An environment for parallel and distributed computation with application to overlapping grids

by G. Chesshire
V. K. Naik

We describe an environment for efficient and scalable implementation of large scientific applications on parallel and distributed computing systems. We show how this environment is used to support overlapping grid methods. In addition to providing a user interface that reduces programming complexity, the environment facilitates dynamic partitioning of data and the scheduling of both computations and communication, transparent to the user. After describing the user interface and some of the implementation issues, we present performance data for a model application executed on two different systems: an eight-processor IBM Power Parallel Prototype (PPP) system and a 32-processor IBM POWER Visualization System™ (PVS).

Introduction

DSK is a portable scientific database package for managing hierarchical data structures in FORTRAN programs [1]. The DSK package maintains, in the form of a database, an image of the data structures in memory, which may be accessed subsequently by other programs. In this paper, we describe some extensions to the DSK package to support parallel and distributed computation. With these extensions, it is possible for a user to define distributed-data structures but not necessarily specify the data distribution explicitly. Furthermore, these extensions provide, at the user level, a uniform method of access to local and remote data and an efficient run-time support environment. The distributed nature of the data is transparent to the user, and so is the scheduling of communication. Under the environment provided by the extended DSK package, a user can write programs for distributed-memory systems that are not very different

from sequential programs. Moreover, execution of these programs does not involve any run-time preprocessing, as is the case in some other systems.

To demonstrate some features of DSK that make it a useful tool in the solution of partial differential equations (PDEs), we explain how we used it in the parallel solution of a time-dependent PDE on a composite overlapping grid [2]. This involved an interesting combination of regular and irregular types of computations: The parallel finite-difference computations within each component grid are characterized by regular data dependencies and communication patterns, whereas the computations associated with the intergrid interpolations exhibit irregular dependencies and unstructured communication patterns. The solution of the problem we selected provides a nontrivial example of the use of the DSK package for parallel computation.

We discuss the extended DSK package in the context of its applicability to a class of problems that has been referred to as *irregularly coupled, regular-mesh* computations [3]. This class includes overlapping-grid, block-structured grid, and adaptive-grid problems. Difficulties encountered in parallelizing the application programs belonging to this class are well known, and several attempts to reduce the programming burden have been made in recent years. These attempts include a suite of library calls, run-time preprocessors, and run-time environments managed by compilers. Class libraries in C++®, such as P++ [4] and LPAR [5], have been developed in recent years. LPAR provides a coarse-grain parallel-programming model and is primarily intended for applications with dependencies that may vary dynamically, such as those in N-body simulations. Multiblock PARTI [6, 7] is a preprocessor that analyzes dependences at run time and performs communication optimizations prior to commencing any numerical computations. Work has been described in [8] that extends the capabilities of multiblock PARTI so that its run-time library can be incorporated by compilers for FORTRAN D [9] and other High Performance FORTRAN (HPF) data-parallel programming languages. Both LPAR and PARTI perform run-time preprocessing to determine communication schedules. Task-oriented parallel languages, such as FORTRAN M, also provide facilities for handling irregularly coupled, regular-mesh applications [10].

The advantage of the DSK package over the other systems described above is that it allows one to efficiently execute FORTRAN programs on a variety of parallel architectures, with minimal user intervention. In addition, the DSK package performs no run-time preprocessing. As described in this paper, the overheads involved are small, and code modifications are minimal.

The organization of the rest of the paper is as follows. In the next section, we provide some background for numerical solution of PDE problems where overlapping grid techniques are commonly used. We also use that discussion to provide a motivation for the work we discuss in the rest of the paper. In the third section, we discuss, in some detail, a model problem that we solve using the overlapping-grid method, and we provide an outline of the parallel implementation of the model problem under the DSK environment. The details of the DSK package are given in the next section. The performance results from our experiments on two parallel systems are presented and discussed in the following section. The next section concludes the paper.

Background and motivation

Among scientific computations, the numerical solution of PDEs, such as the Euler and Navier–Stokes equations describing fluid flow, is of great interest both to the research community and to industry. With the help of an example that requires the solution of PDEs, we motivate the design and development of the extended DSK package; however, the tools we describe are applicable to a wider class of application programs in which large-array data structures are used.

