CEWES MSRC/PET TR/99-18

Dual-level Parallel Analysis of Harbor Wave Response Using MPI and OpenMP

by

Steve Bova
Clay P. Breshears
Christine Cuicchi
Zeki Demirbilek
Henry Gabb

04h01799

Work funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC94-96-C0002
Nichols Research Corporation

Dual-level Parallel Analysis of Harbor Wave Response Using MPI and OpenMP

Steve W. Bova* Clay P. Breshears† Christine Cuicchi‡ Zeki Demirbilek§
Henry A. Gabb¶

April 16, 1999

1 Introduction

It has become clear that the emerging class of high performance computers will have architectures based upon clusters of shared-memory processors (SMPs). The SGI/CRAY Origin2000 is an early example of this class which is currently available and offers non-uniform memory access (NUMA). Furthermore, IBM is reportedly developing a machine that will have small clusters of shared-memory processors connected via a message-passing network. The availablity of Pentium-class motherboards that have four or more processors has also lead to relatively inexpensive machines of this type. It has recently become possible for high-performance applications to run on a network of multi-processor personal computers and communicate via the Message Passing Interface (MPI). A distinguishing characteristic of this architectural class is that it offers dual-mode parallelism: simultaneous shared and distributed-memory programming is possible. In order to exploit such architectures, however, research must be performed to identify and express nested parallelism within applications. Studies such as these are only now beginning to appear in the literature [1].

The primary goal of the current paper is to illustrate the effectiveness of a dual-level parallel algorithm when applied to an existing, serial, production code. The parallel algorithm is implemented using MPI and OpenMP, because they represent the current standards in message-passing and shared-memory programming interfaces, respectively. Although the method is appropriate for a broad class of applications which explore a parameter space, the attention of this report is restricted to a harbor response code, CGWAVE, which is currently used for wave climate forecasts by the Department of Defense (DoD). Results presented herein demonstrate that dual-level parallelism allows substantial increases in model resolution combined with improvements in simulation turnaround time, and yet, contrary to conventional wisdom, requires very little source code alteration.

In Section 2 we summarize the physical and mathematical models used in CGWAVE. Then Section 3 describes our parallel algorithm. Numerical results and a summary are presented in

*Mississippi State University On-site CFD Lead for PET
†Rice University On-site Scalable Parallel Programming Tools Lead for PET
‡Computational Science and Engineering Group
§Coastal and Hydraulics Laboratory
¶Computational Migration Group

Sections 4 and 5, respectively. Finally, because OpenMP is a relatively new and therefore unfamiliar standard, we provide a brief introduction to its salient features in an Appendix.

2 Physical and Mathematical Model

CGWAVE is a Fortran77 code used for predicting wave climate in the nearshore area for military and civil engineering activities, including wave estimation inside harbors. Conceptually, harbors resonate at certain natural frequencies which are influenced by the bathymetry (topography of the ocean floor), the shape of the surrounding coastlines, and the presence of islands and artificial structures such as piers, jetties, breakwaters, *etc.* As periodic, incident waves approach from the open ocean, large-amplitude, standing waves may occur in certain areas of harbors. These waves present problems if naval activities such as mooring, on/off-loading, or amphibious operations are planned in the area. These large waves may also be hazardous to recreational boating and commercial shipping, or may lead to undesirable coastal erosion and subsequent overflowing of defensive structures, thereby causing flooding. CGWAVE is used to simulate the response of a harbor's surface and thereby estimate the locations of such areas of concern. Simulations have also been performed on behalf of agencies and governments for the underwriting of floating platforms, structures, navigable routes, and the examination of oil spills and other pollutants in the near-shore zone..

The details of the physical and mathematical model used in CGWAVE are available elsewhere in the literature[2, 3, 4], and will only be briefly summarized here. The underlying mathematical model is the two-dimensional, elliptic mild-slope wave equation

$$\boldsymbol{\nabla} \cdot (CC_g \boldsymbol{\nabla} \Phi) + \left[\frac{C_g}{C} \sigma^2 + \imath (w + C_g \gamma) \sigma \right] \Phi = 0 \tag{1}$$

where $\Phi = \Phi(x, y)$ is the unknown, complex surface elevation function; $C = C(x, y)$ is the phase velocity; $C_g = C_g(x, y)$ is the group velocity; σ is the wave frequency under consideration; $\imath = \sqrt{-1}$; w is the bottom friction factor; and γ is the wave-breaking parameter. The interested reader may consult the CGWAVE User's Guide [4] to obtain the various empirical models (*e.g.*, the wave-breaking and bottom friction terms) and boundary conditions required for closure.

Equation (1) represents a Helmholtz-type equation for the complex potential resulting from a given incident wave component, characterized by a known direction, amplitude, and frequency. The application of a Galerkin finite element method to the version of (1) associated with the i^{th} incident wave component leads to a large, sparse system of simultaneous linear equations of the form

$$\boldsymbol{A}_i \boldsymbol{\Phi}_i = \boldsymbol{b}_i \tag{2}$$

which is solved via a Krylov subspace method. In (2), $\boldsymbol{\Phi}_i$ denotes the unknown vector of discrete complex potential values associated with each mesh point; $\boldsymbol{A}_i$ denotes the sparse, complex–symmetric (not complex–Hermitian), indefinite coefficient matrix; and $\boldsymbol{b}_i$ denotes the known forcing vector. After all of the component solutions $\boldsymbol{\Phi}_i$ are computed, they are combined to obtain the resultant potential $\boldsymbol{\Phi}$.

