U. S. Department of Commerce
National Oceanic and Atmospheric Administration
National Weather Service
National Centers for Environmental Prediction
4700 Silver Hill Road, Mail Stop 9910
Washington, DC 20233-9910

Technical Note

Running WAVEWATCH III on a Linux cluster[†].

Hendrik L. Tolman [‡]
SAIC-GSO at
Environmental Modeling Center
Marine Modeling and Analysis Branch

September 2003

THIS IS AN UNREVIEWED MANUSCRIPT, PRIMARILY INTENDED FOR INFORMAL EXCHANGE OF INFORMATION AMONG NCEP STAFF MEMBERS

[†] MMAB Contribution No. 228.

[‡] e-mail: Hendrik.Tolman@NOAA.gov

This page is intentionally left blank.

Abstract

The present report assesses the parallel performance of the WAVE-WATCH III wave model a Linux cluster. This model is developed for parallel computer architecture, but implicitly assumes the presence of a parallel file system to assure the integrity of all raw data output files. Alternative output methods for non-parallel file systems are discussed, included in the model and tested.

Acknowledgments. The author thanks Henrique Alves for comments on early drafts of this report, and Bryan Lewis and Sami Mkaddem of Rocketcalc for their help in finding optimal compilers and communication packages for the WAVE-WATCH III code on their hardware. The present study was made possible by funding from the NOAA High Performance Computing and Communication (HPCC) office.

This report is available as a pdf file from

http://polar.ncep.noaa.gov/waves

Contents

Abstract i
Acknowledgments ii
Table of contents iii

1 Introduction 1

2 Hardware and software setup 3

3 Parallel measures and prognosis 5

4 WAVEWATCH III version 2.22. 11

5 I/O modifications. 17
5.1 Model version 2.22. 17
5.2 Modifications for non-parallel file systems. 19
5.3 Timing results. 21

6 Summary and conclusions 25

References 27

This page is intentionally left blank.

1 Introduction

This study describes testing and modification of the WAVEWATCH III wave model (Tolman, 2002c) for running on a Linux cluster. WAVEWATCH III was originally designed for running on supercomputers at NCEP. Its performance has been documented in, for instance, Tolman et al. (2002), and Moon et al. (2003). Its efficiency on NCEP's parallel computers is documented in Tolman (2002a). From its inception code transparency and portability has been deemed important. The resulting code has been applied successfully on a wide range of computer hardware, including single processor, shared memory multiprocessor, and distributed memory multiprocessor computer architectures.

An important reason to enforce portability has been that typically smaller computers have been used to develop WAVEWATCH III. For instance, the distributed memory version of WAVEWATCH III is based on the Message-Passing Interface (MPI) standard for communication between processors (e.g., Gropp et al., 1997). The first MPI version of WAVEWATCH III was mostly developed on a single processor Linux machine, which mimicked a multiprocessor machine using the portable MPICH package[1]. Similarly, most of the tests of WAVEWATCH III version 2.22 as reported in Tolman (2002b) were run on an SGI™ 1450 server, consisting of four 700 MHz Intel Xeon chips with 2 Mb of shared memory. All these tests were run using the shared memory parallel option of the model, using the OpenMP standard.

NCEP recently purchased a new Linux cluster to facilitate wave model development. The cluster is a Rocketcalc™ Titan T8-240, consisting of 8 2.4 GHz Intel Xeon Processors arranged on four mother boards, with 2 Mb of shared memory on each board. The boards are interconnected using a copper Gigabit Ethernet switch, and use a disk pack residing on a front end Linux PC. To run a parallel FORTRAN code, the Portland compiler version 4.1-1 and the LAM-MPI[2] package version 6.5.9 are installed. Both were recommended by the vendor after extensive benchmarking with WAVEWATCH III.

The present report presents timing and implementation issues for WAVEWATCH III on this Titan cluster. In section 2, hardware and software setup are discussed, including the test cases used. In section 3, measures for the level of parallel behavior are discussed, together with expected parallel behavior of the wave model. Much of these two sections are an extension to Tolman (2002a), which will henceforth be denoted as T02a. In section 4 timing results of the latest release of WAVEWATCH III (version 2.22) are presented. Although this version is highly portable, it assumes the presence of a truly parallel file system. On many clusters, such a file system is not available. In section 5 modifications to WAVEWATCH III are discussed, which assure that all output passes through

[1] http://www-unix.mcs.anl.gov/mpi/mpich/

[2] http://www.lam-mpi.org/

a designated processor. After the proper code modifications are discussed, the impact of several approaches on the model economy are discussed. A summary and conclusions are presented in section 6.

2 Hardware and software setup

To test the performance of WAVEWATCH III on the Rocketcalc Titan T8-240 cluster, the global operational model of NCEP is used. A description of this test case can be found in T02a. Because of the limited number of processors on the cluster, it is sufficient to reduce the forecast range of the test from 24 h in T02a to 6 h in the present study. A copy of the test case, including source code, makefile, input files and timing scripts can be found on the WAVEWATCH ftp server[3], and is stored with other distribution material for model version 2.22. Because this test is primarily intended for assessing computational speed on alternative hardware, only the original model version 2.22 code without output is available in this way.

The tests considering model version 2.22 (section 4) are preformed without any model output being generated, other than diagnostic output identifying the progress of the calculations. The proper treatment of all standard model output, and its effect on model efficiency is assessed in section 5.

The wave model has been compiled with the pgf90 compiler using the following compiler options:

-fastsse -tp p7 -byteswapio

The first option enables the SSE instruction set, which vectorizes parts of the code. Because the code is vectorizable, This speeds up the code by a factor of 2 to 3. Note that this compiler option is not implemented in earlier versions of the Portland compiler. The second option enables instructions particular for the Pentium III chip, but was found to have minimal impact. The last option forces binary I/O to use the big endian format. This is necessary because the input files are taken directly from NCEP's IBM systems, which use the big endian IEEE format for binary files.

The new Intel™ chips include a 'hyper-threading' feature (Marr et al., 2002). Hyperthreading is more generically referred to as 'simultaneous multi-threading.' The idea is to keep the many processing units within one processor as busy as possible. Rather than using complicated optimization code in the compiler and OS, the OS is tricked into seeing two virtual processors and the on-chip scheduler is left to the task of keeping the processor busy. With hyper-threading switched on, the virtual cluster contains 16 instead of 8 processors, and in principle two parallel processes should be loaded on each individual processor. This overcommitment is important in assessing parallel behavior of a system. In the following section, a systematic distinction is therefore made between (parallel) processes, and actual processors.