The complexity of the computations for the numerical solution of PDEs, in terms of both data dependencies and programming effort, increases with the complexity of the associated geometry. For cases in which the geometry is simple, the physical domain can be discretized with a Cartesian grid. To properly handle physical domains with a more complex geometry, a boundary-fitted curvilinear grid may be used. Such a grid is generated from a Cartesian grid by a smooth coordinate transformation from the physical domain to a rectilinear computational domain. In general, the difference stencils used in discretizing the PDEs for numerical solution on Cartesian or curvilinear grids give rise to data dependencies that are local and regular in nature. For this reason, implementation of explicit finite-difference methods that use such stencils is easy, and implementation of parallel algorithms for these methods is not very difficult.

When the geometry is nontrivial, a single curvilinear grid is inadequate, and more powerful techniques must be used, such as patched-grid (block-structured) or overlapping-grid methods. With these methods, several curvilinear component grids are used, which together cover the physical domain. For patched-grid methods, adjacent component grids must match exactly along their common boundary. As a result, the data dependencies of explicit finite-difference methods for patched grid are such that computing the difference stencils at the boundary points of component grids requires an exchange of rectangular blocks of data among adjacent grids. This makes patched-grid methods difficult to implement, especially when a parallel implementation is required.

For overlapping-grid methods, functions defined on the composite grid are connected by means of interpolation between the component grids in their region of overlap. As a result, finite-difference methods for overlapping grid use interpolation to complete the computations of the difference stencil at all points in the region of overlap. Once the interpolation functions are determined, these computations turn out to be more straightforward than those with patched grid. However, the parallel implementations of the finite-difference methods for overlapping grid are more complex, since they involve less structured exchange of data between component grids. Moreover, correct implementation of an efficient parallel algorithm is much more difficult. Our interest in the success of patched- and overlapping-grid methods has motivated us to develop tools to manage these difficulties. We consider such tools indispensable for the development of parallel algorithms and their implementations.

We discuss in some detail the implementation of finite-difference methods on overlapping grids, because this class of grids includes as special cases patched grids, single curvilinear grids, and Cartesian grids. The methods and tools we describe are applicable to all of these types of grids. A composite overlapping grid is a set of curvilinear, logically rectangular grids, each of which covers part of a physical domain to be discretized. The component grids that are adjacent to one another overlap, so that functions defined on the composite grid may be interpolated from one component grid to the other. **Figure 1** shows a simple two-dimensional overlapping grid, with two components—a square grid (with some redundant cells eliminated) and an annular grid. This overlapping grid was generated using the program CMPGRD [2, 11], which determines what cells on each component grid may be used in the discretization of the PDEs over the entire grid and what cells may be interpolated from other component grids. In Figure 1, the interpolated cells are marked with small circles at their centers. A typical overlapping grid for a domain of modest complexity consists of two to ten component grids, each containing one thousand to one million grid points.

In recent years, the available physical memory for user data on uniprocessor systems has increased manyfold; however, there will always be PDE problems we would like to solve that are too large for any uniprocessor system to handle. We expect that the largest parallel computers available in the near future will be of the distributed-memory type, so we would like to be able to use these to solve PDE problems on overlapping grids. We have implemented software tools to facilitate the implementation of efficient parallel algorithms for the solution of PDEs with the least possible programming effort. With this system of tools, it is possible to transform an explicit sequential time-stepping or iterative PDE algorithm into a parallel algorithm, with only small changes to the method

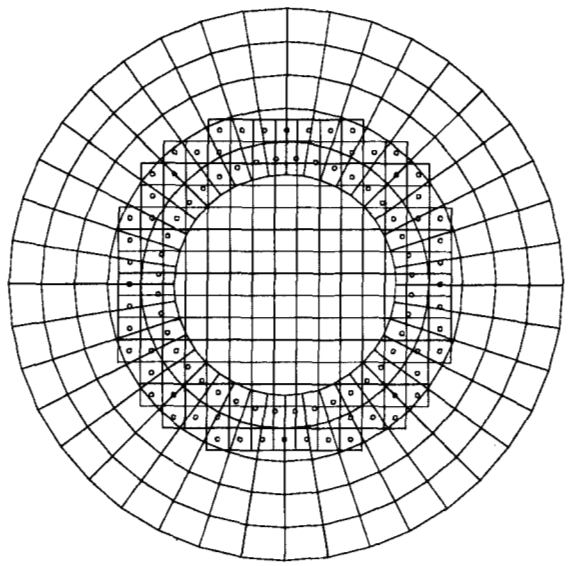

Figure 1

Composite overlapping grid for a disc. Interpolated cells are marked with circles at their centers.

and to its implementation in a FORTRAN program. This provides a framework for the use of overlapping grids to solve PDE problems that are too large to be solved on a uniprocessor system.

