Parallelization of a Three-Dimensional Shallow-Water Estuary Model on the KSR-1

C. FALCÓ KORN[1], J. M. BULL[2], G. D. RILEY[2], AND P. K. STANSBY[3]

[1]*The London Parallel Applications Centre, Queen Mary and Westfield College; Mile End Road, London E1 4NS, U.K.; e-mail: C.Falco-Korn@lpac.ac.uk*

[2]*Center for Novel Computing, University of Manchester, Oxford Rd., Manchester M13 9PL, U.K.; e-mail: {markb, griley}@cs.man.ac.uk*

[3]*Department of Engineering, University of Manchester, Oxford Rd., Manchester M13 9PL, U.K.; e-mail: pkstansby@fs1.eng.man.uk*

ABSTRACT

Flows in estuarial and coastal regions may be described by the shallow-water equations. The processes of pollution transport, sediment transport, and plume dispersion are driven by the underlying hydrodynamics. Accurate resolution of these processes requires a three-dimensional formulation with turbulence modeling, which is very demanding computationally. A numerical scheme has been developed which is both stable and accurate—we show that this scheme is also well suited to parallel processing, making the solution of massive complex problems a practical computing possibility. We describe the implementation of the numerical scheme on a Kendall Square Research KSR-1 multiprocessor, and present experimental results which demonstrate that a problem requiring 600,000 mesh points and 6,000 time steps can be solved in under 8 hours using 32 processors.

1 INTRODUCTION

Environmental impact studies relating to estuarial or coastal regions invariably involve computational flow simulation with additional simulation for the transport of pollution, sediment, or thermal plumes. The equations to be solved are known as the shallow-water equations which are based on the Navier–Stokes and continuity equations, with the assumption that the pressure everywhere in the flow is simply hydrostatic. The formulation may be simplified further by making the "depth-averaged" assumption where velocity is assumed uniform across the water depth. Computational schemes for such two-dimensional (depth-averaged) flows have been in existence since the pioneering work of Leendertse [6] and have proved useful in predicting flows in "well-mixed" conditions.

However, the turbulent boundary layer velocity profile will not be typical of a steady unidirectional current when flow curvature effects and eddy shedding are significant. This has obvious implications for predicting the transport of pollution—usually released near the sea bed—where the vertical distribution of velocity and turbulence (mixing) processes has an important influence. For sediment transport the near bed velocity and turbulence characteristics are also of vital importance. When buoyant plumes are released from power station outfalls, vertical motion is clearly significant to plume dispersion. Overall it can be seen that computation of the shallow-water equa-

Received May 1994
Revised December 1994

Scientific Programming, Vol. 4, pp. 155–169 (1995)
CCC 1058-9244/95/030155-15

tions in three-dimensional form is highly desirable.

Casulli and Cheng [1] developed a semi-implicit, Lagrangian finite-difference scheme as an alternative to a Eulerian, alternating direction-implicit scheme [10] avoiding the need for upwind differencing to give stability and the time step limitation of the Courant condition due to convective terms. Casulli and Cheng applied their scheme to tidal flows in the San Francisco Bay and the Venice Lagoon reporting good results.

Stansby and Lloyd [8] refined this scheme and applied it to the less spectacular, but probably more hydrodynamically demanding, case of flow around a circular island with sloping sides generating vortex shedding (see Fig. 1). The choice of this simple geometry was motivated by the desire to validate the model before applying it to real-world estuaries (see Section 8). Hence, the output of the program was compared to detailed measurements obtained from a laboratory tank, resulting in good agreement [7].

Typical simulations require the order of 10^6 mesh points and several thousand time steps. On scalar computers this would be computationally prohibitive. Even on a modest vector processor, the Cray EL-98, the code required excessive computer time (days) for large problems. In this article we investigate the use of parallel processing for producing such simulations within practical time scales.

Section 2 introduces briefly the underlying physical model and the numerical scheme. The resulting algorithm and its memory requirements are explained in detail in Section 3. Section 4 gives an overview of the target parallel platform, the Kendall Square Research KSR-1, focusing on those aspects of the architecture and programming model relevant to this study. Before embarking on the parallelization process, several optimizations were performed on the original, sequential code; these are described in Section 5. Section 6 details the parallelization strategy and the problems encountered in its stepwise application to the optimized code. Section 7 presents run-time results obtained on the KSR-1, which confirm the suitability of the numerical scheme to parallel processing. Furthermore, the sources of overhead in the parallel version are identified and analyzed. We conclude with Section 8 in which we outline future enhancements in the physical and numerical model and their consequences for parallelization.

2 THE THREE-DIMENSIONAL SHALLOW-WATER METHOD

The three-dimensional shallow-water equations are as follows

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = -g\frac{\partial \eta}{\partial x} + \frac{\mu_H}{\rho}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + \frac{\partial}{\partial z}\left(\frac{\mu_I}{\rho}\frac{\partial u}{\partial z}\right)$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} = -g\frac{\partial \eta}{\partial y} + \frac{\mu_H}{\rho}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) + \frac{\partial}{\partial z}\left(\frac{\mu_I}{\rho}\frac{\partial v}{\partial z}\right)$$

$$0 = \frac{\partial \eta}{\partial t} + \frac{\partial}{\partial x}\int_{z_0}^{\eta} u\,dz + \frac{\partial}{\partial y}\int_{z_0}^{\eta} v\,dz$$

where z_0 is the bed elevation above a reference level and η is the water surface elevation; x, y, z

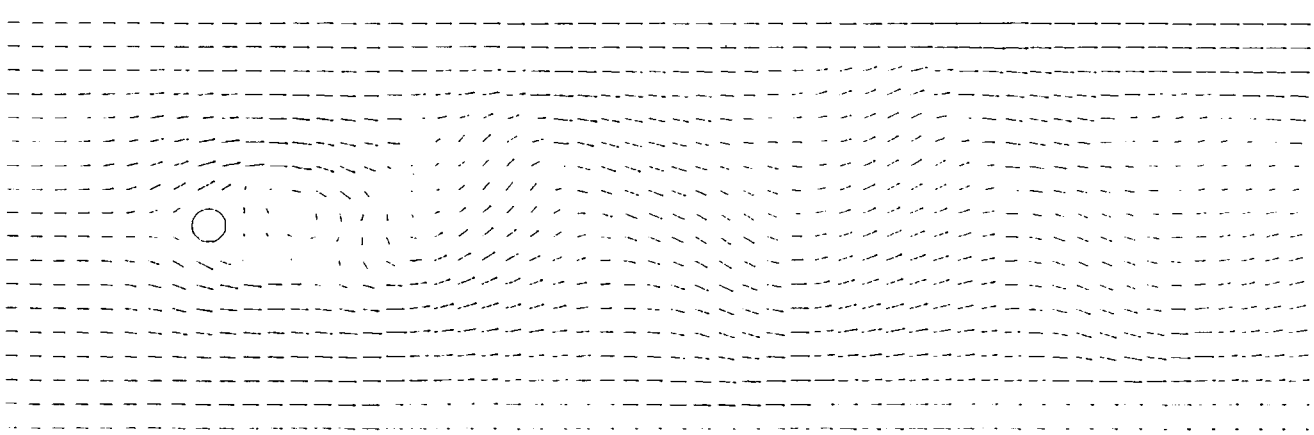

FIGURE 1 Surface flow after 2.000 time steps in the simulation.