The input data consists of the bathymetry, its associated unstructured, triangular finite element mesh (the resolution of which depends on the wave frequencies of interest), and a set of wave components. Hence, the model has two types of resolution: the usual spatial resolution associated with the finite element mesh and a sea state resolution, which is determined by the number of incident wave components. Our solution algorithm exploits parallelism for both types of resolution. Shorter waves (lower wave periods) typically require a greater spatial resolution to the extent that more than 20 mesh points may be necessary over the wavelength distance in order to accurately represent waves in shallow water depths. If the desired wavelengths become relatively small, the required density of the finite element mesh can place severe constraints on modelling capabilities. Parallel computing can ameliorate these issues to the extent that more memory and computational power can be employed.

3 Parallel Implementation

Typically, the number of incident wave components required to obtain an accurate simulation can range from as few as five for long waves (swells) in calm seas to several hundreds or thousands for short-crested, rougher sea states. Since each component can be treated in CGWAVE independently, this leads to a set of independent, differential equations that can be solved in parallel. This work is distributed to multiple processors via MPI[5]. A detailed analysis of the execution profile of CGWAVE reveals that the majority of the CPU time is spent in the conjugate gradient solver. More specifically, the matrix-vector product and inner product kernels are the most time-consuming calculations. We use OpenMP[6] to parallelize these kernels. Hence, MPI is used to simultaneously obtain solutions to multiple incident wave components, and OpenMP is used to accelerate the solution to each component.

3.1 MPI and CGWAVE

Two load-balancing schemes for distributing the incident wave components were tested. The first, Round-Robin, is akin to dealing cards in a poker game. One simple implementation gives access to the file containing component data to a single "dealer" process. Then, individual wave components (as triplets of direction, amplitude, and frequency) are packaged into messages and distributed to participating processors in a Round-Robin fashion until all components have been assigned. As in poker, the dealer process receives an equal share of the components distributed. To reduce message traffic, all wave component triplets to be assigned to a processor are combined into a single message rather than sending multiple small messages.

Round-Robin scheduling is a static strategy, the efficiency of which depends on the set ordering and the length of time required for a component to converge. Since there is no *a priori* knowledge of exactly how much execution time any wave component solution will require, the chance of a poor load balance is high. It is easy to imagine data sets in which all of the longest executing components are assigned to a single processor and all shortest executing components are computed on another.

The second load-balancing method tested, Manager-Worker, is a standard dynamic approach whose efficiency is largely independent of set ordering and can be effective even under erratic system load [7, 8]. The manager process is devoted to assigning computations to all other processes. When a worker process completes a task, a request for more work is sent to the manager process, which replies with one of two messages: a message containing the next task to be computed, or, if all tasks have been assigned, a message signaling that the computation has been completed. Upon receipt of the termination signal from the manager, worker processes exit gracefully. The pseudocode for the MPI Manager-Worker strategy for CGWAVE appears in Algorithms 1 and 2.

Algorithm 1 MPI Manager Pseudocode for CGWAVE

```
MPI_Init
do i = 1 to number_of_wave_components
  blocking receive     ! wait for work request
  blocking send        ! send wave component
end do
      ! All wave components solved
do worker = 1 to nprocs - 1
  blocking receive     ! wait for work request
  blocking send        ! terminate worker
end do
MPI_Finalize
```

Algorithm 2 MPI Worker Pseudocode for CGWAVE

```
MPI_Init
do forever
  blocking send        ! ask manager for work
  blocking receive     ! get wave component
  if not termination signal then
    perform calculations to solve wave component
  else
    exit infinite loop
  end if
end do
MPI_Finalize
```

3.2 MPI_Connect

The Manager-Worker method described above was originally designed and intended to run on a single high performance computing (HPC) platform with all processors able to communicate. Hence, applications which are load-balanced through the Manager-Worker algorithm are restricted to the total number of processors available on that machine. This may be accpetable if the application can use only a small number of MPI processes efficiently. CGWAVE, on the other hand, can make effective use of hundreds of MPI processes, up to one per wave component. Since very few commercially available HPC systems are equipped with thousands of processors, a user must coordinate an application running across multiple machines to harness the computational power of a very large number of processors. While it is possible to run the Manager-Worker model across multiple platforms under PVM [9], MPI has become the *de facto* standard for message passing and is the method of choice for most message-passing codes developed today. MPI_Connect [10] is a system that has the power to coordinate multiple MPI codes across multiple HPC platforms.

MPI_Connect is a metacomputing middleware–executable from either C or Fortran codes–that allows separately initiated MPI applications to interact in a peer-peer fashion. Public domain versions of MPI have been designed to allow a single application to be run across multiple platforms. However, if the particular implementation of MPI is not installed on a machine, the code would be unable to run on that system. One of the advantages with using MPI_Connect is that it will work with public domain versions of MPI as well as vendor-optimized versions. Thus, running MPI codes cooperatively on different vendor platforms with different MPI libraries is now possible.

Using MPI_Connect, MPI codes interoperate by registering themselves with a third party naming service that coordinates the interaction. This naming service and its location are transparent to the code. Once MPI applications have registered they can locate other MPI applications by using the naming service. When both parties agree, an MPI intercommunicator is created between the codes. This is used in the same way as any other MPI intercommunicator. In order to disconnect from all intercommunicators established with other codes, the MPI application removes its name from the naming service.

Configuring current MPI applications to run under MPI_Connect is accomplished with a minimal impact to the code. Three extra commands are added:

- `MPI_Conn_register()` to register the code's name with the naming service,
- `MPI_Conn_intercomm_create()` to create an intercommunicator with another named application, and
- `MPI_Conn_leave()` to remove the code's name from the naming service and consequently sever all MPI_Connect intercommunicators in which the code is involved.

