Parallel Spectral Transform Shallow Water Model: A Runtime–Tunable Parallel Benchmark Code

P. H. Worley

Oak Ridge National Laboratory
P.O. Box 2008
Oak Ridge, TN 37831-6367

I. T. Foster

Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439

Abstract

Fairness is an important issue when benchmarking parallel computers using application codes. The best parallel algorithm on one platform may not be the best on another. While it is not feasible to reevaluate parallel algorithms and reimplement large codes whenever new machines become available, it is possible to embed algorithmic options into codes that allow them to be "tuned" for a particular machine without requiring code modifications. In this paper, we describe a code in which such an approach was taken.

PSTSWM was developed for evaluating parallel algorithms for the spectral transform method in atmospheric circulation models. Many levels of runtime-selectable algorithmic options are supported. We discuss these options and our evaluation methodology. We also provide empirical results from a number of parallel machines, indicating the importance of tuning for each platform before making a comparison.

1 Introduction

Benchmarking parallel (and sequential) computers is a varied activity. It includes determining low level machine and system characteristics, measuring the performance of important or representative kernels, and measuring the performance of full or compact application codes [2]. It is also an activity beset with many difficulties, for example, when trying to fairly compare/contrast the performance of different computer systems.

There are special difficulties when using full or compact application code benchmarks in interplatform comparisons. While it is important to measure the performance of specific parallel algorithms in the context of the application codes, it is equally important to measure the performance in solving the stated problem, where the parallel algorithm is allowed to vary between platforms or as the system characteristics evolve. One solution to this problem is the 'paper and pencil' benchmark, as exemplified by the NAS benchmark suite [1]. For such a benchmark, the numerical algorithm is specified, but the parallel implementation is left to the researcher or the vendor. But it can be very time-consuming to develop parallel application codes from scratch, and paper benchmarks are most useful for algorithmic kernels.

In this paper, we describe the Parallel Spectral Transform Shallow Water Model (PSTSWM) compact application code. PSTSWM supports a significant amount of runtime-selectable algorithm tuning, allowing both the parallel algorithm and the communication protocol used in the code to be specified at runtime. When using PSTSWM for benchmarking, we are able to measure how quickly the numerical simulation can be run without fixing the parallel implementation, but still executing the same numerical code. Thus we consider PSTSWM to be a compromise between paper benchmarks and the usual fixed benchmarks. We have used PSTSWM successfully to determine efficient parallel algorithms for a variety of multiprocessor platforms and to compare performance across platforms. We feel that encapsulating multiple parallel algorithm options into a code is a very powerful approach to benchmarking.

2 PSTSWM

PSTSWM is a message-passing parallel implementation of the sequential Fortran 77 code STSWM 2.0 [7]. STSWM solves the nonlinear shallow water equations on a rotating sphere using the spectral transform method. The nonlinear shallow water equations constitute a simplified atmospheric-like fluid prediction model that exhibits many of the features of more complete models, and that has been used to investigate numerical methods and benchmark a number of machines. The spectral transform algorithm of STSWM follows closely how CCM2, the NCAR Community Climate Model, handles the dynamical part of the primitive equations [6].

PSTSWM differs from STSWM in one major respect: fictitious vertical levels have been added in order to get the correct communication and computation granularity for three dimensional weather and climate codes, and to allow parallel algorithms that decompose the vertical dimension to be evaluated. The number of vertical levels is an input parameter. In all other respects, we changed the algorithmic aspects of STSWM as little as possible. While the algorithmic structure of STSWM was maintained, the code structure was altered radically in developing PSTSWM. These changes were made to support the algorithm comparison, and to support reuse of the code developed in implementing the parallel algorithms.

PSTSWM uses the spectral transform method to solve the shallow water equations. 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 is a tensor product longitude-latitude-vertical grid. The spectral domain is the set of spectral coefficients in a truncated spherical harmonic expansion of the state variables. When a "triangular" truncation of the spectral coefficients is used, as is common, the indices for the spectral coefficients for a single vertial level make up a triangular array.