are Cartesian coordinates: u, v, w are the corresponding velocity components; g is acceleration due to gravity; ρ is water density (assumed constant); and μ_I and μ_{II} are vertical and horizontal mixing coefficients. The boundary conditions at the bed are

$$\mu_I \frac{\partial u}{\partial z} = \tau_x^b, \quad \mu_I \frac{\partial v}{\partial z} = \tau_y^b$$

where τ_x^b and τ_y^b are the x and y components of the shear stress, respectively. At the surface no wind is assumed, so that

$$\mu_I \frac{\partial u}{\partial z} = \mu_I \frac{\partial v}{\partial z} = 0.$$

In the laboratory experiment described in the previous section, the water is initially stationary and the water level horizontal. The inlet flow rate is then increased with time as a quarter sinusoid and maintained after a specific time step at a constant value to represent a steady current. At the outlet boundary, the velocities u and v are given zero normal gradients and the water depth is fixed. At the two side walls, v and the normal gradients of u and η are set to zero.

The original formulation of the numerical scheme proposed by Casulli and Cheng [1] used a uniform mesh in the vertical direction, constant vertical and horizontal mixing coefficients, and the Chezy coefficients to give bed boundary conditions. In order to give an accurate representation of bed and water-surface conditions, Stansby and Lloyd [8] introduced the σ-coordinate system $\sigma = (z - \eta)/(\eta - z_0)$ for the vertical direction, defining the bed surface by its roughness height. This enables a turbulence model for the vertical direction to be incorporated; Stansby and Lloyd proposed a simple two-layer mixing length model for rough-turbulent flow. Furthermore, they introduced for horizontal mixing a mixing coefficient proportional to depth and friction velocity.

The finite-difference mesh used in the numerical computation is a staggered rectangular system with a "wet/dry" boundary crossing the horizontal mesh obliquely (giving wet and dry cells). This is not a severe limitation since velocities close to the shoreline with gently sloping beds tend to be quite small. The σ-coordinate system entails a fixed number of vertical cells at each horizontal mesh point. We will refer to the number of mesh points in each spatial direction by n_x, n_y, and n_z, and to the corresponding coordinates by $x_i (i = 1, \ldots, n_x)$, y_j $(j = 1, \ldots, n_y)$, and z_k $(k = 1, \ldots, n_z)$.

An important feature of the numerical scheme is the Lagrangian treatment of the convective terms. This avoids the need in conventional Eulerian schemes (e.g., TRISULA [10]) to generate stability through upwind differencing with some inevitable numerical viscosity. The terms involving surface elevation gradient and vertical mixing are handled implicitly for stability, whereas the terms involving horizontal mixing are handled explicitly. The equations are solved as fully coupled in both horizontal directions producing at each time step a pentadigonal system of equations for the new values of η at each grid point in the horizontal plane. Schemes which involve uncoupling (alternating direction schemes) require smaller time steps to be used for equivalent accuracy.

3 THE APPLICATION PROGRAM: SW3D

In this section we describe the structure of a Fortran 77 program, SW3D, which implements the three-dimensional shallow-water method described in Section 2. The version of SW3D which forms the starting point for the parallelization process had previously been run on a Cray EL-98 system.

The main computational effort of SW3D is contained within a subroutine called LXY, which is sketched in the pseudo code shown in Figure 2. We distinguish between actual array elements (written in `truetype` font) and mathematical objects and operations (using standard notation). For instance, $A^{(i,j)}$ denotes a $n_z \times n_z$ tridiagonal matrix which depends on the index pair `(i, j)`, while `u(i, j, k)` represents the `(i, j, k)`-th element of the array storing the values of u. The vectors b_1 and b_2 in BU_CU and BV_CV are fixed, and n_{ij} in SETUP implies a numbering scheme of the $n_x \times n_y$ pentadiagonal matrix P.

Most of the work in LXY is devoted to setting up the matrix P and right-hand side r of the linear system $Pe = r$ which is solved for the new surface elevation. For each time step the sequence of operations is as follows: first, code segments FU and FV evaluate, for every grid point, the finite-difference operator arising from the explicit terms for convection and horizontal mixing, and store the values into arrays `fu` and `fv`, respectively. Next, segments BU_CU and BV_CV each solve (for every `(i, j)`) two tridiagonal linear systems of dimen-

do t = 1, maxt

FU

do i = 1, n_x
  do j = 1, n_y
    do k = 1, n_z
      compute Lagrangian convection and horizontal diffusion terms in u
      store result in `fu(i,j,k)`

FV

do i = 1, n_x
  do j = 1, n_y
    do k = 1, n_z
      compute Lagrangian convection and horizontal diffusion terms in v
      store result in `fv(i,j,k)`

BU_CU

do i = 1, n_x
  do j = 1, n_y
    setup $n_z \times n_z$ tridiagonal matrix $A^{(i,j)}$ using values stored in `u`
    `bu(i,j)` = $b_1^T \left(A^{(i,j)}\right)^{-1} b_1$
    setup n_z-dimensional vector $b^{(i,j)}$ using values stored in `fu`
    `cu(i,j)` = $b_1^T \left(A^{(i,j)}\right)^{-1} b^{(i,j)}$

BV_CV

do i = 1, n_x
  do j = 1, n_y
    setup $n_z \times n_z$ tridiagonal matrix $A^{(i,j)}$ using values stored in `v`
    `bv(i,j)` = $b_2^T \left(A^{(i,j)}\right)^{-1} b_2$
    setup n_z-dimensional vector $b^{(i,j)}$ using values stored in `fv`
    `cv(i,j)` = $b_2^T \left(A^{(i,j)}\right)^{-1} b^{(i,j)}$

SETUP

do i = 1, n_x
  do j = 1, n_y
    compute 5 non-zero entries of row n_{ij} of P using `bu(i,j)`, `bu(i+1,j)`, `bv(i,j)`, `bv(i,j+1)`
    compute element n_{ij} of r using `cu(i,j)`, `cu(i+1,j)`, `cv(i,j)`, `cv(i,j+1)`