[3] see http://polar.ncep.noaa.gov/waves/wavewatch/wavewatch.html#documentation for access to ftp server.

Any parallel application requires a loading scheme, i.e., a way to distribute the parallel processes over actual processors. Most MPI applications use a resource file in which a list of 'machines' is identified, together with the number of processes (m) to be loaded onto each machine. In the present cluster, each motherboard is treated as a separate machine with two processors. In the following such a unit will be called a node.

There are several ways to define the resource list on the present cluster. Each node could be assigned with $m = 1$, which would imply that the first process goes to node 1, the second goes to node 2, and so on until all nodes have one process assigned. The scheduler then goes back to the top of the list to assign another process to each node, until all processes are assigned. Because there are two processors per node, another logical way to distribute processes over nodes is by setting $m = 2$. In this case, processes 1 and 2 go to the first node, processes 3 and 4 to the second node, etc. Finally, hyper-threading suggests that $m = 4$ could be used. In the timing test, all these options, with and without hyper-threading have been tried.

3 Parallel measures and prognosis

The basis of assessing or predicting the parallel behavior of a code is Amdahl's law. This law will first be discussed in terms of processes, assuming that sufficient resources are available to assigns no more than one process to each processor (as in T02a). After this discussion, the effects of sparse resources (more processes than processors) will be discussed.

Amdahl's law states the the shortest possible run time for a parallel code with n processes ($\tilde{t}_n$) can be calculated from the run time of a single process t_1 and the fraction p of the code that is parallel as

$$\tilde{t}_n = (1-p)\, t_1 + p\, \frac{t_1}{n} = t_1 \left(1 - p\, \frac{n-1}{n} \right) \quad . \tag{3.1}$$

The actual run time with n processes (t_n) is always larger

$$t_n = t_o + \tilde{t}_n = t_o + t_1 \left(1 - p\, \frac{n-1}{n} \right) \quad , \tag{3.2}$$

where t_o represent the parallel overhead time, consisting of additional computer time required for communication between processes. The overhead time also includes effects of imbalances in the amount of work to be done at each individual process. From Eq. (3.2), the parallel fraction of the code can be estimated from measured run times t_1 and t_n as

$$p = \frac{n}{n-1} \left(1 - \frac{\tilde{t}_n}{t_1} \right) = \frac{n}{n-1} \left(1 - \frac{t_n - t_o}{t_1} \right) \quad , \tag{3.3}$$

which also requires an assessment of the overhead time t_o. This overhead time generally cannot be estimated or measured in an accurate way. Instead the effective parallel fraction p_e can be estimated by assuming that $t_o = 0$,

$$p_e = \frac{n}{n-1} \left(1 - \frac{t_n}{t_1} \right) \quad . \tag{3.4}$$

Note that ignoring t_o leads to an overestimation of $\tilde{t}_n$ in Eq. (3.3), and hence in an underestimation of p. Hence, by definition

$$p_e \leq p \quad . \tag{3.5}$$

In Fig. 5 of T02a the effective parallel fraction for WAVEWATCH III on NCEP's IBM supercomputer is presented. For the present study, results of up to 8 processes are most relevant. For such numbers n, p_e ranges from 0.91 to about 0.96. Deviations from the ideal value of $p_e \approx 1$ are shown to be mostly due to communication overhead. The latter is furthermore shown to generally diminish with increasing n.

The implications of Amdahl's law as discussed above are only valid in cases where resources are not a limiting factor. In other words, if there are a sufficient number of processors available to assign no more than one process to each processor. Particularly in the case of hyper-threading on the present cluster, resources will be systematically overcommitted, i.e., more than one process will be assigned to each processor. In this case, load balancing in terms of processes per processor counteracts Amdahl's law.

For convenience immediately considering the effective parallel fraction p_e, the run time per process for n processes ($\hat{t}_n$) becomes [Cf. Eq. (3.1)]

$$\hat{t}_n = t_1 \left(1 - p_e \, \frac{n-1}{n} \right) \quad . \tag{3.6}$$

For an overcommitted system, however, $\hat{t}_n$ does not represent the actual run time t_n. The latter is governed by both $\hat{t}_n$ and the number of processes per processor on the most heavily loaded node of the system (α).

$$t_n = \alpha \, \hat{t}_n = \alpha \, t_1 \left(1 - p_e \, \frac{n-1}{n} \right) \quad , \tag{3.7}$$

and the corresponding effective parallel fraction of the code is estimated from measured run times t_1 and t_n as

$$p_e = \frac{n}{n-1} \left(1 - \frac{t_n}{\alpha \, t_1} \right) \quad . \tag{3.8}$$

Note that $\alpha \geq 1$, but that α can be a fraction. For instance, when 3 processes are assigned to a node with 2 processors, $\alpha = \frac{3}{2}$.

The value of α depends on the configuration of the cluster, and on the loading scheme used. Let N be the total number of processors on the cluster, M_p be the number of processors per node, and M_n the number of nodes ($N \equiv M_p M_n$). Let furthermore n again be the total number of processes, which are assigned to individual nodes in groups of size m (section 2). The maximum number of processes per node n_n and α then become

$$n_n = \{ n \div (m M_n) \} \; m + \min \{ \mathrm{mod}(n, m M_n), m \} \quad , \tag{3.9}$$

$$\alpha = n_n / M_p \quad , \tag{3.10}$$

where $\div$ denotes integer division.

A final measure for parallel performance as used in T02a is the bulk parallel efficiency

$$\zeta = \frac{t_1}{n t_n} \tag{3.11}$$

which represents the ratio of the actual speed up of the code to the ideal speed up for $p \equiv 1$. This measure was also defined without considering limited resources.

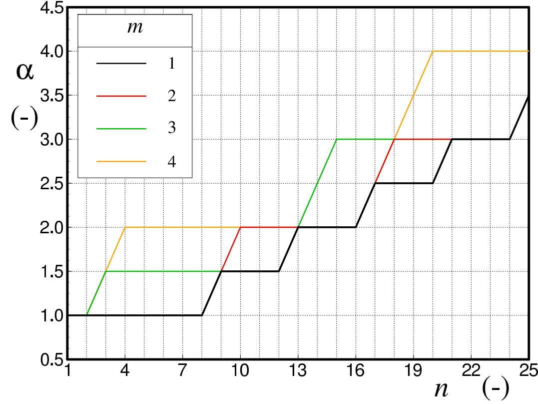

Fig. 3.1 : Process load per processor α as a function of the number of processes n for several loading schemes (m) for the present cluster with $M_n = 4$, $M_p = 2$ and $N = 8$.

