The CMU Task Parallel Program Suite

Peter Dinda Thomas Gross David O'Hallaron
Edward Segall James Stichnoth Jaspal Subhlok
Jon Webb Bwolen Yang

March 1994
CMU-CS-94-131

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The idea of exploiting both task and data parallelism in programs is appealing. However, identifying realistic, yet manageable example programs that can benefit from such a mix of task and data parallelism is a major problem for researchers. We address this problem by describing a suite of five applications from the domains of scientific, signal, and image processing that are of reasonable size, are representative of real codes, and can benefit from exploiting task and data parallelism. The suite includes fast Fourier transforms, narrowband tracking radar, multibaseline stereo imaging, and airshed simulation. Complete source code for each example program is available from the authors.

This research was sponsored by the Advanced Research Projects Agency/CSTO monitored by SPAWAR under contract N00039-93-C-0152, and also by the Air Force Office of Scientific Research under contract F49620–92–J–0131. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either express or implied, of the U.S. government.

Contact author: David O'Hallaron, droh@cs.cmu.edu.

Keywords: Parallel programming, task parallelism, functional parallelism, parallelizing compilers, program mapping

1. Introduction

There is growing interest in the idea of exploiting both task and data parallelism [1, 3, 4, 5, 6, 7, 19]. There are a number of practical reasons for this interest. For many applications, especially in the domains of image and signal processing, data set sizes are limited by physical constraints and cannot be easily increased [19]. In such cases the amount of available data parallelism is limited. For example, in the multibaseline stereo application described later in this report, the size of an image is determined by the circuitry of the video cameras and the throughput of the camera interface. Increasing the image size means buying new cameras and building a faster interface, which may not be feasible. Since the data parallelism is limited, additional parallelism must come from tasking.

Another reason for the increased interest in task parallelism is that simulations are becoming increasingly sophisticated as they attempt to capture interactions among different physical phenomena. The phenomena might represent different scientific disciplines, and different parts of the simulation might even be written by different groups. For example, the airshed model described later in this report characterizes the formation of air pollution as the interaction between the wind blowing and reactions among various chemical species. It is natural to model such interactions using tasking, where one task models the wind blowing, and the other task models the chemical reactions. Further, if the codes are written by different groups, task parallelism may be the only feasible way to integrate the codes.

Applications that can benefit from a mix of task and data parallelism tend to be somewhat complicated because they are typically composed of a collection of nontrivial functions, each of which is a sequence of data parallel operations. Identifying and building representative example programs that are a manageable size is a major stumbling block for computer science researchers who are not application domain experts. We address this problem by describing a set of realistic example programs from the domains of scientific, signal, and image processing that can benefit from a mix of task and data parallelism:

1. 1D fast Fourier transform.
2. 2D fast Fourier transform.
3. Narrowband tracking radar.
4. Multibaseline stereo.
5. Airshed simulation.

We identified these applications in the course of developing an integrated model of task and data parallelism for the Fx parallelizing Fortran compiler [9, 19, 17, 18, 23] and have found them to be extremely helpful.

Complete Fortran 77 sources of the programs are available from the authors. Each program is fewer than 500 lines of code. The Fortran 77 sources are useful for a number of reasons. First, the sources provide an unambiguous specification of each application, including input and output data sets. Second, there are many models and dialects for task parallelism. Fortran 77 represents a lowest common denominator of sorts, available to everyone, for describing the applications. Finally, the source code clearly identifies the obvious sources of course-grained parallelism in the form of calls to subroutines. This enabled us, in all but one case, to port the Fortran 77 programs to the Fx system with only minor modifications (the only exception being the airshed simulation, which requires a dynamic model of task parallelism that Fx does not currently support). We invite other researchers who are interested in task parallelism to use these Fortran 77 sources as the basis for writing the applications in their favorite task parallel dialect.

Section 2 briefly outlines a simple space of the different ways that task and data parallelism can be used in the same program. Sections 2–6 describe the example programs and discuss briefly how they can mapped onto a parallel system using a mix of task and data parallelism. Section 7 describes where to find the online Fortran 77 codes and provides some more detail on their structure.