PENTA

solve pentadiagonal system $Pe = r$; store results in array `e`

COPY

copy `u` into `uold`, `v` into `vold`

UPDATE-U

do i = 1, n_x
  do j = 1, n_y
    setup $n_z \times n_z$ tridiagonal matrix $A^{(i,j)}$ using values stored in `uold`
    setup n_z-dimensional vector $b^{(i,j)}$ using values stored in `fu` and `e`
    `u(i,j,k)` = k-th element of $\left(A^{(i,j)}\right)^{-1} b^{(i,j)}$

UPDATE-V

do i = 1, n_x
  do j = 1, n_y
    setup $n_z \times n_z$ tridiagonal matrix $A^{(i,j)}$ using values stored in `vold`
    setup n_z-dimensional vector $b^{(i,j)}$ using values stored in `fv` and `e`
    `u(i,j,k)` = k-th element of $\left(A^{(i,j)}\right)^{-1} b^{(i,j)}$

UPDATE-W

do i = 1, n_x
  do j = 1, n_y
    do k = 1, n_z
      compute `w(i,j,k)` using values stored in `u` and `v`

FIGURE 2 SW3D's main computational cycle—subroutine LXY.

sion n_z, which result from the implicit vertical mixing term. The dot product of the solution of these systems with a given vector is stored in arrays bu, cu and bv, cv, respectively. Segment SETUP uses this information to compute the entries of P and r. PENTA solves the pentadiagonal system, leaving the solution in array e.

Having copied the current values of u and v into uold and vold (COPY), the new u and v velocities are computed in segments UPDATE-U and UPDATE-V, respectively. Here a further tridiagonal system is solved for each index pair (i, j), giving the new velocities for all values of k. The final segment of LXY, UPDATE-W, computes the new w velocities.

It is important to clarify the treatment of dry points. For those indices (i, j), where (x_i, y_j, z_1) is above the water, the loops FU, FV, BU_CU, BU_CV, UPDATE-U, UPDATE-V, and UPDATE-W do nothing except set the corresponding array elements to zero.

Figure 2 contains only the most computationally significant segments of LXY. Further code such as the calculation of the new water depth using the new surface elevation or the flooding of dry points are not included. This code will, however, not be neglected when analyzing the runtime.

To determine the memory requirements we introduce the notation $(n_1, \ldots, n_m)$ to denote m-dimensional arrays with $n_1 \times \ldots \times n_m$ floating-point elements. Hence, the arrays bu and bv are of type (n_x, n_y), while P is stored in an array of type $(5n_x, n_y)$. Since the tridiagonal matrices are set up on demand, one array of type $(n_z, 3)$ suffices, otherwise it would be necessary to store $2n_xn_y$ of these. Clearly the memory needed is dominated by the seven three-dimensional arrays fu, fv, u, v, w, uold, and vold of type (n_x, n_y, n_z). The original code uses six further arrays of the same type (see Section 5) leading, for 64-bit floating-point numbers, to a memory requirement of at least $13 \times n_x \times n_y \times 8$ bytes. For $n_x = 329$, $n_y = 105$, and $n_z = 22$, the values used in this study, this represents a memory requirement of nearly 80 Mbyte.

4 THE KENDALL SQUARE RESEARCH KSR-1

The KSR-1 is a virtual shared memory multiprocessor. The machine consists of processor–memory pairs (cells) arranged in a hierarchy of search groups, each group containing 32 cells. The virtual memory is implemented on the physically distributed memories by a combination of operating system software and hardware support through the KSR ALLCACHE search engine. The OS manages page migration and fault handling in units of 16 Kbyte. The ALLCACHE engine manages movement of 128 byte subpages within the system. Movement of subpages is therefore cheap compared to the movement of pages. The implementation described in this work is for the 64 cell, double search group, KSR-1 installed at Manchester University.*

Each cell is a 20 MHz, super-scalar, RISC chip with a peak 64-bit floating-point performance of 40 Mflop/s (achieved with a multiply–add instruction) and 32 Mbyte of memory. Two instructions may be issued per cycle: the instruction pair consists of one load/store or i/o instruction and one floating-point or integer instruction. The cells in a single group are connected by a unidirectional slotted ring network with a bandwidth of 1 Gbyte/s. The two search groups of the Manchester machine are connected by a further unidirectional slotted ring network with a bandwidth of 4 Gbyte/s, where up to 34 groups can be attached.

The ALLCACHE memory system is a directory-based system which supports full cache coherency in hardware. Data movement is request driven: a memory read operation which cannot be satisfied by a cell's own memory generates a request which traverses the hierarchy of rings and returns a copy of the data item to the requesting cell. A memory write request which cannot be satisfied by a cell's own memory results in that cell obtaining exclusive ownership of the data item—the data item moves to the requesting cell. In the process, as the request traverses the memory system, all other copies of the data item are invalidated, thus maintaining cache coherence through an invalidate-on-write policy.

The machine has a Unix-compatible distributed operating system—the Mach-based OSF/1—allowing multiuser operation. The programming model supported is primarily that of program directives placed in the user code (Fortran 77 and to some extent, C, [5]). The directives may be placed manually or automatically (by a pre-processor, KAP). A run-time support system, PRESTO, and underlying Posix-based threads model support the user directives. The run-time

* Running KSR OS version R1.1.4.1, October 20, 1993 and compiler version 1.0, May 11, 1993.

system and threads are also directly accessible through a standard library interface.

4.1 KSR-1 Memory Latencies

The KSR-1 processor has a level 1 cache, known as the subcache. The subcache is 0.5 Mbyte in size, split equally between instructions and data. The data subcache is two-way set associative with a random replacement policy. The cache line of the data subcache is 64 bytes (half a subpage).

There is a two-cycle pipeline from the subcache to registers. A request satisfied within the main cache of a cell results in the transfer of half a subpage to the subcache with a latency of 18 cycles (0.9 μs). A request satisfied remotely from the main cache of another cell on the same ring results in the transfer of a whole subpage with a latency of around 150 clock cycles (7.5 μs). This value has to be multiplied by a factor of 3 if the request is satisfied by a cell of the second ring. A request for data not currently cached in any cell's memory results in a traditional, high latency, page fault to disk.

4.2 Memory System Behavior—Alignment and Padding

In order for a thread to access data on a subpage, the page in which the subpage resides must be present in the cache of the processor on which the thread executes. If the page is not present, a page miss occurs and the operating system and ALLCACHE system combine to make the page present. If a new page causes an old page in the cache to be displaced, the old page is moved to the cache of another cell if possible. If no room can be found for the page in any cache, the page is displaced to disk. Moving a page to the cache of another cell is much cheaper than paging to disk.

