ((rewind_clock_end.tv_usec
              - rewind_clock_start.tv_usec) / 1000000.0);

  partial_clock_time = (float) (partial_clock_end.tv_sec
                        - partial_clock_start.tv_sec)
    + (float) ((partial_clock_end.tv_usec
              - partial_clock_start.tv_usec) / 1000000.0);

  bcast_clock_time = (float) (bcast_clock_end.tv_sec
                        - bcast_clock_start.tv_sec)
    + (float) ((bcast_clock_end.tv_usec
              - bcast_clock_start.tv_usec) / 1000000.0);

  step1_clock_time = (float) (step1_clock_end.tv_sec
                        - step1_clock_start.tv_sec)
    + (float) ((step1_clock_end.tv_usec
              - step1_clock_start.tv_usec) / 1000000.0);

  step2_clock_time = (float) (step2_clock_end.tv_sec
                        - step2_clock_start.tv_sec)
    + (float) ((step2_clock_end.tv_usec
              - step2_clock_start.tv_usec) / 1000000.0);

  cfar_clock_time = (float) (cfar_clock_end.tv_sec
                        - cfar_clock_start.tv_sec)
    + (float) ((cfar_clock_end.tv_usec
              - cfar_clock_start.tv_usec) / 1000000.0);

  report_clock_time = (float) (report_clock_end.tv_sec
                        - report_clock_start.tv_sec)
    + (float) ((report_clock_end.tv_usec
              - report_clock_start.tv_usec) / 1000000.0);

/*
 * Use the mpc_reduce operation to find the largest CPU time in each section
 * (user and system time separately).
 */

  rc = mpc_reduce (&all_user, &all_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&all_sys, &all_sys_max, sizeof (float), 0, s_vmax, allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }

  rc = mpc_reduce (&disk_user, &disk_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&disk_sys, &disk_sys_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
```

```
rc = mpc_reduce (&fft_user, &fft_user_max, sizeof (float), 0, s_vmax,
                 allgrp);
if (rc == -1)
  {
    printf ("Error - unable to call mpc_reduce.\n");
    exit (-1);
  }
rc = mpc_reduce (&fft_sys, &fft_sys_max, sizeof (float), 0, s_vmax, allgrp);
if (rc == -1)
  {
    printf ("Error - unable to call mpc_reduce.\n");
    exit (-1);
  }

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

rc = mpc_reduce (&rewind_user, &rewind_user_max, sizeof (float), 0, s_vmax,
                 allgrp);
if (rc == -1)
  {
    printf ("Error - unable to call mpc_reduce.\n");
    exit (-1);
  }
rc = mpc_reduce (&rewind_sys, &rewind_sys_max, sizeof (float), 0, s_vmax,
                 allgrp);
if (rc == -1)
  {
    printf ("Error - unable to call mpc_reduce.\n");
    exit (-1);
  }

rc = mpc_reduce (&partial_user, &partial_user_max, sizeof (float), 0,
                 s_vmax, allgrp);
if (rc == -1)
  {
    printf ("Error - unable to call mpc_reduce.\n");
    exit (-1);
  }
rc = mpc_reduce (&partial_sys, &partial_sys_max, sizeof (float), 0, s_vmax,
                 allgrp);
if (rc == -1)
  {
    printf ("Error - unable to call mpc_reduce.\n");
    exit (-1);
  }

rc = mpc_reduce (&bcast_user, &bcast_user_max, sizeof (float), 0, s_vmax,
                 allgrp);
if (rc == -1)
  {
    printf ("Error - unable to call mpc_reduce.\n");
    exit (-1);
  }
rc = mpc_reduce (&bcast_sys, &bcast_sys_max, sizeof (float), 0, s_vmax,
                 allgrp);
if (rc == -1)
```

```
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }

  rc = mpc_reduce (&step1_user, &step1_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&step1_sys, &step1_sys_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }

  rc = mpc_reduce (&step2_user, &step2_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&step2_sys, &step2_sys_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }

  rc = mpc_reduce (&cfar_user, &cfar_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&cfar_sys, &cfar_sys_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }

  rc = mpc_reduce (&report_user, &report_user_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }
  rc = mpc_reduce (&report_sys, &report_sys_max, sizeof (float), 0, s_vmax,
                   allgrp);
  if (rc == -1)
    {
      printf ("Error - unable to call mpc_reduce.\n");
      exit (-1);
    }

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

```
  if (taskid == 0)
    {
      printf ("\n\n\n*** 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 ("  fft_user_max     = %.2f s, fft_sys_max     = %.2f s\n",
              fft_user_max, fft_sys_max);
      printf ("  index_user_max   = %.2f s, index_sys_max   = %.2f s\n",
              index_user_max, index_sys_max);
      printf ("  rewind_user_max  = %.2f s, rewind_sys_max  = %.2f s\n",
              rewind_user_max, rewind_sys_max);
      printf ("  partial_user_max = %.2f s, partial_sys_max = %.2f s\n",
              partial_user_max, partial_sys_max);
      printf ("  bcast_user_max   = %.2f s, bcast_sys_max   = %.2f s\n",
              bcast_user_max, bcast_sys_max);
      printf ("  step1_user_max   = %.2f s, step1_sys_max   = %.2f s\n",
              step1_user_max, step1_sys_max);
      printf ("  step2_user_max   = %.2f s, step2_sys_max   = %.2f s\n",
              step2_user_max, step2_sys_max);
      printf ("  cfar_user_max    = %.2f s, cfar_sys_max    = %.2f s\n",
              cfar_user_max, cfar_sys_max);
      printf ("  report_user_max  = %.2f s, report_sys_max  = %.2f s\n",
              report_user_max, report_sys_max);

      printf ("\nWall clock timing -\n");
      printf ("  all_clock_time     = %f\n", all_clock_time);
      printf ("  disk_clock_time    = %f\n", disk_clock_time);
      printf ("  fft_clock_time     = %f\n", fft_clock_time);
      printf ("  index_clock_time   = %f\n", index_clock_time);
      printf ("  rewind_clock_time  = %f\n", rewind_clock_time);
      printf ("  partial_clock_time = %f\n", partial_clock_time);
      printf ("  bcast_clock_time   = %f\n", bcast_clock_time);
      printf ("  step1_clock_time   = %f\n", step1_clock_time);
      printf ("  step2_clock_time   = %f\n", step2_clock_time);
      printf ("  cfar_clock_time    = %f\n", cfar_clock_time);
      printf ("  report_clock_time  = %f\n", report_clock_time);
    }
  return;
}  /* main */
```

