ornl

ORNL/TM-12779

OAK RIDGE NATIONAL LABORATORY

MARTIN MARIETTA

A Users' Guide to PSTSWM

Patrick H. Worley
Brian Toonen

MANAGED BY
MARTIN MARIETTA ENERGY SYSTEMS, INC.
FOR THE UNITED STATES
DEPARTMENT OF ENERGY

ORNL/TM-12779

Computer Science and Mathematics Division

Mathematical Sciences Section

A USERS' GUIDE TO PSTSWM

Patrick H. Worley †
Brian Toonen •

† Oak Ridge National Laboratory
Mathematical Sciences Section
P. O. Box 2008
Oak Ridge, TN 37831-6367

• Argonne National Laboratory
Mathematics and Computer Science Division
Argonne, IL 60439-4801

Date Published: July, 1995

Research was supported by the Atmospheric and Climate Research Division and by the Applied Mathematical Sciences Research Program, both of the Office of Energy Research, U.S. Department of Energy

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
managed by
LOCKHEED MARTIN ENERGY RESEARCH CORP.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-960R22464

CONTENTS

A USERS' GUIDE TO PSTSWM

Patrick H. Worley
Brian Toonen

The Parallel Spectral Transform Shallow Water Model (PSTSWM) is a code developed to evaluate parallel algorithms for the spectral transform method in global atmospheric circulation models. PSTSWM is also useful for benchmarking parallel platforms that use the message-passing parallel programming paradigm. In this report, we describe how to obtain, compile, and use the code. We also discuss what is involved in porting the code to a new parallel platform.

1. Introduction

PSTSWM Version 4.0 is a message-passing benchmark code and parallel algorithm testbed that solves the nonlinear shallow water equations on a rotating sphere using the spectral transform method. PSTSWM was developed to evaluate parallel algorithms for the spectral transform method as it is used in global atmospheric circulation models [6]. Multiple parallel algorithms are embedded in the code and can be selected at run-time, as can the problem size, number of processors, and data decomposition. Six different problem test cases are also supported, each with associated solution and error analysis options.

The extensive selection of run-time options are included to make a fair parallel algorithm comparison tractable. On each platform, each major algorithm is first tuned to achieve optimum performance before comparing between the algorithms. Developing, validating, maintaining, and executing separate versions of the code for each variant of each parallel algorithm would have been impossible.

The algorithm comparison is also sensitive to problem specifics, motivating the run-time selection of the problem size and problem test case, and to the parallel platform. To avoid maintaining significantly different versions of the code for outwardly similar parallel architectures, PSTSWM has been structured to be easily ported. PSTSWM is written in Fortran 77 with VMS extensions and a small number of C preprocessor directives. Message passing is implemented using MPI [2], PICL [8], PVM [7], or native message passing libraries, with the choice being made at compile time. Additionally, all message passing is encapsulated in three high level routines for broadcast, global minimum and global maximum, and in two classes of low level routines representing variants or stages of the swap operation and the send/receive operation. Porting the code to another message passing system requires either porting the MPI, PICL, or PVM libraries or implementing the (few) communication routines in PSTSWM using native message passing primitives. As of 4/1/95, PSTSWM has been run on the Intel iPSC/2, iPSC/860, DELTA, and Paragon (on both GP and MP nodes and using either the NX or SUNMOS operating systems), the nCUBE/2 and nCUBE/2S, the IBM SP-1 and SP-2, the Cray Research T3D, across a network of workstations, and on a Cray vector machine (as a serial application). In principle, it should also run on any other platform on which MPI, PICL, or PVM is available.

To aid in tuning and in understanding the parallel performance, PSTSWM has been instrumented for the collection of performance data using the PICL trace and profile collection interface. The PICL implementation of the code must be used in order to collect performance

data on interprocessor communication[1] but a mixed PICL/native implementation is also provided that can be used to collect data on events not related to message passing. In the mixed implementation, the performance sensitive message passing uses native commands and PICL is only used in the collection of the performance data.

The ability to easily port and tune PSTSWM on different message-passing platforms has made the code valuable as a fair benchmark. By comparing the run-times for the best parallel algorithm options on each platform, PSTSWM allows the parallel platform to be evaluated on its ability to run the *numerical simulation*, not just a particular parallel implementation. Thus, PSTSWM is a compromise between paper benchmarks [1], where most everything can be varied, at the cost of developing a parallel simulation code from scratch on each platform, and fixed benchmarks, where nothing can be varied even if the parallel implementation is unsuitable. The results on the best algorithm options also provide guidance on how to use a given platform most efficiently. Note that all parallel algorithms have been carefully implemented, eliminating unnecessary buffer copying and exploiting our knowledge of the context in which they are called.

In this report, we describe the practical issues of how to use PSTSWM. In a future report, we will describe the code structure and embedded parallel algorithms in detail. Algorithm comparison results are described in [6]. Benchmark results are described in [18] and [5]. The benchmarking philosophy inspired by PSTSWM is described in [17].

The rest of this report is as follows. Chapter 2 gives a brief history of the development of PSTSWM. Chapter 3 describes how to obtain, build, and run the code. Chapter 4 describes the underlying problem and problem specification options. Chapter 5 describes the approach to parallelization and the parallel algorithm specification options. Chapter 6 describes performance data that can be collected and the performance data collection options. Chapter 7 describes the compile time options, which specify maximum problem size and platform-specific parameters. Chapter 8 describes the model and timing data output produced by running the code. Chapter 9 briefly describes our benchmarking methodology and features in PSTSWM that support this usage. Chapter 10 describes how PSTSWM has been adapted to peculiarities of some of the target platforms. Chapter 11 discusses what is involved in porting PSTSWM to a new platform. Appendix A lists the sample problem input files included with the PSTSWM source code. Appendix B describes the differences between the general distribution of PSTSWM, the PVM-only version used in the ParkBench v1.0 suite of benchmark codes, and the MPI-only version developed for inclusion in the next generation of the ParkBench suite.

[1] The PICL message passing model is fairly rich, representing a substantial subset of both the NX [12] and MPI [2] low level primitives, and has been adequate for obtaining efficient message passing performance on most current message passing systems. But PICL message passing is not as efficient as the native commands on all systems. For example, the SHMEM remote read/write commands are significantly less expensive to use than are tagged message passing commands (like those used in PICL) on the T3D.

2. History

PSTSWM is a parallel implementation of the sequential Fortran 77 code STSWM 2.0, written by J. J. Hack and R. Jakob at the National Center for Atmospheric Research (NCAR) [10]. STSWM was developed to provide the reference solutions to the seven test cases proposed by Williamson et. al. in [14], which were chosen to test the ability of numerical methods to simulate important flow phenomena. In addition, STSWM was meant to serve as an educational tool for numerical studies of the shallow water equations, and is available from the netlib software repository [3] and via the World Wide Web from `http://www.epm.ornl.gov/chammp/stswm`.

PSTSWM was developed as an experimental vehicle for evaluating parallel algorithms for inclusion in PCCM2 [4], the message passing parallel implementation of NCAR's Community Climate Model CCM2 [9]. The data structures and algorithms used to solve the shallow water equations in STSWM are similar to those employed in CCM2 to handle the horizontal dynamics component of the primitive equations on a single vertical level [9]. By modifying PSTSWM to solve the shallow water equations on multiple (independent) vertical levels during each timestep of the simulation, the parallel performance of PCCM2's horizontal dynamics can be studied in isolation from the other aspects of the full model.

PSTSWM represents a complete rewrite of STSWM, but the underlying numerical algorithms, most of the test cases and analysis routines, and much of the user interface are unchanged. In particular, the primary index and loop orderings are unchanged, preserving the close correspondence to the numerical algorithms in CCM2.

From the user's point of view, there are four major differences between STSWM and PSTSWM. First, the graphics options in the analysis routines are not supported in PSTSWM. Second, reading in the solutions for test cases #5, #6, and #7 and writing out the spectral coefficients for the calculated solution are not supported in PSTSWM. This eliminates test case #7, which depends on reading in real weather data. Both of these changes make PSTSWM easier to port by eliminating dependence on the NCAR graphics and netCDF libraries that are used to implement the graphics options and to input and output solution values, respectively.

The other two differences between STSWM and PSTSWM represent additional capabilities. First, as mentioned above, PSTSWM solves the shallow water equations on multiple (fictitious) vertical levels. The number of levels is a run-time selectable parameter, allowing the correct communication and computation granularity for any given 3-D weather or climate code to be specified. Second, two additional input files are used to specify the numerous parallel algorithm and performance measurement options, as will be described in Chapter 5 and Chapter 6.

3. General Description

PSTSWM is packaged as a makefile and five subdirectories: `bin`, `doc`, `input`, `picl2.0`, and `src`. The `bin` subdirectory will contain any executables, i.e., the makefile puts executables there. The `doc` subdirectory contains any documentation, like this document. The `input` subdirectory contains sample input files. The `picl2.0` subdirectory contains the latest version of the PICL communication library.

The `src` subdirectory contains approximately 30,000 lines of source code, about half of which are comments, divided into approximately 80 separate files. These files are mostly Fortran source code with C preprocessor directives (`*.F`), but there are also some include files (`*.i`) and C source code files (`*.c`). `src` also contains the `lib` subdirectory, which contains the communication library-specific implementations of the low-level communication routines, and the `makefiles` subdirectory, which contains one makefile for each different parallel platform and communication library combination. The (top level) driver makefile is used to define the compile-time options and to select the appropriate system-specific makefile.

Obtaining PSTSWM. PSTSWM is available via the World Wide Web from location `http://www.epm.ornl.gov/chammp/pstswm` or via *netlib*, a software distribution service set up on the Internet [3]. To obtain source code and documentation for PSTSWM via netlib, send e-mail to `netlib@ornl.gov` with the message: `send index from chammp`. A mail handler will return a list of available files and further instructions by e-mail. If all else fails, you can also contact the authors at `worleyph@ornl.gov` or `foster@mcs.anl.gov`. The code from the World Wide Web location will generally be the most current. Note that the PVM-only and MPI-only distributions of PSTSWM are available from the WWW location as well.

Building PSTSWM. For most of the supported systems you simply need to execute make at the top level of the PSTSWM distribution with the appropriate arguments to build PSTSWM. Executing make with no arguments will print a brief description of the arguments and the currently available options. These options are described in detail in Chapter 7.

Certain systems may require that the appropriate makefile in `src/makefiles` be modified to, for example, point to the correct C preprocessor, compilers, libraries, and/or include files. What is currently used is what works in the authors' environments.

Multiprocessor	Load Commands
CRI T3D	`pstswm -npes 64`
IBM SP	`setenv MP_PROCS 64` `poe pstswm`
Intel iPSC	`getcube -c pstswm_cube -t 64` `load -c pstswm_cube pstswm`
Intel Paragon - OSF	`mkpart -sz 8x8 pstswm_partition` `pstswm -pn pstswm_partition`
Intel Paragon - SUNMOS nCUBE 2 series	`yod -proc 1 -sz 8x8 pstswm` `xnc -d 6 pstswm`

Figure 3.1: Example commands to load PSTSWM on to 64 processors of a multiprocessor.