Performance of applications is virtual memory systems can suffer from the phenomenon of false sharing: if two threads, running on different cells, request separate data items which reside on the same subpage, that subpage may continually thrash back and forth between cells. Most virtual memory systems have to contend with false sharing at the OS page level, which is typically several kilobytes in size. On the KSR-1 the unit of movement around the system is the relatively small 128-byte subpage. At this size, ensuring that data structures accessed by several threads do not cause thrashing can be achieved simply by ensuring that the structures are padded out to a subpage boundary and that they are aligned so as to begin on a subpage boundary. This is most simply achieved through suitable declaration of data structures: e.g., padding the inner dimension of multidimensional arrays.

4.3 KSR Fortran Directives

The directives provided support the following three forms of parallel construct:

1. **Parallel sections** support the execution of multiple code segments in parallel.
2. **Parallel regions** support the execution of multiple copies of the same code segment in parallel.
3. **Tile families** support the execution of loop nests in parallel. A loop nest is considered to define an iteration space which may be partitioned into tiles. Multiple tiles may be executed in parallel. The tile family is a specialized version of a parallel region, tailored to the regular iteration spaces found in Fortran Do loops. This form of parallelism is the most common in Fortran programs. The syntax was described previously [4], but we shall outline the most important features here. The `tile` directive takes the following form:

```
c*ksr* tile (index_list, [options])
        .
        .
    [loop nest]
        .
        .
c*ksr* end tile
```

This divides the iteration space of the loop nest into a number of rectangular pieces (tiles). These tiles are then scheduled to be executed in parallel. The `index_list` allows the programmer to specify which iterators are to be tiled. The options allow specification of the number of threads to be used, and a choice of scheduling strategies. There are two strategies which are of interest in this study: `slice` and `mod`. The `slice` strategy divides the iteration space into p roughly equally sized tiles. The `mod` strategy divides the iteration space into more than p tiles (where possible), and schedules them on p threads in a modulo fashion. For either strategy the size of the tiles can be fixed by the programmer, or determined at run-time. In the latter case the tile size will normally be chosen

as a multiple of 16 to help avoid false sharing of subpages. The options also allow scalar variables to be declared as private or reduction variables. In the case of a reduction variable results are accumulated in local copies of the variable, and code is generated which reduces these to a single variable at the end of the tiled loop.

5 SEQUENTIAL OPTIMIZATIONS

The original code consisted of nearly 1,000 lines of Fortran code, handling PENTA through ITPACK [3], a 9,000 line Fortran package which offers seven iterative methods to solve sparse linear systems with symmetric positive definite or mildly nonsymmetric coefficient matrices. The Jacobi conjugate gradient (JCG) method was chosen because of its convergence properties. As proposed by Casulli and Cheng [1], the code was developed for vector processors, running initially on a Cray EL-98 with an optimized version of ITPACK. The code was transferred to the KSR-1 and compiled without any change. We always used the highest optimization level of the compiler (`-O2` option). Furthermore, ITPACK was compiled on the KSR-1 with the `-r8` option. Otherwise all floating-point variables, which are declared as DOUBLE PRECISION, would be handled as 128-bit values.

It is important to note that one cell does not have enough memory to cope with the required 80 Mbyte, causing a considerable amount of data to be placed on the memory of other cells. Hence, the sequential program suffers communication overhead since it has to perform some remote data accesses. Analysis of the code led to following optimizations.

1. **Reducing memory requirements:** From Figure 2 we can see that the most natural loop orders are `ijk` (i.e., `i` outermost, `k` innermost) or `jik`, where `i` runs over the x dimension, `j` over the y dimension, and `k` over the z dimension. Because the algorithm is applied to shallow-water problems, the index space of `k` is much smaller than that of `i` or `j`. Casulli and Cheng [1] suggest that the proposed algorithm is suited for vectorization: efficient vectorization would required flipping the loop order to make the innermost loops the longest, i.e., `kij` or `kji` order. This "unnatural" loop ordering was implemented in the original code provided here only in FU and FV: the remaining computations used loop order `ijk`. As the Cray vectorizing compiler reported that loop bodies in FU and FV were too long to vectorize, the loop bodies were split in two. This involved the storage of intermediate data into six arrays of type (n_x, n_y, n_z). By reversing this splitting we avoided the temporary arrays, reducing the number of three-dimensional arrays to seven, and the total memory required to around 43 Mbyte. This in turn reduced the number of remote accesses.
2. **Avoiding bad stride:** All loops over `i`, `j`, and `k` were converted to `ijk` order. Since Fortran arrays are stored column wise, the seven three-dimensional arrays were declared of type (n_z, n_y, n_x) thus achieving a correlation between loop nest order and the layout of arrays in memory. This is vital for achieving a high rate of data reuse in a hierarchical memory system. A minor side effect of `k` being the innermost array index is the fact that the solution of the tridiagonal systems in UPDATE-U and UPDATE-V can be stored directly into `u` and `v`, respectively, rather than having to use an intermediate vector.
3. **Stripping ITPACK:** As a first step, the path followed by the JCG call through the library routines was identified and isolated: almost 7,500 lines of unnecessary code were deleted. Furthermore, the routines SCAL and UNSCAL were modified. ITPACK calls the former before the first iteration to scale the matrix, the right-hand side and the initial solution. After convergence, the scaling is reversed. In LXY the unscaling of the matrix and right-hand side is not necessary since they are not used after PENTA. Hence UNSCAL was reduced to a single loop, which was inlined, to unscale the solution. The modification of SCAL was motivated retroactively by the necessity to parallelize ITPACK. The matrix is scaled by ITPACK such that all diagonal elements have the value 1. To perform the unscaling, the original diagonal elements are stored at the beginning of the one-dimensional array containing all nonzero elements of the sparse matrix. This implies shifting the off-diagonal elements, an operation that is inherently sequential. Therefore, the sparse matrix structure is constructed accordingly in

SETUP, i.e., the diagonal elements were stored at the beginning, and the rest afterwards. The consequences for SCAL are the avoidance of the shifting, and the simplification of the search for the diagonal elements.

The correctness of these code transformations was confirmed in separate runs by dumping test data to a file after each time step and comparing them with the values from the original version of the code. A new version was only accepted if the files were identical.

The effort invested in these sequential changes has a significant payoff—the elapsed time for five time steps was reduced from nearly 2,750 seconds in the original code to about 600 seconds. Nearly 86% of this enhancement is as a result of the avoidance of bad stride by declaring the three-dimensional arrays as (n_z, n_y, n_x). On a vector processor (on which the code was developed) stride has little impact, since the memory on such an architecture is basically "flat." In a hierarchically structured memory, however, ensuring maximum reuse of data is vital to obtain efficient code. The reduction of data and stripping of ITPACK resulted in 13% and 1% improvement in execution time, respectively. The total amount of work invested in these optimisations, including the time required to become acquainted with the algorithm and the code, was about 7 person-days (we consider 1 person-day to be 8 hours of dedicated work).