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.

For more details on the steps in solving the shallow water equations using the spectral transform algorithm, see [7].

3 Parallel algorithm options

Parallel algorithms are used to compute the FFTs and to compute the vector sums used to approximate the forward and inverse Legendre transforms. There are four "levels" of runtime options supported in the code.

3.1 Processor grid

The physical and spectral domains are decomposed over a logical two-dimensional processor mesh of size $P_X \times P_Y$. The longitude dimension of the physical domain is decomposed over P_X processors and the latitude dimension is decomposed over P_Y processors. 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 total number of processors and the logical aspect ratio, and thus the number of processors dedicated to computing the different parallel transforms, are input parameters. While "squarish" meshes are generally best, the FFT and LT algorithms have different computational and communication complexities, and nonsquare meshes are sometimes better, depending on the problem size and the multiprocessor.

The mapping of the logical grid to physical processors is also a runtime option. A set of standard mappings is provided, suitable for hypercube and for mesh interconnect topologies, as well as the options to input an arbitrary mapping.

3.2 Parallel algorithms

Two classes of parallel algorithms are available for each transform: distributed algorithms, using a fixed data decompostion 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 Legendre 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 [4]. Note that the simple characterizations described here ignore link contention in the physical network, and, for example, the $\Theta(\log Q)$ distributed Legendre transform algorithm is not automatically better than the $\Theta(Q)$ distributed algorithm.

In summary, PSTSWM provides 12 different parallel algorithms for implementing the spectral transform method: 3 FFT $\times$ 4 LT.

3.3 Parallel algorithm variants

Each FFT and LT parallel algorithm also has a number of implementation options that can be selected at runtime.

$\Theta(Q)$ transpose. This algorithm proceeds in $Q-1$ steps on Q processors, where, at each step, a processor sends $1/Q$ of its data to another processor [8]. To be efficient, some care must be taken with the order of the data communication. We support two different schedules in PSTSWM. In the XOR schedule, processor q swaps data with processor XOR(q, i) at step i. This avoids competition for bandwidth in hypercube interconnection networks [8], and is often more efficient than a general send/receive pattern on other networks. In the MOD schedule, processor q sends to processor MOD$(q + i, Q)$ at step i.

The most robust and portable implementation of the $\Theta(\log Q)$ transpose is to pair each send request with a receive request. But, since all the data to be sent is available at the beginning of the transpose, it can also all be sent immediately (*send-ahead*), delaying any receive requests until later. Depending on the details of the underlying message passing system and the size of the problem, send-ahead can be an efficient technique, or it can consume all system buffer space and cause deadlock.

Since all data that is received is retained and where the data is to go is known beforehand, all receive requests can be posted before the send requests (*receive-ahead*) if nonblocking receives are supported by the native message passing system. Completion of the receive requests would then be checked after individual send requests, or after all the send requests in a send-ahead implementation.

In summary, PSTSWM supports two different communication schedules (denoted by the indicated symbol): XOR (X) and MOD (M), and four different communication options: send/receive (0), recv-ahead/send (1), send-ahead/receive (2), and receive-ahead/send-ahead(3).

$\Theta(\log Q)$ transpose. The schedule for this algorithm is fixed. Also, each send involves data received at the previous step, so a send-ahead implementation is not possible. But, if extra buffer space is allocated, the receive requests can be posted early, hopefully eliminating some buffer copying by guaranteeing that messages are placed in user buffer space when they arrive. Both send/receive (0) and receive-ahead/send (1) implementations are supported.

Distributed FFT. Walker, et al. [10] showed that on computers that permit overlapping of computation and communication, it can be advantageous to divide the single block FFT into two and interleave the communication for one block with the computation for the other. PSTSWM supports both overlap (O) and nonoverlap (N) versions of the distributed FFT.