The only other change required is to use the intercommunicator returned from the `MPI_Conn_intercomm_create()` call in the appropriate MPI communication functions.

Using [11] as a guideline for implementing the Manager-Worker model as an MPI_Connect code, CGWAVE was first broken up into two separate programs: one for the manager process and the other for the workers. The pseudocode for the MPI_Connect version of the Manager-Worker algorithm is shown in Algorithms 3 and 4.

Algorithm 3 MPI_Connect Manager Pseudocode for CGWAVE

```
MPI_Init
connect with worker groups          ! intercomm set-up
do $i$ = 1 to number_of_wave_components
  probe for worker request          ! busy wait on comm
  blocking receive
  blocking send          ! send wave component
end do
        ! All wave components solved
do worker = 1 to num_workers
  probe for worker request          ! busy wait on comm
  blocking receive
  blocking send          ! terminate worker
end do
MPI_Finalize
```

Algorithm 4 MPI_Connect Worker Pseudocode for CGWAVE

```
MPI_Init
connect with manager          ! intercomm set-up
do forever
  blocking send          ! ask manager for work
  blocking receive          ! get wave component
  if not termination signal then
    perform calculations to solve wave component
  else
    exit infinite loop
  end if
end do
MPI_Finalize
```

3.2.1 Worker Details

The first thing that each worker process does is to register the group name and create an intercommunicator with the manager process. The process with rank zero is designated as the lead worker. Before any worker in the group requests a wave component task, this process sends an informational message to the manager which contains the total number of workers that are active in its group. The manager adds this count to the total of active workers. The lead worker synchronizes all other workers in its group by broadcasting the reply to the informational message from the manager process. In this way all workers within a group are guaranteed that the manager has created all group intercommunicators and is ready to handle task requests.

3.2.2 Manager Details

The first thing that the manager process must do is register its name and create the intercommunicators between itself and the worker processes. One intercommunicator per group is sufficient since all workers within the group are part of the common MPI_Connect intercommunicator and have a unique rank within that intercommunicator. Intercommunicators that are returned from each call to `MPI_Conn_intercomm_create()` are kept in an array within the manager process.

The MPI standard allows the use of wildcard placeholders to match on any sender and/or any message tag within the `MPI_Recv()` routine, but there is no wildcard for the communicator argument in the parameter list. Thus, to execute the blocking receive steps within the manager process indicated in Algorithm 3, the explicit intercommunicator from which the message is to be received must be known. Since all workers are in a different communicator than the manager, and there can be many different worker group intercommunicators, the manager process must probe for worker requests.

To implement this step, the manager continuously loops over each intercommunicator within its array, using it as an argument to `MPI_Iprobe()` with a sender wildcard placeholder. By using the sender wildcard as an argument to `MPI_Iprobe()`, the manager probes for the arrival of any message from the stated intercommunicator. When the probe function returns TRUE, the probe loop is exited since it is known which explicit intercommunicator to use for the `MPI_Recv()` routine call that follows.

A case selection structure to handle both task requests and informational messages hides the message latency between the manager and worker processes. The manager process is able to assign wave components to workers before all informational messages from other groups (which may be delayed due to network traffic or other congestion) have been received.

3.2.3 Message Details

Task request messages from workers to the manager process need not contain any data; the tag is sufficient to denote that the message is a request for more work. Similarly, messages from the manager to worker processes use the tag to indicate whether the message carries component data or the termination signal.

3.2.4 Multiple Platforms

With MPI_Connect, CGWAVE may be run on multiple, geographically separate HPC platforms. Since the manager remains idle during the bulk of the execution, it may be desireable to run the manager process on a workstation. In this way, no expensive HPC resources are used to execute a process that does little computation.

Each HPC system can be running one or more copies of the worker program and each group of workers will execute as if it were the only group participating in the computation. This allows the user the flexibility to compute with as many or as few workers distributed among as many or as few groups which are appropriate for the expected amount of wave computations or physical resources available. The independence of groups allows a dynamic allocation of resources from one run to the next. That is, while the static allocation of the number of workers and groups must be fixed at the outset of execution, subsequent computations need not be required to use the same configuration as a previous execution.

3.3 OpenMP and CGWAVE

OpenMP confers additional benefits to CGWAVE. Since CGWAVE is a legacy Fortran code and an established engineering tool, it is undersireable to extensively change the underlying code structure. Fortunately, computation is largely confined to the conjugate gradient routine, particularly the matrix-vector and inner products. Therefore, our incremental approach began with the conjugate gradient solver.

Since an important goal is to run CGWAVE across multiple systems, the parallel implementation must be portable. All of the major hardware vendors have announced support for OpenMP, so portability is not an issue. For CGWAVE, only loop-level parallel directives are used. The KAP/Pro Toolset (Kuck & Associates, Inc.) was used for parallel assurance testing of the OpenMP implementation.

Typically, programs using exclusively loop-level parallelism scale poorly. On SMPs with uniform memory access, existing bus technology does not provide sufficient bandwidth to supply data to large numbers of processors. To achieve hardware scalability, the SGI/CRAY Origin2000 (O2K) uses a cache-coherent non-uniform memory access (ccNUMA) [12], distributed shared-memory hybrid architecture. Unfortunately, parallel loops also scale poorly on ccNUMA systems unless two things occur. First, data must be distributed to the processors that are to use that data. Second, the program must minimize data sharing between processors to avoid the system overhead of maintaining coherency.