Loading PSTSWM. PSTSWM is a hostless parallel program. Loading the program on the multiprocessor is the responsibility of the user, and what is required differs from platform to platform and from site to site. Example load commands are listed in Fig. 3.1. Note that there are numerous environment variables that control parallel execution in the IBM SP environment, and more may be required than is shown here.

Input. Once loaded, PSTSWM normally looks for three input files: `problem`, described in Chapters 4, `algorithm`, described in Chapter 5, and `measurements`, described in Chapter 6. Example algorithm and measurements input files and six example problem input files are located in the subdirectory `input` that comes with the code distribution. To use a particular example problem input file, you must rename it `problem` and copy it to where the executable will look for it, usually in the same directory from which the executable is loaded onto the multiprocessor. Similarly, the example algorithm and measurements input files must be copied to the appropriate location.

If the file `script` is found by PSTSWM, then PSTSWM runs a sequence of experiments, using a sequence of problem, algorithm, and measurements input files. `script` indicates how many experiments to run and the names of the input files to be used in each experiment. See Chapter 9 for more information.

Depending on the value of the algorithm input parameter `MESHOPT`, PSTSWM may also look for a file named `meshmap`, which specifies how the problem should be partitioned over the processors. See Chapter 5 for details.

Output. PSTSWM sends model output to standard out (Fortran Unit 6). Model output includes a summary of problem and algorithm specifications and the results of any requested solution analyses, as described in Chapter 8. If timing measurements are requested, timing data is appended to a file indicated in `measurements`, as described in Chapter 6. If PICL performance data is requested, it is also output to a file indicated in `measurements`. Error output is sent to standard error.

4. Problem Description and Specification

4.1. Spectral Transform Method for the Shallow Water Equations

The shallow water equations in the form solved by the spectral transform method describe the time evolution of three *state* variables: vorticity, horizontal divergence, and a perturbation from an average geopotential. The horizontal velocities are computed from these variables. PSTSWM advances the solution fields in a sequence of timesteps. During each timestep, the state variables of the problem are transformed between the physical domain, where the physical forces are calculated, and the spectral domain, where the terms of the differential equation are evaluated. The physical domain for a given vertical level is a tensor product longitude-latitude grid. The spectral domain for a given vertical level is the set of spectral coefficients in a truncated spherical harmonic expansion of the state variables of the form

$$\xi(\lambda,\mu) = \sum_{m=-M}^{M} \sum_{n=|m|}^{N(m)} \xi_n^m P_n^m(\mu) e^{im\lambda}, \tag{4.1}$$

where

$$\xi_n^m = \int_{-1}^{1} \left[\frac{1}{2\pi} \int_0^{2\pi} \xi(\lambda,\mu) e^{-im\lambda} d\lambda \right] P_n^m(\mu) d\mu \ .$$

Here $i = \sqrt{-1}$, $\mu = \sin\theta$, θ is latitude, λ is longitude, m is the wavenumber or Fourier mode, $P_n^m(\mu)$ is the associated Legendre function, and $\{P_n^m(\mu)e^{im\lambda}\}$ are the spherical harmonics. In the truncated expansion, M is the highest Fourier mode and $N(m)$ is the highest degree of the associated Legendre function in the north-south representation.

Transforming from physical coordinates to spectral coordinates involves performing a real fast Fourier transform (FFT) for each line of constant latitude, followed by integration over latitude using Gaussian quadrature (approximating the Legendre transform (LT)) to obtain the spectral coefficients. The inverse transformation involves evaluating sums of spectral harmonics and inverse real FFTs, analogous to the forward transform. The basic outline of each timestep is the following:

1) Evaluate non-linear product and forcing terms.

2) Fourier transform non-linear terms as a block transform.

3) Compute forward Legendre transforms and advance in time the spectral coefficients for state variables. (Much of the calculation of the time update is "bundled" with the Legendre transform for efficiency.)

4) Transform state variables back to gridpoint space using

   a) an inverse LT and associated computations, and

   b) an inverse real block FFT,

   simultaneously calculating the horizontal velocities from the updated state variables.

For more details on the steps in solving the shallow water equations using the spectral transform algorithm see [10].

4.2. Problem Specification

Unlike STSWM, all problem parameters in PSTSWM are specified at run-time and are input from a file named `problem`. Included with the code distribution are example problem input files corresponding to the first 6 test cases described in [14] and a common problem resolution. Figure 4.1 contains the input file corresponding to test case #2. Problem input files for the other test cases can be found in Appendix A. A brief description of each of the problem parameters follows:

1: `CHEXP` is a 4 character string used to label the experiment. Typically, it is a numeric label indicating which test case is being run (see `ICOND` below). The default is a string of blanks.

2-4: The integer parameters `MM`, `NN`, and `KK` define the total number of spectral coefficients and the spectral truncation used. `MM` is the maximum number of wavenumbers retained, i.e., is equivalent to M in equation 4.1, and

$$N(m) = \begin{cases} \texttt{NN} + m & \text{if } \texttt{NN} + m \leq \texttt{KK}; \\ \texttt{KK} & \text{otherwise.} \end{cases}$$

Note that `KK` must be at least as large as both `MM` and `NN`, but no larger than `MM` + `NN`. All the included problem input files use a triangular truncation `MM` = `NN` = `KK`, so called because the (m, n) indices of the spectral coefficients for a single vertical layer form a triangular grid. The default values are `MM` = 21, `NN` = 21, and `KK` = 21.

5-7: The integer parameters `NLAT`, `NLON`, and `NVER` define the tensor-product physical grid of size `NLON` × `NLAT` × `NVER`. `NLAT` must be an even number no less than $(3\texttt{NN}+1)/2$ if `NN` = `KK`, and no less than $(3\texttt{NN} + 2\texttt{MM} + 1)/2$ otherwise. `NLON` must be a power of two no less than $3\texttt{MM} + 1$. `NVER` must be a positive integer. The default values are `NLAT` = 32, `NLON` = 64, and `NVER` = 9.

```
'0002'                  / CHEXP
42                      / MM
42                      / NN
42                      / KK
64                      / NLAT
128                     / NLON
16                      / NVER
                        / NGRPHS
                        / A
                        / OMEGA
                        / GRAV
                        / HDC
.78539816339744830961 / ALPHA
2000.0                  / DT
999.0                   / EGYFRQ
1.0                     / ERRFRQ
999.0                   / SPCFRQ
120.0                   / TAUE
                        / AFC
.TRUE.                  / SITS
                        / FORCED
                        / MOMENT
2                       / ICOND
```

Figure 4.1: Example problem input file for test case #2.

8: The parameter NGRPHS is not used currently, and should not be set.

9-11: The floating point parameters A, OMEGA, and GRAV are the radius, angular velocity, and gravitational acceleration of the sphere, respectively. The default values are those for the Earth.

12: The floating point parameter HDC is the linear diffusion constant K_4. For a description, see [15]. The default value is 0.0.

13: The floating point parameter ALPHA is the rotation angle of the coordinate system in radians. It is used in test cases #1, #2, and #3. The default value is $\pi/4$ for these test cases.

14: The floating point parameter DT is the timestep in seconds used in advancing the state variables. The size of the timestep is subject to a stability condition defined by the resolution of the computational grids, the numerical method, and the test case. A stability condition estimate is printed as part of the output of the model run. A necessary condition for stability is that the condition estimate be less than 1.0. Due to unresolved problems with the estimator, this is not a sufficient condition when using explicit timestepping (see SITS below). For test cases #2–#6, a condition number less than 0.5 does appear to be sufficient. For test case #1, the estimate should be kept below 0.1. As a first

approximation, the condition estimate scales linearly with the timestep, and a reasonable timestep can be determined quickly. The default value is 2400.0.

15-17: The floating point parameters EQYFRQ, ERRFRQ, and SPCFRQ specify the time interval in simulation hours between output of conservation, error, and spectral analyses of the model state, respectively. The default values are all 999.0. These parameters are ignored when doing timing runs, so that calculation and output of these analyses will not affect the timings.

18: The floating point parameter TAUE is the duration of the model run in simulation hours. The default value is 120.0, representing a 5 day simulation.

19: The floating point parameter AFC is the Asselin filter coefficient, which is used to prevent the modal splitting between even and odd timesteps that can occur in the leapfrog timestepping procedure. The default value is 0.0.

20: The logical parameter SITS specifies whether to use semi-implicit timestepping (.TRUE.) or explicit timestepping (.FALSE.). The default value is .TRUE. . Note that explicit timestepping must be used with test case #1, and that stability restrictions require that explicit timestepping use smaller timesteps than are needed by semi-implicit timestepping for the same problem. Also note that the stability condition estimate for explicit timestepping is not reliable, as described earlier.

21: The logical parameter FORCING specifies whether or not to enable external forcing. The default value is .FALSE. . Note that forcing must be used with test case #4, and will invalidate the error analysis if used in test cases #2 and #3.

22: The logical parameter MOMENT specifies whether to use momentum forcing (.TRUE.) or vorticity-divergence forcing (.FALSE.) in test case #4. The default value is .FALSE..

23: The integer parameter ICOND specifies the test case to be executed. Currently the following cases are supported:

#1 - advection equation for solid body flow
#2 - solid body rotation steady state flow
#3 - jetstream steady state flow
#4 - forced low in jetstream
#5 - zonal flow over isolated mountain
#6 - Rossby-Haurwitz waves

For full descriptions of these cases, see [14]. Note that a 5 day simulation of test case #2 is the designated parallel benchmark test.

5. Parallel Algorithm Description and Specification

5.1. Approach to Parallelization

The parallel algorithms in PSTSWM are based on decompositions of the physical and spectral computational domains over a logical two-dimensional processor mesh of size $P_X \times P_Y$. Initially, the longitude dimension of the physical domain is decomposed over the processor mesh "row" dimension and the latitude dimension is decomposed over the "column" dimension. Thus, FFTs in different processor rows are independent, and each row of P_X processors collaborates in computing a block FFT. Similarly, the Legendre transforms in different processor columns are independent, and each column of P_Y processors collaborates in computing a block of Legendre transforms. The computation of the nonlinear terms at a given location on the physical grid is independent of that at other locations, and the domain decomposition requires no collaboration between processors for this phase of the algorithm. The spectral domain decomposition is a function of the parallel algorithm used.