A.2 cell_avg_cfar.c

```
/*
 * cell_avg_cfar.c
 */

/*
 * This file contains the procedure cell_avg_cfar (), and is part of the
 * parallel APT 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 APT benchmark program was originally written by Tony Adams
 * on 7/12/93.
 *
 * This procedure was largely untouched during our parallelization effort,
 * and therefore almost identical to the sequential version. Modifications
 * were made to perform the cell averaging and CFAR for only a slice of the
 * data cube, instead of the entire data cube (as was the case in the
 * sequential version of the program). As such, most, if not all, of the
 * comments in this procedure are taken directly from the sequential version
```

```
 * of this program.
 */

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



/*
 * cell_avg_cfar ()
 *   inputs: threshold, dopplers, beams, rng, detect_cube
 *   outputs: target_report
 *
 *   threshold: when the power at a certain cell exceeds this threshold,
 *     we consider it as having a target
 *   dopplers: the number of dopplers in the input data
 *   beams: the number of beams (rows) in the input data
 *   rng: the number of range gates (columns) in the input matrix
 *   target_report: a 25-element target report matrix (each element consisting
 *     of four floating point numbers: RANG, BEAM, DOPPLER, and POWER), to
 *     output the 25 closest targets found in this slice of detect_cube
 *   detect_cube: input beam/doppler/rnggate matrix
 */

cell_avg_cfar (threshold, dopplers, beams, rng, target_report, detect_cube)
     float threshold;
     int dopplers;
     int beams;
     int rng;
     float target_report[][4];
     COMPLEX detect_cube[][DOP/NN][RNG];

{

/*
 * Variables: Most, if not all, of the variable comments were taken directly
 * from the sequential version of this program.
 *
 * sum: a holder for the cell sum
 * num_seg: number of range segments
 * rng_seg: number of range gates per segment
 * targets: holder for the number of targets to include in the target report
 * num_targets: the number of targets to include in the target report
 * N: holder for the number of range gates per range segment
 * guard_range: number of cells away from cell of interest
 * rseg, r, i: loop variables
 * beam, dop: loop variables
 * rng_start: address of the first range gate in each range segment
 * rng_end: address of the last range gate in each range segment
 * ncells: holder for cells: guard_range per range segment
 * taskid: the identifier for this task (added for parallel execution)
 */

  float sum;
  int num_seg = NUMSEG;
  int rng_seg = RNGSEG;
  int targets;
  int num_targets = 25;
  int N;
  int guard_range = 3;
  int rseg, r, i;
  int beam, dop;
  int rng_start;
  int rng_end;
  int ncells;
```

```
  extern int taskid;

/*
 * Begin function body: cell_avg_cfar ()
 *
 * Process cell averaging cfar by range segments: do cell averaging in ranges.
 */

  N = rng_seg;  /* Number of range cells per range segment */
  targets = 0;  /* Start with 0 targets in target report number */
  for (i = 0; i < num_targets; i++)
    {
      target_report[i][0] = 0.0;
      target_report[i][1] = 0.0;
      target_report[i][2] = 0.0;
      target_report[i][3] = 0.0;
    }

/*
 * Do the entire cell average CFAR algorithm starting from the first range
 * segment and continuing to the last range segment. This ensures getting
 * 25 least-range targets first, so the process can be stopped without
 * looking further into longer ranges.
 */

  for (rseg = 0; rseg < num_seg; rseg++)
    {  /* for (rseg...) */

/*
 * Get start and end range gates for each of the range segments: starts at
 * low ranges and increments to higher ranges.
 */

      rng_start = rseg * N;
      rng_end = rng_start + N - 1;

/*
 * 1st get the range cell power and store it in detect_cube[][][].real. For
 * each beam and each doppler, get the power for each range cell in the
 * current range segment.
 */

      for (beam = 0; beam < beams; beam++)
       {  /* for (beam...) */
         for (dop = 0; dop < dopplers; dop++)
           {  /* for (dop...) */
             for (r = rng_start ; r <= rng_end; r++)
              {  /* for (r...) */
                sum = detect_cube[beam][dop][r].real
                  * detect_cube[beam][dop][r].real
                    + detect_cube[beam][dop][r].imag
                     * detect_cube[beam][dop][r].imag;
                detect_cube[beam][dop][r].real = sum;
              }  /* for (r...) */
           }  /* for (dop...) */
       }  /* for (beam...) */

/*
 * Now, get the range cell's average power, using cell averaging CFAR described
 * above, and store it in detect_cube[][][].imag. For each beam and each
 * doppler, get the average power for each range cell in the current range
 * segment.
 */

      for (beam = 0; beam < beams; beam++)
       {  /* for (beam...) */
         for (dop = 0; dop < dopplers; dop++)
           {  /* for (dop...) */
             sum = 0.0;
```

```
            ncells = N - guard_range - 1;

/*
 * Do a summation loop for the first range cell at the start of a range
 * segment.
 */

            for (r = rng_start + guard_range + 1; r <= rng_end; r++)
             {
               sum += detect_cube[beam][dop][r].real;
             }

/*
 * The average power is the sum divided by the number of cells in the
 * summation.
 */

            detect_cube[beam][dop][rng_start].imag = sum / ncells;

/*
 * Now, perform loops until the guard band is fully involved in the data.
 */

            for (r = rng_start + 1; r <= rng_start + guard_range; r++)
             {  /* for (r...) */
               sum = sum - detect_cube[beam][dop][r + guard_range].real;
               --ncells;
               detect_cube[beam][dop][r].imag = sum / ncells;
             }  /* for (r...) */


/*
 * Now, perform loops until the guard band reaches the range segment border.
 */

            for (r = rng_start + guard_range + 1;
                r <= rng_end - guard_range; r++)
             {
               sum = sum + detect_cube[beam][dop][r-guard_range-1].real
                 - detect_cube[beam][dop][r + guard_range].real;
               detect_cube[beam][dop][r].imag = sum / ncells;
             }

/*
 * Now, perform loops to the end of the range segment.
 */

            for (r = rng_end - guard_range + 1; r <= rng_end; r++)
             {
               sum += detect_cube[beam][dop][r - guard_range - 1].real;
               ++ncells;
               detect_cube[beam][dop][r].imag = sum / ncells;
             }
          }  /* for (dop...) */
      }  /* for (beam...) */

/*
 * Compare the range cell power to the range cell average power. Start at
 * the minimum range and increase the range until 25 targets are found. Put
 * the target's RANGE, BEAM, DOPPLER, and POWER level in a matrix called
 * "target_report". Store all values as floating point numbers. Some can get
 * converted to integers on printout later as required.
 */

      for (r = rng_start ; r <= rng_end; r++)
       {  /* for (r...) */
         for (beam = 0; beam < beams; beam++)
           {  /* for (beam...) */
             for (dop = 0; dop < dopplers; dop++)
```

```
            {  /* for (dop...) */
              if (((detect_cube[beam][dop][r].real
                   - detect_cube[beam][dop][r].imag) >
                   threshold) && (targets <= num_targets))
                {  /* if (((...))) */
                  target_report[targets][0] = (float) r;
                  target_report[targets][1] = (float) beam;
                  target_report[targets][2] = (float) dop
                   + taskid * PRI / NN;
                  target_report[targets][3]
                   = detect_cube[beam][dop][r].real;
                  targets += 1;
                  if (targets >= num_targets)
                   {
                     goto quit_report;
                   }
                }  /* if (((...))) */
            }  /* for (dop...) */
          }  /* for (beam...) */
        }  /* for (r...) */
    }  /* for (rseg..) */
quit_report: ;
 return;
}
```