6 PARALLELIZATION

The version containing all sequential optimizations proposed in Section 5 was the starting point for parallelization. Table 1 shows in detail the contribution of the different parts to the total runtime of LXY: computations of similar structure have been grouped together since they can be parallelized in a similar way. REST accounts for all minor computations scattered throughout LXY, including COPY.

LXY was parallelized stepwise, the sequence order—reflected in the following subsections—being determined by the magnitude of the execution time given in Table 1. The only exception was PENTA which was left to the end, because the loops in ITPACK have a different structure to all the others in LXY. The parallelization strategy for loops not in PENTA is already implied in Figure 2. The obvious and successful approach is to split the x, y plane evenly among all threads and let each one work independently on its portion of the plane. This is achieved by tiling the loops over i and j, thus:

```
c*ksr* tile (i, j, strategy=slice,
private=(k))
    do i = 1, nx
        do j = 1, ny
            do k = 1, nz
                ...
c*ksr* end tile
```

Here the index space of i and j is partitioned by PRESTO (the KSR run-time system) into contiguous tiles which are distributed between all threads such that each gets exactly one (slice strategy). All variables are shared (i.e., just one copy exists which can be accessed by all threads) except the tiled index variables and those explicitly listed as private.

A detailed analysis showed that the proposed approach is indeed valid. The shared data consist basically of all arrays of size at least $n_x \times n_y$ (e.g., fu, v, and e). Tiling of the loops results in correct execution since only the thread "owning" an index pair (i, j) updates the corresponding element of any shared array, and the tile statements impose the necessary synchronization points which prevent threads from starting the execution of a subsequent loop nest until all other threads have completed the current loop nest.

Note that the island is mapped onto the threads depending on the chosen partitioning of the x, y plane. This could lead in some partitionings to load imbalance, since no computation is performed on dry grid points (see Section 3). Because

Table 1. Elapsed Times in Seconds for Five Time Steps for the Optimized Sequential Version of LXY

FU, FV	BU_CU, UPDATE_U, BV_CV, UPDATE_V	SETUP	PENTA	UPDATE_W	REST	Total
375	158	2	41	15	13	608

the size of the island is small compared to the estuary, the load imbalance is negligible. For a complex estuary with irregular wet/dry boundaries a more sophisticated load-balancing strategy is required (see Section 8).

Figure 2 shows that some communication of data between threads is necessary in SETUP, caused when a thread owning index pair `(i, j)` but not `(i+1, j)` accesses, for instance, `bu(i+1, j)`. The set up of the tridiagonal matrices (and to a lesser extent some of the right-hand sides) as well as the computation in UPDATE-W cause similar, regular communication patterns. In FU and FV, however, the communication pattern depends on the data and is therefore unpredictable and irregular (see Section 6.1). The situation in PENTA will be described in Section 6.6.

The performance results presented in Section 7 confirm the validity of this approach. The following subsections report insights and experiences gained during the process of parallelizing the various segments of LXY. This parallelization process required approximately 10 person-days.

6.1 FU and FV

The main difficulty encountered in adding the tile directives (to these and all other loops) was the identification of the private variables. Having done this for FU and FV, we discovered, using PRESTO information, an important amount of load imbalance cause by an uneven assignment of indices to threads. We therefore decided to take manual control of the size and distribution of tiles in order to improve load balance. In Section 7 we will give more details and report on the results obtained.

We would like to stress the ease of parallelizing FU and FV on a virtual shared memory architecture like the KSR-1. As we have already mentioned, the communication pattern in these steps is unpredictable: for each `(i,j,k)` we need the velocity at that grid point (x_i, y_j, z_k) and at the point $(x_i - a, y_j - b, z_k = c)$, where a, b, and c depend on the actual values of `u(i, j, k)`, `v(i, j, k)`, and `w(i, j, k)`. Since $(x_i - a, y_j - b, z_k - c)$ is usually not a grid point, its velocity is obtained by interpolating the velocities of the eight cell corners containing it (the two-dimensional analog is shown in Fig. 3).

On a message-passing architecture each process would have to find out where the information concerning $(x_i - a, y_j - b, z_k - c)$ is stored, send a message to the corresponding process, and wait for the data to arrive. Note that this protocol is complicated by the fact that the "owner" of $(x_i - a, y_j - b, z_k - c)$ does not know who is going to contact him, or when. Alternatively, the processes could exchange "halo" data, but significant amounts of the communicated data would be unused. On the KSR-1, the remote accesses to array elements at $(x_i - a, y_j - b, z_k - c)$ are automatically handled by the ALLCACHE memory system.

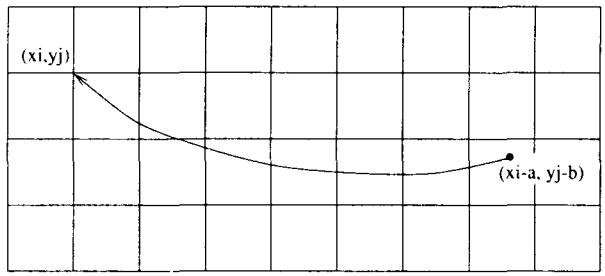

FIGURE 3 Two-dimensional Lagrangian interpolation.

6.2 BU_CU, BV_CV, UPDATE-U, UPDATE-V

Since the tridiagonal systems in these four segments are solved in parallel, each thread needs its own copy of the coefficient and right-hand side arrays. Because KSR Fortran does not support private arrays, a technique called array expansion had to be applied. Hereby, an array is expanded from $(n_1, \ldots, n_m)$ to $(n_1, \ldots, n_m, p)$ where p is the number of threads, so that thread $i (i = 1, \ldots, p)$ uses the memory locations starting at index $(1, \ldots, m, i)$.

Having done this, the measured run-time of the parallel version was disappointing. Using GIST, a tool for logging and visualizing events, serious load imbalance was detected.

Information from PRESTO revealed that some loops over `i` run from 1 to n_x and others from 2 to n_x. Tiling the former causes the first thread to start with index 1 and finish with some m_1, while in the latter case the same thread handles the range $2, \ldots, m_1 + 1$. Hence, data locality is not preserved, a situation which can lead to a significant number of remote data accesses. This was avoided by embracing all loops in LXY by a KSR Fortran affinity region [4], which ensures that for different tiled loops, the same values of indices are scheduled to the same threads, even though the loop bounds may be different. Although this mea-

sure did not solve the load imbalance problem and showed almost no performance benefit, we maintained it to ease experimenting with different tile sizes and strategies.