Two classes of parallel algorithms are available for each transform: distributed algorithms, using a fixed data decomposition and computing results where they are assigned, and transpose algorithms, remapping the domains to allow the transforms to be calculated sequentially. These represent four classes of parallel algorithms:

1) distributed FFT/distributed LT

2) transpose FFT/distributed LT

3) distributed FFT/transpose LT

4) transpose FFT/transpose LT

There are two transpose algorithms, which differ primarily in the number of messages sent and the cumulative message volume. Assume that the transpose algorithms are implemented on Q processors and that each processor contains D data to be transposed. Then the per processor communication costs for the two algorithms can be characterized by

- $\Theta(Q)$ messages, $\Theta(D)$ total volume

- $\Theta(\log Q)$ messages, $\Theta(D \log Q)$ total volume

respectively. In the first ($\Theta(Q)$ transpose) algorithm, every processor sends data to every other processor. In the second ($\Theta(\log Q)$ transpose) algorithm, every processor exchanges data with its neighbors in a logical $\log_2 Q$ dimensional hypercube.

There are also two distributed LT algorithms. Assume that the Legendre transform is parallelized over Q processors and that each processor will contain D spectral coefficients when the transform is complete. Then the per processor communication costs for these two algorithms can be characterized by

- $\Theta(Q)$ messages, $\Theta(DQ)$ total volume
- $\Theta(\log Q)$ messages, $\Theta(DQ)$ total volume

respectively. The $\Theta(Q)$ algorithm works on a logical ring, sending messages to and receiving them from nearest neighbors only. The $\Theta(\log Q)$ algorithm uses the same communication pattern as the $\Theta(\log Q)$ transpose algorithm.

There is only one distributed FFT algorithm. It has the same characterization of communication costs and communication pattern as the $\Theta(\log Q)$ transpose algorithm.

All parallel algorithms execute essentially the same computations, and, modulo load imbalances, differ only in communication costs. Load balance issues are discussed in detail in [6]. Each FFT and LT parallel algorithm also has a number of implementation options that can be selected at runtime, as indicated below.

5.2. Parallel Algorithm Specification

The parallel algorithm specification is input from a file named `algorithm`. Included with the code distribution is the example algorithm input file given in Fig. 5.1. A brief description of each of the algorithm parameters follows.

1-2: The integer parameters `NPLON` and `NPLAT` define the logical processor grid $P_X \times P_Y$, determining how many processors are allocated to the parallel FFT and the parallel LT, respectively. The default values are both 1.

3: The integer parameter `MESHOPT` indicates how to map the logical processor mesh

$$[0, \texttt{NPLON} - 1] \times [0, \texttt{NPLAT} - 1]$$

to the "physical" processors numbered 0 to (`NPLON`·`NPLAT`)-1. The nine options currently supported are listed in Fig. 5.2. The default value is 1. For more details, see the comments in the source code file `map.F`.

```
1               / NPLON
1               / NPLAT
1               / MESHOPT
1               / RINGOPT
0               / FTOPT
0               / LTOPT
1               / COMMFFT
1               / COMMIFT
1               / COMMFLT
1               / COMMILT
0               / BUFSFFT
0               / BUFSIFT
0               / BUFSFLT
0               / BUFSILT
6               / PROTFFT
6               / PROTIFT
6               / PROTFLT
6               / PROTILT
0               / SUMOPT
0               / EXCHSIZE
```

Figure 5.1: Sample algorithm input file.

Options -1 and 1 are generally the best options on mesh based machines. The sign of the option should be set to map the long logical dimension to the long physical dimension. On platforms with hypercube interconnects, use a linear encoding of the longitude coordinate (options $\pm$ 1 and $\pm$ 2) for the distributed FFT or $\Theta(\log Q)$ transpose FFT algorithms and a Gray code encoding (options $\pm$ 3 and $\pm$ 4) for the $\Theta(Q)$ transpose FFT algorithm. Similarly, use a linear encoding of the latitude coordinate (options $\pm$ 1 and $\pm$ 3) for the $\Theta(\log Q)$ transpose LT or distributed LT algorithms and a Gray code encoding (options $\pm$ 2 and $\pm$ 4) for the $\Theta(Q)$ LT algorithm. Note that options -5 and 5 decrease the communication cost of the parallel LT or FFT, respectively, on mesh-based multiprocessors, but at the expense of increasing the communication cost of the other parallel transform.

4: The integer parameter `RINGOPT` is not used currently.

5: The integer parameter `FTOPT` specifies whether to use a distributed parallel FFT algorithm (0), a single transpose parallel FFT algorithm (1), or a double transpose parallel FFT algorithm (2). The single transpose algorithm undecomposes the longitude dimension, to allow the use of a serial FFT, by instead decomposing over the vertical dimension. This can lead to severe load imbalances if there are not enough vertical layers in the model. The double transpose algorithm decomposes over both vertical levels and fields, improving load balance, but requires an additional transpose at the end of the FFT to undecompose the vertical dimension and fields, since the fields must be together for the Legendre transform. `NPLON` must be a power of two to use the distributed algorithm or the $\Theta(\log Q)$ transpose algorithm. The default value is 0.

MESHOPT	Mapping
0	Read mapping from the file meshmap in row major order: $(0,0)\ ,\ (1,0)\ ,\ (2,0)\ ,\ \ldots\ ,\ (\texttt{NPLON}-1,0)\ ,\ (0,1)\ ,\ \ldots$ where meshmap contains a list of physical node numbers, one per line.
-1	Use linear code encoding for both coordinates and column major ordering $(i,j) \rightarrow j + i \cdot \texttt{NPLON}\ .$
1	Use linear code encoding for both coordinates and row major ordering $(i,j) \rightarrow i + j \cdot \texttt{NPLAT}\ .$
-2	Use linear encoding for longitude coordinate, Gray code encoding for latitude coordinate, and column major ordering $(i,j) \rightarrow \texttt{GRAY}(j + \texttt{NPLAT} \bmod \texttt{NPLAT}) + (i \cdot \texttt{NPLON} \bmod \texttt{NPLON}) \cdot \texttt{NPLAT}\ .$
2	Use linear encoding for longitude coordinate, Gray code encoding for latitude coordinate, and row major ordering $(i,j) \rightarrow (i \cdot \texttt{NPLON} \bmod \texttt{NPLON}) + \texttt{GRAY}(j + \texttt{NPLAT} \bmod \texttt{NPLAT}) \cdot \texttt{NPLON}\ .$
-3	Use Gray code encoding for longitude coordinate, linear encoding for latitude coordinat, and column major ordering.
3	Use Gray code encoding for longitude coordinate, linear encoding for latitude coordinat, and row major ordering.
-4	Use Gray code encoding for both coordinates and column major ordering.
4	Use Gray code encoding for both coordinates and row major ordering.
-5	Map latitude coordinate j to a square subgrid in the logical NPLON $\times$ NPLAT grid, scattering longitude coordinate.
5	Map longitude coordinate i to a square subgrid in the logical NPLON $\times$ NPLAT grid, scattering latitude coordinate.

Figure 5.2: Options for mapping logical processor grid onto physical multiprocessor.

6: The integer parameter LTOPT specifies whether to use a distributed LT algorithm (0) or a transpose-based parallel LT algorithm (1). NPLAT must be a power-of-two to use the $\Theta(\log Q)$ transpose algorithm. The transpose-based parallel LT algorithm cannot be used with the double transpose parallel FFT. This is an uninteresting algorithm combination, and is unlikely ever to be supported. The default value is 0.

7-8: The integer parameters COMMFFT and COMMIFT specify which algorithm variants to be used in the parallel forward and inverse FFT algorithms, respectively.

Distributed. The four options for the distributed algorithm are listed in Fig. 5.3, where the default value is 1. The distributed algorithm uses a series of swap operations (between pairs of processors) to move data between processors. These swaps can be implemented in two ways: send/receive (simple)

Processor a	Processor b
`send to b`	`send to a`
`receive from b`	`receive from a`

or send/receive by one processor and receive/send by the other (ordered)

Processor a	Processor b
`send to b`	`receive from a`
`receive from b`	`send to a`

Also, if the block transform is divided into two blocks, communication and computation can be overlapped: during each stage of the transform one block is being communicated while the other block is being used in computation.

Transpose. The twelve options for the transpose algorithms are listed in Fig. 5.4, where the default value is 1. The $\Theta(Q)$ transpose parallel algorithm has two options for scheduling how the information is sent and received, linear and exclusive-OR. During step k of the linear schedule, processor q sends data to processor $(q + k \bmod Q)$ and receives data from $(q - k \bmod Q)$. During step k of the exclusive-OR schedule, processor q swaps data with processor XOR(q, k). The send/receives and swaps in these algorithms can both be implemented in two ways: simple or ordered, as in the distributed FFT. (For the linear ordering, the ordered option uses send/receive by even numbered processors and receive/send by odd numbered processors.)

In the $\Theta(Q)$ transpose algorithm, what is being sent is known beforehand and all of the data can be sent before any data is received. This is the send-ahead option. Similarly, all data that is received is retained and the destination of the data is known beforehand. Thus, all receive requests can be posted before any data is sent, assuming that nonblocking receives are supported by the native message passing system. This is the receive-ahead option. If nonblocking receives are not specified in PROTFFT or PROTIFT (see below), then the receive-ahead option is ignored.

0	one block distributed FFT using simple swap
1	one block distributed FFT using ordered swap
2	two block distributed FFT using simple swap
3	two block distributed FFT using ordered swap

Figure 5.3: Distributed FFT algorithm options.

0	$\Theta(Q)$ transpose algorithm using simple send/receive and linear schedule
1	$\Theta(Q)$ transpose algorithm using ordered send/receive and linear schedule
2	$\Theta(Q)$ transpose algorithm using simple send/receive with receive-ahead and linear schedule
3	$\Theta(Q)$ transpose algorithm using ordered send/receive with receive-ahead and linear schedule
4	$\Theta(Q)$ transpose algorithm using simple send/receive with receive-ahead, send-ahead, and linear schedule
10	$\Theta(Q)$ transpose algorithm using simple swap and exclusive-OR schedule
11	$\Theta(Q)$ transpose algorithm using ordered swap and exclusive-OR schedule
12	$\Theta(Q)$ transpose algorithm using simple swap with receive-ahead and exclusive-OR schedule
13	$\Theta(Q)$ transpose algorithm using ordered swap with receive-ahead and exclusive-OR schedule
14	$\Theta(Q)$ transpose algorithm using simple send/receive with receive-ahead, send-ahead, and exclusive-OR schedule
20	$\Theta(\log Q)$ transpose algorithm using simple swaps. Q must be a power of two.
21	$\Theta(\log Q)$ transpose algorithm using ordered swaps. Q must be a power of two.

Figure 5.4: Transpose algorithm options.

0	interleaved $\Theta(Q)$ distributed LT algorithm using simple send/receive
1	interleaved $\Theta(Q)$ distributed LT algorithm using ordered send/receive
2	interleaved $\Theta(Q)$ distributed LT algorithm using delayed receive
10	localized $\Theta(Q)$ distributed LT algorithm using simple send/receive
11	localized $\Theta(Q)$ distributed LT algorithm using ordered send/receive
12	localized $\Theta(Q)$ distributed LT algorithm using simple send/receive with receive-ahead
13	localized $\Theta(Q)$ distributed LT algorithm using ordered send/receive with receive-ahead
20	$\Theta(\log Q)$ distributed LT algorithm using simple swap
21	$\Theta(\log Q)$ distributed LT algorithm using ordered swap

Figure 5.5: Distributed forward Legendre transform algorithm options.