A.3 cmd_line.c

```
/*
 * cmd_line.c
 */

/*
 * This file contains the procedure cmd_line (), and is part of the parallel
 * APT 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 APT benchmark program was originally written by Tony Adams
 * on 7/12/93.
 *
 * 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.4 fft.c

```
/*
 * fft.c
 */

/*
 * This file contains the procedures fft () and bit_reverse (), and is part
 * of the parallel APT 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 APT benchmark program was originally written by Tony Adams
 * on 7/12/93.
 *
 * 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 MHPCC. 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.5 fft_APT.c

```
/*
 * fft_APT.c
 */

/*
 * This file contains the procedure fft_APT (), and is part of the parallel
 * APT 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 APT benchmark program was originally written by Tony Adams
 * on 7/12/93.
 *
 * The procedure fft_APT () performs an FFT along the PRI dimension for
 * each column of data (there are a total of RNG*EL such columns) by calling
 * the procedure fft ().
 */

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

/*
 * fft_APT ()
 *   inputs: fft_out
 *   outputs: fft_out
 *
 *   fft_out: the FFT output data cube
 */

fft_APT (fft_out)
     COMPLEX fft_out[][RNG / NN][EL];

{

/*
 * Variables
 *
 * pricnt: pricnt globally defined in main
 * output_report: not used in parallel version
 * fft (): a pointer to the FFT function (which performs a radix 2
 *   calculation)
 * taskid: an identifier for this task (included for parallel execution)
```

```
 * el: the maximum number of elements in each PRI
 * beams: the maximum number of beams
 * rng: the maximum number of range gates in each PRI, in this data cube slice
 *   (modified for parallel execution)
 * i, j, k: loop counters
 * logpoints: log base 2 of the number of points in the FFT
 * pix2, pi: 2*pi and pi
 * x: storage for the temporary FFT vector
 * w: storage for the twiddle factors table
 */

  extern int pricnt;
  extern int output_report;
  extern void fft ();
  extern int taskid;

  int el = EL;
  int beams = BEAM;
  int rng = RNG / NN;
  int i, j, k;
  int logpoints;

  float pix2, pi;
  static COMPLEX x[PRI];
  static COMPLEX w[PRI];

/*
 * Begin function body: fft_APT ()
 *
 * Generate twiddle factors table w.
 */

  logpoints = log2 ((float) pricnt) + 0.1;
  pi = 3.14159265358979;
  pix2 = 2.0 * pi;

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

/*
 * For each range gate and each element, move all the PRIs into real and
 * imaginary vectors for the FFT.
 */

  for (i = 0; i < rng; i++)
    {
      for (k = 0; k < el; k++)
       {
         for (j = 0; j < pricnt; j++)
           {
             x[j].real = fft_out[j][i][k].real;
             x[j].imag = fft_out[j][i][k].imag;
           }

/*
 * Perform the FFTs.
 */

         fft (x, w, pricnt, logpoints);

/*
 * Move the FFT'ed data back into fft_out.
 */

         for (j = 0; j < pricnt; j++)
           {
```

```
            fft_out[j][i][k].real = x[j].real;
            fft_out[j][i][k].imag = x[j].imag;
          }
      }
    }
  return;
}
```

A.6 forback.c

```
/*
 * forback.c
 */

/*
 * This file contains the procedure forback (), and is part of the parallel
 * APT 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 APT benchmark program was originally written by Tony Adams
 * on 7/12/93.
 *
 * 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.7 house.c

```
/*
 * house.c
 */

/*
 * This file contains the procedure house (), and is part of the parallel APT
 * benchmark 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 APT benchmark program was originally written by Tony Adams
```