On the O2K, data placement is often determined according to the "first touch" rule. Specifically, data retain an affinity to the first thread that touches them. This form of data distribution is transparent to the programmer. So if the master thread accesses a variable first (*e.g.*, initializes an array), the data will reside on the master thread's processor for the duration of program execution. Initializing important arrays in parallel allows other threads to touch parts of the data first and avoids this performance bottleneck. It should be noted that if threads are created and destroyed dynamically (controlled by the `OMP_DYNAMIC` environment variable), the first touch rule is void and data must be explicitly distributed. Dynamic threading is not used in CGWAVE.

The OpenMP coding in CGWAVE is limited to simple loop-level parallel directives (Figure 1). The sample code calculates a matrix-vector product using standard ELLPACK storage format for a sparse linear system. This is followed by a dot product calculation. The OpenMP directives interact with the Fortran source code in several ways. The parallel region encloses both loops. This reduces the overhead of thread creation. (The parallel region in the CGWAVE Krylov solver encloses eight, work-intensive loops.) Each thread does the same amount of work, so static scheduling is specified for both loops. Since most variables are accessed by all threads, the default scope is set to shared.

Figure 1 OpenMP code fragment from parallel Krylov solver.

```
!$omp parallel default(shared)
!$omp do schedule(static) private(l,j,aa)
do j=1,nn
    aa=(0.d0,0.d0)
    do i=1,nm
        aa=aa+a(i,j)*r(id(i,j))
    enddo
    q(j)=aa
enddo
!$omp end do
!$omp do schedule(static) private(l) reduction(+:pnew)
do l=1,nn
    pnew=pnew+r(l)*q(l)
enddo
!$omp end do
!$omp end parallel
```

Only loop indices and a place holder variable are declared private. A summation variable is declared using the reduction directive.

It is not an unusual practice for Fortran77 programs to use `COMMON` blocks as temporary, or "scratch", workspace. This practice can have detrimental effects on parallel performance, as was the case with our unoptimized version of CGWAVE. During parallel execution under OpenMP, all threads must share access to this work space. Clearly, this situation can lead to synchronization problems if multiple threads attempt simultaneous access. Thus, we replaced these `COMMON` blocks with local work arrays for each thread, thereby eliminating synchronization overheads. This also helped reduce the effect of system load on performance.

In some situations, these local work arrays may be copies of each other and therefore need to be kept consistent across all threads. The cost of actively maintaining data coherence across separate copies (both in execution time and programming complexity) could potentially negate the benefits of eliminating synchronization overheads. Since the data stored in the work space is local to each thread and CGWAVE initializes it prior to each use, no coherence between copies is needed.

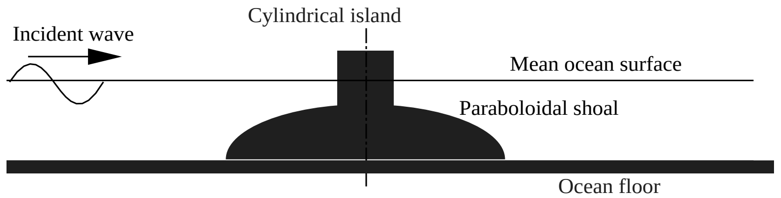
3.4 Combining MPI and OpenMP: Caveats

In CGWAVE, MPI and OpenMP express parallelism in two distinct layers. Specifically, MPI distributes wave components across processors and OpenMP creates threads to solve each component more quickly. However, it is possible for MPI and OpenMP to interact more closely. First and foremost, the MPI implementation used must be thread-safe to guarantee correct and consistent execution of both MPI and OpenMP.

Each thread within an OpenMP parallel region of a single MPI process will share the same communicator and rank of the master thread. Thus, `MPI_Init()` must not be called from within a parallel region since this would violate the rule that a MPI process may only call `MPI_Init()` once.

Point-to-point communications can be placed within an OpenMP parallel region, but it is the

Figure 2 Cross-sectional schematic of axisymmetric bathymetry for coarse mesh test problem (not to scale). Incident waves approach the cylindrical island from the left. After [2].



responsibility of the programmer to guarantee that message sends and receives are correctly paired. Because of the sharing of MPI rank between threads of a MPI process, targeting a message from a thread of one process to a specific thread of another process can be achieved by use of an explicit message tag. Another level of complexity and coordination is introduced if each MPI process creates different numbers of threads; *i.e.*, `OMP_DYNAMIC` is true. MPI collective communication routines (*e.g.*, broadcast, scatter, gather) called from within parallel regions present problems of coordination between threads of different processors and data coherence between threads of the same process.

MPI programmers seek to minimize the total number of communications needed. Communication calls in multithreaded regions drastically increase message traffic. Also, MPI communication routines often act as synchronization points. Since the goal of threading is to express concurrency, excessive synchronization is something that thread programmers try to avoid. Taking the above into consideration, for efficiency and ease of coding, it is recommended that message passing be performed outside of OpenMP parallel regions.

4 Numerical Results

We now present results obtained from sample problems analyzed using CGWAVE. First, the scalability of the dual-level parallel algorithm is examined. Next, a practical simulation of an inlet on the Atlantic coast of Florida is performed.

4.1 Scalability Study

In this section, we report our experiences with two well-studied configurations (*e.g.* see [2, 3]) that were used to assess scalability. Although the bathymetry differs in each case, we use identical sea states defined by seventy–five incident wave components combining five periods, fifteen directions, and forty amplitudes. The first case has a relatively coarse mesh (50,000 elements), and its bathymetry is shown in Figure 2. This case represents an axisymmetric mound that rises up from the ocean floor and terminates in a cylindrical island. The second case has a finer mesh (150,000 elements), and it models a sediment mound on a wedge-shaped incline (Figure 3).