The exclusive-OR option for $\Theta(Q)$ transpose is most efficient on hypercubes, and has similar performance to the linear option on meshes. The send-ahead options can be highly efficient for small problems, but can also consume all available system buffer space and cause deadlock.

Like the distributed algorithm, the $\Theta(\log Q)$ transpose algorithm uses a series of swap operations to move data between processors, and these swaps can be implemented in two ways: simple or ordered. The amount of data to be received at each step is known beforehand, and there is also a receive-ahead option. But this option is invoked implicitly by specifying additional buffer space via the `BUFSFFT` or `BUFSIFT` parameters (ee below).

9: The integer parameter `COMMFLT` specifies which algorithm variants to be used in the parallel forward LT algorithms.

Distributed. The nine options for the distributed algorithms are described in Fig. 5.5. The default value is 1. The $\Theta(Q)$ distributed LT algorithm has two options for scheduling when interprocessor communication occurs: interleaved and localized. The interleaved option intersperses communication with computation in a series of send/receive/compute steps. The localized option isolates communication from the body of the computation. If the interleaved option is used in the forward or the inverse LT, it must also be used for the transform in the other direction.

For the interleaved algorithm, the send/receive/compute schedule can also be organized as send/compute/receive, permitting some communication/computation overlap. This is the delayed-receive option. The amount of data to be received at each communication step is known beforehand, and there is also a receive-ahead option. This option is invoked implicitly by specifying additional buffer space via the `BUFSFLT` parameter (see below).

0	interleaved $\Theta(Q)$ distributed LT algorithm using simple send/receive
1	interleaved $\Theta(Q)$ distributed LT algorithm using ordered send/receive
2	interleaved $\Theta(Q)$ distributed LT algorithm using delayed receive
>9	do nothing

Figure 5.6: Distributed inverse Legendre transform algorithm options.

For the localized algorithm, all data that is received is retained and the destination of the data is known beforehand. Thus, all receive requests can be posted before any data is sent. This is the receive-ahead option.

For both interleaved and localized algorithms, each processor receives data from one processor and sends data to another during each communication step. Both simple and ordered send/receive options are supported.

The $\Theta(\log Q)$ distributed LT algorithm uses a series of swaps operations to move data between processors, and both the simple and ordered swap options are available.

Transpose. For the transpose parallel forward LT algorithms, there are the same 12 options as for the transpose parallel FFT algorithms, listed in Fig 5.4. The default value is 1.

10: The integer parameter `COMMILT` specifies which algorithm variants to be used in the parallel inverse LT algorithms.

Distributed. The four options for the distributed algorithms are described in Fig. 5.6. The default value is 1. If the $\Theta(\log Q)$ or the localized $\Theta(Q)$ distributed forward LT algorithm is used, then no communication is required during the inverse transform. If the interleaved algorithm is used for the forward transform, then an interleaved algorithm must be used for the inverse, and the same options hold.

Transpose. For the transpose parallel inverse LT algorithms, there are the same 12 options as for the transpose parallel FFT algorithms, listed in Fig 5.4. The default value is 1.

11-12: The integer parameters `BUFSFFT` and `BUFSIFT` specify the number of communication buffers to be used in receive-ahead variants of the parallel forward and inverse FFT algorithms, respectively. They have an effect only for the $\Theta(\log Q)$ transpose algorithm. The other algorithms do not need extra buffer space to enable receive-ahead, which is invoked instead by the `COMMFFT` and `COMMIFT` parameters. Receive-ahead for the $\Theta(\log Q)$ transpose in the forward FFT is invoked by specifying nonblocking receives (see PROTFFT below) and `BUFSFFT` > 3 if `NPLON` is a power of two and `BUFSFFT` > 5 otherwise. Similarly, specify `BUFSIFT` > 3 or `BUFSIFT` > 5 and nonblocking receives to invoke receive-ahead for the inverse FFT. At most $\log_2(\texttt{NPLON}) + 1$ buffers can be used if `NPLON` is a power of two,

Algorithm	minimum	maximum
Forward Fourier Transform	BUFSFFT	
$\Theta(\log Q)$ transpose		
- NPLON a power of two	4	$1+\log_2$ NPLON
- NPLON not a power of two	6	$3+2\log_2$ NPLON
Inverse Fourier Transform	BUFSIFT	
$\Theta(\log Q)$ transpose		
- NPLON a power of two	4	$1+\log_2$ NPLON
- NPLON not a power of two	6	$3+2\log_2$ NPLON
Forward Legendre Transform	BUFSFLT	
$\Theta(\log Q)$ transpose		
- NPLAT a power of two	4	$1+\log_2$ NPLAT
- NPLAT not a power of two	6	$3+2\log_2$ NPLAT
Interleaved $\Theta(Q)$ distributed	2	NPLAT
$\Theta(\log Q)$ distributed	2	3
Inverse Legendre Transform	BUFSILT	
$\Theta(\log Q)$ transpose		
- NPLAT a power of two	4	$1+\log_2$ NPLAT
- NPLAT not a power of two	6	$3+2\log_2$ NPLAT
Interleaved $\Theta(Q)$ distributed	3	NPLAT + 1

Figure 5.7: Number of buffers required to enable receive-ahead algorithms.

and at most $2\log_2$(NPLON)+3 buffers can be used otherwise. If more buffers are specified, only the maximum allowable will be used. See Fig. 5.7 for a summary of this information.

13: The integer parameter BUFSFLT specifies the number of communication buffers to be used in receive-ahead variants of the parallel forward LT algorithms. The localized $\Theta(Q)$ distributed algorithm and $\Theta(Q)$ transpose algorithm do not need additional buffers to enable receive-ahead, which is invoked instead by COMMFLT. For the interleaved $\Theta(Q)$ distributed algorithm, receive-ahead is invoked by specifying BUFSFLT > 1 and nonblocking receives (see PROTFLT). Up to BUFSFLT $- 1$ receive requests can be posted early, and each request requires an additional buffer. At most NPLAT buffers can be used. If more than this are specified, only NPLAT will be used.

To invoke receive-ahead for the $\Theta(\log Q)$ distributed algorithm, BUFSFLT should be set to 2 if NPLAT is a power of two and to 3 otherwise, and nonblocking receives should be specified. Receive-ahead for the $\Theta(\log Q)$ transpose is invoked by specifying BUFSFLT > 2 and nonblocking receives, with the same bounds as for BUFSFFT except that NPLON is replaced by NPLAT.

See Fig. 5.7 for a summary of this information.

14: The integer parameter BUFSILT specifies the number of communication buffers to be used in receive-ahead variants of the parallel inverse LT algorithms. BUFSILT has an effect only in the interleaved $\Theta(Q)$ distributed algorithm and in the $\Theta(\log Q)$ transpose algorithm. The other algorithms either do not need additional buffers to enable receive-

0	blocking send/blocking receive
1	nonblocking send/blocking receive
2	blocking send/nonblocking receive
3	nonblocking send/nonblocking receive
4	blocking ready send/nonblocking receive
5	nonblocking ready send/nonblocking receive
6	synchronous blocking send/blocking receive (for ordered operation only).

Figure 5.8: Communication protocol options.

ahead, or perform no interprocessor communication during the inverse transform. For the interleaved $\Theta(Q)$ distributed algorithm, BUFSILT $-$ 2 receive requests can be posted early, and receive-ahead is invoked by specifying BUFSILT $>$ 2 and nonblocking receives (see PROTILT). At most NPLAT $+$ 1 buffers can be used.

Receive-ahead for the $\Theta(\log Q)$ transpose is invoked by specifying BUFSILT $>$ 2 and nonblocking receives, with the same bounds as for BUFSILT.

See Fig. 5.7 for a summary of this information.

15-18: The integer parameters PROTFFT, PROTIFT, PROTFLT, PROTILT specify the communication protocol to be used with the parallel algorithms for the forward FFT, the inverse FFT, the forward LT, and the inverse LT, respectively. The five options for simple send/receive and swap operations and the six options for the ordered operations are listed in Fig. 5.8. The default value is 6.

The interpretation of blocking and nonblocking is somewhat system dependent. For PSTSWM, nonblocking commands are assumed to spawn communication requests that then proceed independent of the main thread of control. With respect to the parallel algorithms in PSTSWM, blocking commands are faster and are available on all platforms. The nonblocking commands are somewhat slower and are not universally supported, but they enable the overlap of communication and computation. The nonblocking commands also do not require system buffering in order to avoid deadlock when using the simple swap and simple send-receive orderings. The ready send option, available on Intel iPSC systems (native or PICL) and when using MPI, uses a different communication protocol, one which is faster for large messages in PSTSWM. In the synchronous option, handshaking messages are sent to guarantee that each processor is ready to receive a message before it is sent.

19: The integer parameter SUMOPT specifies the order of summation in local calculations. Options are in-place linear ordering (0) and binary tree ordering (1). In-place is generally faster, because of better data locality. When used with the $\Theta(\log Q)$ distributed LT

algorithm, the binary tree algorithm insures the "reproducibility" of results, i.e., the same order of operations holds independent of the number of processors used. Note that reproducibility automatically holds for the transpose algorithms is impossible to (efficiently) impose on the $\Theta(Q)$ distributed LT algorithm. The default value is 0.

20: The $\Theta(\log Q)$ distributed LT algorithm is a hybrid algorithm that switches from a high(er) latency/low(er) communication volume algorithm to a low latency/high volume algorithm, depending on the length of the vectors it is operating on [13]. The integer parameter `EXCHSIZE` specifies the length at which it switches. The optimal value is primarily a function of the platform characteristics and the algorithm and communication protocol options, and so can be estimated once for each platform for algorithm options of interest. The default value is 0, i.e., do not switch to the lower latency algorithm.

5.3. Discussion

As evident in the previous section, there are *many* parallel algorithm options. Not all of the options work on all platforms or for all problem sizes or numbers of processors, and not all options are consistent with one another. Consistency problems are checked for internally, as are options that are illegal on a given platform. The only problems that cannot be identified within the code prior to running an experiment have to do with algorithm and protocol options that require additional system or user communication buffers in order to avoid deadlock conditions. The potential for such problems is identifiable *a priori*, and the code prints warning messages to this effect before starting a run. Additional detail on algorithm option restrictions follows.

Communication Protocols. All but communication protocol options 0 and 6 require nonblocking communication commands. The Intel NX, IBM MPL, and the MPI communication libraries provide these, but nCUBE VERTEX and PVM do not. Thus options 1–5 are illegal when using the latter two systems.