One way of making this measure more meaningful for applications with limited resource is to replace it by

$$\zeta = \frac{t_1}{\min(n, N) t_n} \tag{3.12}$$

The latter definition will be used here.

In the remainder of this section, the above relations will be used to predict parallel model behavior on the present cluster.

For the cluster considered here, $M_n = 4$, $M_p = 2$ and $N = 8$. As discussed in the previous section, logical choices for m for this system would be 1, 2 or 4. In Fig. 3.1, α is presented for m ranging from 1 through 4 and for a number of processes up to $n = 25$.

For $n \leq 8$ and $m \leq 2$, $\alpha \equiv 1$. In these conditions, the system is not overcommitted. For any larger n, the system is always overcommitted. As could be expected, the smallest load factors α are always found for $m = 1$. Note that for the in principle undercommitted cases with $n < 8$ but $m > 2$, larger values of α are found. This points to the obvious load imbalance created by loading more than M_p processes on some nodes, while other nodes are underused.

Figure 3.2 presents expected run times t_n and bulk parallel efficiencies ζ as a function of m and n as in Fig. 3.1. The different panels correspond to different (constant) effective parallel fractions p_e.

Results for the ideal parallel system ($p_e \equiv 1$, upper panels) identify some fairly trivial features of (overcommitted) parallel applications. As long as $n \leq N$ and $m \leq M_p$ (no more processes than processors in total or per node), ideal parallel behavior can be achieved ($\zeta \equiv 1$, right panel of Fig. 3.2a). If however, the system is overcommitted in total ($n > N$) or per node ($m > M_p$), ideal parallel behavior

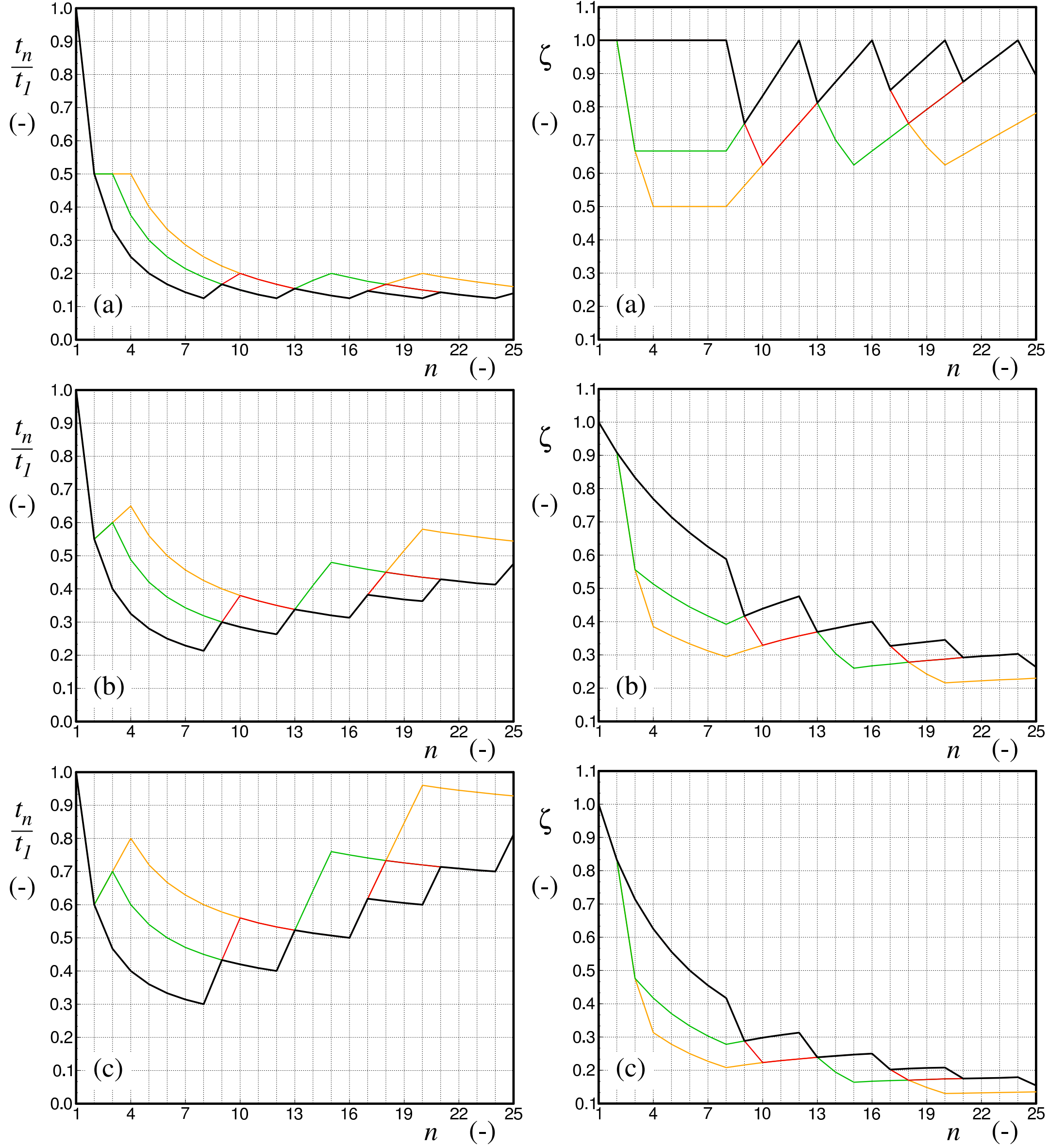

Fig. 3.2 : Predicted run time t_n [Eq;. (3.7), left panels] and overall parallel efficiency ζ [Eq. (3.12), right panels] as a function of the number of processes n for several parallel fractions p. (a) $p_e = 1$, (b) $p_e = 0.9$, (c) $p_e = 0.8$. Legend as in Fig. 3.1.

is only achieved in specific conditions, and generally $\zeta < 1$. An overcommitted parallel application shows ideal parallel behavior $\zeta \equiv 1$ only if each node is loaded identically, which implies that

$$n = \beta \, M_n \, m \quad \cup \quad (n \leq N \, \cap \, m \leq M_p) \quad , \tag{3.13}$$

where β is a positive integer number. For instance for $m = 1$ this requires $n = 1 - 8$, 12, 16, 20, 24, ..., for $m = 2$ it requires $n = 1 - 8$, 16, 24, ..., for $m = 3$ it requires $n = 12$, 24, ..., and for $m = 4$ it requires $n = 16$, 32, ...