Load balancing is necessary because the CPU time required to solve a given component can vary widely, depending on its period and amplitude (see Figure 4). Generally, the larger the period, the less time is required. For each problem considered here, the times associated with each component vary by as much as a factor of four.

Figure 3 Cross-sectional schematic of bathymetry for fine mesh test problem (not to scale). Incident waves approach the sediment mound in shallow water from the right. After [3].

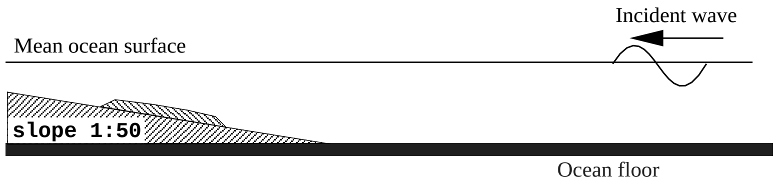
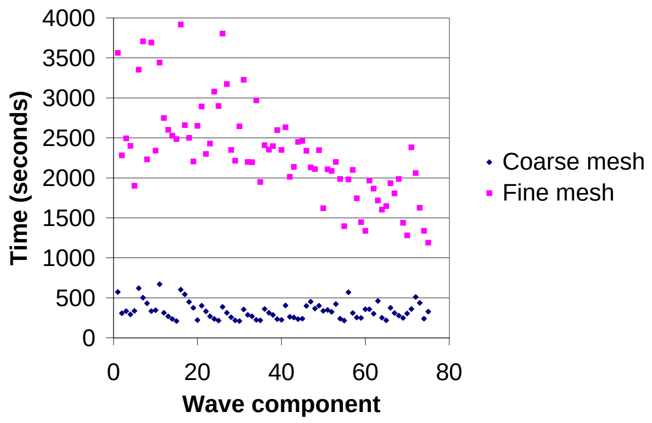


Figure 4 CPU time required for each incident wave component for coarse and fine mesh test problems.



Tables 1 and 2 present the required wallclock times (in minutes) for several combinations of MPI processes and OpenMP thread counts for the coarse and fine mesh problems, respectively. The number of OpenMP threads is denoted NT and the number of MPI processes is denoted NP. The total number of processors used in a given case is the product of NT and NP. The blank fields correspond to total processor counts in excess of available resources.

First, note that for a given NP, OpenMP provides near-linear speedup. Similarly, for a fixed number of OpenMP threads, scalability with respect to the MPI algorithm is also very good. Note that performance is best when using a mixture of OpenMP and MPI. For example, consider Table 2: if 64 total processors are used, using (NT,NP)=(4, 16) performs substantially better than (64, 1) or even (1, 64). Furthermore, note that if only a single kind of parallelism is used, NT=1 or NP=1, and the scalability breaks down beyond 32 total processors. By introducing a second kind of parallelism, the range of nearly linear speedup is extended to include 64 processors.

Table 3 presents the required wallclock times (in minutes) for several larger combinations of

Table 1 Wallclock times (minutes) for coarse mesh problem having 75 incident wave components.

NT	NP							
	1	2	4	8	16	32	64	75
1	393	197	98	50	27	16	10	9
2	197	105	48	25	13	8		
4	94	42	25	12	6			
8	44	21	11	5				
16	25	13	6					
32	25	12						
64	31							

Table 2 Wallclock times (minutes) for fine mesh problem having 75 incident wave components.

NT	NP							
	1	2	4	8	16	32	64	75
1	3100	1571	998	475	243	145	251	284
2	1877	941	462	307	153	78		
4	980	447	238	167	74			
8	483	178	127	69				
16	205	91	60					
32	125	59						
64	192							

Table 3 Wallclock times (minutes) for fine mesh problem having 75 incident wave components under MPI_Connect.

NT	NP					
	2	4	8	16	32	64
2					47	41
4				38	25	
8			31	17	33	
16		27	21	12		
32	40	22	21			
64	48					

MPI processor and OpenMP thread counts for the test problem. The timing tests were run using MPI_Connect across combinations of SGI O2K systems located at CEWES MSRC, Vicksburg, Mississippi, the Aeronautical Systems Center (ASC) MSRC, Dayton, Ohio, and the National Center for Supercomputing Applications (NCSA), Champaign-Urbana, Illinois. The total number of processors (64, 128, or 256 CPUs) used for a given case is the product of the number of MPI processes and number of OpenMP threads used by each process. (Blank entries in the table above the data shown were thought to use too few processors to justify MPI_Connect.) Due to the differences in the processor speeds (195 MHz for CEWES and ASC machines, 250 MHz for NCSA machines) and the different ratios of processors employed between configurations, these results may not display the true scalability of this code when run on processors of the same speed.