2. Combinations of task and data parallelism

Each program in our task parallel suite has the following form:

```
do i = 1,m
   call T1()
   call T2()
   call T3()
enddo
```

There is an outer loop that iterates over m input data sets. The body of the loop consists of calls to three *task-subroutines*. Each task-subroutine typically consists of a sequence of data-parallel statements (e.g., array assignment statements or DOALL-like parallel loops) that can run on multiple processors. Depending on the application, there may or may not be dependences across iterations of the outer loop and among the task-subroutines. We can graphically represent the program using a task graph, as shown in Figure 1. Each node corresponds to a task-subroutine, and each arc corresponds to a data dependence between a pair of task-subroutines.

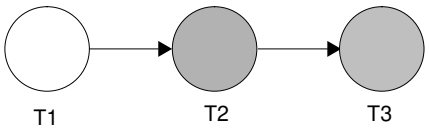

Figure 1: Example task graph

We can view different task and data parallel mappings for an application as points in a two-dimensional space [18]. The first axis corresponds to different *clusterings* of task-subroutines, where each task-subroutine in cluster runs on the same set of processors, and each cluster runs on a unique set of processors. In some cases, assigning two task subroutines to the same cluster reduces communication overhead, at the cost of reduced parallelism. In these cases, there is a tension between reducing communication cost and increasing parallelism.

If the input data sets for a cluster are independent, then that cluster can be *replicated*. For example, one replicated instance of a cluster could process even numbered data sets and the other replicated instance could process the odd numbered data sets. The degree of replication is captured by the second axis. Replication is beneficial for task-subroutines that do not scale well.

Some examples of different combinations of task and data parallelism are shown in Figure 2. Figure 2(a) shows the usual data parallel mapping where all task-subroutines are clustered onto all processors, with no replication. Figure 2(b) shows a task parallel mapping where each task-subroutine gets its own cluster, again with no replication. It may be desirable in some cases to replicate the data-parallel clusters, as in Figure 2(c). Finally, a mix of task and data parallelism, using both clustering and replication, is shown in Figure 2(d).

3. Fast Fourier transform

The fast Fourier transform (FFT) converts an input data set from the temporal/spatial domain to the frequency domain, and vice versa. While it is an important algorithm in its own right, with numerous applications in scientific, signal, and image processing, it is an especially interesting example program for studying task parallelism because it exemplifies a common computational pattern, i.e., manipulate the data

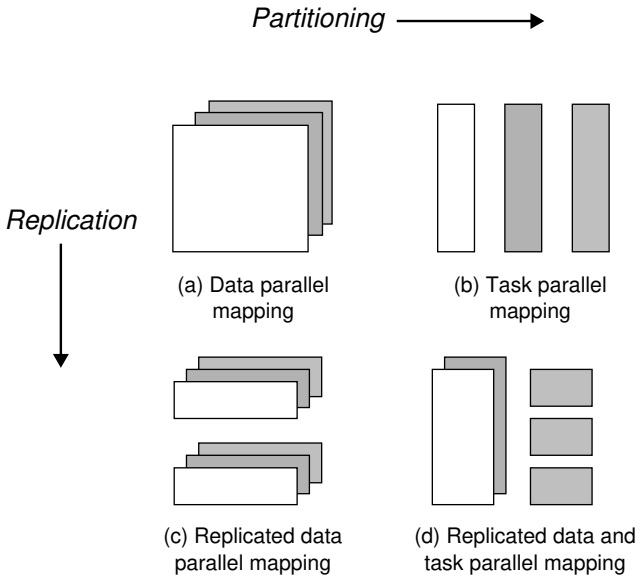

Figure 2: Combinations of task and data parallel mappings

row-wise, then manipulate the data column-wise. This pattern appears in diverse applications such as medical imaging [13], synthetic aperture radar imaging [15], ADI solvers, sonar beamforming, and the radar and airshed programs described later in this document.

More precisely, a discrete Fourier transform (DFT) is a complex matrix-vector product

$$y = F_n x$$

where x and y are complex vectors, and $F_n = (f_{pq})$ is an $n \times n$ matrix such that $f_{pq} = \omega_n^{pq}$, where

$$\omega_n = \cos(2\pi/n) - i\sin(2\pi/n) = e^{-2\pi i/n}$$

and $i = \sqrt{-1}$. A fast Fourier transform (FFT) is an efficient procedure for computing the DFT that exploits structure in F_n to compute the DFT matrix-vector product in $O(n \log n)$ time. Higher dimensional FFTs are also defined. In general, an m-dimensional FFT operates on an m-dimensional matrix with n elements in each dimension in $O(n^m \log n)$ time. See [20] for an excellent description of the numerous FFT algorithms that have been developed over the past 40 years.