For realistic parallel applications, the effective parallel fraction p_e is always less than 1. Examples of projected run times t_n and bulk parallel efficiencies ζ are presented in Figs. 3.2b and c for $p_e = 0.9$ and $p_e = 0.8$, respectively. For overcommitted systems, $m = 1$ and loadings schemes satisfying Eq. (3.13) still give the most efficient parallel behavior. However, due to the impact of the non-parallel contribution in Amdahl's law, any run time for $n > N$ will always be larger than the optimum run time for $n = N$ and $m \leq M_p$ (left panels in Figs. 3.2b and c).

On first inspection, the latter finding might be interpreted as suggesting that it is never prudent to overcommit the processors. Another argument against overcommitment is that generally the operating system overhead increases a little for overcommitted processors due to for instance, frequent swapping of processes in and out of memory. This would effectively reduce p_e with increasing n (or α). Counterarguments are found in the design of hyper-threading (see section 2), and in the fact that the WAVEWATCH III code has been shown to display an increasing parallel behavior (increasing p_e) with increasing n (T02a, Fig. 5 and discussion). It is, therefore, still prudent to test the parallel behavior of WAVEWATCH III for $n > N$.

This page is intentionally left blank.

4 WAVEWATCH III version 2.22.

Based on the results of the previous sections, the parallel performance of WAVEWATCH III on the Titan cluster is assessed by running a six-hour segment of NCEP's global wave model. The timing runs consider 1 through 16 processes n, loading schemes with $m = 1$, 2 or 4, and with hyper-threading switched on or off. For each case, 10 individual runs are made. Actual run times t_n as a function of the number of processes n are presented in Fig, 4.1. It should be noted that run times are measured from within the source code. This implies that the time estimates do not include overhead for getting a parallel system started (Cf. T02a). This guarantees that the results are scalable to longer model runs, and hence makes it feasible to do timing runs with relatively short simulations.

A comparison of Fig. 4.1 with the left panels in Fig. 3.2 shows that the wave model displays much of the expected timing behavior. Particularly obvious are the impacts of the loading scheme (m) on the run times, as extensively discussed in section 3. However, differences can also be observed. Particularly interesting is the observation that t_n in Fig. 4.1 does not appear to show the systematic increase associated with Amdahl's law for $n > 8$ as predicted in Fig. 3.2. This can be explained by the increasing parallel fraction p_e with increasing n, as already shown in T02a, and as discussed in more detail below.

Results with or without hyper-threading (right and left panels of Fig. 4.1, respectively), show largely the same behavior. Nevertheless, two significant differences stand out. First, hyper-threading in general seems to result in more variability in the timing results, as indicated by the minima and maxima of t_n (thin lines). Secondly, timings for the test with hyper-threading seem marginally slower than without. To illustrate the latter, mean run time results for all six test cases (thick lines in Fig. 4.1) are gathered in one figure in Fig. 4.2. The latter figure clearly indicates that run times t_n with hyper-threading (red lines) are systematically larger than without (black lines). Both differences between runs with and without hyper-threading are small, but due to their systematic nature, there seems to be no justification to switch on hyper-threading for the WAVEWATCH III model. Therefore, only results obtained without hyper-threading are discussed in more detail below.

From the timing results t_n the effective parallel fractions p_e can be calculated using Eq. (3.8). The corresponding results are presented in Fig. 4.3. Note that this figure does not show a clear impact of loading characteristics as found in the raw run times t_n in Fig. 4.1. This is due to the fact that the loading characteristics are explicitly accounted for in Eq. (3.8) through α. For $n > 4$, the parallel efficiency shows minimal impact of the loading scheme (m), and shows a systematic increase with n. For $n = 5$, $p_e \approx 0.92$. For $n = 16$, $p_e \approx 0.97$. This compares well with NCEP's IBM supercomputer, for which the respective values for p_e are 0.92 and 0.98 (T02a, Fig. 5).