```
 * on 7/12/93.
 *
 * 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.8 read_input_APT.c

```
/*
 * read_input_APT.c
 */

/*
 * This file contains the procedure read_input_APT (), and is part of the
 * parallel APT 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 APT benchmark program was originally written by Tony Adams
 * on 7/12/93.
 *
 * The procedure read_input_APT () reads the input data files (containing
 * the data cube and the steering vectors).
 *
 * In this parallel version of read_input_APT (), each task reads its portion
 * of the data cube in from the data file simultaneously. 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

/*
 * Externally defined variables (needed for parallel execution)
 *
 * taskid: an identifier for this task
 * numtask: the total number of tasks running this program
 * allgrp: an identifier for all the tasks running this program (used for
 *   collective communications and aggregated computations)
 */

extern int taskid, numtask, allgrp;

/*
 * read_input_APT ()
 *   inputs: data_name, str_name
 *   outputs: str_vecs, fft_out
 *
 *   data_name: the name of the file containing the data cube
 *   str_name: the name of the file containing the steering vectors
 *   str_vecs: a matrix holding the steering vectors
 *   fft_out: this task's slice of the data cube
 */

read_input_APT (data_name, str_name, str_vecs, fft_out)
     char data_name[];
     char str_name[];
     COMPLEX str_vecs[][DOF];
     COMPLEX fft_out[][RNG / NN][EL];

{

/*
 * Variables
 *
 * el: the maximum number of elements in each PRI
 * beams: the maximum number of beams
 * rng: the maximum number of range gates
 * temp: a buffer for binary input integer data
 * temp1, temp2: holders for integer data
 * i, j, k: loop counters
 * fopen (): a pointer to the file open function
 * f_str, f_dat: pointers to the input files
 * local_int_cube: the local portion of the data cube
 * pricnt: the pricnt variable globally defined in main ()
 * threshold: the threshold variable globally defined in main ()
```

```
 * blklen, rc: not used
 */

  int el = EL;
  int beams = BEAM;
  int rng = RNG / NN;
  unsigned int temp[1];
  long int temp1, temp2;
  int i, j, k;
  FILE *fopen();
  FILE *f_str, *f_dat;
  unsigned int local_int_cube[RNG/NN][PRI][EL];
  extern int pricnt;
  extern float threshold;
  long blklen, rc;

/*
 * Begin function body: read_input_APT ()
 *
 * Every task: load the steering vector file.
 */

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

/*
 * The first item in the steering vector file is the number of PRIs. The
 * second item in the steering vector file is the target detection threshold.
 */

  fscanf (f_str, "%d", &pricnt);
  fscanf (f_str, fmt, &threshold);

/*
 * Read in the rest of the steering vector file.
 */

  for (i = 0; i < beams; i++)
    {
      for (j = 0; j < el; j++)
       {
         fscanf (f_str, fmt, &str_vecs[i][j].real);
         fscanf (f_str, fmt, &str_vecs[i][j].imag);
       }
    }
  fclose (f_str);

/*
 * Read in the data 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 * rng * pricnt * el * sizeof (unsigned int), 0);
  fread (local_int_cube, sizeof (unsigned int), pricnt * rng * el, f_dat);
  fclose (f_dat);

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

```
  for (i = 0; i < rng; i++)
    {  /* for (i...) */
      for (j = 0; j < pricnt; j++)
       {  /* for (j...) */
         for (k = 0; k < el; k++)
           {  /* for (k...) */
             temp[0] = local_int_cube[i][j][k];
             temp1 = 0x0000FFFF & temp[0];
             temp1 = (temp1 & 0x00008000) ? temp1 | 0xffff0000 : temp1;
             temp2 = (temp[0] >> 16) & 0x0000FFFF;
             temp2 = (temp2 & 0x00008000) ? temp2 | 0xffff0000 : temp2;
             fft_out[j][i][k].real = (float) temp1;
             fft_out[j][i][k].imag = (float) temp2;
           }  /* for (k...) */
       }  /* for (j...) */
    }  /* for (i...) */
  return;
}
```

A.9 step1_beams.c

```
/*
 * step1_beams.c
 */

/*
 * This file contains the procedure step1_beams (), and is part of the
 * parallel APT 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 APT benchmark program was originally written by Tony Adams
 * on 7/12/93.
 *
 * The procedure step1_beams () performs the first step adaptive beamforming
 * algorithm. In this algorithm, we apply the Householder transform to all
 * rows of a matrix to get a lower triangular Cholesky. Then, we foward/back
 * solve with the steering vectors to get a set of weights to form the T
 * matrix.
 *
 * In the original sequential version of this procedure, forming the
 * Householder matrix was done here. However, in the parallel version, the
 * formation and broadcasting of the Householder matrix was moved into
 * main ().
 */

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

/*
 * step1_beams ()
 *   inputs: str_vecs, partial_temp
 *   outputs: partial_temp
 *
 *   str_vecs: steering vectors used to solve for the T matrix
 *   partial_temp: the Householder matrix
 */

step1_beams (str_vecs, partial_temp)
     COMPLEX str_vecs[][DOF];
```

```
    COMPLEX partial_temp[RNG_S][EL];

{

/*
 * Variables
 *
 * temp_mat: MAX temporary matrix holding various data
 * el: the maximum number of elements in each PRI
 * beams: the maximum number of beams
 * rng: the maximum number of range gates in each PRI
 * num_cols: the number of samples in the first step beamforming matrix
 * i, k: loop counters
 * lower_triangular_rows: the number of rows to be lower triangularized
 * num_rows: the number of rows on which the Householder transform is applied
 * start_row: the row number on which to start the Householder transform
 * weight_vec: storage for the weight solution vectors
 * make_t: 1 = generate T matrix from the solution weight vectors; 0 = don't
 *   generate the T matrix, but pass back the weight vector
 * numtask: the total number of tasks running this program
 * taskid: an identifier for this task
 * fp1, fp2: pointers to files; used for diagnostic purposes
 * e, c: loop counters; used for diagnostic purposes
 */

  static COMPLEX temp_mat[EL][COLS];
  int el = EL;
  int beams = BEAM;
  int rng = RNG;
  int num_cols = RNG_S;
  int i, k;
  int lower_triangular_rows;
  int num_rows;
  int start_row;
  COMPLEX weight_vec[DOF];
  int make_t;

  int numtask, taskid;
  FILE *fp1, *fp2;
  int e, c;

/*
 * Begin function body: step1_beams ()
 *
 * Rearrange partial_temp, and store into temp_mat.
 */

  for (i = 0; i < 320; i++)
    {
      for (k = 0; k < el; k++)
       {
         temp_mat[k][i].real = partial_temp[i][k].real;
         temp_mat[k][i].imag = partial_temp[i][k].imag;
       }
    }