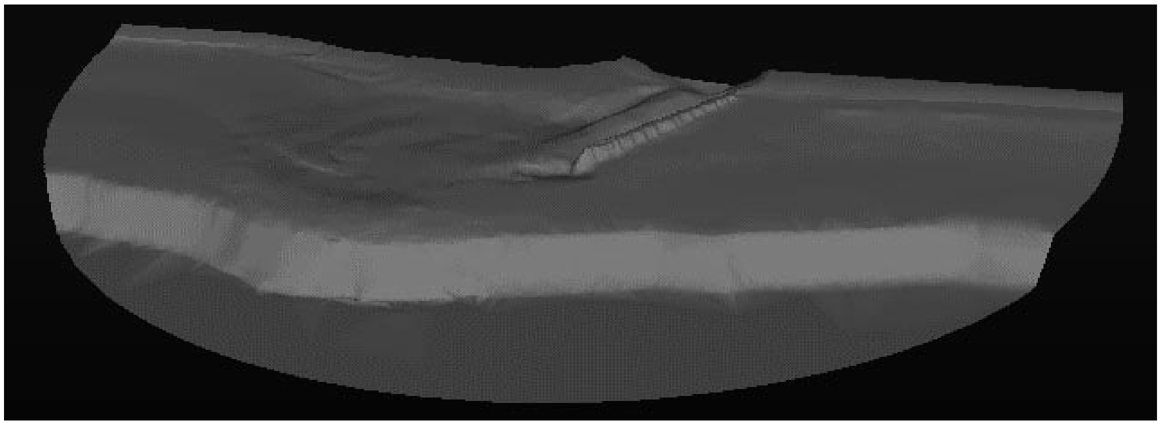
Table 2 shows execution times for the CGWAVE code run on different numbers of MPI processes and OpenMP threads on a single 195 MHz O2K system without MPI_Connect. From this data it was concluded that, for a fixed number of total processors, a combination of both parallel models is required to achieve the fastest execution time. The data in Table 3 also support this conclusion under MPI_Connect. Comparing data from Tables 3 and 2, we see the time taken to run the test problem was reduced from 3100 minutes on a single SGI O2K to just over 12 minutes running on 256 CPUs (16 MPI processes, each with 16 OpenMP threads) coordinated with MPI_Connect.

It should be emphasized that the performance data shown in Tables 1 and 2 are only be obtained when the "first touch" rule is observed, *i.e.,* when arrays associated with the parallel regions are initialized in parallel. When only the master thread initializes the relevant data, scalable performance is consistently lower and drastically affected by system load.

4.2 Application to Ponce Inlet, FL

Ponce Inlet is located approximately forty-five miles northeast of Orlando, Florida, and is notorious for its patches of rough water, which have capsized boats as they enter or leave the inlet. Previous studies were performed using computational models [13] as well as physical model basins built to scale [14]. The model bathymetry, which represents approximately 25 square kilometers, is shown in Figure 5 and was discretized using 235,000 mesh points. The sea state of interest can be adequately represented using 293 incident wave components. However, a common approximation necessitated by CPU time considerations is to use the dominant incident wave component. For example, the fastest of the 293 components to converge (not the most dominant) consumed 14 hours on a single

Figure 5 Model bathymetry for Ponce Inlet. The depths have been scaled by a factor of ten for clarity. The deep ocean is in the foreground, with the jetty clearly visible in the center. To the left of the jetty is the inlet channel, with the beaches rising up on either side.



processor. Hence, a conservative estimate of the required CPU time is

14 hrs x 300 components $\approx$ 4,200 hrs (six months)

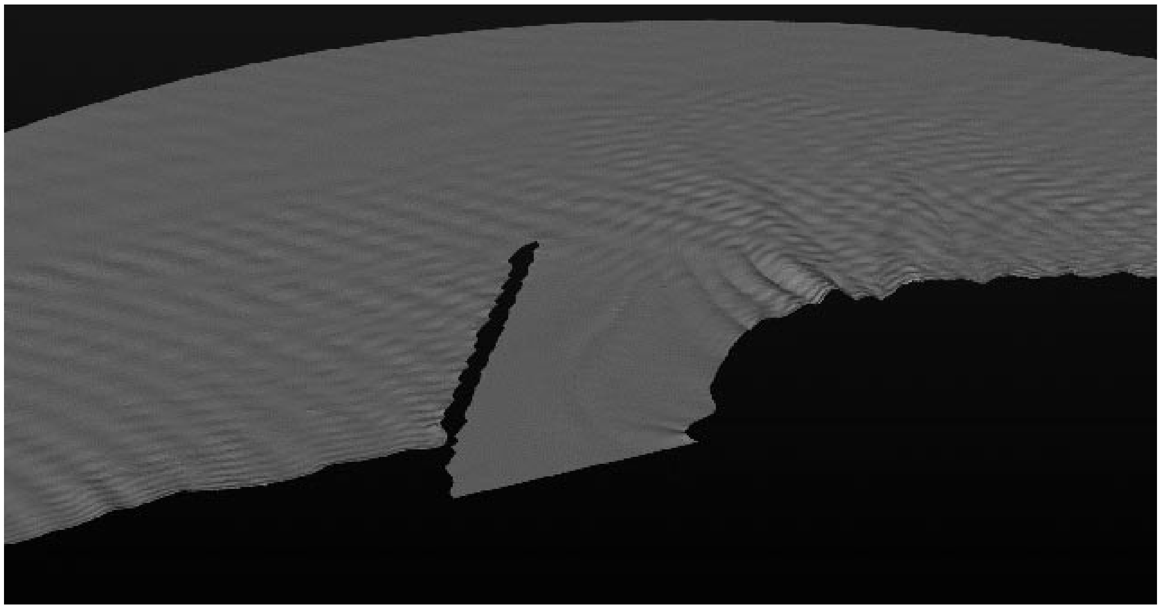
Using our parallel algorithm (MPI only) the entire 293 component problem was solved in less than 72 hrs using 60 SGI/CRAY Origin2000 processors. (Precise timings are unavailable since our application was automatically halted and subsequently restarted because the system was taken offline for maintenance.)