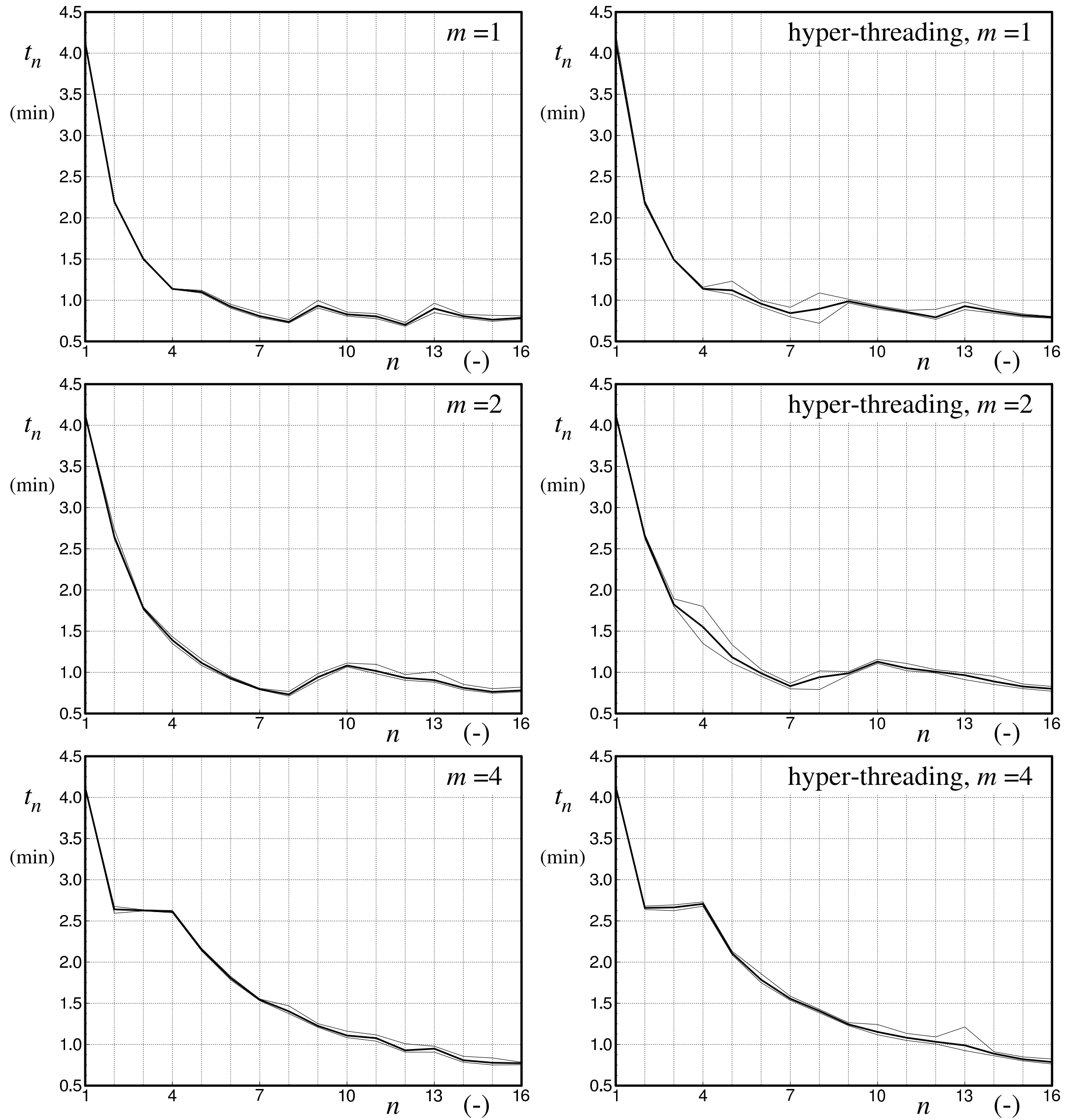

Fig. 4.1 : Measured run times t_n as a function of the number of processes n for several loading parameters m and with or without hyper-threading as indicated in the panels. Thick line: average over 10 runs. This lines: minima and maxima.

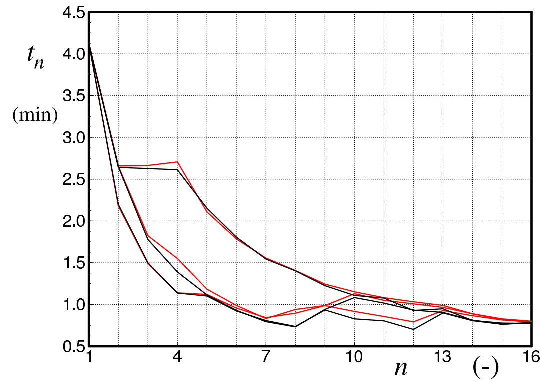

Fig. 4.2 : Composite of mean measured run times t_n from Fig. 4.1 (thick lines). Black lines without hyper-threading, read lines with hyper-threading.

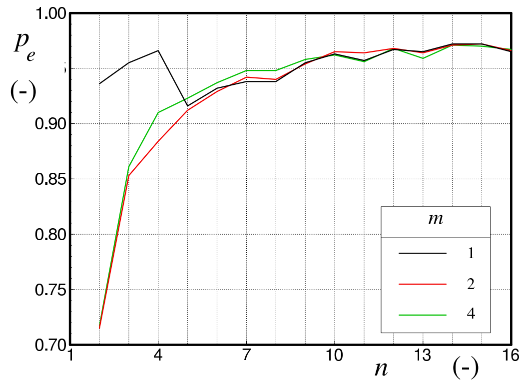

Fig. 4.3 : Effective parallel fractions p_e as a function of the number of processes n for various loading schemes m. Based on mean run times for 10 runs, no hyper-threading.

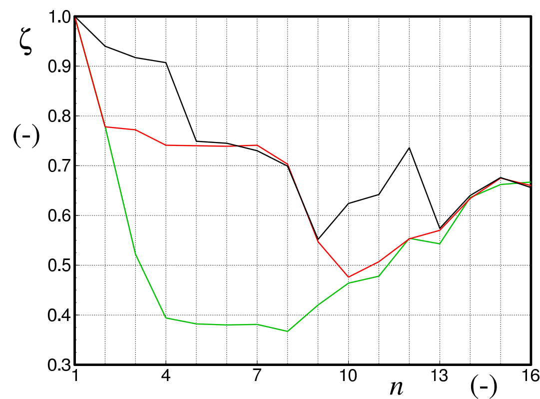

Fig. 4.4 : Bulk parallel efficiency ζ as a function of the number of processes n for various loading schemes m. Based on mean run times for 10 runs, no hyper-threading. Legend as in Fig. 4.3

Significant differences in parallel fractions as a function of m are found only for $n \leq 4$. In this range of n, the results for $m = 1$ deviate significantly from the results obtained with the other loading schemes, and show systematically higher parallel fractions p_e. This behavior appears to be a consequence of the characteristics with $n \leq 4$ and $m = 1$, where the first four processes all are assigned to a node with two processors. Hence, one processor is dedicated to the wave model, whereas the other processor on the node provides resources for the communication software. As soon as the fifth process is assigned to node 1, both resulting wave model processes are loaded on the two processors of the node, and have to contend with the communication software for resources. This results in a reduction of the effective parallel fraction as displayed in the figure. Even though for $n = 5$ this happens on only a single node, it will become the bottleneck in computational effort and hence dictate the run time and p_e.

It should be noted that similar behavior with larger parallel fractions p_e for $n \leq 4$ was found in T02a for NCEP's supercomputer. In T02a, however, this is not attributed to the loading scheme, but to the more efficient communication within a single node (see T02a for details).

Finally, Fig. 4.4 shows the bulk parallel efficiency ζ [Eq. (3.12)] as a function of the number of processes n and the loading scheme (m). Due to the definition of ζ, it again shows a distinct impact of the loading scheme. If the system is not overcommitted ($n \leq N$ and $m \leq 2$), the bulk efficiency $\zeta > 0.7$, with $\zeta \approx 0.70$ for a fully utilized system ($n = N$). Based on results of the previous section, the latter would be expected to correspond to the shortest run times of the code. However, overcommitting the system with $n = 12$ and $m = 1$ results in the shortest run times and hence a better bulk efficiency $\zeta \approx 0.74$. This is confirmed in Fig. 4.2, although not obvious due to the scaling in this figure.