/*
 * Save temp_mat for diagnostic purposes.
 */

/*
 *  mpc_environ (&numtask, &taskid);
 *  if (taskid == 0)
 *    {
 *      printf ("  saving temp_mat...\n");
 *    }
 *  if ((fp1 = fopen ("/scratch1/masa/temp_mat_par.real", "w")) == NULL)
 *    {
 *      printf ("Error - unable to open /scratch1/masa/temp_mat_par.real.\n");
```

```
*       exit (-1);
*     }
*   if ((fp2 = fopen ("/scratch1/masa/temp_mat_par.imag", "w")) == NULL)
*     {
*       printf ("Error - unable to open /scratch1/masa/temp_mat_par.imag.\n");
*       exit (-1);
*     }
*
*   for (e = 0; e < EL; e++)
*     {
*       for (c = 0; c < 320; c++)
*      {
*        fwrite (&(temp_mat[e][c].real), sizeof (float), 1, fp1);
*        fwrite (&(temp_mat[e][c].imag), sizeof (float), 1, fp2);
*      }
*     }
*
*   fclose (fp1);
*   fclose (fp2);
*/

/*
 * Perform Householder transform.
 */

  start_row = 0;  /* Start Householder on 1st row */
  num_rows = el;  /* Do Householder on all rows */
  lower_triangular_rows = el;  /* Lower triangularize all rows */

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

/*
 * Save temp_mat for diagnostic purposes.
 */

/*
 *  mpc_environ (&numtask, &taskid);
 *  if (taskid == 0)
 *    {
 *      printf ("  saving temp_mat2...\n");
 *    }
 *  if ((fp1 = fopen ("/scratch1/masa/temp_mat2_par.real", "w")) == NULL)
 *    {
 *      printf ("Error - unable to open /scratch1/masa/temp_mat2_par.real.\n");
 *      exit (-1);
 *    }
 *  if ((fp2 = fopen ("/scratch1/masa/temp_mat2_par.imag", "w")) == NULL)
 *    {
 *      printf ("Error - unable to open /scratch1/masa/temp_mat2_par.imag.\n");
 *      exit (-1);
 *    }
 *
 *  for (e = 0; e < EL; e++)
 *    {
 *      for (c = 0; c < 320; c++)
 *     {
 *       fwrite (&(temp_mat[e][c].real), sizeof (float), 1, fp1);
 *       fwrite (&(temp_mat[e][c].imag), sizeof (float), 1, fp2);
 *     }
 *    }
 *
 *  fclose (fp1);
 *  fclose (fp2);
 */

/*
 * Now, temp_mat has lower triangular matrix as the first MxM rows. Set make_t
 * to 1 to generate the T matrix.
 */
```

```
  make_t = 1;

/*
 * Call forback to solve M times with a different steering vector, putting
 * the Hermitian of the solution weight vectors into the rows of the T matrix.
 */

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

/*
 * The T matrix will have N main beams in 1st N rows, and A auxiliary beams in
 * the remaining A rows.
 */

  return;
}
```

A.10 step2_beams.c

```
/*
 * step2_beams.c
 */

/*
 * This file contains the procedure step2_beams (), and is part of the parallel
 * APT 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 APT benchmark program was originally written by Tony Adams
 * on 7/12/93.
 *
 * The procedure step2_beams () performs the second adaptive beamforming
 * step. This parallel version has been modified for parallel execution in
 * that each task performs the beamforming algorithm on only its slice of
 * the data cube. Other than that, the algorithm has been left alone. Also,
 * most, if not all, of the comments in this file are taken directly from
 * the sequential version.
 *
 * For each of the dopplers:
 *
 *   1. Does T_matrix * doppler (BEAM number of beams by RNG range gates).
 *      Nominally, BEAM = 32.
 *   2. For each of NUMSEG segments (of RNGSEG range gates each):
 *      Nominally, NUMSEG = 7, and RNGSEG = 40.
 *      a. Finds the maximum power in all beams, other than the main beams.
 *         (First MBEAM beams are main beams)
 *      b. Selects PWR_BM most power beams and then appends MBEAM main beams
 *         to the PWR_BM making (nominally 9+12=21) PWR_BM + MBEAM beams.
 *      c. Does a Householder on the PWR_BM + MBEAM beams beams but lower
 *         triangularizing only the first PWR_BM beams.
 *      d. Then, for each main beam, lower triangularize it and append to the
 *         PWR_BM genearting a lower triangular MxM matrix (nominally M = 10).
 *      e. Forward and back solve each MxM matrix applying steering vectors
 *         for forward solutions. (Steering vectors have been modified by Tb
 *         (the maximum power beams of the T matrix)).
 *      f. Apply weights from (e.) above to input dopplers to get output data
 *         cube to be used for target detection. The output data cube is MBEAM
 *         beams by PRI dopplers by RNG range gates divided, in NUMSEG segments
 *         of RNGSEG range gates each.
 *
 * This routine requires 6 input parameters as follows:
```

```
 *   dopplers: number of PRIs in the input data
 *   beams: number of beams (rows) in the input data
 *   rng: number of range gates in the input data
 *   str_vecs: pointer to the steering vectors
 *   local_cube[][][]: input data cube
 *   detect_cube[][][]: detection data cube
 */

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

extern int taskid;

/*
 * step2_beams ()
 *   inputs: dopplers, beams, rng, str_vecs, local_cube
 *   outputs: detect_cube
 *
 *   dopplers: number of PRIs in the input data
 *   beams: number of beams in the input data
 *   rng: number of range gates in the input data
 *   str_vecs: a pointer to the array of COMPLEX steering vectors
 *   local_cube: input data cube
 *   detect_cube: detection data cube
 */

step2_beams (dopplers, beams, rng, str_vecs, local_cube, detect_cube)
     int dopplers;
     int beams;
     int rng;
     COMPLEX str_vecs[][DOF];
     COMPLEX local_cube[][RNG][EL];
     COMPLEX detect_cube[][PRI / NN][RNG];