$\Theta(Q)$ distributed LT. The computation preceding the communication in this algorithm can be delayed and interleaved with the communication steps, enabling significant overlap of communication and computation. Even if a computer does not support concurrent computation and communication, overlap techniques can still overlap idle time and computation, but contention for shared resources such as the system bus may reduce overall performance. Also, if extra buffer space is allocated, the receive requests can be posted early. PSTSWM supports both overlap (O) and nonoverlap (N) and both send/receive (0) and receive-ahead/send (1) implementations of this algorithm.

$\Theta(\log Q)$ distributed LT. Overlap of communication and computation is not efficient for this algorithm. But, if extra buffer space is allocated, the receive requests can be posted early. Both send/receive (0) and receive-ahead/send (1) implementations are supported.

3.4 Communication protocols

All of the parallel algorithms and their implementations are based on SWAP and SENDRECV primitives, where, during each communication step, a processor sends a message to one processor and receives a message from another. The send-ahead and receive-ahead implementations use subcommands, e.g. SWAP1, SWAP2, and SWAP3, that together are equivalent to the basic primitive.

SWAP and SENDRECV have a number of different communication protocols that can be selected at runtime. For brevity, we will restrict our discussion to the SWAP primitive, but the same options also apply to SENDRECV.

First, in a SWAP two processors exchange messages. In a simple SWAP, each processor sends a message and then receives one. In an ordered SWAP, one processor sends while the other receives, after which the first processor receives and the second processors sends. The ordered SWAP will work on synchronous message passing systems and potentially avoids some buffer copying. The simple swap takes advantage of the full bandwidth in a bidirectional link and concurrency in initiating the SWAP on the two processors.

For each of the two types of SWAP, the exchange can be implemented with blocking or nonblocking sends, blocking or nonblocking receives, and regular or forcetype messages, as supported by the native message passing system. The forcetype messages, available on Intel systems, do without the normal handshaking protocol that guarantees that messages are not lost if they arrive before the corresponding receive request is made. In certain circumstances, the user can guarantee that a receive request will be posted before the send, in which case the forcetype message is much more efficient. Correctness is guaranteed in the SWAP and SENDRECV commands when the forcetype option is chosen.

For the ordered SWAP command, there is also a *synchronous* option, where handshaking messages are used to guarantee that each processor is ready to receive a message before it is sent.

In summary, PSTSWM supports six "simple" options and seven "ordered" options. We denote the simple options by Sn, for some number n, and the ordered options by On. The simple options are

0: blocking send/blocking receive,

1: nonblocking send/blocking receive,

2: blocking send/nonblocking receive,

3: nonblocking send/nonblocking receive,

4: blocking send/nonblocking receive using forcetypes, and

5: nonblocking send/nonblocking receive using forcetypes.

The ordered options are the same six plus

6: synchronous blocking send/blocking receive.

Note that some options are not supported for some algorithm variants. For example, it is meaningless to specify a receive-ahead algorithm that uses blocking receives.

4 Tuning methodology

PSTSWM incorporates so many options that a comprehensive study of all possible combinations of algorithm parameters, problem size, and processor count is unreasonable. Instead, we proceed in two stages.

In the first stage, we perform a series of tuning experiments designed to identify an "optimal" set of algorithm variants and communication protocol parameters for each FFT and LT algorithm. These tuning experiments are performed with either $P_X = 1$ or $P_Y = 1$, i.e., one-dimensional decompositions, allowing the FFT and LT algorithms to be studied in isolation. One or more of the problem dimensions is reduced to provide the computation granularity and communication volumes that would obtain in a two-dimensional decomposition. For example, when evaluating the distributed FFT algorithm for a 16×8 processor grid on a problem with 16 vertical levels, a 16×1 processor grid is used with the number of vertical levels reduced by a factor of 8 (from 16 to 2).