To facilitate timing comparisons with other hardware, the actual timing results for $m = 1$ without hyper-threading are presented in Table 4.1. Extrapolating the results for the 6 h runs to the run time of the operational model at NCEP (7 days and 6 h), suggests that this operational model can be run on the Titan cluster in 20 min. This has been validated with several runs (figures not presented here).

Table 4.1: Timing results for tests with $m = 1$ and without hyper-threading.

n (-)	$t_{n,\min}$ (s)	$t_{n,\text{avg}}$ (s)	$t_{n,\max}$ (s)	p_e (-)	ζ (-)
1	246.80	247.45	248.95	—	—
2	130.99	131.68	132.40	0.936	0.940
3	89.46	89.95	90.76	0.955	0.917
4	68.05	68.19	68.36	0.966	0.907
5	65.11	66.04	67.19	0.916	0.749
6	54.11	55.32	56.99	0.932	0.745
7	47.23	48.42	50.74	0.938	0.730
8	43.35	44.25	45.84	0.938	0.699
9	54.26	56.02	59.71	0.955	0.552
10	48.35	49.60	51.27	0.963	0.624
11	46.54	48.21	50.24	0.957	0.642
12	41.00	42.04	43.80	0.967	0.736
13	50.94	53.91	57.78	0.965	0.574
14	47.04	48.35	49.61	0.972	0.640
15	44.52	45.75	48.93	0.972	0.676
16	46.32	47.17	48.61	0.965	0.656

This page is intentionally left blank.

5 I/O modifications.

5.1 Model version 2.22.

WAVEWATCH III generates diagnostic output identifying the progress of the model run, and output of raw data for later processing. The diagnostic output consists of two parts. The actual wave model subroutine (w3wave) maintains a log file (log.ww3), and the shell around this routine (then main wave model driver ww3_shel) writes output to standard output (stdout). In the MPI version of the code the diagnostic output to both stdout and to the file log.ww3 is performed by the first processor assigned to the parallel job. This implies that the file log.ww3 will be written to the file system as seen by the first processor, and that the stdout output will be written to the console as seen by the first processor.

On the present cluster setup used here, a single file system residing on the front end machine is NFS mounted on all nodes of the cluster, and hence the file log.ww3 will automatically appear on this file system, when the program is started from a directory on this file system. Depending on the actual setup of a cluster, proper capturing of stdout output may require that the parallel job is started from the actual first node or processor assigned to the parallel job run.

The raw data output of WAVEWATCH III consists of five data types, that can be controlled independently (with starting time, ending time and increment, or switched off altogether):

1) Fields of mean wave parameters for the entire spatial grid of the wave model.
2) Output of full spectral data for selected output points. The spectra are interpolated from the four surrounding discrete grid points in the spatial grid.
3) Output along tracks. Full spectral data for grid points surrounding a user-supplied track in space and time are saved in a file.
4) Restart files. All spectra for all spatial 'wet' grid points as well as all additional data needed to restart the calculation is written to file.
5) Boundary data. Spectra needed as boundary conditions for nested runs are written to one or more files.

Output types 1 and 2 require data to be gathered from several processors before the output can be performed. Therefore, these output files are written by a single processor in version 2.22 of the model. Because the work load on the higher numbered processors typically is slightly less than on the lower numbered processors, the field output is controlled by processor $n-1$, while the point output is controlled by process $n-2$, where n again is the number of processes assigned to the model run. Although not strictly necessary from a logistics perspective, the output of boundary data (type 5) is also performed by a single process (process

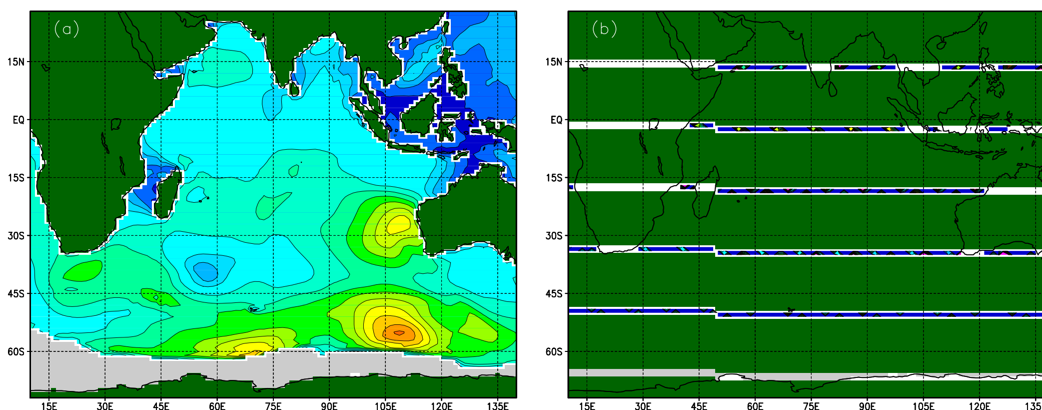

Fig. 5.1 : Wave height field from operational global NWW3 model for the Indian Ocean for Aug. 11, 2003, 06z, as generated from the properly written retest file valid for that time (panel a), or from a corrupted file written using $n = 8$ (panel b). Color scheme as on web site.

$n - 3$). As long as the file system from which the program is executed is visible from all nodes, output types 1, 2 and 5 will therefore generate the expected output. If the file systems is not shared between nodes, the corresponding output files will need to be gathered from the appropriate node.

The restart and track output files (type 3 and 4) are direct accesses files where each processor directly writes spectra to the proper record in the file. The restart files furthermore contain several fields of parameters on the full spatial grid. The latter fields are gathered on processor n, and written to the restart file by this processor. Retaining the integrity of such direct access files when many processes write to the file at the same time requires a truly parallel file system. A simple NFS mount of a file system across all nodes of a cluster, as is the case on our Titan cluster, does not provide a proper parallel file system.

The break down of the integrity of the restart file on a non-parallel file system is obvious from the model behavior. This is illustrated in Fig. 5.1, which shows the wave height fields for the Indian Ocean from the operational NWW3 model at NCEP for a hindcast for Aug. 11, 2003, 06z as it should be (panel a) and as it is reconstructed from a corrupted restart file valid for the same time. The corrupt restart file was generated on by a model run with $n = 8$ processes. Note that the actual corruption of the file depends on the order in which processors try to write the file, and hence is not fully reproducible.