On the Cray Research T3D, the native messaging is implemented on top of SHMEM, using the remote read/write commands, and the available protocols are "defined" to be 1, 2, and 6, representing nonblocking send/blocking receive, blocking send/nonblocking receive, and synchronous, respectively. PSTSWM can also be run on the T3D using the PICL library, which is layered on top of Cray Research's implementation of PVM and works with protocol options 0 and 6.

Note that PICL is a compatibility library, and thus inherits the restrictions of the underlying native communication library. For example, on nCUBE machines, only communication protocols 0 and 6 are legal, while on Intel machines, all protocols are legal.

Protocol options 4 and 5, as defined, use the ready send command, which sends messages without any handshaking. If the destination is not ready to receive a ready message (by already having posted a receive request), then the message is thrown away. Options 4 and 5

provide the necessary additional handshaking, with the sender waiting for a go-ahead message before initiating the ready send. In actuality, if the ready send command is not supported in the given communication library, options 4 and 5 simply use normal blocking and nonblocking sends with this explicit handshaking protocol.

Figure 5.9 summarizes this information.

Message Buffering. Using simple swap and simple send/receive orderings and blocking send require that messages be buffered, usually at the sender. Under MPL, NX, PVM, and VERTEX, system buffers are used, and either deadlock or a program abort can occur if the buffer space fills up. The amount of system buffer space under NX or VERTEX can be specified at load time. The amount of system buffer space under MPL can be specified at run time (once), and PSTSWM calculates the amount needed and makes this request. If a sequence of experiments are being run (see Chapter 9) and a later experiment needs more space than the first, then the code may yet deadlock. MPI uses user space for buffering messages. PSTSWM passes a pointer to unused work space to MPI for this purpose (see `WORKSPACE` in Chapter 7). If there is not enough space to guarantee correct execution, then a warning message is output, but execution is still attempted.

Receive-ahead Algorithms. The receive-ahead variants of the parallel algorithms post one or more receive requests before the message is likely to be sent. Receive-ahead requires that nonblocking receives be used and that space in which to receive the message(s) be set aside. For some of the parallel algorithms, this space is available naturally and the receive-ahead variant is chosen explicitly via the `COMMFFT`, `COMMIFT`, `COMMFLT`, or `COMMILT` parameters. For other algorithms, the receive-ahead variants require additional buffer space, and the number of receive requests posted early is determined by the number of extra buffers allocated. For these algorithms, the receive-ahead variant is chosen implicitly by specifying both nonblocking receives and enough extra buffer space, as indicated in Fig 5.7. Figure 5.10 summarizes which algorithms enable receive-ahead explicitly, and which do so implicitly.

Communication Library	0	1	2	3	4	5	6
CRI T3D native		X	X				X
IBM SP native	X	X	X	X	X	X	X
Intel native	X	X	X	X	X	X	X
nCUBE native	X						X
PVM	X						X
MPI	X	X	X	X	X	X	X

Figure 5.9: Legal communication protocol options.

Algorithm	Explicit	Implicit
Forward Fourier Transform		
$\Theta(Q)$ transpose	X	
$\Theta(\log Q)$ transpose		X
distributed	X	
Inverse Fourier Transform		
$\Theta(Q)$ transpose	X	
$\Theta(\log Q)$ transpose		X
distributed	X	
Forward Legendre Transform		
$\Theta(Q)$ transpose	X	
$\Theta(\log Q)$ transpose		X
Interleaved $\Theta(Q)$ distributed		X
Localized $\Theta(Q)$ distributed	X	
$\Theta(\log Q)$ distributed		X
Inverse Legendre Transform		
$\Theta(Q)$ transpose	X	
$\Theta(\log Q)$ transpose		X
Interleaved $\Theta(Q)$ distributed		X

Figure 5.10: Parallel algorithms specifying receive-ahead explicitly and implicitly

6. Performance Measurement Description and Specification

6.1. Approach to Performance Measurement

Three types of performance data can be collected automatically. The first type is execution time, both per timestep and total time, measured using real time clocks. The per timestep data reports the minimum and maximum times over all processors for that timestep. Since the code is only loosely synchronous, the individual timestep timings are not guaranteed to be accurate. The total time reports the maximum total time over all processors. Since the processors are synchronized before beginning timing, the total time measurement is accurate. Note that timing begins only when the timestepping begins. The code set-up phase (input and calculation of initial values) is not included, as this is an artifact of the experiment and is not representative of production codes. (Separate benchmarks are needed to measure I/O performance for meteorological codes.) Execution time data is either appended to the output file indicated in the `measurements` input file, or is appended to the file `timings` (the default).

The second type of performance data is profile data, indicating the amount of time spent in various user and system level events. The user level events (and associated event types) are

- 0: total time,
- 1: evaluation of nonlinear terms in physical space,
- 2: forward FFT,
- 3: forward Legendre transform,
- 4: calculation of tendencies (time update) in spectral space,
- 5: addition of linear diffusion (in spectral space) and first half of Asselin filter (in physical space),
- 6: inverse Legendre transform,
- 7: inverse FFT,
- 8: second half of Asselin filter (in physical space), and
- 100+i: timestep `i`

```
.FALSE.          / TIMING
.FALSE.          / TRACING
.FALSE.          / TRACEFILE
0                / VERBOSE
100000           / TRSIZE
1                / TRSTART
10000            / TRSTOP
0                / TL1
0                / TL2
0                / TL3
                 / TOUTPUT
                 / TMPNAME
'pstswm.trace'   / PERMNAME
```

Figure 6.1: Sample measurements input file.

The system level events profiled are primarily PICL communication calls and have negative event types, as documented in [16].

The third type of performance data is event traces, indicating when each of the above mentioned events started and stopped. This data can be visualized with the performance visualization tool ParaGraph [11]. Both profile and trace data are saved in a file indicated in the `measurements` input file.

Both profile and trace data are collected using PICL instrumentation logic. To collect user event data requires that either a mixed PICL/native or a pure PICL implementation of PSTSWM be used. System event data data can only be collected from a pure PICL implementation.

6.2. Performance Measurement Specification

The performance measurement specification is input from the `measurements` input file. Included with the code distribution is the example measurements input file given in Fig. 6.1. A brief description of each of the measurement parameters follows.

1: The logical parameter `TIMING` specifies whether to time the current run of PSTSWM. If so, all model output during timestepping is disabled, and both per timestep and total execution time is measured. The default value is `.FALSE.` .

2: The logical parameter `TRACING` specifies whether to trace the execution of PSTSWM. Instrumentation has been added to the PSTSWM source code, marking each timestep calculation and the major phases of each timestep as separate tasks. When `TRACING` = `.TRUE.`, the PICL trace collection facility is used to collect trace and profile data on these user-defined events and on PICL communication calls, as specified by tracing level parameters. The default value is `.FALSE.` .

3: The logical parameter TRACEFILE specifies whether a trace file should be opened. If not, then any profile or trace data is lost. The default value is .FALSE. .

4: The integer parameter VERBOSE specifies the format in which the trace data is written to disk. The options are:

(0): a compact form that can be read by the performance visualization program ParaGraph [11]

(1): a verbose form with labels that make the trace output more human-readable

5: PICL trace data is saved in internal buffers, to minimize the perturbation of the performance measurements. The integer parameter TRSIZE specifies how much buffer space to allocate for trace data on each node. The default value is 0.

6-7: The integer parameters TRSTART and TRSTOP specify the timestep at which trace collection is to begin and the timestep at which it is to end, respectively. The default values are both -1.

8-9: The integer parameters TL1, TL2, and TL3 specify what PICL trace data is collected. TL1 controls the collection of data on PICL communication events, TL2 controls the collection of data on user-defined events, and TL3 controls the collection of data on tracing events. All three have the following options:

(-1): Do not collect data.

(0): Collect profile data.

(1): Collect both profile data and detailed trace data.

10: The character string parameter TMPNAME is not used currently.

11: The character string parameter PERMNAME is the name of the disk file where the profile and trace data is to be written at the end of the execution of PSTSWM (if TRACEFILE = .TRUE.). If PERMNAME = '', then the data is not saved. Note that only the first 32 characters are used, and that the default value is the blank string (''), i.e., do not save the profile and trace data.

7. Compile Time Options

7.1. Parameter File Specifications

Unlike STSWM, PSTSWM need not be recompiled to run different problem sizes. But, to increase portability, PSTSWM does not allocate memory dynamically. (Neither dynamic memory allocation nor the Fortran to C interface is part of the Fortran 77 standard.) Instead, parameter values in the include file `params.i` are used to specify the maximum problem sizes and number of processors, and the storage is allocated accordingly. An edited version of the `params.i` file included in the code distribution is given in Fig. 7.1. A brief description of each of the parameters follows.

1-3: The integer parameters `MMX`, `NNX`, and `KKX` specify the maximum values for `MM`, `NN`, and `KK`, respectively.

4-6: The integer parameters `NLATX`, `NLONX`, and `NVERX` specify the maximum values for `NLAT`, `NLON`, and `NVER`, respectively.

7-8: The integer parameters `NPROCSX` and `LGPROCSX` specify the maximum number of processors and the base-2 logarithm of the maximum number of processors, respectively.

9: The integer parameter `NGRPHSX` specifies the amount of storage to allocate for time series data of solution and error analysis statistics. These statistics are not collected currently, but the logic (from STSWM) is retained for future development.

7.2. Makefile Parameter Specifications

The above mentioned compile time parameters are used only for declaring one dimensional arrays. For good performance on RISC-based microprocessors, it is crucial that multidimensional arrays be packed into contiguous storage (to increase data locality). To ensure this, all field arrays are allocated from a single large buffer, whose size is specified by a makefile command-line parameter. The default value is 500000 floating point values, or 4 megabytes if using 64 bit precision.

```
C*************************************************************************
C* Include file 'params.i'                                               *
C*************************************************************************
      INTEGER MMX, NNX, KKX, NLATX, NLONX, NVERX
      PARAMETER
     &       (MMX=340,
     &        NNX=340,
     &        KKX=340,
     &        NLATX=512,
     &        NLONX=1024,
     &        NVERX=72)
C*                                                                       *
      INTEGER NPROCSX, LGPROCSX, NGRPHSX
      PARAMETER
     &       (NPROCSX=1024,
     &        LGPROCSX=10,
     &        NGRPHSX=100)
C*                                                                       *
C*************************************************************************
C* end include file                                                      *
C*************************************************************************
```

Figure 7.1: Compile time parameters in include file.

Makefile command-line parameters are also used to specify the target platform, the communication library, the precision of the different variable types, and the word alignment used when allocating arrays. Executing make without specifying any parameters will generate the description of these parameters listed in Fig. 7.2.

Figure 7.3 summarizes the parallel platform/communication library combinations that are supported currently. Both native and PICL are supported on all parallel platforms listed, and the underlying communication library for the native and PICL implementations is listed in the figure. Also, the particular MPI library implementation is listed in the MPI column.