The number of experiments performed in this first stage is further reduced by first evaluating the full range of parameters only on "largest" and "mid-sized" multiprocessor configurations for each parallel algorithm. For example, $P = 512$ and $P = 64$ are considered on the 512-processor Intel Paragon at Oak Ridge National Laboratory. If these studies all show one set of communication parameters and algorithm variants to be consistently superior, no further experiments are performed. We expect the impact of communication parameters on performance to be relatively insensitive to the problem granularity and number of processors, and have found this to be true in many cases. When there is a significant difference, the purpose of the tuning exercise must be taken into account. For benchmarking "peak" performance, we give preference to the parameters that worked best for the larger configurations. In all cases, subsequent experiments for a given platform and parallel algorithm use a fixed set of communication parameters. The experiments performed in the first stage are also used to eliminate FFT and LT algorithms from further consideration (on a given platform) if it is clear that they are not competitive.

Once we have identified optimal communication parameters and promising parallel algorithms, we perform a second series of experiments that compares all possible combinations of the remaining FFT and LT algorithms on all possible aspect ratios for each power of two number of processors supported by each computer. For example, on the Intel Paragon, we use 1, 2, 4, 8, 16, 32, 64, 128, 256, and 512 processors; for 32 processors, we try the aspect ratios 1×32, 2×16, 4×8, 8×4, 16×2, and 32×1. While powers of two are not necessary, they reduce the number of aspect ratio experiments to a reasonable number without giving up much information. Also, the distributed FFT and $\Theta(\log Q)$ transpose algorithms require that the number of processors be a power of two.

5 Empirical results

In this section we describe empirical results that indicate the importance of algorithm tuning on various machines when benchmarking using PSTSWM. Other papers will describe the algorithm comparison and benchmarking results in detail. Here, we concentrate on the variance in timings observed during the tuning experiments.

We describe experiments run on the four parallel computer systems listed below. These systems vary considerably in their communication or computational capabilities, despite similarities in their architectures and programming models.

Name	Processor	Network	P
nCUBE/2	nCUBE 2	hypercube	1024
iPSC/860	i860	hypercube	128
Paragon:			
OSF/1	i860SP	16×32 mesh	512
SUNMOS	i860SP	16×32 mesh	512

The nCUBE/2 and the Intel iPSC/860 use hypercube interconnects in which each of P processors is connected to $\log_2 P$ neighbors via bidirectional links. The Intel Paragon use a two-dimensional bidirectional mesh interconnect and cut-through routing. Measurements were run on the Paragon using both R1.1.2, an OSF-based operating system, and SUNMOS, a low overhead operating system developed at Sandia National Laboratories and the University of New Mexico. Both of these operating systems are still evolving, and any conclusions as to their performance or optimal tuning parameters will be short-lived.

Low level tuning. The best algorithm variants and communication protocols for the different parallel algorithms are listed below. The optimal parameters are denoted by (a, b), where a is the algorithm variant and b is the communication protocol. Both a and b are encoded as indicated in earlier sections. The multiprocessors are denoted as follows: nCUBE/2

(N), iPSC/860 (I), Paragon-OSF (P), and Paragon-SUNMOS (S).

Transpose Algorithms	N	I	P	S
$\Theta(Q)$	(X2,S0)	(X0,S5)	(X1,S5)	(X0,S0)
$\Theta(\log Q)$	(0,S0)	(0,S5)	(0,S5)	(0,O6)

Distributed Algorithms	N	I	P	S
FFT	(O,O0)	(O,S5)	(N,S5)	(N,O6)
$\Theta(Q)$	(O0,S0)	(O1,S5)	(O1,S4)	(O1,S2)
$\Theta(\log Q)$	(0,S0)	(0,S5)	(1,S4)	(1,S5)

The important thing to notice in these tables is that the optimal parameters do differ between platforms. Note that the best parameters for one platform may not be supported on another. For example, the native message passing system on the nCUBE/2 does not support nonblocking sends or receives, nor does it have a special forcetype protocol.