The track output files are generated in a similar manner, using a direct access file containing one full spectrum combined with its environmental data per record. On a non-parallel file system, the track output file will be corrupted in the same way the restart file is corrupted.

These problems require changes to th model source code to avoid the gen-

eration of corrupted direct access files in parallel machines without parallel file systems, as described in the following section.

5.2 Modifications for non-parallel file systems.

With respect to the restart files, it is desirable to retain the direct access file structure, because it allows for fast reading of restart data for a model run on multiple processors, and because it facilitates a change in the number of processors between consecutive model runs. The obvious solution to guarantee file integrity is to have the entire file written by a single process. This process then needs to receive all necessary data through MPI communications. The actual data storage structure has a significant impact on how this communication can most easily be performed.

If the WAVEWATCH III model is run in MPI mode using n processes, and the model grid contains I discrete spectra, each processor only has $I_l = I \div n$ or $(I \div n) + 1$ spectra stored in memory. If i is a counter over all spectra I, j is a counter over the locally stored spectra I_l, and k is the rank number of the processes, ranging from 1 through n, the storage scheme used in WAVEWATCH III is such that (Tolman, 2002c, page 117)

$$i = k + (j-1)n \quad , \tag{5.1}$$

$$j = 1 + (i-1) \div n \quad , \tag{5.2}$$

$$k = 1 + \mathrm{mod}(i-1, n) \quad . \tag{5.3}$$

In the restart file, consecutive records contain spectra with consecutive counters i. Only every n^{th} spectrum from the full wave model grid are locally stored in each process with a continuous counter j to save memory.

Considering this, a simple and memory efficient way to generate a data server for the restart file is to send each local storage array in full to the data server process. This server process will receive one such array at at time, and write it to file before processing the data from the next processor. This approach generates a fairly limited additional usage of memory on the server node, because only a single storage array from other nodes is needed at each time. The disadvantage of this method is that the spectra are not written contiguously, but in a 'striped' mode; instead of writing spectra (or records) i, $i+1$, $i+2$ ... in sequence, the spectra are written in the order i, $i+n$, $i+2n$ .... This way of writing is known to be very inefficient for virtually any file system.

To test the efficiency of writing the restart file, 6h forecast runs were made with or without writing this file (with all other data output options switched off). The differences in run time were then used as an estimate of time needed to generate this file. Test results are presented in Table 5.1. From the timing results it is obvious that the striped writing takes two orders of magnitude longer

Table 5.1: Timing estimates of writing a single restart file using different methods. n is the number of processes, t is the time needed to write the file.

Method of writing	n (-)	t (s)
Single processor code (cont. write)	1	1.8
Striped writing	8	230
Cont. writing, all data to server	8	3.3
Cont. writing, blocked data at server (using 10 blocks)	8	3.6
Cont. writing, blocked and buffered data (using 10 blocks, 2 buffers)	8	2.5

than continuous writing, confirming the well known inefficiency of the stripped writing.

The most simple way to set up a data server process that can write the data contiguously, is to first gather all spectra to the server, and then write them out in order. This approach is expected to reduce the time needed to write significantly, but has the disadvantage that it requires much additional memory for the server process. As is shown in Table 5.1, this method indeed is much faster than the striped writing technique, and comes to within a factor of two of the time needed to write the file without communication (single processor results).

The major disadvantage of the method where all spectra are gathered at a single processor is the large increase of memory needed. This can be alleviated by gathering blocks of contiguous data on the target process, and making this block an order of magnitude smaller than the total spectral storage used. This blocked storage can then be reused until all data is written. This method proved to give similar results as the method where all data is gathered before writing (see Table 5.1), with a modest increase of time to generate the file due to additional communication overhead and synchronization between processes. The time needed to generate the file still is within twice the optimal time for a single processor model version.

The results in Table 5.1 suggest that about half the time needed to write the restart file with the blocked method is spent on communication. The necessary communication time can conceptually be reduced by using buffering techniques, where one buffer is being filled, while another is being written to disk. This technique is also fairly easy to implement, once the blocking has been set up properly. This technique indeed reduces the writing time further (see Table 5.1), and brings it close to the writing time required for a single processor set up. This technique removes some of the communication overhead at the cost of a modest

increase in memory use. The latter can be reduced further my increasing the number of blocks while leaving the number of buffers unchanged. This blocked and buffered writing technique will be available as in forthcoming distribution versions of WAVEWATCH III. The non-server version will also remain available as it might still be useful if memory usage is an issue.

The track output file was originally written to a direct access file from each process simultaneously to avoid the need of moving individual spectra to a target processor. The present experience with the restart file makes it clear that this concept is not efficient, and that a gathering of all data at a single process before writing will be more efficient in almost any case. Rather than adding a data server option to the generation of this output file, the model has been modified to always process the track output file through process $n-4$. With this modification, there is no reason to use a direct access file structure. The raw data file is therefore converted to a sequential file for both disk usage and writing time economy. Because this conversion is fairly trivial, it's effects will not be discussed in any detail here.

With the above modifications to the restart and track output files, WAVEWATCH III can be run on arbitrary clusters. It may nevertheless still be inconvenient that the different output files are all written by different processes. In cases where it is preferred that all files are written by the same process to the same disk pack, an option has been added to the code to have all output routed through the first process assigned to the model. It should be noted that this removes possible parallelisms in writing different files from different processes (see T02a). The impact of this option on writing times for files will be assessed in the following section.

5.3 Timing results.

In this section some timing results for writing output from the wave model are presented. The amount of data produced by the wave model varies enormously per application. NCEP's global wave model will again be used as a representative case. With this model a one day run is made producing the following output, similar to the output of the operational global model.

1) Hourly output (24 times) of 10 fields of input and mean wave parameter on the spatial grid.
2) Hourly output (24 times) of 86 full spectra at selected output locations.
3) No output along tracks.
4) A single restart file (29,792 spectra and two fields of mean wave parameters).
5) Hourly output (24 times) of boundary data to three separate files. The total number of spectra in these files is 284.

Table 5.2: Time in seconds needed to write different output types for a one day run as described in the text. Average time based on 99 runs for each model setup, with std in brackets. All output types processed by different processes. The run time of the model without output is 158.7 (2.7) s. Producing all data requires 5.8 (4.4) s.