Note that some of the MPI and PVM combinations are missing for platforms on which these libraries are supported. To add support for these combinations merely requires the generation of an appropriate makefile. Additional comments follow.

1) Currently, PVM is not an explicit communication library choice, but rather is the native communication library for `sun-pvm` and `rs6k-pvm` (for SUN and IBM RS/6000 workstations, respectively). The Cray Research implementation of PVM is also the basis for the PICL communication library on the T3D. The native T3D implementation is a mix of PVM and SHMEM library calls.

2) The paragon platform uses the OSF operating system and NX is the native communication library. The `paragon-sunmos` uses the SUNMOS operating system and the low level SUNMOS communication commands are used in the native implementation. The

```
% make

To compile :  make MACH=<machine>
To clean up:  make MACH=<machine> clean

Optional make parameters are
  COMM=<communication library>
  WORKSPACE=<number of REALS>
  PRECISION=<4|8>
  ALIGN=<number of REALS>

MACH is the platform being compiled for, currently one of
  crayvector, ipsc2, ipsc860, ncube2, ncube2S, paragon,
  paragon-sunmos, paragon-mp, rs6k-pvm, sp, sun-pvm, t3d
COMM is the communication library to use, one of
  native, mpich, mpif, picl, mixed,
  (mixed uses native routines for performance sensitive code and PICL
   for the rest. To collect profile data, choose mixed or picl. To
   collect event traces, choose picl. native is the default.)
WORKSPACE is the amount of space (in REALS) to be allocated for
  problem storage. If omitted, a default value of 500000 is used.
PRECISION specifies whether REALs are 4 bytes or 8 bytes long.
  The default is 8.
ALIGN is the (REAL) word boundary the starting address of each array
  must fall on. ALIGN must be 1 or a multiple of 2. The default is 2.
```

Figure 7.2: make command-line arguments

Platform	native	MPI	PICL	PVM
CRI T3D	SHMEM + PVM	–	PVM	–
CRI vector	(serial)	–	–	–
IBM SP	MPL	mpich mpif	MPL	–
Intel iPSC	NX	–	NX	–
Intel Paragon - OSF	NX	mpich	NX	–
Intel Paragon - SUNMOS	SUNMOS	–	NX	–
nCUBE 2 series	VERTEX	–	VERTEX	–
workstation networks				
SUN	PVM	–	PVM	–
IBM RS6000	PVM	–	PVM	–

Figure 7.3: Supported parallel platform/communication library combinations.

`paragon-mp` is the same as `paragon` except that certain tasks are spawned to be run on the additional compute processor. Note that specifying `paragon` will work correctly on an MP system. We have not yet developed a `paragon-mp-sunmos` implementation, although it will be simple to do.

3) `mpich` is the Argonne/Mississippi State implementation of MPI, and we have run it on the Intel Paragon and on the IBM SP2. It should also run on most of the other platforms, but we have not developed the corresponding makefiles yet. `mpif` is the experimental IBM version of MPI for the SP, MPI-F.

4) As mentioned above, not all platform/communication library combinations are currently supported. If an unsupported combination is requested, then the information in Fig. 7.2 is generated.

8. OUTPUT

8.1. Model Output

Except for trace and timing data and error messages, all PSTSWM output is generated by one process and is sent to the standard output (unit number 6). Figure 8.1 contains the (edited) output from a 5-day model run of test case #2 using a 2×4 processor grid on an Intel iPSC/860 and 32-bit precision.

The first part of the output summarizes the problem and algorithm specifications. Note that both the value of `COMPSZ` and the amount of space actually used are given.

The second part of the output contains the solution field analysis, which is identical to that provided by STSWM. The `COURANT NUMBER` is the stability estimate, which must be less than 1.0, but is safest if it is no more than 0.5. `HEIGHT` refers to a normalized value of the geopotential.

For this model run, both error analysis (`ERRFRQ` = 1.0) and timing (`TIMING` = `.TRUE.`) were enabled. This disabled all but the first and last error analysis output. The error in the simulation is a function of the test case, the problem resolution, the machine precision (`MACHINE EPSILON`), and the number of timesteps. The following guidelines refer to runs of no longer than 5 simulated days (120 hours).

For test case #1, the error should not exceed 10^{-1} for T21 (`MM` = `NN` = `KK` = 21), and the error should be reduced by an additional factor of 10^{-1} for each doubling of the resolution: T42, T85, T170, etc. For test cases #2 and #3, the error should be within a few orders of magnitude of the machine precision. For test case #4, the L_1, L_2, and l_∞ errors should not exceed approximately 10^{-2}. For test cases #5 and #6, the exact solutions are not known, and the conservation and spectral analyses must be used to evaluate the solutions. Sample output files generated on a Cray Y-MP for all test cases are included with the STSWM code distribution.

8.2. Timing Data

Figure 8.2 contains the timing output for this run, which is written into a file specified in `measurements`. If this file does not already exist, it is created and the column labels are

```
PARALLEL SPECTRAL TRANSFORM SHALLOW WATER MODEL, VERSION 4.0
(BASED ON SPECTRAL TRANSFORM SHALLOW WATER MODEL, VERSION 2.0
          COPYRIGHT (C) 1992
          UNIVERSITY CORPORATION FOR ATMOSPHERIC RESEARCH
          ALL RIGHTS RESERVED)

READING PARAMETERS FROM FILE problem:

EXPERIMENT 0002

SPECTRAL TRUNCATION TYPE: TRIANGULAR
M = N = K =   42

NUMBER OF GRIDPOINTS IN MODEL
NORTH-SOUTH GAUSSIAN GRID:  NLAT =   64
EAST-WEST EQUIDISTANT GRID: NLON =  128
VERTICAL LAYERS:            NVER =   16

READING PARAMETERS FROM FILE algorithm:

NUMBER OF PROCESSORS IN PARALLELIZATION
ROW PROCESSORS:    PX =    2
COLUMN PROCESSORS: PY =    4

PARALLEL ALGORITHMS
  FT: DISTRIBUTED
  LT: DISTRIBUTED (OVERLAPPED RING VECTOR SUM)

AVAILABLE WORK SPACE (IN BYTES): 5200000
REQUIRED WORK SPACE            : 1843552

MACHINE EPSILON (1.0 + EPS > 1.0) =  1.192092900E-07

TEST CASE #2: STEADY STATE NONLINEAR GEOSTROPHIC FLOW
ROTATED BY AN ANGLE ALPHA = 0.785
MAX. WIND      = 3.861068300E+01
COURANT NUMBER = 0.5091

GLOBAL MEAN STEADY GEOPOTENTIAL =  2.317216400E+04

ERRANL: INITIAL VALUES FOR NORMALIZATION OF RELATIVE ERRORS
(SLIGHTLY GRID DEPENDENT!)
HEIGHT MIN./MAX. =  1.093466800E+03/ 2.998115500E+03
HEIGHT AVG./VAR. =  2.363021500E+03/ 3.226756200E+05

ERRANL: ERROR ESTIMATES FOR NSTEP =  216, TAU = 120.00 HRS
HEIGHT ERROR
L1 =  2.377001000E-06, L2 =  2.857407900E-06 L(INF) =  8.143136300E-06
VECTOR WIND ERROR
L1 =  3.304713400E-05, L2 =  3.853516700E-05 L(INF) =  1.234512900E-04
HEIGHT MIN./MAX.  =  1.093454000E+03/ 2.998118200E+03
HEIGHT AVG./VAR.  =  2.363021000E+03/ 3.226713100E+05
```

Figure 8.1: Sample model output.

```
   total          min           max        overhead    nstep
 0.38209E+03  0.17586E+01  0.17836E+01  0.00000E+00   216

spectral space  physical space  problem  processors   map
( 42, 42, 42)  ( 128,  64, 16) (2,T,F,F) (   2,   4)   2

      parallel ft              parallel lt         sum  exchsize
(0, 0, 0,   1,   1,0,0) (0, 0, 0,   1,   2,0,0)   0         0
```

Figure 8.2: *Sample timing output.*

output. If the file already exists, subsequent timings are appended to the file. Note that model timing data is output on a single line. In Fig. 8.2, this is broken into three pieces.

1: `total` is the total execution time for this model run, where timing begins *after* initialization: reading in problem and algorithm specifications, setting up the work space, and calculating initial data.

2-3: `min` and `max` are the minimum and maximum execution time for a single timestep, respectively, taken over all processors.

4: `overhead` is not measured currently.

5: `nstep` is the number of timesteps required to reach `TAUE`, which equals 120.0 hours for this run.

6: `spectral space` gives `MM`, `NN`, and `KK`.

7: `physical space` gives `NLON`, `NLAT`, and `NVER`.

8: `processors` gives `NPLON` and `NPLAT`.

9: `map` is `MESHOPT`.

10: `parallel ft` gives `FTOPT`, `COMMFFT`, `COMMIFT`, `BUFSFFT`, `BUFSIFT`, `PROTFFT`, and `PROTIFT`. Note that the number of buffers listed is the actual number used, not the number requested. Each parallel algorithm has a minimum number of buffers it needs and a maximum number that it can use.

11: `parallel lt` gives `LTOPT`, `COMMFLT`, `COMMILT`, `BUFSFLT`, `BUFSILT`, `PROTFLT`, and `PROTILT`.

12: `sum` is `SUMOPT`.

11: `exchsize` is `EXCHSIZE`.

The information saved in the `timings` file is important for interpreting performance experiments, but is not sufficient for specifying the model run. For example, while `TAUE` or `DT` are

important when interpreting the accuracy of the solution, performance is solely a function of `nstep`, and only `nstep` is recorded.

The computation required to compute a single timestep of a model run is a function of the test case and of the problem resolution. The hardware performance monitor on a single processor of a Cray C-90 measured the following number of floating point operations (flops) per timestep for test case #2 using semi-implicit timestepping for the indicated resolutions:

flops	MM	NN	KK	NLON	NLAT	NVER
4,129,674	42	42	42	128	64	1
24,235,196	85	85	85	256	128	1
153,013,770	170	170	170	512	256	1

The measurements were run with the Cray compiler options that indicated the smallest number of flops. The complexity scales linearly with `NVER`, but is a more complicated function of the other parameters. These counts will increase when forcing is enabled, and will decrease (slightly) when explicit timestepping is used.

9. Benchmarking Methodology

We have developed a particular methodology for fair algorithm comparison and benchmarking:

to the extent feasible, compare tuned algorithms

PSTSWM was designed with this in mind, and is partially a product of research into what is "feasible". When using the code on a new platform, or after the platform has undergone a significant modification, we follow the following procedure:

1. determine best communication options and protocols for each parallel algorithm;
2. determine the best combination of tuned parallel FFT and parallel LT algorithms and the corresponding logical processor mesh for each total number of processors;
3. compare performance of the optimal parallel algorithms on each machine when making intermachine comparisons.

For more details, see [17] and [18].