Of more relevance to this paper is the sensitivity of performance to the parameter selection. To indicate this, the following tables give statistics about the range of timings for two problem sizes: twelve timesteps each of T42L16, which has a physical computational grid of size $128\times64\times16$, and T85L32, which has a physical computational grid of size $256\times128\times32$. T42 is the design point for CCM2 [6], and T85 is representative of the size of problem that climate researchers would like to solve next. The tuning experiments reported here were designed assuming a logical 16×32 processor for the nCUBE/2 and Intel Paragon, and assuming a logical 8×16 processor mesh for the Intel iPSC/860. For example, for the Intel Paragon, a 1×32 logical processor grid is used and the problem size is scaled by 1/16 when evaluating parallel LT algorithms, to get the correct granularity. For brevity, only the transpose FFT tuning experiments are reported. The other transpose experiments have similar statistics.

The statistics used are the first quartile (Q1) and largest time (MAX), both normalized by the minimum time. The MAX statistic indicates how much is to be gained by tuning. The Q1 statistic gives some indication as to how important it is to get the optimal parameters. For example, for the $\Theta(\log Q)$ transpose algorithm on the iPSC/860, the worst case and the first quartile are both between 40% and 50% worse than the optimal algorithm, indicating that there is a strong optimum. In contrast, on the Paragon, the first quartile is only 3% worse than the optimum, and, while tuning is worthwhile, getting the optimal parameters is not as important.

$\Theta(Q)$ Transpose FFT Algorithm	N	I	P	S
# of Options	8	50	46	50
T42L16 - Q1	1.03	1.22	1.11	1.05
T42L16 - MAX	1.16	1.38	1.56	1.49
T85L32 - Q1	1.03	1.24	1.11	1.01
T85L32 - MAX	1.10	1.37	1.24	1.10

$\Theta(\log Q)$ Transpose FFT Algorithm	N	I	P	S
# of Options	3	19	18	19
T42L16 - Q1	–	1.40	1.03	1.05
T42L16 - MAX	1.14	1.45	1.29	1.53
T85L32 - Q1	–	1.40	1.03	1.01
T85L32 - MAX	1.12	1.48	1.37	1.12

Distributed FFT Algorithm	N	I	P	S
# of Options	5	21	18	21
T42L16 - Q1	1.00	1.30	1.04	1.03
T42L16 - MAX	1.21	1.76	1.29	1.50
T85L32 - Q1	1.00	1.27	1.05	1.04
T85L32 - MAX	1.17	1.69	1.37	1.09

$\Theta(Q)$ Distributed LT Algorithm	N	I	P	S
# of Options	7	64	61	64
T42L16 - Q1	1.13	1.44	1.20	1.10
T42L16 - MAX	1.41	1.76	1.97	1.85
T85L32 - Q1	1.10	1.29	1.09	1.09
T85L32 - MAX	1.31	1.70	1.63	3.97

$\Theta(\log Q)$ Distributed LT Algorithm	N	I	P	S
# of Options	3	19	18	19
T42L16 - Q1	–	1.35	1.07	1.08
T42L16 - MAX	1.15	1.46	1.21	1.73
T85L32 - Q1	–	1.29	1.06	1.01
T85L32 - MAX	1.11	1.40	1.22	1.12

As can be seen from the data, tuning is very important for performance for some algorithm/multiprocesor combinations, and is relatively unimportant for others. Similarly, for some algorithms and multiprocessors, there are many parameter settings that produce good results, while for others there are few. The way that the statistics change with problem granularity is also strongly algorithm and machine dependent. All of the algorithm variants and communication protocols used in these experiments are expected to be reasonable choices in some environment, and cannot be dismissed out of hand. In particular, the optimal parameters for one of our target machines are sometimes bad choices for a different target, as will be described in an expanded paper. Thus, given the complexity of the behavior, we feel that empirical low level tuning experiments are an important step in benchmarking when using PSTSWM.