Output type	alone	with fields	with points	with restart	with bound.
Fields	1.8 (4.2)		2.1 (4.2)	6.1 (4.5)	2.8 (4.5)
Points	-0.6 (3.7)	2.1 (4.2)		2.8 (4.3)	-0.8 (3.6)
Restart	2.7 (4.2)	6.1 (4.5)	2.8 (4.3)		3.7 (4.2)
Boundary	-0.1 (3.8)	2.8 (4.5)	-0.8 (3.6)	3.7 (4.2)	

Table 5.3: Like Table 5.2 with all output processed by the first process. The run time of the model without output is 158.5 (2.9) s. Producing all data requires 6.6 (3.9) s

Output type	alone	with fields	with points	with restart	with bound.
Fields	1.8 (3.7)		2.6 (4.4)	6.0 (3.9)	2.5 (4.2)
Points	-0.6 (4.0)	2.6 (4.4)		2.2 (4.2)	0.1 (4.8)
Restart	2.1 (4.2)	6.0 (3.9)	2.2 (4.2)		2.4 (4.3)
Boundary	-0.6 (4.1)	2.5 (4.2)	0.1 (4.8)	2.4 (4.3)	

Note that this model normally runs for 7 days and 6 hours. The amount of data written and hence the time required to write scales linearly with the run time for all outputs, except for the restart file. In the operational wave model only one restart file is written independent of the forecast range of the model. Hence, the time estimate for writing the restart file as presented here is an absolute time, whereas all other times are times per day of forecast.

The results of model tests using individual processes for each type of output, or with having all output processed by the first process are presented in tables 5.2 and 5.3, respectively. All results have been obtained with $n = 8$ processes and with the loading scheme with $m = 1$.

Both Tables clearly indicate that the time required to write the different output files is of the same order of magnitude as the variability in the run times, or even significantly smaller. This generally makes the estimated write times statistically unreliable, and is the reason for using a large number of runs (99) to make run time estimates. Even without a detailed statistical analysis, it is clear that there is still a significant sampling error in the timing estimates, for instance

from comparing the run times of the models without output (which should be identical), or from the models with only one type of output (which should be nearly identical). The sampling error also most likely is the main reason for the negative writing time estimates in the tables. Note that the timing results presented in Table 5.1 are statistically much more reliable, because the run time variability is reduced by the shorter forecast period. The corresponding std of the run time estimates was typically well below 1 s in the latter case.

The fact that the run time estimates are statistically unreliable may be considered disappointing, because it does not allow for a detailed investigation of the times needed to generate the output. However, one of the reasons why the results are unreliable is due to a positive result from these tables: generally, write times required to generate output files from the wave model are insignificant, adding less than 5% to the run times. Note that this was not necessarily the case for the restart files before the data server options were implemented (see Table 5.1).

Even with the sampling variability, some observations can be made considering times needed to write different types of output. First, the restart file is the biggest file written (72.5 Mb), followed by the field output file (41.5 Mb), the boundary data files (three files, in total 16.6 Mb) and the point output file (5.3 Mb). The two biggest files require the larges writing times, the smallest two 'disappear in the noise'. Note that a track output files would typically generate data amounts between the boundary data and point output files, and thus is also expected to result in inconsequential write times. Differences between the table are too small to have statistical significance, and will therefore not be discussed.

It should be noted that the data server version of the restart file might also be beneficial for use on parallel file systems. This has been tested briefly by applying the test of Table 5.1 on NCEP's IBM supercomputer, also using $n = 8$ processors. The computational run time on this system is about 41s, comparable to the run time for 8 processes on the present cluster. In the non-server mode, the writing of the restart file takes 8.6s, and in the data server model it takes 2.5s. Hence, the data server mode of the model may also be useful for parallel file systems.

This page is intentionally left blank.

6 Summary and conclusions

In the present study, the WAVEWATCH III model has been installed on a Linux based Titan cluster, consisting of 8 Intel Xeon processors. In section 4 it is shown that this model displays excellent scaling behavior on this cluster, comparable to the scaling behavior of this model on an IBM super computer (see Tolman, 2002a). The timing result indicate that the model benefits moderately from overloading the system by running 12 processes equally distributed over the 8 processors. In this case, the detrimental impacts of Amdahl's law, and the overloading of individual processors is offset by the increased parallel behavior of the model for increasing numbers of processes. It is also shown that the present model does not benefit from the hyper threading option on the Intel chips.

The presently distributed version of WAVEWATCH III (version 2.22, Tolman, 2002c) requires a parallel file system to assure the integrity of raw data output files for restarts and for output along tracks, as discussed in section 5. The latter output is modified to always be processed by a dedicated process, hence removing the need of a parallel file system for this output file. For the restart files, the parallel file system options are retained, based on its memory efficiency. Alternative data server methods are also made available, that produce proper restart files on non-parallel file systems. For the data server version of the model, the time required to write output data proved inconsequential compared to the computational time required by the model. The data server version for writing the restart file might also by useful on parallel file systems, because the writing technique utilized is inherently more efficient than in the original approach.

This page is intentionally left blank.

References

Gropp, W., E. Lusk and A. Skjellum, 1997: *Using MPI: Portable parallel programming with the message-passing interface.* MIT Press, 299 pp.

Marr, D. B., F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller and M. Upton, 2002: Hyper-Threading technology architecture and microarchitecture. *Intel Technology Journal*, **6**, 1–12.

Moon, I. J., I. Ginnis, T. Hara, H. L. Tolman, C. W. Wright and E. J. Walsh, 2003: Numerical modeling of sea surface directional wave spectra under hurricane wind forcing. *J. Phys. Oceanogr.*, **33**, 1680–1706.

Tolman, H. L., 2002a: Distributed memory concepts in the wave model WAVEWATCH III. *Parallel Computing*, **28**, 35–52.

Tolman, H. L., 2002b: Testing of WAVEWATCH III version 2.22 in NCEP's NWW3 ocean wave model suite. Tech. Note 214, NOAA/NWS/NCEP/OMB, 99 pp.

Tolman, H. L., 2002c: User manual and system documentation of WAVEWATCH III version 2.22. Tech. Note 222, NOAA/NWS/NCEP/MMAB, 133 pp.

Tolman, H. L., B. Balasubramaniyan, L. D. Burroughs, D. V. Chalikov, Y. Y. Chao, H. S. Chen and V. M. Gerald, 2002: Development and implementation of wind generated ocean surface wave models at NCEP. *Wea. Forecasting*, **17**, 311–333.