These steps all involve running numerous experiments using one or a small number of problem input files, but many different algorithm input files. To support this methodology, PSTSWM will look for a file named `script` before reading in `problem`, `algorithm`, and `measurements`. `script` should contain, first, the number of experiments to run. Each subsequent line should contain the names of the files specifying the problem, algorithm, and measurements options for an experiment, as indicated in the following example.

```
3
"problem.t42.16" "alg.16.1.0.1.0" "mea.42.16"
"problem.t42.16" "alg.16.1.0.2.0" "mea.42.16"
"problem.t42.16" "alg.16.1.0.3.0" "mea.42.16"
```

Figure 9.1: Example experiment script file.

Not only does this make extensive parameter studies possible, but it also eliminates the necessity of reloading PSTSWM for every experiment. On some systems, loading the executable is very expensive, especially relative to the short execution times required in most of the parameter

studies. Note that the sequence of experiments controlled by `script` must all use the same total number of processors, but that the logical aspect ratio used can be varied.

10. Machine Specifics

For PSTSWM, we have attempted to make a portable code that will run efficiently on a variety of machines. These goals are somewhat conflicting, but, when possible, we have supported machine-specific peculiarities if they do not have negative impacts on the performance on other platforms. The following list is not exhaustive, listing Intel-motivated modifications primarily, but it indicates some of the issues that have been addressed in the code.

1. The Intel i860 family of microprocessors support IEEE-standard denormalized numbers, but very inefficiently. The spectral transform method often works with very small numbers, ones which have no effect on the solution accuracy. To avoid unnecessary performance degradation, a special routine is provided to set denormalized numbers to zero on i860 based machines. (In general, the `-noieee` compiler option also improves the performance of PSTSWM on Intel i860-based platforms.)

2. On the current version of the Intel Paragon, the first time through a timestep is much slower than later ones due to paging the instructions, data, and work storage into local memory. This is partially an artifact of the current state of the operating system, and is unimportant for long runs. To eliminate this, the first timestep is calculated once before timing is enabled, then the initial conditions are restored, timing is enabled, and the first timestep is calculated again.

3. The real solution to the paging problem on the Paragon, and a mechanism that improves performance throughout the model simulation, is to allocate the work space dynamically, using the ALLOCATE extension to Fortran. This requires a modification to two lines of code, but is nonportable, and is included on the Paragon machines using the C preprocessor and compile time switches.

4. An additional preprocessor test is used to include the multithreading constructs used on the Paragon MP node systems.

5. Performance on the T3D is sensitive to the number of distinct arrays (or cache lines) being updated in a single loop. To improve performance, some compute intensive loops have been split so that only one or two arrays are being updated simultaneously.

Note that we are willing to make additional changes to PSTSWM to make it run efficiently on other platforms (on a case-by-case basis). We also encourage vendors and other researchers

to insert library calls for the global communication routines, or other similar modifications to PSTSWM, to increase efficiency on a given platform. PSTSWM as it is written currently represents a generic version of the code. As part of our methodology research, we are interested in how much efficiency we give up by taking our approach.

11. Porting the Code

After passing PSTSWM source code through the C preprocessor `/lib/cpp`, it is a Fortran 77 code with the following "standard" extensions:

1. IMPLICIT NONE
2. DO WHILE .. ENDDO
3. DO .. ENDDO
4. variable names greater than 6 characters long
5. variables names containing underscore
6. COMMON containing both character and noncharacter variables

The code also calls the intrinsic function `XOR`. For systems that do not provide this function, like the nCUBE/2, a Fortran-callable C routine must be written.

All other porting issues deal with message-passing. If the MPI, PICL, or PVM libraries have already been implemented on the machine, then the port of PSTSWM should be simply a matter of generating a new machine-specific makefile. If these communication libraries do not exist, then the most useful approach, to the authors, is to port the PICL message-passing library, since this retains the full performance data collection functionality. If this is not feasible, reimplementing the basic message-passing routines is straightforward. Example native ports can be found in the `src/lib` subdirectory. Note that the only performance-sensitive interprocessor communication is in `sendrecv.f` and `swap.f`. If `sendrecv.f` and `swap.f` are reimplemented, please check for unsupported communication protocol options and output meaningful error messages.

Note that all of the message-passing routines, and all of PSTSWM, contain extensive internal documentation, specifying exactly what the code is doing, and why.

12. Conclusions and Future Plans

PSTSWM was developed for, and has been used in, numerous research projects, primarily in parallel algorithm research, the evaluation of early systems. benchmarking methodology, and performance modeling. It has also been used very effectively for finding bugs and performance problems in communication libraries. Other than ports to additional systems, PSTSWM's development is essentially complete, but we expect to continue using the code to test and evaluate parallel platforms and communication libraries.

Much of the code for the collective communication operations in PSTSWM is also being used in other application codes, in particular PCCM2, the message passing version of the Community Climate Model, providing these codes with both efficient communication algorithms and portability.

We are also developing new parallel algorithm testbeds for other numerical methods used in atmospheric circulation models, and many of the ideas (and some of the code) from PSTSWM will be reused.

13. ACKNOWLEDGMENTS

We are grateful to Ian Foster for his collaboration in developing the parallel algorithms that are embodied in PSTSWM, and acknowledge that, without his participation, this code would not have been written.

We are grateful to members of the CHAMMP Interagency Organization for Numerical Simulation, a collaboration involving Argonne National Laboratory, the National Center for Atmospheric Research, and Oak Ridge National Laboratory, for sharing codes and results. In particular, we thank Ruediger Jakob for his initial help in obtaining and understanding STSWM.

This research was performed using Intel iPSC/2 and iPSC/860 and IBM SP-2 multiprocessor systems at Oak Ridge National Laboratory; Intel Paragon XP/S 5, XP/S 35, and XP/S 150 systems located in the Oak Ridge National Laboratory Center for Computational Sciences, funded by the Department of Energy's Mathematical, Information, and Computational Sciences Division of the Office of Computational and Technology Research; the IBM SP-1, IBM SP-2, and Intel iPSC/860 at Argonne National Laboratory; the IBM SP-2 at NASA Ames; the nCUBE/2 and Intel Paragon systems at Sandia National Laboratories; the nCUBE/2 and nCUBE/2S systems at Ames National Laboratory; the Intel Touchstone DELTA System operated by Caltech on behalf of the Concurrent Supercomputing Consortium; a Cray T3D and a Cray Y/MP at Cray Research; and a Cray Y/MP at the National Energy Research Supercomputer Center.

14. BIBLIOGRAPHY

[1] D. H. BAILEY, E. BARSZCZ, J. T. BARTON, D. S. BROWNING, R. L. CARTER, L. DAGUM, R. A. FATOOHI, P. O. FREDERICKSON, T. A. LASINSKI, R. S. SCHREIBER, H. D. SIMON, V. VENKATAKRISHAN, AND S. K. WEERATUNGA, *The NAS Parallel Benchmarks*, Internat. J. Supercomputer Applications, 5 (1991), pp. 63–73.

[2] M. COMMITTEE, *MPI: a message-passing interface standard*, Internat. J. Supercomputer Applications, 8 (1994), pp. 165–416.

[3] J. J. DONGARRA AND E. GROSSE, *Distribution of mathematical software via electronic mail*, Comm. Assoc. Comput. Mach., 30 (1987), pp. 403–407.

[4] J. B. DRAKE, I. T. FOSTER, J. J. HACK, J. G. MICHALAKES, B. D. SEMERARO, B. TOONEN, D. L. WILLIAMSON, AND P. H. WORLEY, *PCCM2: A GCM adapted for scalable parallel computer*, in Fifth Symposium on Global Change Studies, American Meteorological Society, Boston, 1994, pp. 91–98.

[5] I. T. FOSTER, B. TOONEN, AND P. H. WORLEY, *Performance of parallel computers for spectral atmospheric models*, Tech. Report ORNL/TM–12986, Oak Ridge National Laboratory, Oak Ridge, TN, April 1995.

[6] I. T. FOSTER AND P. H. WORLEY, *Parallel algorithms for the spectral transform method*, Tech. Report ORNL/TM–12507, Oak Ridge National Laboratory, Oak Ridge, TN, May 1994.

[7] G. A. GEIST, A. L. BEGUELIN, J. J. DONGARRA, W. JIANG, R. J. MANCHEK, AND V. S. SUNDERAM, *PVM: Parallel Virtual Machine - A Users Guide and Tutorial for Network Parallel Computing*, MIT Press, Boston, 1994.

[8] G. A. GEIST, M. T. HEATH, B. W. PEYTON, AND P. H. WORLEY, *PICL: a portable instrumented communication library, C reference manual*, Tech. Report ORNL/TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990.

[9] J. J. HACK, B. A. BOVILLE, B. P. BRIEGLEB, J. T. KIEHL, P. J. RASCH, AND D. L. WILLIAMSON, *Description of the NCAR Community Climate Model (CCM2)*, NCAR Tech. Note NCAR/TN–382+STR, National Center for Atmospheric Research, Boulder, Colo., 1992.

[10] J. J. HACK AND R. JAKOB, *Description of a global shallow water model based on the spectral transform method*, NCAR Tech Note NCAR/TN-343+STR, National Center for Atmospheric Research, Boulder, CO, February 1992.

[11] M. T. HEATH AND J. A. ETHERIDGE, *Visualizing the performance of parallel programs*, IEEE Software, 8 (1991), pp. 29–39.

[12] P. PIERCE, *The NX message passing interface*, Parallel Computing, 20 (1994), pp. 463–480.

[13] R. A. VAN DE GEIJN, *On global combine operations*, LAPACK Working Note 29, Computer Science Department, University of Tennessee, Knoxville, TN 37996, April 1991.

[14] D. L. WILLIAMSON, J. B. DRAKE, J. J. HACK, R. JAKOB, AND P. N. SWARZTRAUBER, *A standard test set for numerical approximations to the shallow water equations on the sphere*, J. Computational Physics, 102 (1992), pp. 211–224.

[15] D. L. WILLIAMSON, J. T. KIEHL, V. RAMANATHAN, R. E. DICKINSON, AND J. J. HACK, *Description of NCAR community climate model (CCM1)*, NCAR Tech. Note NCAR/TN-285+STR, NTIS PB87–203782/AS, National Center for Atmospheric Research, Boulder, CO, June 1987.

[16] P. H. WORLEY, *A new PICL trace file format*, Tech. Report ORNL/TM–12125, Oak Ridge National Laboratory, Oak Ridge, TN, October 1992.

[17] P. H. WORLEY AND I. T. FOSTER, *Parallel Spectral Transform Shallow Water Model: a runtime–tunable parallel benchmark code*, in Proc. Scalable High Performance Computing Conf., IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 207–214.

[18] P. H. WORLEY, I. T. FOSTER, AND B. TOONEN, *Algorithm comparison and benchmarking using a parallel spectral transform shallow water model*, in Parallel Supercomputing in Atmospheric Science: Proceedings of the Sixth ECMWF Workshop on Use of Parallel Processors in Meteorology, World Scientific Publishing Co. Pte. Ltd., Singapore. (in press).