{

/*
 * Variables
 *
 * temp_mat: a temporary matrix holding various data
 * rng_seg: number of range gates per segment
 * main_beams: number of main beams in the input data
 * aux_beams: number of auxiliary beams in the input data
 * max_pwr_beams: number of max pwr beams used in Cholesky
 * num_seg: number of range segments for Cholesky
 * el: number of elements in the steering vectors
 * num_rows: numbe rof rows on which to apply the Cholesky
 * start_row: the row number on which to start applying the Cholesky
 * lower_triangular_rows: number of rows to lower triangularize
 * i, j, k, m, n: loop counters
 * row: row number
 * col: column number
 * appl: column number for application data
 * dop: loop counter for doppler number
 * seg_col: column number for correct 1 of 7 segments
 * next_seg: start column number for next 1 of 7 segments
 * indx: index desired column number
 * xreal: storage for real data for pointer use
 * ximag: storage for imag data for pointer use
 * weight_vec: storage for weight solution vector
 * mod_str_vec: storage for one modified steering vector
 * x_beams: hold each doppler for step 2 beamforming
 * make_t: flag to make or not make T matrix. If make_t is set to 0, T
 *   matrix is not made and the weight vector is returned to the caller
 * real_ptr: pointer to next real data
 * imag_ptr: pointer to next imaginary data
```

```
 * max_sum: max sum of the ABEAM auxiliary beams: nominally ABEAM = 20.
 * pwr_sums: pwr sums of ABEAM auxiliary beams
 * row_indx: row index of pwr sums of SBEAM aux beams
 * max_indx: row index of PWR_BM max pwr beams
 * sum: COMPLEX variable to hold sum of complex data
 */

  static COMPLEX temp_mat[BEAM][COLS];
  int rng_seg = RNGSEG;
  int main_beams = MBEAM;
  int aux_beams = ABEAM;
  int max_pwr_beams = PWR_BM;
  int num_seg = NUMSEG;
  int el = EL;
  int num_rows;
  int start_row;
  int lower_triangular_rows;
  int i, j, k, m, n;
  int row;
  int col;
  int appl;
  int dop;
  int seg_col;
  int next_seg;
  int indx;
  float xreal[EL];
  float ximag[EL];
  COMPLEX weight_vec[DOF];
  COMPLEX mod_str_vec[1][DOF];
  COMPLEX x_beams[EL][RNG_S];
  int make_t;
  float *real_ptr;
  float *imag_ptr;
  float max_sum;
  float pwr_sums[ABEAM];
  int row_indx[ABEAM];
  int max_indx[PWR_BM];
  COMPLEX sum;

  extern void house();
  extern void forback();

/*
 * Begin function body: step2_beams ()
 */

  for (dop = 0; dop < dopplers; dop++)
    {   /* for (dop...) */

/*
 * For each doppler, form beams by multiplying T_matrix by each doppler data
 * in the local_cube. MBEAM main beams are first then ABEAM auxiliary beams.
 * Nominally, MBEAM = 12 and ABEAM = 20. Perform T_matrix * doppler data and
 * store into the x_beams matrix.
 */

      for (j = 0; j < rng; j++)
       {  /* for (j...) */

/*
 * For each doppler move EL elements of each range gate into temporary
 * holding vectors to multiply all rows of T matrix using pointers.
 */

        real_ptr = xreal;
        imag_ptr = ximag;

        for (k = 0; k < beams; k++)
        {  /* for (k...) */
```

```
          *real_ptr++ = local_cube[dop][j][k].real;
          *imag_ptr++ = local_cube[dop][j][k].imag;
         }  /* for (k...) */

/*
 * Multiply 1 range gate of each doppler by all rows of T_matrix summing the
 * products of each row and storing the sums in a corresponding row of x_beams
 * doppler output matrix.
 */

        for (row = 0; row < beams; row++)
          {  /* for (row...) */
            sum.real = 0.0;
            sum.imag = 0.0;
            real_ptr = xreal;
            imag_ptr = ximag;
            for (col = 0; col < beams; col++)
             {  /* for (col...) */
               sum.real += (t_matrix[row][col].real * *real_ptr -
                            t_matrix[row][col].imag * *imag_ptr);
               sum.imag += (t_matrix[row][col].imag * *real_ptr++ +
                            t_matrix[row][col].real * *imag_ptr++);
             }  /* for (col...) */
            x_beams[row][j].real = sum.real;
            x_beams[row][j].imag = sum.imag;
          }   /* for (row...) */
      }   /* for (j...) */

/*
 * For each doppler we now have BEAM beams formed in x_beams matrix. Main
 * beams are the first MBEAM beams then ABEAM auxiliary beams. Divide RNG
 * range gates into NUMSEG segments of RNGSEG range gates. Process RNGSEG
 * range gates per secgment for each of the NUMSEG segments for both the
 * Cholesky data and application of weights data. The PWR_BM max pwr beams
 * are the first beams and following these beams are the MBEAM main beams.
 */

     for (j = 0; j < num_seg; j++)
      {  /* for (j...) */
        next_seg = j * rng_seg;

/*
 * Begin by moving MBEAM main beams to both Cholesky area and weights
 * application area of temp_mat as the MBEAM+1 through the MBEAM+PWR_BM
 * beams.
 */

        for (k = 0; k < main_beams; k++)
          {  /* for (k...) */
            for (col = 0; col < rng_seg; col++)
             {  /* for (col...) */
               row = max_pwr_beams + k;
               appl = col + rng_seg; /* application region RNGSEG    */
                                     /*   range gates after cholesky */
                                     /*   data region                */
               seg_col = next_seg + col;
                                     /* gets the data from the       */
                                     /*   correct range segments col */
                                     /*   which divides the range    */
                                     /*   gates                      */
               temp_mat[row][col].real = x_beams[k][seg_col].real;
               temp_mat[row][col].imag = x_beams[k][seg_col].imag;
               temp_mat[row][appl].real = x_beams[k][seg_col].real;
               temp_mat[row][appl].imag = x_beams[k][seg_col].imag;
             }   /* for (col...) */
          }  /* for (k...) */