Note that twelve timsteps results in .5-20 second times for these experiments. We have observed longer runs to have exactly the same behavior, and are confident in the validity of these experiments. While there is some variability in the run times for some of the experiments, all of the good parameter settings show little variability in the time to compute a timestep, and the minimum timings are very consistent.

High level tuning. Unlike the low level tuning parameters (algorithm variants and communication protocols), it is unreasonable to choose a single aspect ratio or parallel algorithm for all problem sizes and machine configurations. The different parallel algorithms have different asymptotic behaviors, and the best choices vary significantly as problem and machine sizes change.

For climate modeling, the problem size seldom changes. Once the optimal low level parameters have been determined, it is straightforward to evaluate all relevant machine configurations to determine the best aspect ratio and parallel algorithms for a given number of processors. This information can then be saved and used whenever a given number of processors becomes available for a run.

We defer the complete description of the algorithm comparison to [4], and only discuss the high level tuning for the largest configurations on each of the target machines.

The best parallel algorithm combinations and logical aspect ratios are listed below. The parallel algorithms are denoted as follows: $\Theta(Q)$ transpose (TQ), $\Theta(\log Q)$ transpose (TL), distributed FFT (DF), $\Theta(Q)$ distributed Legendre transform (DQ), and $\Theta(\log Q)$ distributed Legendre transform (DL). Remember that a $P_X \times P_Y$ aspect ratio indicates, among other things, that P_X processors collaborate to compute a block FFT and P_Y processors collaborate to compute a block LT.

T42L16	N	I	P	S
FFT	DF	TQ	TL	TL
LT	DL	DQ	TL	DL
aspect	32×32	16×8	16×32	16×32
T85L32	**N**	**I**	**P**	**S**
FFT	TQ	TQ	TQ	TQ
LT	TQ	DQ	TQ	TQ
aspect	32×32	16×8	32×16	32×16

As before, the important point to be drawn from this data is that the best algorithms can differ with the problem size and the multiprocessor. In almost all cases, there is a cluster of "good" algorithms close to the optimal one, and the particular one mentioned here is not particularly important. But the members of this "optimal set" vary with problem size and between multiprocessors.

The following tables give statistics about the sensitivity of performance to the high level tuning. Since large numbers of processors allow us to examine extreme logical aspect ratios that are expected to be poor performers, we give both the largest time (MAX) and the largest time in the set of experiments using 1:1, 1:2, and 2:1 aspect ratios (MAXSQ). We also give the time for a "generic" algorithm (GEN), one that has optimal asymptotic behavior and a simple protocol that is supported on all message passing systems that the authors have experience with. This algorithm uses $\Theta(Q)$ transpose FFT and LT algorithms, a 1:1 or 1:2 logical aspect ratio, a MOD communication schedule, and an ordered, blocking communication protocol, i.e., (M0,O0) using our earlier encoding. The generic algorithm is meant to represent what a reasonable choice would have been if we had forgone tuning. As before, all statistics are normalized by the minimum execution time.

	N	I	P	S
T42L16 - GEN	1.48	1.53	1.43	1.95
T42L16 - MAXSQ	2.11	1.59	2.47	2.20
T42L16 - MAX	3.95	7.69	6.40	6.75
T85L32 - GEN	1.12	1.50	1.19	1.06
T85L32 - MAXSQ	1.92	1.46	2.28	1.92
T85L32 - MAX	3.95	2.75	4.43	8.74

As would be expected, the generic algorithm is very good for the cases where the optimal algorithm is the tuned version of $\Theta(Q)$ transpose FFT/$\Theta(Q)$ transpose LT, and is significantly worse otherwise. The MAXSQ values indicate that high level tuning can lead to significant improvements (50%-100%). As before, all of the parallel algorithms, if not the aspect ratios, are expected to be reasonable choices in some environment, and cannot be dismissed out of hand. Even nonsquare aspect ratios are optimal for smaller numbers of processors. See [4] for more details.