The surface elevation at the start of a period is shown in Figure 6(a) when only a single, dominant wave component is used as input. The dramatically improved surface elevation obtained when a sea state composed of 293 incident wave components is used is shown in Figure 6(b). In both cases, the solution is smooth in the inlet itself, but the single component case fails to capture the extent of the large standing waves that are evident to the right of the inlet in Figure 6. The location and magnitude of these waves correlates well with field observations. It is in this area that boats have been capsized as they pass through the inlet. A more subtle difference between the two solutions is the influence of bathymetric features such as the shelf. In Figure 6(b) the shelf is clearly visible as a line which separates the smoother surface associated with the deeper water from the choppy waves associated with the shallow water. This feature is less evident in the lower resolution result shown in Figure 6(a).

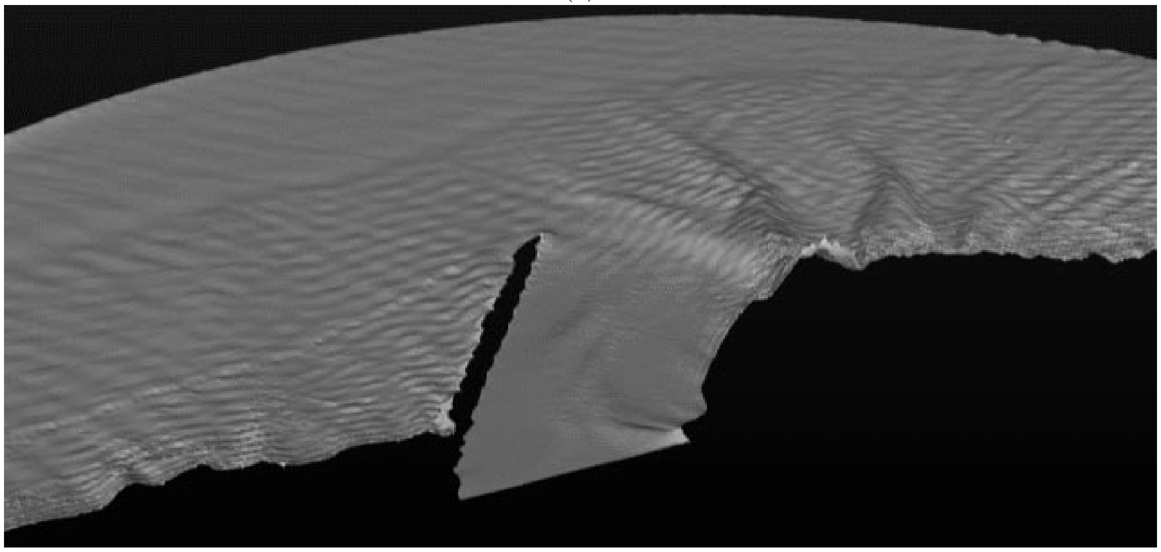
5 Summary

We have demonstrated that dual-level parallelism simultaneously using MPI and OpenMP is both feasible and beneficial. In particular, for a fixed problem size, the range of nearly linear speedup can be extended by introducing more than one level of parallelism. The proposed approach is suitable to a broad class of engineering applications which explore a parameter space. We have also

Figure 6 Surface elevations at the start of a period for Ponce Inlet obtained using (a) one incident wave component, and (b) 293 incident wave components. The depths have been scaled by a factor of 100 for clarity.



(a)



(b)

demonstrated that the use of MPI_Connect allows our nested algorithm to run across geographically distributed computer systems. Using the dual-level algorithm presented here, we were able to solve in twelve minutes what previously took over two days.

Such a dramatic performance improvement allows modeling of larger regions. This is particularly important in light of the following considerations: the Ponce Inlet grid covers only about 25 square kilometers. The DoD is interested in modeling regions on the order of a few hundred square kilometers. The dual-level algorithm also allows simulating more realistic sea states, *i.e.* spectral sea conditions. The previous state-of-the-art was only able to consider approximately 50 components for spectral sea states. These simulations required weeks to compute, whereas we used 293 components in the Ponce Inlet simulation, a number that was impractical with the original, serial code. Ideally, the ultimate goal would be to use approximately 1,000 wave components for simulating a sea state. MPI_Connect can be used to link multiple HPC platforms together in the solution of such extremely large problems.

The dual-level approach described here offers great flexibility. Through the use of conditional compilation, the MPI statements can be easily included or excluded. OpenMP parallelism is controlled via environment variables, and directives appear as comments to a compiler that does not support OpenMP. Hence, it is simple to have a fully serial code, an MPI-only code, an OpenMP-only code, or the dual-level code, depending on available resources, the desired turnaround time, and problem resolution.

5.1 Future Work

If a computation contains heterogeneous tasks that are better suited for different HPC systems, Manager-Worker codes under MPI_Connect can be programmed to assign tasks to processors on appropriate architectures. Future work on the CGWAVE code will include assigning tasks to different HPC platforms best suited to execute those particular tasks. For example, the computation time for wave components has been found to be proportional to the wave period under consideration. With this in mind, it will be possible to assign long period wave components to processors on one parallel machine (*e.g.*, IBM SP) while the shorter period components could be assigned to processors on an O2K able to compute with OpenMP threads. In fact, along with the wave component data, the number of threads to be used in the execution could also be sent to a worker process on an O2K.

It is important to note that such dynamic threading (controlled by the `OMP_DYNAMIC` environment variable and the library routine `omp_set_dynamic`) voids the first touch rule. This process is analogous to the prohibition on threadprivate `COMMON` blocks when `OMP_DYNAMIC` is true. In the former case the distribution is transparent and occurs at the system level. In the latter case, the distribution is explicit and is the responsibility of the programmer. It is impossible to maintain data affinity to a thread if the number of threads changes from one parallel region to another. In programs where dynamic thread creation is required, SGI provides data distribution extensions to OpenMP. These directives allow the programmer to specify distribution patterns or explicitly set data affinity to threads.