3.1. 1D fast Fourier transform

If $n = n_1 n_2$, then a 1D FFT can be computed using a collection of smaller independent 1D FFTs [2, 8, 20]. Starting with a view of the input vector x as an $n_1 \times n_2$ array A, stored in column major order, perform n_1 independent n_2-point FFTs on the rows of A, scale each element a_{pq} of the resulting array by a factor of ω_n^{pq}, and then perform n_2 independent n_1-point FFTs on the columns of A. The final result vector y is obtained by concatenating the rows of the resulting array. Figure 3 shows the task graph for the 1D FFT.

Input and output are sequences of vectors reshaped as 2D arrays. Nodes labeled `trans` perform a transpose operation, nodes labeled `col FFTs` perform a set of 1D FFTs on the columns of its input array, and the node labeled `scale` multiplies each element of its input array by a constant. To exploit locality in the memory subsystem, the program implements each set of row-wise operations as a transpose followed by a set of column-wise operations. This specific order is an artifact of the fact that Fortran 77 stores matrices

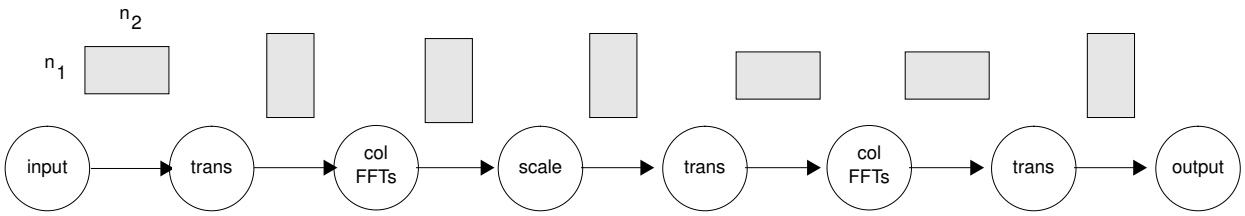

Figure 3: 1D FFT task graph for one input vector

in column-major order. If the example program were written in C, each column-wise operation would be implemented as a transpose followed by a set of row-wise operations.

Mapping the 1D FFT onto a parallel system is easy in some ways and challenging in other ways. The problem is easy in the sense that the column-wise FFTs and the scaling operation are perfectly parallel and easily represented by conventional data parallel constructs. The column-wise FFTs are naturally expressed with DOALL-like parallel loop construct such as the HPF independent DO statement [10]. The scaling operation can be expressed with a Fortran 90 array assignment statement. However, the problem is challenging because the transpose operation requires an efficient redistribution of data, usually in the form of a complete exchange where each processor must send data to every other processor. Since the input arrays are independent, both replication and clustering of the task graph are possible.

3.2. 2D fast Fourier transform

Computing the 2D FFT is similar to computing the 1D FFT. Given an $n_1 \times n_2$ input array A, perform n_2 independent n_1-point FFTs on the columns of A, followed by n_1 independent n_2-point 1D FFTs on the rows of A. Figure 4 shows the task graph for the 2D FFT. As with the 1D FFT, row-wise FFTs are replaced by a transpose followed by a set of column-wise FFTs. Notice that the 2D FFT is simpler than the 1D FFT. No scaling is required and there is one fewer transpose operation. Again, the input and output are sequences of arrays, and since these arrays are independent, both replication and clustering of the task graph are possible.

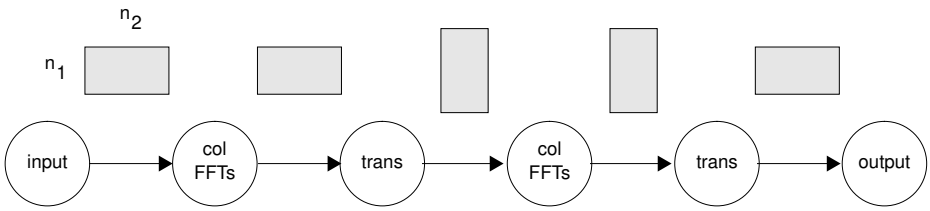

Figure 4: 2D FFT task graph for one input array

4. Narrowband tracking radar

The narrowband tracking radar benchmark was developed by researchers at MIT Lincoln Laboratories to measure the effectiveness of various multicomputers for their radar applications [16]. It is an interesting program for studying task parallelism because of its hard real-time requirements, and because the size of the input data set is limited by physical properties of the radar sensor. The task graph for the radar application is shown in Figure 5.