Finally we turned to PMON, a tool that gathers hardware monitoring statistics. This information showed that some threads had an extremely high rate of cache subpage misses, that is they were accessing a large number of data items located on other cells. Since this did not occur in BU or BV, we searched for some differences in the code which might account for this, and identified the "privatized" arrays as the source of the problem. The scalar expansion of arrays is an example where the effect of false sharing (see Section 4.2) can be severe. For instance, the elements `A(nz, 3, i)` and `A(1, 1, i+1)` lie in consecutive memory locations, the first being written to by thread i, the second by thread $i + 1$. If these two 8-byte data items happen to be on the same 128-byte subpage, this subpage is moved back and forth, causing unnecessary communication. The smaller the original array is (in our case it has only $3n_z$ elements), the higher the degree of false sharing in the extended array. By padding all extended arrays to subpage boundaries, we eliminated this effect and achieved much better load balancing.

6.3 UPDATE-W

After the experience gained in the previous steps, tiling this loop, including identifying the private variables and expanding some arrays, was trivial.

6.4 REST

The REST segments consist mainly of smaller loops scattered throughout LXY. Although they account for very little of the sequential execution time, it was important to parallelize them, as otherwise significant data movement will occur. Most of these loops are similar to COPY and were tiled trivially. Some other loops cover only the boundaries of the domain. The bodies of these loops should ideally be performed by the threads that own the corresponding x, y index pairs. However this is not easy to perform on the KSR-1 and since the performance gain would not justify the effort, we did not parallelize them. At the end of LXY, the flooding and drying of cells in the horizonal plane is handled. Introducing parallelism in these final loops would result in several threads writing to the same memory location, making the use of locks or critical regions necessary, and again any performance gain would not justify the effort involved.

6.5 SETUP

Due to the steps undertaken when optimizing the sequential code, it was straightforward to set up the pentadiagonal matrix in parallel maintaining the correct (sequential) order of the rows.

6.6 PENTA

In the optimized sequential version the call of the JCG routine in ITPACK took only 6.7% of the total time, but this increased as the parallelization steps progressed. Eventually, having carried out parallelization of all other segments, about half of the elapsed time (using 16 cells) was spent in PENTA.

ITPACK handles vector operations through level 1 BLAS-like routines while matrix–vector multiplications are adapted to the structure of the data type containing the sparse matrix. These subroutines contain a single loop of length N (where N is the dimension of the linear system—in our case $N = n_x n_y$) as opposed to two outermost loops of length n_x and n_y encountered in the previous sections. Therefore we tiled ITPACK loops specifying that they should not be part of the enclosing affinity region.

As a consequence, some data movement will occur at the beginning and the end of the iterative procedure. After the first iteration, most data are local and do not move to other threads. Communication takes place in each iteration due to the scatter and gather of vectors in GAXPY-like operations ($Ax + b$) with a sparse matrix A, and the reduction phase in the parallel execution of dot products.

Note that the influence of rounding errors can change the result of parallelized floating-point vector sums. Therefore we maintained one sequential and one parallel version of the dot product. The former was used to check the correctness of all changes (as mentioned in Section 5), the latter for run-time measurements.

Finally, it is important to note that we are coping with some load imbalance in the parallelized version of JCG. Tile sizes produced by PRESTO are by default a multiple of 16 (see Section 4.3). Changing PRESTOs default (for instance to a multiple of one) leads to better load balance, but results in false sharing. Our experiments showed that in this trade-off between load imbalance and

false sharing. the former caused the smaller performance penalty. This can be explained by noting that most operations in JCG are level 1 BLAS-like routines. where the ratio of accessed data to computation is high. We therefore maintained the original PRESTO default value.

7 RESULTS

To obtain consistent run-time measurements we removed from our experiments the output of data. During a typical production run output is only produced infrequently. Furthermore. we performed six time steps but measured only the elapsed time for the second to the sixth. Hereby we masked out the influence of the first time step which on the KSR-1 is more expensive than subsequent ones. due to page misses caused by accessing uninitialized data for the first time. In a normal run consisting of thousands of time steps the effect of this is negligible.

When comparing the sequential and parallel run-times. we had to consider that the parallel execution of floating-point sums in PENTA will perturb the data. and consequently the number of iterations performed by JCG may differ in the sequential and parallel version. In our experiments—time steps two to six—this did not occur. resulting in a "fair" comparison. Longer runs using the parallel version have shown that the number of iteration steps performed by JCG has a small variance. It is therefore reasonable to extrapolate the performance results of short runs to long ones.

A profile of the code showed that subroutine TRI. which merely interpolates the velocity of an interior cell point from the velocities at the eight corner points. is called nearly 1 million times in every time step. All other subroutines are called considerably fewer times and contain considerably more computation. We decided therefore to inline TRI at compile time. but no other routines.

Before presenting the obtained results. we describe the chosen tile size and tiling strategy for all loops outside PENTA. Load balance is usually achieved by having more tiles than cells and distributing the tiles in a modulo fashion. When we did this however. we discovered that the memory requirements per thread were not decreasing with the number of cells. Remember that our problem requires around 43 Mbyte of memory and that each thread should access roughly one pth part of it, where p is the number of cells. This effect can be explained by considering the layout of Fortran arrays within the KSR-1 memory architecture. Note that $n_z \times n_y \times 8$ bytes $\approx$16 Kbyte, which is the size of a page. Thus, the access for instance of `u(k0, j0, i0)` will cause the page containing the elements `u(k, j, i0)` ($k = 1, \ldots, n_z$, $j = 1, \ldots, n_y$) to become resident on the requesting cell. With the modulo tiling strategy. we can expect that each thread will use almost every value of i. Hence. almost every page is requested by every thread. even though only a few of its subpages are actually used by any one thread. Note that this false sharing of pages is different from the false sharing of subpages encountered in Section 6.2. To avoid this problem we let each thread work on exactly one tile of size $\lceil n_x/p \rceil \times n_y$ (equivalent to tiling over `i` in a `slice` fashion) so that each of the p threads requires only a pth of all pages and accesses all subpages within them. This does however result in some load imbalance—see below.

Table 2 contains the run-times obtained following the above experimental description. Each run was repeated three times: Table 2 presents the best value of three. Furthermore. all runs were executed with the *allocate_cells* command. which ensures exclusive use of a given number of cells. By specifying in the tile statements the same number of threads as allocated cells. we achieved a one-to-one mapping between cells and threads. Finally. for the experiments using up to 32 cells. we ensured that all the cells were on the same ring.