Model application

In this section, we describe our model problem in some detail. In the following sections, we illustrate the application of the extended DSK package for the parallel implementation of the numerical solution of this model problem. The problem we consider is that of the solution of Burger's equation (which describes a nonlinear wave equation) over a circular disc. This is a relatively simple example with sufficiently complex geometry. A discretization of such a body using overlapping grid is shown in Figure 1. Note that, for simplicity, the model problem is a two-dimensional example, although the techniques described are equally applicable to three-dimensional problems.

We consider a two-dimensional analogue of Burger's equation,

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left(\frac{1}{2}u^2\right) = \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right), \tag{1}$$

with the initial condition

$$u(x, y, 0) = u_0(x) := c - \tanh\frac{x - x_0}{2\nu},$$

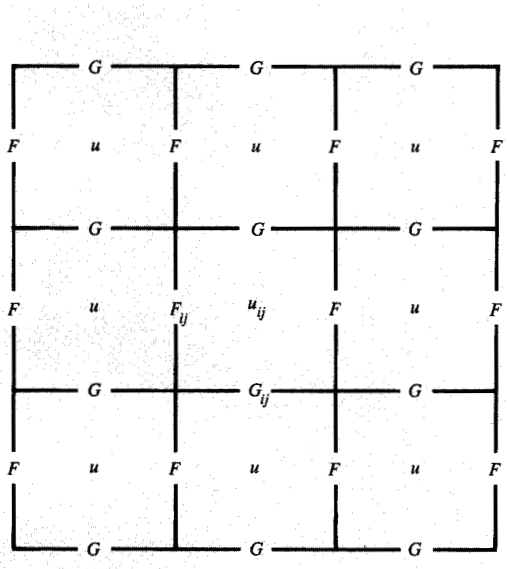

Figure 2

Grid functions for discretization of model problem. The u's are at the cell centers, and F and G are the fluxes at the cell edges.

where ν is the coefficient of diffusion. The exact solution to this equation is

$$u(x, y, t) = u_0(x - ct),$$

which is a shock layer moving to the right with speed c. Equation (1) has the form

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) + \frac{\partial}{\partial y} g(u) = 0, \tag{2}$$

where

$$f(u) = \frac{1}{2} u^2 - \nu \frac{\partial u}{\partial x}$$

and

$$g(u) = -\nu \frac{\partial u}{\partial y}.$$

For the solution of the above PDE by finite-difference methods, overlapping boundary-fitted curvilinear grids may be used. To discretize, we first transform Equation (2) from the (x, y) coordinate system to an (r, s) curvilinear coordinate system, so that the grid spacing in the new system is uniform and of unit length. This transforms the physical domain in x–y space into an r–s computational space that is a rectangular domain and has a regular uniform mesh. Then, Equation (1) becomes

$$\frac{\partial u}{\partial t} + J^{-1}\left[\frac{\partial}{\partial r} F(u) + \frac{\partial}{\partial s} G(u)\right] = 0, \tag{3}$$

where

$$J = \frac{\partial x}{\partial r}\frac{\partial y}{\partial s} - \frac{\partial y}{\partial r}\frac{\partial x}{\partial s},$$

$$F(u) = \frac{\partial y}{\partial s} f(u) - \frac{\partial x}{\partial s} g(u),$$

and

$$G(u) = \frac{\partial x}{\partial r} g(u) - \frac{\partial y}{\partial r} f(u).$$