The program inputs data from a single sensor along $c = 4$ independent *channels*. Every 5 milliseconds, for each channel, the program receives $d = 512$ complex vectors of length $r = 10$, one after the other in the form of an $r \times d$ complex array A (assuming the column major ordering of Fortran). At a high-level,

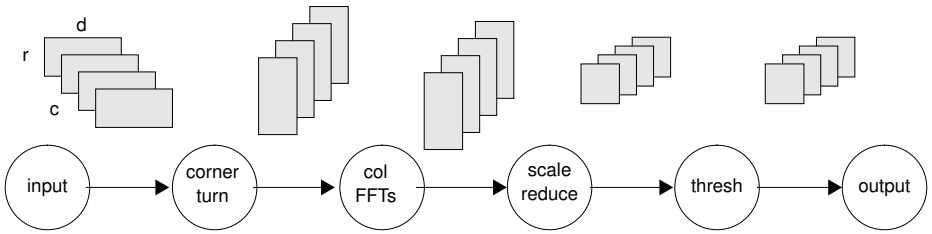

Figure 5: Radar task graph for one input array

each input array A is processed in the following way: (1) *Corner turn* the $r \times d$ input array to form a $d \times r$ array. (2) Perform r independent d-point FFTs. (3) Convert the resulting complex $d \times r$ array to a real $w \times r$ subarray, $w = 40$, by replacing each element $a + ib$ in the $w \times r$ subarray with its scaled magnitude $\sqrt{a^2 + b^2}/d$. (4) Threshold each element a_{jk} of the subarray using a cutoff that is a function of a_{jk} and the sum of the subarray elements. Elements that are above the threshold are set to unity; elements below the threshold are set to zero.

The corner turn operation is equivalent to a transpose, so it can potentially induce a complete exchange where each processor communicates with every other processor. As with the 1D FFT, the column FFTs, scaling, and thresholding operations can be naturally expressed using conventional data parallel constructs. Further, the reduction operation requires an efficient reduction mechanism. However, the most interesting computational property of the radar benchmark is the fact that the size parameters r, d, c, and w are determined by mother nature and the properties of current sensor technology. The luxury of simply increasing the data set size simply does not exist in this case. The amount of available low-level data parallelism is limited, so additional parallelism must come from higher-level task parallelism. Like the FFT examples, input data sets are independent, so both replication and clustering of the task graph are possible.

5. Multibaseline stereo

The multibaseline stereo uses an algorithm developed at Carnegie Mellon that gives greater accuracy in depth through the use of more than two cameras [14]. It is an interesting program for studying task parallelism because it contains significant amounts of both inter-task and intra-task communication[22], and because, like the radar example, the size of the input data sets cannot be easily increased. Our implementation is adapted from a previous data-parallel implementation written in a specialized image processing language [21].

Figure 6 shows the task graph for the stereo program. Input consists of three $m \times n$ images acquired from three horizontally aligned, equally spaced cameras. One image is the *reference image*, the other two are *match images*. For each of 16 disparities, $d = 0, \ldots, 15$, the first match image is shifted by d pixels, the second image is shifted by $2d$ pixels. A *difference image* is formed by computing the sum of squared differences between the corresponding pixels of the reference image and the shifted match images. Next, an *error image* is formed by replacing each pixel in the difference image with the sum of the pixels in a surrounding 13×13 window. A *disparity image* is then formed by finding, for each pixel, the disparity that minimizes error. Finally, the depth of each pixel is displayed as a simple function of its disparity.

The stereo program requires efficient mechanisms for broadcasting and reducing large data sets. The computation of the difference images requires simple pointwise operations on the three input images and can thus be naturally expressed with Fortran 90 array statements. The computation of the error images is somewhat more interesting, being similar to a convolution operation. The convolution can be modeled as a DOALL where the loop iterations operate on overlapping regions of the image, which means that processors must communicate before the loop iterations can begin executing. As with the FFT and radar programs, the data sets are independent, so both replication and clustering of the task graph are possible.