Note the discrepancy between the sequential and the one-cell parallel time. This can be explained by the increase in memory requirements caused by array expansion. Note also that the four cell time is less than half that on two cells. When using one or two cells. there is not enough memory

Table 2. Elapsed Time Per Time Step in Seconds for the Sequential and Parallel Version of SW3D

		Number of Cells								
1 (seq)	1 (par)	2	4	8	16	24	32	40	48	56
122	126	62.9	30.3	15.8	8.3	5.6	4.7	4.3	3.6	3.5

to hold all of SW3D's data, and it is necessary for the operating system to place some data in other cells' memories. With four or more cells, however there is enough memory, and the number of remote accesses are considerably reduced. We also see that the use of more than 48 cells gives almost no further reduction of the run-time.

Figure 4 shows the simulation performance in time steps per second. The naive ideal performance is the reciprocal of the naive ideal time, which is computed by simply dividing the execution time of the optimized sequential code by the number of cells. To check that extrapolation of our results to long runs is indeed valid, we ran 6,000 time steps on 32 cells (using both rings). This simulation took 7 hours 47 minutes, which corresponds to 0.21 timesteps per second, the same value as we obtained from measuring five time steps.

We have performed an analysis of the parallel overheads in order to identify the major factors causing the discrepancy between the naive ideal and the actual performance. We define the total overhead as the difference between the actual measured time and the naive ideal time. We then apportion the total overhead into four categories as follows:

1. Unparallelised code. This is the overhead incurred due to the parallel version containing sections of sequential code.
2. Load imbalance. This is the overhead that results from processors having to wait at a synchronization point for other processors to finish their parallel tasks.
3. Memory accesses. This the overhead due to the parallel and sequential versions spending different amounts of time accessing data. Note that data accesses include both local (same cell) and remote accesses.
4. Synchronization and scheduling. This is the overhead caused by the implementation of synchronisation points (in SW3D these are all barrier synchronisations), and the scheduling of tiles to threads by the PRESTO run-time system. These are grouped together because they are both associated with the addition of tile directives to the code.

Note that this analysis is somewhat complicated by the fact that the sequential version makes a substantial number of remote memory accesses, because there is too much data to fit in the memory of a single cell. This makes the naive ideal time

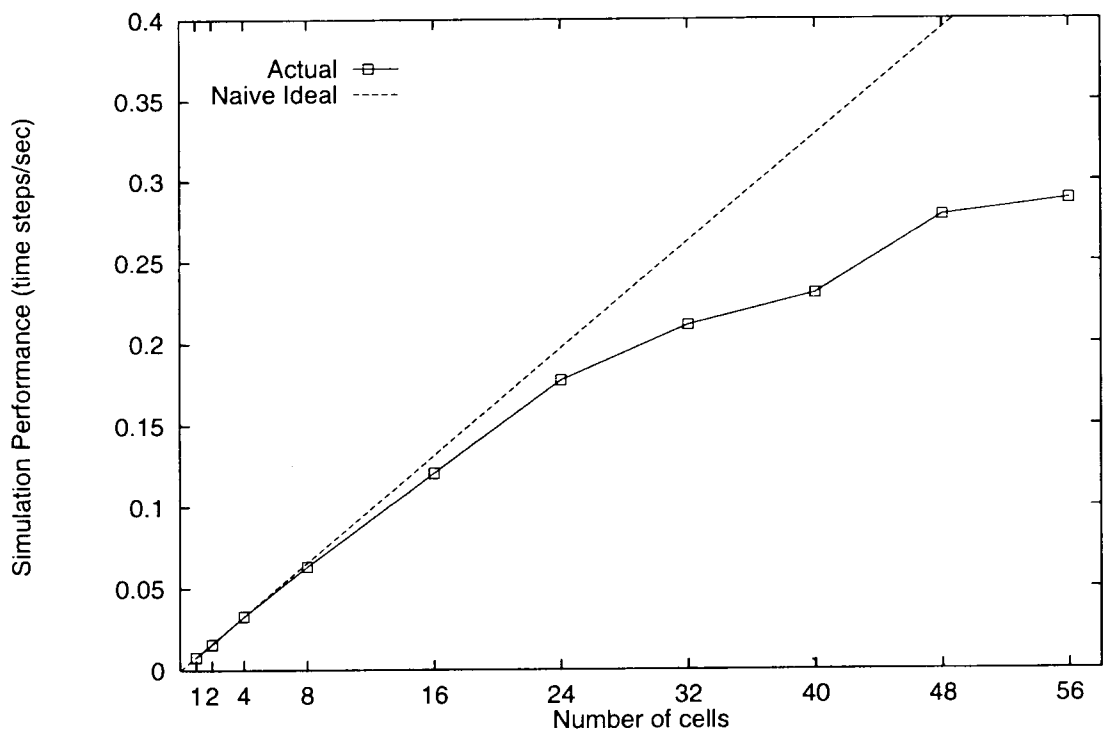

FIGURE 4 Simulation performance of SW3D.

somewhat pessimistic, and results in negative values of memory access overhead. In such a situation the time spent by the parallel version in making remote data accesses is a useful additional statistic, as it gives a better impression of the performance loss resulting from communication of data between processors.

The results of this analysis for 16, 32, and 48 cells are displayed in Table 3. The analysis shows that the total overheads are increasing as the number of cells increases, such that on 48 processors the overhead accounts for nearly 30% of the measured execution time. This suggests that with the present parallelization strategy, it is unlikely that the execution time could be reduced below 3 seconds per time step for a problem of this size, no matter how many processors were used.

It is clear that load imbalance is the most significant source of overhead. About half of this load imbalance can be attributed to the uneven assignment of indices to threads, since 16, 32, and 48 are not divisors of n_x. For example, using 32 processors the tile size is 11, which results in two processors being idle. This could be ameliorated by transforming all double-nested loops over i and j into one loop from 1 to $n_x \times n_y$.

The remaining load imbalance cannot be explained by uneven assignment of indices to threads in PENTA (see Section 6.6). Using hardware monitoring information we discovered that in many of the tiled loops, some threads are stalled waiting for data almost twice as long as others, although all have the same workload. This additional stalling is not caused by remote accesses, but by subcache misses. The implication of this is that for certain values of i and j there is considerably more overhashing (and hence displacement) of subcache lines, than for others. The precise cause of this is currently not clear, but we believe it may be a side effect of having a number of large arrays with a variety of sizes. More research is needed to understand and overcome this overhead source, since it causes significant degradation of performance with increasing numbers of cells.