/*
 * For each segment, find the pwr sums of the ABEAM auxiliary beams.
```

```
 */

        for (k = 0; k < aux_beams; k++)
          {  /* for (k...) */
            pwr_sums[k] = 0.0;
            row_indx[k] = main_beams + k;
            row = k + main_beams;
            for (col = 0; col < rng_seg; col++)
             {  /* for (col...) */
               seg_col = next_seg + col;
                                    /* gets the data from the       */
                                    /*   correct range segments col */
                                    /*   which divides the range    */
                                    /*   gates                      */
               pwr_sums[k] += (x_beams[row][seg_col].real *
                               x_beams[row][seg_col].real +
                               x_beams[row][seg_col].imag *
                               x_beams[row][seg_col].imag);
             }   /* for (col...) */
          }   /* for (k...)  */

/*
 * For each segment, find PWR_BM max pwr beams of the ABEAM aux beams. Sort
 * them by max pwr in max_indx vector.
 */

        for (k = 0; k < max_pwr_beams; k++)
          {  /* for (k...) */

/*
 * Init max_index[k] since pwr_sums may all be 0 or Not a Number.
 */

            max_indx[k] = main_beams + k;
            max_sum = 0.0;
            for (m = 0; m < aux_beams; m++)
             {  /* for (m...) */

/*
 * The > in the next if statement was changed to a >=
 */

               if (pwr_sums[m] >= max_sum)
                 {
                   max_sum = pwr_sums[m];
                   max_indx[k] = row_indx[m];
                   n = m;
                 }
             }  /* for (m...) */
            pwr_sums[n] = 0.0;
      }   /* for (k...) */

/*
 * For each segment, move PWR_BM max pwr beams to both Cholesky area and
 * weights application area of temp_mat as the first PWR_BM beams.
 */

        for (k = 0; k < max_pwr_beams; k++)
          {  /* for (k...) */
            indx = max_indx[k];       /* Indexes the correct max_pwr  */
                                      /*   beam row                   */
            for (col = 0; col < rng_seg; col++)
             {  /* for (col...) */
               appl = col + rng_seg; /* application region RNGSEG    */
                                     /*   range gates after cholesky */
                                     /*   data region                */
               seg_col = next_seg + col;
                                     /* gets the data from the       */
                                     /*   correct range segments col */
```

```
                                        /*   which divides the range    */
                                        /*   gates                      */
                temp_mat[k][col].real = x_beams[indx][seg_col].real;
                temp_mat[k][col].imag = x_beams[indx][seg_col].imag;
                temp_mat[k][appl].real = x_beams[indx][seg_col].real;
                temp_mat[k][appl].imag = x_beams[indx][seg_col].imag;
              }   /* for (col...) */
            }   /* for (k...) */

/*
 * In 2nd step adaptive beam forming we apply householder to all PWR_BM
 * + MBEAM rows but only lower triangularize the PWR_BM max_pwr_beams rows.
 */

        num_rows = max_pwr_beams + main_beams;
        lower_triangular_rows = max_pwr_beams;
        start_row = 0;                  /* start housholder on 1st row  */

/*
 * Call the second step adaptive beamforning Householder routine.
 */

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

/*
 * In addition call householder routine MBEAM more times with one of the main
 * beam rows moved to the PWR_BM+1 row in order to lower triangularize one
 * more row to make it a MxM lower triangular matrix. Then this will be
 * forward and back solved for weights. Nominally M = 10.
 */

        for (k = 0; k < main_beams; k++)
          {  /* for (k...) */

/*
 * Move a main beam row to PWR_BM+1 row except for the main beam 1 which is
 * already in the Mth row and doesn't have to be moved.
 */

            if (k)
             {  /* if (k) */
               row = k + max_pwr_beams;
                                        /* desired 1 of MBEAM main beam */
                                        /*   numbers                    */
               for (col = 0; col < rng_seg; col++)
                 {  /* for (col...) */
                   appl = col + rng_seg;
                                        /* application region RNGSEG    */
                                        /*   range gates after cholesky */
                                        /*   data region                */
                   temp_mat[max_pwr_beams][col].real
                    = temp_mat[row][col].real;
                   temp_mat[max_pwr_beams][col].imag
                    = temp_mat[row][col].imag;
                   temp_mat[max_pwr_beams][appl].real
                    = temp_mat[row][appl].real;
                   temp_mat[max_pwr_beams][appl].imag
                    = temp_mat[row][appl].imag;
                 }  /* for (col...) */
             }  /* if (k) */

            num_rows = max_pwr_beams + 1;
                                        /* Allows 1 row to be processed */
                                        /*   in house()                 */
            lower_triangular_rows = max_pwr_beams + 1;
            start_row = max_pwr_beams;
                                        /* start housholder on PWR_BM + */
                                        /*   1 (last) row. This does    */
```

```
                                        /*   Householder on only 1 row  */
/*
 * Call second step adaptive beamforming Householder routine.
 */

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

/*
 * The temp_mat matrix now has a lower triangular matrix as the first MxM
 * rows. Modify the main beam steering vector by the max pwr beam rows of the
 * T_matrix plus 1 selected main beam row to get a 1xM modified steering
 * vector. Move EL elements of each selected main beam steering vector into
 * a temporary holding vector to multiply all max pwr rows of T matrix using
 * pointers.
 */

            real_ptr = xreal;
            imag_ptr = ximag;
            for (n = 0; n < el; n++)
             {  /* for (n...) */
               *real_ptr++ = str_vecs[k][n].real;
               *imag_ptr++ = str_vecs[k][n].imag;
             }  /* for (n...) */

/*
 * Multiply the steering vector by all max pwr beams of the T_matrix.
 */

            for (m = 0; m < max_pwr_beams; m++)
             {  /* for (m...) */
               sum.real = 0.0;
               sum.imag = 0.0;
               real_ptr = xreal;
               imag_ptr = ximag;
               row = max_indx[m];
               for (col = 0; col < el; col++)
                 {  /* for (col...) */
                   sum.real += (t_matrix[row][col].real * *real_ptr -
                               t_matrix[row][col].imag * *imag_ptr);
                   sum.imag += (t_matrix[row][col].imag * *real_ptr++ +
                               t_matrix[row][col].real * *imag_ptr++);
                 }  /* for (col...) */