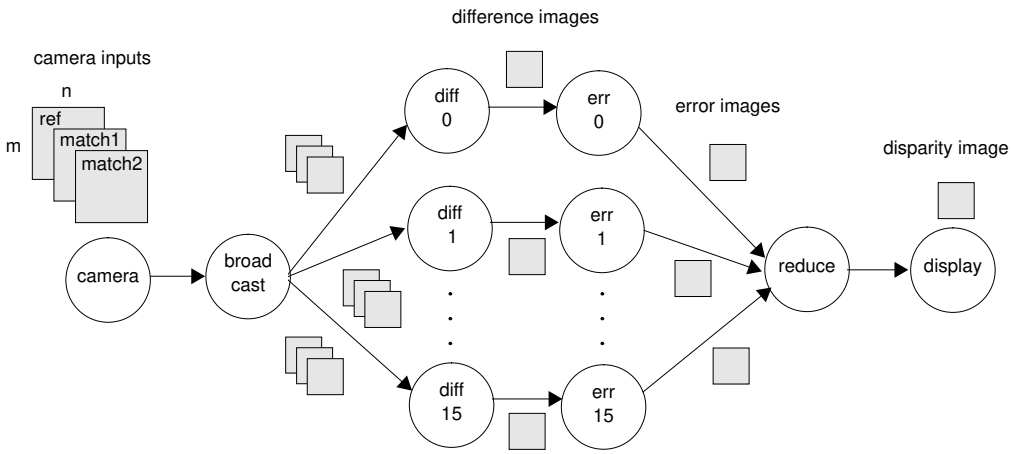

Figure 6: Multibaseline stereo task graph for one input data set

6. Airshed simulation

The airshed simulation is significantly more complex than the previous examples. The multiscale airshed model captures the formation, reaction, and transport of atmospheric pollutants and related chemical species [11, 12]. It is an interesting application because it requires a dynamic task parallel model, and because different parts of the application exhibit widely varying amounts of DOALL parallelism.

The airshed application simulates the behavior of the airshed model when it is applied to s chemical species, distributed over domains containing p grid points in each of l atmospheric layers. Typical values are $s = 35$ species, $500 \leq p \leq 5000$ grid points, and $l = 5$ atmospheric layers. Because of the multiscale grid, the entire northeastern United States can be modeled with problems in this size range. A total of about 200 chemical reactions are modeled.

The program computes in two principle phases: (1) horizontal transport (using a finite element method with repeated application of a direct solver), followed by (2) chemistry/vertical transport (using an iterative, predictor-corrector method). Figure 7 depicts the task graph for one hour of simulated time. Input is an $l \times s \times p$ concentration array. Initial conditions are input from disk (`inputhour`), and in a preprocessing phase for the horizontal transport phases to follow, the finite element stiffness matrix for each layer is assembled and factored (`pretrans`). The atmospheric conditions captured by the stiffness matrix are assumed to be constant during the simulated hour, so this step is performed just once per hour. This is followed by a sequence of steps — the number of steps is one of the initial conditions — where each step consists of a horizontal transport phase, followed by a chemistry/vertical transport phase, followed by another horizontal transport phase. Each horizontal transport phase performs ls backsolves, one for each layer and species. All may be computed independently; however, for each layer l, all backsolves use the same factored matrix A_l. The chemistry/vertical transport phase performs an independent computation for each of the p grid points. Output for the hour is an updated concentration array, which is then input to the next hour.

A number of interesting issues arise when we map the airshed to a parallel system. In the other example programs we have discussed, the number of tasks is known at compile time. However, in the airshed program, the number of transport/chemistry/transport steps for each hour is not known until runtime, which implies a dynamic model of task parallelism. Also, since the output concentration array of one hour is the input to the next hour, replication of the task graph is not feasible, as it was with the previous example programs.