Acknowledgements

The authors are grateful to Graham Fagg (UT, Knoxville) for his assistance with MPI_Connect. John Towns of the National Center for Supercomputing Applications (NCSA) and K.V. Rao of

Silicon Graphics, Inc. graciously arranged access to computer facilities at NCSA. Randy Kleinman and Alex Carrillo (CEWES MSRC) provided valuable assistance with scientific visualization. This work was supported by the Department of Defense Major Shared Resource Center at the Corps of Engineers Waterways Experiment Station. Views, opinions, and/or findings contained in this report are those of the authors and should not be construed as an official Department of the Defense position, policy, or decision unless so designated by other official documentation. Permission to publish this paper granted by the Headuarters, U.S. Army Corps of Engineers.

References

[1] S.W. Bova, Clay P. Breshears, Rudolf Eigenmann, Henry Gabb, Greg Gaertner, Bob Kuhn, Bill Magro, and Stefano Salvini. Parallel programming with message passing and directives. *SIAM News*, In press, 1999.

[2] Bingyi Xu, Vijay Panchang, and Zeki Demirbilek. Exterior reflections in elliptic harbor wave models. *Journal of Waterway, Port, Coastal, and Ocean Engineering*, 122:118–126, May,June 1996.

[3] Vijay Panchang, Bryan Pearce, Ge Wei, and Benoit Cushman-Roisin. Soloution of the mild-slope wave problem by iteration. *Applied Ocean Research*, 13:187–199, 1991.

[4] Zeki Demirbilek and Vijay Panchang. CGWAVE: A coastal surface water wave model of the mild-slope wave equation. Technical Report CHL-98-26, U.S. Army Waterways Experiment Station, September 1998.

[5] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI—The Complete Reference: Volume 1, the MPI Core.* MIT Press, Cambridge, second edition, 1998.

[6] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface, Version 1.0. `http://www.openmp.org`, October 1997.

[7] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Publishing Company, Reading, MA, 1997.

[8] Hesham El-Rewini and Ted G. Lewis. *Distributed and Parallel Computing.* Manning Publications Co., Greenwich, CT, 1998.

[9] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: A Users' Guide and Tutorial for Network Parallel Computing.* MIT Press, Cambridge, 1994.

[10] Graham E. Fagg and Jack J. Dongarra. PVMPI: An Integration of the PVM and MPI Systems. Technical Report UT-CS-96-328, University of Tennessee, Knoxville, May 1996.

[11] Clay P. Breshears and Graham E. Fagg. A Computation Allocation Model for Distributed Computing Under MPI_Connect. In *Proceedings of First Southern Conference on Computing*, 1998.

[12] Kevin Dowd and Charles Severance. *High Performance Computing.* O'Reilly and Associates Inc., Sebastopol, CA, 1998.

[13] S.J. Smith and J.M. Smith. Numerical modeling of waves at ponce de leon inlet, florida. *Journal of Waterway, Port, Coastal, and Ocean Engineering*, 1999. Submitted for publication.

[14] G.S. Harkins, P. Puckette, and C. Dorrell. Physical model studies of ponce de leon inlet, florida. Technical Report CHL-97-23, Waterways Experiment Station, 1997.

[15] Bruce Leasure (editor). Parallel Processing Model for High Level Programming Languages. Technical Report X3H5/94-SD2 Revision L, ANSI, 1994.

[16] Bradford Nichols, Dick Buttlar, and Jacquelin Prolux Farrell. *Pthreads Programming.* O'Reilly and Associates, Inc., Sebastopol, CA, 1996.

Appendix: Overview of OpenMP

The OpenMP Architecture Review Board was formed in order to propose a standard set of parallel compiler directives for shared-memory multiprocessors (SMP). Previously, each SMP vendor used a different set of parallel directives. While the directive syntax differed between vendors, the functionality was similar. The OpenMP Fortran API (Version 1.0 released in October 1997) provides portable, hardware-independent parallel directives for SMPs. Architecture-specific extensions (*e.g.*, data placement and affinity), when necessary, are left to the implementor and do not interfere with the standard syntax.

Previous attempts at standardization (*e.g.*, Parallel Computing Forum [15]) were unsuccessful for two reasons: limited functionality and limited interest in SMP hardware. Recently, improved shared-memory architectures have sparked new interest in SMP programming. OpenMP, which is derived from PCF directives, also provides newer features that make scalable parallel codes possible. Shortly after its release, nearly all major hardware vendors announced support for the OpenMP specification.

OpenMP provides robust support for loop-level parallelism by spawning threads of execution for loop iterations. However, OpenMP is designed to give greater functionality than just loop-level parallelism. The directives give the programmer fine control over variable scope, thread scheduling, and thread synchronization within more general portions of code designated for parallel execution called *parallel regions*. The OpenMP specification also defines nested parallel regions, but existing commercial implementations do not currently support this feature.

The syntax of OpenMP allows the programmer to parallelize as much or as little of a program as desired. This is unlike message-passing methods which require a complete parallelization of codes. By using OpenMP, parallel regions of code can be created incrementally and tested for correctness, efficiency, and scalability.

External control over program behavior is possible through four environment variables that set the number of threads and default scheduling type, and enable/disable dynamic thread creation and nested parallelism. These variables and behaviors can also be controlled internally using the OpenMP run-time library. Run-time library routines are available to query the system and to determine the program environment. A collection of lock subprograms allows the programmer to declare mutual exclusion variables, providing low-level synchronization similar to POSIX threads[16].