The next most important source of overhead is the unparallelized code. Most of this code is concerned with setting boundary conditions, and again with some more effort it may be possible to parallelize some of these sections, although, of course, doing so may increase the overheads from other sources. There is little that can be done to reduce the cost of synchronization and tile scheduling. In each time step around 200 parallel loops are executed (there is some variation depending on the number of steps required in the conjugate gradient solver). Experiments have shown that each parallel loop incurs a synchronization and scheduling overhead of 1 ms for small numbers of cells, rising to 1.4 ms on 56 cells. Finally we note that remote data accesses are the least significant source of performance loss, hence there is no point in attempting to reduce the number of remote accesses before the other more significant sources of overhead have been addressed.

Table 3. Overhead Analysis per Time Step

	Number of Cells		
	16	32	48
Measured time	8.30	4.75	3.60
Naive ideal time	7.60	3.80	2.55
Total overhead	0.70	0.95	1.05
Unparallelized code overhead	0.40	0.40	0.40
Load-imbalance overhead	0.70	0.75	0.45
Memory access overhead	−0.60	−0.45	0.00
Synchronization/ scheduling overhead	0.20	0.25	0.30
Remote access time	0.15	0.15	0.15

Note. All times are in seconds and are given to the nearest 0.05 seconds.

8 FUTURE WORK

After the validation of the code with a simple geometry, we intend to apply it to a real-world estuary, Bideford Bay (southwest United Kingdom), which is used as a benchmark to allow a standardized approach to the testing and comparison of modeling software used for hydrodynamic and bacterial dispersion modeling [11]. This will require a more sophisticated load-balancing strategy in order to cope efficiently with highly irregular wet/dry boundaries which may be changing markedly with time. One possible strategy consists of allowing the boundaries between the partitions of the horizontal plane to change as the simulation proceeds, in such a way as to equalize the time spent by each processor [2].

Section 7 showed that further analysis is necessary to understand and overcome some overhead

sources, particularly the load imbalance, and the remaining unparallelized code. Note that the load imbalance problems should be solved by a dynamic load-balancing strategy. It would also be interesting to verify the parallelization strategy on other virtual shared memory architectures, such as the Cray T3D and Convex Exemplar.

There are several planned enhancements to the computational model, including pollution transport, sediment transport, and plume dispersion. Also a more sophisticated turbulence model involving the transport of Reynolds stresses directly is planned. Algorithms for biological and chemical behavior of pollutants are also desirable. Since the governing equations for each of these additions are of essentially similar form to those studied here, and are assumed to be largely uncoupled from the hydrodynamics equations, we can expect to apply the same algorithmic structure and parallelization strategy.

We also envisage some enhancements to the numerical scheme, including the avoidance of the time step limitation due to the explicit treatment of horizontal mixing. This could be achieved by an implicit treatment, possibly as a fractional step process. A greater challenge, especially for parallelization, will arise from the introduction of adaptive mesh refinement. For example we might adopt a strategy where mesh sizes are successively halved in proportion to the inverse of water depth and spatial flow gradients, e.g., vorticity [9].

9 CONCLUSIONS

We have described the parallelization of a three-dimensional shallow-water estuary model on the Kendall Square Research KSR-1. Although the semi-implicit Lagrangian scheme was initially described as an algorithm well suited for vectorization [1], we have found that its parallelization is natural and easy to perform, resulting in exceptionally efficient execution.

Recall that the time stepping solution process revolves around the solution of a pentadiagonal system of equations describing the evolution of the surface elevation. This system of equations can itself be solved in parallel, but parallelism can also be exploited in all other major computation segments such as the set up of the matrix and right-hand side coefficients for the system describing the new surface elevation. These matrix coefficients and right-hand side terms result from the solution of a set of independent tridiagonal systems of equations, one at each grid point in the horizontal plane. The parallel algorithm partitions the horizontal plane equally between threads, each one setting up and solving a group of independent tridiagonal systems. This partitioning approach is used for all other code segments, except for the conjugate gradient solver itself.

In practical terms we have demonstrated that a simulation which would require several days of CPU time on a powerful workstation or a modest vector processor can be run overnight on 32 cells of a KSR-1. We have also found that the development process, consisting of sequential optimizations followed by an incremental parallelization strategy, has given very good performance without an excessive amount of programmer effort. We have performed an analysis of the sources of overhead in the parallel version of the code, which has allowed us to identify the aspects of the parallelization strategy which are most in need of attention should it prove desirable to further reduce the run-time by using more processors.

ACKNOWLEDGMENTS

This work was funded by ESPRIT project 6253 (SHIPS).

REFERENCES

[1] V. Casulli and R. T. Cheng. "Semi-implicit finite difference methods for three-dimensional shallow water flow," *Int. J. Numerical Methods Fluids*, vol. 15, pp. 629–648. 1992.

[2] G. K. Egan, G. D. Riley, and J. M. Bull. *Proceedings of the 8th ACM International Conference on Supercomputing, 1994*.

[3] D. R. Kincaid, J. R. Respess, D. M. Young, and R. G. Grimes, "ITPACK 2C: A Fortran package for solving large sparse linear systems by adaptive accelerated iterative methods." *ACM. Trans. Math. Software*, vol. 8, pp. 302–322. 1982.

[4] K. S. R., KSR FORTRAN Programming. Kendall Square Research. 170 Tracer Lane, Waltham, MA, February 15, 1992.

[5] K. S. R., KSR Parallel Programming Guide, Kendall Square Research, 170 Tracer Lane, Waltham, MA, February 15, 1992.

[6] J. J. Leendertse, "Aspects of a computational model for long period water wave propagation," Memorandum RM-5294-PR. Rand. Corp., Santa Monica, California, 1967.

[7] P. M. Lloyd and P. K. Stansby. *Proceedings of*

IAHR symposium on Waves—Physical and Numerical Modelling, Vancouver, Canada, 1994.

[8] P. K. Stansby and P. M. Lloyd, "A semi-implicit Lagrangian scheme for 3-D shallow-water flow with 2-layer turbulence model," Engineering Department Report, University of Manchester, 1994 (Submitted to *Int. J. Numerical Methods Fluids*).

[9] C. P. Thompson, G. K. Leaf, and J. Van Rosendale, "A dynamically adaptive multigrid algorithm for the incompressible Navier-Stokes equations—validation and model problems," Argonne National Laboratory Preprint MCS-103-0989, 1991.

[10] Delft Hydraulics, "TRISULA: a program for the computation of non-steady flow and transport phenomena on curvilinear coordinates in 2 or 3 dimensions," Delft Hydraulics, April 1993.

[11] WRc, "Specification for benchmark testing programme for models for the purpose of marine hydrodynamic and bacterial dispersion modelling," FWR Project P-007, WRc, Henley Road, Buckinghamshire SL7 2HD, England, October 1993.