A. Problem Input Files

```
'0001'          / CHEXP
42              / MM
42              / NN
42              / KK
64              / NLAT
128             / NLON
16              / NVER
                / NGRPHS
                / A
                / OMEGA
                / GRAV
                / HDC
0.0             / ALPHA
2400.0          / DT
999.0           / EGYFRQ
1.0             / ERRFRQ
999.0           / SPCFRQ
120.0           / TAUE
                / AFC
.FALSE.         / SITS
                / FORCED
                / MOMENT
1.              / ICOND
```

Figure A.1: Example problem input file for test case #1.

```
'0002'          / CHEXP
42              / MM
42              / NN
42              / KK
64              / NLAT
128             / NLON
16              / NVER
                / NGRPHS
                / A
                / OMEGA
                / GRAV
                / HDC
0.0             / ALPHA
2000.0          / DT
999.0           / EGYFRQ
1.0             / ERRFRQ
999.0           / SPCFRQ
120.0           / TAUE
                / AFC
.TRUE.          / SITS
                / FORCED
                / MOMENT
2               / ICOND
```

Figure A.2: Example problem input file for test case #2.

```
'0003'          / CHEXP
42              / MM
42              / NN
42              / KK
64              / NLAT
128             / NLON
16              / NVER
                / NGRPHS
                / A
                / OMEGA
                / GRAV
                / HDC
1.5207963       / ALPHA
2000.0          / DT
999.0           / EGYFRQ
1.0             / ERRFRQ
999.0           / SPCFRQ
120.0           / TAUE
                / AFC
.TRUE.          / SITS
                / FORCED
                / MOMENT
3               / ICOND
```

Figure A.3: Example problem input file for test case #3

```
'0004'          / CHEXP
42              / MM
42              / NN
42              / KK
64              / NLAT
128             / NLON
16              / NVER
                / NGRPHS
                / A
                / OMEGA
                / GRAV
                / HDC
0.0             / ALPHA
2000.0          / DT
999.0           / EGYFRQ
1.0             / ERRFRQ
999.0           / SPCFRQ
120.0           / TAUE
                / AFC
.TRUE.          / SITS
.TRUE.          / FORCED
.FALSE.         / MOMENT
4               / ICOND
```

Figure A.4: Example problem input file for test case #4

```
'0005'          / CHEXP
42              / MM
42              / NN
42              / KK
64              / NLAT
128             / NLON
16              / NVER
                / NGRPHS
                / A
                / OMEGA
                / GRAV
                / HDC
0.0             / ALPHA
4000.0          / DT
1.0             / EGYFRQ
999.0           / ERRFRQ
1.0             / SPCFRQ
120.0           / TAUE
                / AFC
.TRUE.          / SITS
                / FORCED
                / MOMENT
5               / ICOND
```

Figure A.5: Example problem input file for test case #5

```
'0005'          / CHEXP
42              / MM
42              / NN
42              / KK
64              / NLAT
128             / NLON
16              / NVER
                / NGRPHS
                / A
                / OMEGA
                / GRAV
0.5E16          / HDC
0.0             / ALPHA
800.0           / DT
1.0             / EGYFRQ
999.0           / ERRFRQ
1.0             / SPCFRQ
120.0           / TAUE
                / AFC
.TRUE.          / SITS
                / FORCED
                / MOMENT
6               / ICOND
```

Figure A.6: Example problem input file for test case #6

B. PVM-only and MPI-only PSTSWM Implementations

A network PVM-only distribution of PSTSWM is available from

`http://www.epm.ornl.gov/chammp/pstswm`

and from the PARKBENCH benchmark suite distribution on `netlib`. The source code in this version is identical to that of the full distribution except that only the PVM implementations of the communication routines are retained. The distribution comes with an extensive README file, a driver makefile, and 3 subdirectories: `bin`, `input`, and `src`. The `input` subdirectory contains three example problem input files, corresponding to the small, medium, and large problems associated with all PARKBENCH compact application codes. It also contains the usual example algorithm and measurements input files. The `src` subdirectory contains no subdirectories, with the communication library-specific files and the platform-specific makefiles residing with all of the (other) source code. Currently, makefiles are provided only for the SUN and IBM RS/6000 workstations.

An MPI-only distribution of PSTSWM is also available from

`http://www.epm.ornl.gov/chammp/pstswm`.

The MPI-only distribution has the same structure as the PVM-only distribution. Currently, makefiles are provided only for the Intel Paragon, using mpich, and the IBM SP, using mpich or MPI-F.

INTERNAL DISTRIBUTION

1. T. S. Darland
2. J. B. Drake
3. G. A. Geist
4. K. L. Kliewer
5–9. M. R. Leuze
10. D. R. Mackay
11. C. E. Oliver
12–16. S. A. Raby
17–21. R. F. Sincovec
22. D. W. Walker
23–27. P. H. Worley
28. Central Research Library
29. Laboratory Records - RC
30–31. Laboratory Records Depart.
32. K-25 Applied Technology Library
33. ORNL Patent Office
34. Y-12 Technical Library

EXTERNAL DISTRIBUTION

35. David C. Bader, Environmental Sciences Division, Office of Health and Environmental Research, Office of Energy Research, ER-76, U.S. Department of Energy, Washington, DC 20585

36. David H. Bailey, NASA Ames, Mail Stop 258-5, NASA Ames Research Center, Moffet Field, CA 94035

37. Edward H. Barsis, Computer Science and Mathematics, P. O. Box 5800, Sandia National Laboratory, Albuquerque, NM 87185

38. Robert E. Benner, Sandia National Laboratories, MS 1109, Parallel Computing Science Dept. 1424, P. O. Box 5800, Albuquerque, NM 87185

39. Michael Berry, Department of Computer Science, University of Tennessee, 107 Ayres Hall, Knoxville, TN 37996-1301

40. Roger W. Brockett, Wang Professor of EE and CS, Division of Applied Sciences, 29 Oxford Street, Harvard University, Cambridge, MA 02138

41. Edward Brocklehurst, DITC, Bldg. 93, National Physical Laboratory, Teddington, Middlesex TW11 0LW, UNITED KINGDOM

42. Thomas A. Callcott, Director, The Science Alliance Program, 53 Turner House, University of Tennessee, Knoxville, TN 37996

43. Larry Dowdy, Computer Science Department, Vanderbilt University, Nashville, TN 37235

44. Ian Foster, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439

45. Geoffrey C. Fox, NPAC, 111 College Place, Syracuse University, Syracuse, NY 13244-4100

46. J. Alan George, Vice President, Academic and Provost, Needles Hall, University of Waterloo, Waterloo, Ontario N2L 3G1, CANADA

47. Myron Ginsberg, EDS Advanced Computing Center, 30500 Mound Road, Bldg. I-6, Warren, MI 48090-9055

48. Gene Golub, Computer Science Department, Stanford University, Stanford, CA 94305

49. John Gustafson, 236 Wilhelm, Ames Laboratory, Iowa State University, Ames, IA 50011

50. James J. Hack, National Center for Atmospheric Research, P. O. Box 3000, Boulder, CO 80307

51. Christian Halloy, Assistant Director of JICS, 104 South College, Joint Institute for Computational Science, University of Tennessee, Knoxville, TN 37996-1301

52. Michael T. Heath, National Center for Supercomputing Applications, 4157 Beckman Institute University of Illinois, 405 North Mathews Avenue, Urbana, IL 61801-2300

53. Tom Henderson, NOAA/Forecast Systems Laboratory, R/E/FS5, 325 Broadway, Boulder, CO 80303

54. John L. Hennessy, CIS 208, Stanford University, Stanford, CA 94305

55. Anthony J. G. Hey, Dept. of Electronics and Computer Science, University of Southampton, Highfield, Southampton SO9 5NH, UNITED KINGDOM

56. Dan Hitchcock, ER-31, Mathematical, Information, and Computational Sciences Division, Office of Computational and Technology Research, Office of Energy Research, U.S. Department of Energy, Washington, DC 20585

57. Hans Kaper, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Bldg. 221, Argonne, IL 60439

58. Kenneth Kennedy, Department of Computer Science, Rice University, P. O. Box 1892, Houston, Texas 77001

59. Tom Kitchens, ER-31, Mathematical, Information, and Computational Sciences Division, Office of Computational and Technology Research, Office of Energy Research, Washington, DC 20585

60. Peter D. Lax, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012

61. Robert Malone, Los Alamos National Laboratory, C-3, Mail Stop B265, Los Alamos, NM 87545

62. James McGraw, Lawrence Livermore National Laboratory, L-306, P. O. Box 808, Livermore, CA 94550

63. David B. Nelson, Associate, Director, Office of Computational and Technology Research, ER-30, Office of Energy Research, U.S. Department of Energy, Washington, DC 20585

64. Joseph Oliger, Computer Science Department, Stanford University, Stanford, CA 94305

65. James M. Ortega, Department of Applied Mathematics, Thornton Hall, University of Virginia, Charlottesville, VA 22901

66. James C. T. Pool, Deputy Director, Caltech Concurrent Supercomputing Facility, California Institute of Technology, MS 158-79, Pasadena, CA 91125

67. Daniel A. Reed, Computer Science Department, University of Illinois, Urbana, IL 61801

68. Ahmed Sameh, Department of Computer Science, 200 Union Street, S.E., University of Minnesota, Minneapolis, MN 55455

69. Richard K. Sato, National Center for Atmospheric Research, P. O. Box 3000, Boulder, CO 80307

70. David S. Scott, Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006

71. Robert Schreiber, RIACS, MS 230-5, NASA Ames Research Center, Moffet Field, CA 94035

72. Burton Smith, Tera Computer Company, 400 North 34th Street, Suite 300, Seattle, WA 98103

73. David Snelling, Centre for Novel Computing, Department of Computer Science, University of Manchester, Manchester M13 9PL, UNITED KINGDOM

74. Paul N. Swarztrauber, National Center for Atmospheric Research, P. O. Box 3000, Boulder, CO 80307

75–79. Brian Toonen, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439

80. Jim Tuccillo, Cray Research, Inc., 200 Westpark Drive, Suite 270, Peachtree City, GA 30269

81. Robert Ward, Head, Computer Science Department, University of Tennessee, 107 Ayres Hall, Knoxville, TN 37996-1301

82. Andrew B. White, Los Alamos National Laboratory, P. O. Box 1663, MS-265, Los Alamos, NM 87545

83. Hans Zima, University of Vienna, Institute for Statistics and Computer Science, Brunner Str. 72, 1210 Vienna, AUSTRIA

84. Office of Assistant Manager for Energy Research and Development, U.S. Department of Energy, Oak Ridge Operations Office, P. O. Box 2001, Oak Ridge, TN 37831-8600

85–86. Office of Scientific & Technical Information, P. O. Box 62, Oak Ridge, TN 37831