6 Summary and conclusions

We found runtime tuning options to be a useful technique for determining efficient programming techniques (low level tuning) and for determining appropriate parallel algorithms (high level tuning). Our experience with PSTSWM has also convinced us that conventional benchmark studies, using fixed algorithms, can be very misleading. PSTSWM is not an exceptionally complex code, yet relatively minor changes in the algorithms can change performance by a factor of two or more. In particular, for climate modeling research, a fair comparison and evaluation of multiprocessors requires the use of optimized parallel algorithms.

Structuring PSTSWM so as to support multiple parallel algorithms efficiently was not a simple task, but maintaining and porting the code to other message passing systems will be straightforward. PSTSWM is primarily written in standard Fortran 77, with message passing implemented using the PICL message passing libraries [5]. All message passing is encapsulated in 3 high level routines for broadcast, global minimum, and global maximum, two classes of low level routines representing variants and/or stages of the swap operation and the send/receive operation, repsectively, and one synchronization primitive. Porting the code to another message passing system will require simply porting the PICL library or reimplementing the few communication routines in PSTSWM using either native message passing primitives or the proposed MPI message passing standard [9], which supports all of the communication protocols currently implemented in PSTSWM.

A similar effort to ours to introduce runtime algorithm options into a parallel code may not always be reasonable, but some options are simple to insert, and can make a big difference on some machines. It is generally a good idea to encapsulate message passing primitives in parallel codes [3], and once that is accomplished, a variety of communication protocols and algorithm variants can be embedded easily. In summary, it is our opinion that the low level tuning described in this paper is a generally applicable technique that should be used before commencing any benchmarking effort on message passing systems. Depending on the application code, it may also be useful to support a variety of multiple algorithms in the same code.

Acknowledgements

This work was supported in part by the Atmospheric and Climate Research Division and in part by the Applied Mathematical Sciences Research Program, both of the Office of Energy Research, U.S. Department of Energy.. 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, to Brian Toonen for help with the computational experiments, and to the SUNMOS development team for help in getting PSTSWM running under SUNMOS on the Intel Paragon.

This research was performed using Intel iPSC/860 and Paragon multiprocessor systems at Oak Ridge National Laboratory (ORNL), nCUBE/2 and Intel Paragon systems at Sandia National Laboratories, and the nCUBE/2 system at Ames National Laboratory.

References

[1] D. H. Bailey, *et al.*, *The NAS Parallel Benchmarks*, Internat. J. Supercomputer Applications, 5 (1991), pp. 63–73.

[2] J. J. Dongarra and W. Gentzsch, eds., *Computer Benchmarks*, North-Holland, Amsterdam, 1993.

[3] I. T. Foster, *Language constructs for modular parallel programs*, MCS-P391-1093, Argonne National Laboratory, Argonne, IL, 1993.

[4] I. T. Foster and P. H. Worley, *Parallel algorithms for the spectral transform method*, ORNL/TM–12507, Oak Ridge National Laboratory, Oak Ridge, TN, May 1994.

[5] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley, *PICL: a portable instrumented communication library*, ORNL/TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990.

[6] J. J. Hack, *et al.*, *Description of the NCAR Community Climate Model (CCM2)*, NCAR/TN–382+STR, National Center for Atmospheric Research, Boulder, Colo., 1992.

[7] J. J. Hack and R. Jakob, *Description of a global shallow water model based on the spectral transform method*, NCAR/TN-343+STR, National Center for Atmospheric Research, Boulder, CO, February 1992.

[8] S. L. Johnsson and C.-T. Ho, *Algorithms for matrix transposition on boolean N-cube configured ensemble architectures*, SIAM J. Matrix Anal. Appl., 9 (1988), pp. 419–454.

[9] D. W. Walker, *The design of a standard message passing interface for distributed memory concurrent computers*, Parallel Computing, (1994). (to appear).

[10] D. W. Walker, P. H. Worley, and J. B. Drake, *Parallelizing the spectral transform method. Part II*, Concurrency: Practice and Experience, 4 (1992), pp. 509–531.