Another issue is that the preprocessing phase, the transport phase, and the chemistry phase have very different levels of obvious DOALL parallelism because the sizes of the different dimensions of the concentration array differ by orders of magnitude. For example, the preprocessing phase independently computes

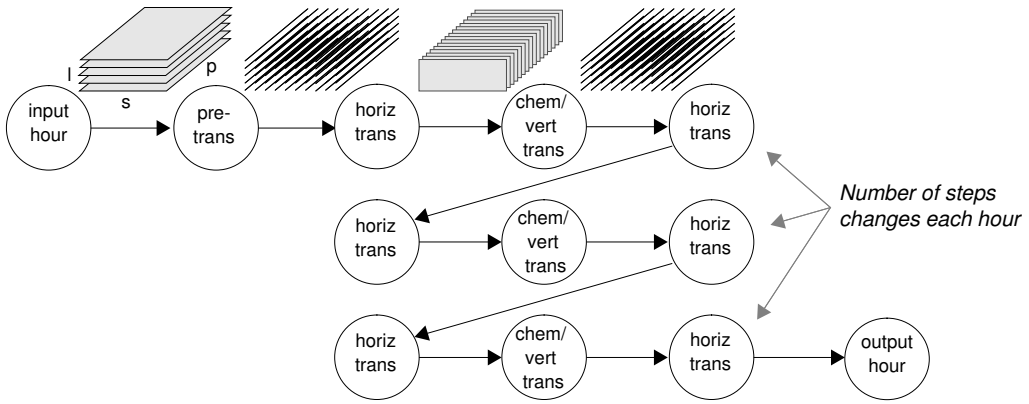

Figure 7: Task graph for one hour of the airshed simulation

stiffness matrices for each layer; unfortunately there are only 5 layers, so the obvious DOALL approach will use 5 processors. To get better utilization, we must parallelize the computation for each layer, or we must try to employ task parallelism to pipeline the computation for each layer, or both.

The issues involved in mapping the transport phase are particularly interesting. Since there are $l = 5$ layers and $s = 35$ species, the transport phase could be easily implemented with doubly nested DOALLs that consist of 175 independent loop iterations. For moderate sized parallel systems, with say 64 processors, this approach might work well. However, for larger systems, with say 512 processors, this approach uses only a fraction of the processors. As with the preprocessing phase, we can get better utilization by either parallelizing the sparse finite element computation (a difficult task) or trying to use task parallelism to pipeline the computation.

The final issue stems from the fact that the preprocessing, horizontal transport, and chemistry/vertical transport phases each operate on different dimensions of the concentration array. To exploit locality in the memory hierarchy, an implementation will most likely insert the appropriate transpose operation before each phase. On a parallel system, this can induce a complete exchange where each processor communicates with every other processor. Again, as with the FFT and radar examples, we see the need for an efficient complete exchange mechanism.

7. Distribution

Complete Fortran 77 sources for the application described in this report are available via anonymous FTP from `warp.cs.cmu.edu` in file `fx-codes/tpsuite/tpsuite.tar`. World Wide Web clients like Mosaic and Cello can use the following URL:

```
ftp://warp.cs.cmu.edu/usr/anon/fx-codes/tpsuite/tpsuite.tar
```

Each source program is a vanilla Fortran 77 code that operates on a sequence of inputs and produces a sequence of outputs. There are no data files; all input data sets are produced automatically by the program. Each program has fewer than 500 lines of code and consists of a single source and include file. The include file contains size constants that can be changed if the researcher wants to measure scalability. Each program (except for radar) checks its output automatically. The radar code prints a few lines of output which can be easily verified by the user; directions are provided in the source code. Sample outputs of the programs running on a DEC 3000 Alpha workstation are also provided. Each program (except the airshed) computes physically meaningful results. To keep the program at a manageable size, we have provided a version of the airshed simulation that uses a synthetic workload for the innermost loops.

8. Concluding remarks

We have described a suite of realistic programs that can benefit from a mix of task and data parallelism. Researchers in the areas of mapping, parallelizing compilers, and parallel programming environments are invited to use these programs to test and validate their ideas for exploiting task and data parallelism in applications.

Acknowledgements

Many thanks to Takeo Kanade and his group at Carnegie Mellon for developing the original stereo code, to Dennis Ghiglia and his imaging group at Sandia National Laboratories who, in generously providing us with source code for their spotlight SAR application, showed us the importance of the 2D FFT as a research tool, and to G. Shaw and fellow researchers at MIT Lincoln Labs for providing C source code for the radar benchmark in their technical report.

References

[1] Agrawal, G., Sussman, A., and Saltz, J. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. Tech. Rep. CS-TR-3143 and UMIACS-TR-93-94, University of Maryland, Department of Computer Science and UMIACS, Oct. 1993.