/*
 * Put the modified steering vector in slot 0 of the mod_str_vec matrix since
 * only one forward back solution is done in the forback routine.
 */

               mod_str_vec[0][m].real = sum.real;
               mod_str_vec[0][m].imag = sum.imag;
             }  /* for (m...) */

/*
 * One more row to multiply. Multiply the steering vector by the selected
 * main beam row of the T_matrix.
 */

            sum.real = 0.0;
            sum.imag = 0.0;
            real_ptr = xreal;
            imag_ptr = ximag;
            row = k;  /* T matrix row = selected main beam row */
            for (col = 0; col < el; col++)
             {  /* for (col...) */
               sum.real += (t_matrix[row][col].real * *real_ptr -
                           t_matrix[row][col].imag * *imag_ptr);
               sum.imag += (t_matrix[row][col].imag * *real_ptr++ +
```

```
                        t_matrix[row][col].real * *imag_ptr++);
                } /* for (col...) */

                mod_str_vec[0][max_pwr_beams].real = sum.real;
                mod_str_vec[0][max_pwr_beams].imag = sum.imag;

/*
 * Set make_t = 0 to NOT generate a T matrix. Do only one forward back
 * solution and pass back the solution weight vector.
 */

                make_t = 0;

/*
 * Call forback to solve one weight vector with modified steering vector.
 */

                forback ( num_rows, mod_str_vec, weight_vec, make_t, temp_mat);

/*
 * Multiply the conjugate of the weight vector with the RNGSEG application
 * data range gates for each of the MBEAM main beams. Store the results in
 * the output detection cube. Use temp storage for the weight vector with
 * pointers to speed multiply.
 */

                real_ptr = xreal;
                imag_ptr = ximag;
                for (n = 0; n < max_pwr_beams + 1; n++)
                { /* for (n...) */

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

                  *real_ptr++ = weight_vec[n].real;
                  *imag_ptr++ = - weight_vec[n].imag;
                } /* for (n...) */

/*
 * Multiply the weight vector by all range gates of the application data set.
 * The application data set is from RNGSEG column to RNGSEG+RNGSEG columns
 * of "temp_mat" matrix.
 */

                for (m = rng_seg; m < rng_seg + rng_seg; m++)
                { /* for (m...) */
                  sum.real = 0.0;
                  sum.imag = 0.0;
                  real_ptr = xreal;
                  imag_ptr = ximag;
                  for (row = 0; row < max_pwr_beams + 1; row++)
                    { /* for (row...) */
                      sum.real += (temp_mat[row][m].real * *real_ptr -
                                   temp_mat[row][m].imag * *imag_ptr);
                      sum.imag += (temp_mat[row][m].imag * *real_ptr++ +
                                   temp_mat[row][m].real * *imag_ptr++);
                    } /* for (row...) */

/*
 * Store the result into the detection data cube.
 */

                  col = j * rng_seg + m - rng_seg;
                  detect_cube[k][dop][col].real = sum.real;
                  detect_cube[k][dop][col].imag = sum.imag;
                } /* for (m...) */
              } /* for (k...) */
            } /* for (j...) */
```

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

A.11 defs.h

```
/* defs.h */

#define NN 64

#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;}

#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;

#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)/60.0)
#define SYSTIME(T1,T2)  ((t2.tms_stime-t1.tms_stime)/60.0)
#define USERTIME1(T1,T2)  ((time_end.tms_utime-time_start.tms_utime)/60.0)
#define SYSTIME1(T1,T2)  ((time_end.tms_stime-time_start.tms_stime)/60.0)
#define USERTIME2(T1,T2)  ((end_time.tms_utime-start_time.tms_utime)/60.0)
#define SYSTIME2(T1,T2)  ((end_time.tms_stime-start_time.tms_stime)/60.0)

/* #define LINE_MAX 256 */
#define TRUE 1
#define FALSE 0

/* The following default dimensions are MAXIMUM values. The actual  */
/* dimension for PRIs will be the 1st entry in file "testcase.str". */
/* The 2nd entry in the file is the Target Detection Threshold.     */
/* The rest of the file will contain steering vectors starting with */
/* the main beams then all auxiliary beams. The file "testcase.hdr" */
/* will have descriptions  of entries in "testcase.str" file. Also  */
/* a description of the data file "testcase.dat" will be given. This*/
/* file contains data to fill the input "data_cube[][][]" array.    */

#define COLS 1500  /* 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 1500
```

```
#define V4 64
#define DIM1 64     /* Number for dimension1 in input data cube */
#define DIM2 128    /* Number for dimension2 in input data cube */
#define DIM3 1500   /* 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 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 */

#define PRI_CHUNK PRI/NN

#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 */


void xu_target ( in1, in2, out, len )
float in1[][4], in2[][4], out[][4];
int *len ;
{
  int i,i1,i2, n ;

  i1=i2=0;
  for (i=0; i<25; i++) { out[i][0] = 0.0; }

  for (i=0; i<25; i++)
    {
      if (in1[i1][3] != 0.0 && (in2[i2][3]==0.0 || in1[i1][0] < in2[i2][0]))
        {
          out[i][0] = in1[i1][0] ;
          out[i][1] = in1[i1][1] ;
```

```
            out[i][2] = in1[i1][2] ;
            out[i][3] = in1[i1][3] ;
           i1++;
          }
        else if
         (in2[i2][3] != 0.0 && (in1[i1][3]==0.0 || in1[i1][0] > in2[i2][0]))
          {
            out[i][0] = in2[i2][0] ;
            out[i][1] = in2[i2][1] ;
            out[i][2] = in2[i2][2] ;
            out[i][3] = in2[i2][3] ;
           i2++;
          }
        else break ;
      }
}
```

A.12 compile_apt

```
mpcc -qarch=pwr2 -O3 -DAPT -DIBM -o apt bench_mark_APT.c cell_avg_cfar.c
  forback.c cmd_line.c house.c read_input_APT.c step1_beams.c step2_beams.c -lm
```

A.13 run.256

```
poe apt /scratch2/zxu/new_data -procs 256
```