[2] Bailey, D. H. FFTs in external or hierarchical memory. *The Journal of Supercomputing 4* (1990), 23–35.

[3] Chandy, M., Foster, I., Kennedy, K., Koelbel, C., and Tseng, C. Integrated support for task and data parallelism. *International Journal of Supercomputer Applications* (1993). to appear.

[4] Chapman, B., Mehrotra, P., Van Rosendale, J., and Zima, H. A software architecture for multidisciplinary applications: Integrating task and data parallelism. working paper, Feb. 1994.

[5] Cheung, A., and Reeves, A. Function-parallel computation in a data-parallel environment. In *Proceedings of the 1993 International Conference on Parallel Processing* (St Charles, IL, August 1993).

[6] Crovella, M., and LeBlanc, T. The search for lost cycles: A new approach to parallel program performance evaluation. Tech. Rep. 479, Computer Science Department, University of Rochester, Dec. 1993.

[7] Foster, I., and Chandy, K. Fortran M: A language for modular parallel programming. Tech. Rep. MCS-P327-0992, Argonne National Laboratory, June 1992.

[8] Gentleman, W. M., and Sande, G. Fast Fourier transforms for fun and profit. In *Proc. AFIPS* (1966), vol. 29, pp. 563–578.

[9] Gross, T., Hasegawa, A., Hinrichs, S., O'Hallaron, D., and Stricker, T. The impact of communication style on machine resource usage for the iWarp parallel processor. *IEEE Computer* (1994). to appear.

[10] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Tech. Rep. CRPC-TR92225, Center for Research on Parallel Computation, Rice University, May 1993.

[11] McRae, G., Goodin, W., and Seinfeld, J. Development of a second-generation mathematical model for urban air pollution - 1. model formulation. *Atmospheric Environment 16*, 4 (1982), 679–696.

[12] McRae, G., Russell, A., and Harley, R. *CIT Photochemical Airshed Model - Systems Manual.* Carnegie Mellon University, Pittsburgh, PA, and California Institute of Technology, Pasadena, CA, Feb. 1992.

[13] Noll, D., Pauly, J., Meyer, C., Nishimura, D., and Macovski, A. Deblurring for non 2d-fourier transform magnetic resonance imaging. *Magnetic Resonance in Medicine 25* (1992), 319–333.

[14] Okutomi, M., and Kanade, T. A multiple-baseline stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence 15*, 4 (1993), 353–363.

[15] Plimpton, S., Mastin, G., and Ghiglia, D. Synthetic aperture radar image processing on parallel supercomputers. In *Proceedings of Supercomputing '91* (Albuquerque, NM, November 1991), pp. 446–452.

[16] Shaw, G., Gabel, R., Martinez, D., Rocco, A., Pohlig, S., Gerber, A., Noonan, J., and Teitelbaum, K. Multiprocessors for radar signal processing. Tech. Rep. 961, MIT Lincoln Laboratory, Nov. 1992.

[17] Subhlok, J. Automatic mapping of task and data parallel programs for efficient execution on multicomputers. Tech. Rep. CMU-CS-93-212, School of Computer Science, Carnegie Mellon University, November 1993.

[18] Subhlok, J., O'Hallaron, D., Gross, T., Dinda, P., and Webb, J. Communication and memory requirements as the basis for mapping task and data parallel programs. Tech. Rep. CMU-CS-94-106, Department of Computer Science, Carnegie-Mellon University, January 1994.

[19] Subhlok, J., Stichnoth, J., O'Hallaron, D., and Gross, T. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, CA, May 1993), pp. 13–22.

[20] Van Loan, C. *Computational Frameworks for the Fast Fourier Transform.* SIAM, Philadelphia, PA, 1992.

[21] Webb, J. Implementation and performance of fast parallel multi-baseline stereo vision. In *Computer Architectures for Machine Perception* (Dec. 1993).

[22] Webb, J. Latency and bandwidth consideration in parallel robotics image processing. In *Supercomputing '93* (Nov. 1993).

[23] Yang, B., Webb, J., Stichnoth, J. M., O'Hallaron, D. R., and Gross, T. Do&Merge: Integrating parallel loops and reductions. In *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing* (Portland, OR, Aug. 1993), vol. 768 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 169–183.