We discretize u, F, and G on a staggered Cartesian grid with unit mesh-size, as shown in **Figure 2**. We use second-order centered averaging and difference operators to compute F_{ij} in terms of $u_{i-1,j}$, $u_{i-1,j\pm1}$, u_{ij}, and $u_{i,j\pm1}$, and to compute G_{ij} in terms of $u_{i,j-1}$, $u_{i\pm1,j-1}$, u_{ij}, and $u_{i\pm1,j}$. Then we discretize Equation (3) in space, to second-order accuracy, as

$$\left(\frac{du}{dt}\right)_{ij} + A_{ij}(\Delta_{+i} F_{ij} + \Delta_{+j} G_{ij}) = 0, \tag{4}$$

where

$$A_{ij} = (J^{-1})_{ij},$$

$$\Delta_{+i} F_{ij} = F_{i+1,j} - F_{ij},$$

and

$$\Delta_{+j} G_{ij} = G_{i,j+1} - G_{ij}.$$

That is, we use a nine-point difference stencil to compute $(du/dt)_{ij}$.

To discretize Equation (3) on an overlapping grid, we must supply interpolation boundary conditions in the regions of overlap among the component grids. On any component grid k, a cell where we want to update u from Equation (4) is referred to as a *discretization cell*. Clearly, such a cell must be surrounded by other cells where the solution to u is known. These surrounding cells may be other discretization cells, they may be cells for which the Dirichlet boundary condition provides the value of u, or they may be cells where u is interpolated from another grid k'. A cell where u is computed using a specified interpolation boundary condition is referred to as an *interpolated cell*. We use biquadratic Lagrange interpolation from a stencil of nine cells centered on the cell of grid k' nearest to the interpolated cell on grid k. As mentioned in the previous section, we use the program CMPGRD to generate the overlapping grids and to determine, for each grid k, the list of interpolated cells that should receive values interpolated from other grids.

In the example of Figure 1, there are only two overlapping component grids, but in general, there may be more than two. In all such cases, CMPGRD determines from which grid k' each interpolated cell should receive its values and provides the location of that cell within grid k'. From this information, the interpolation coefficients for each interpolated cell can be computed.

To discretize in time, we use the classical fourth-order four-stage Runge–Kutta time-stepping method, in which u^{n+1} (the solution at time step $n + 1$) is computed from u^n (the solution at time step n) as follows:

$$u_{ij}^{n+1} = u_{ij}^n + \frac{1}{6}(v_{ij}^{(1)} + 2v_{ij}^{(2)} + 2v_{ij}^{(3)} + v_{ij}^{(4)}), \tag{5a}$$

where

$$v_{ij}^{(1)} = -\Delta t A_{ij}[\Delta_{+i} F_{ij}(u^n) + \Delta_{+j} G_{ij}(u^n)], \tag{5b}$$

$$v_{ij}^{(2)} = -\Delta t A_{ij}\left[\Delta_{+i} F_{ij}\left(u^n + \frac{1}{2}v^{(1)}\right) + \Delta_{+j} G_{ij}\left(u^n + \frac{1}{2}v^{(1)}\right)\right], \tag{5c}$$

$$v_{ij}^{(3)} = -\Delta t A_{ij}\left[\Delta_{+i} F_{ij}\left(u^n + \frac{1}{2}v^{(2)}\right) + \Delta_{+j} G_{ij}\left(u^n + \frac{1}{2}v^{(2)}\right)\right], \tag{5d}$$

and

$$v_{ij}^{(4)} = -\Delta t A_{ij}[\Delta_{+i} F_{ij}(u^n + v^{(3)}) + \Delta_{+j} G_{ij}(u^n + v^{(3)})]. \tag{5e}$$

The initial condition of Equation (1) defines u_{ij}^0. The time step for the above Runge–Kutta method is chosen close to the stability limit. Because of the nonlinearity, we experimentally determined an appropriate time step. In particular, we use

$$\Delta t = 0.3\frac{h^2}{\nu},$$

where

$$h = \min\left\{\sqrt{\left(\frac{\partial x}{\partial r}\right)^2 + \left(\frac{\partial y}{\partial r}\right)^2}\,\Delta r,\ \sqrt{\left(\frac{\partial x}{\partial s}\right)^2 + \left(\frac{\partial y}{\partial s}\right)^2}\,\Delta s\right\}.$$

The minimum here is taken over all grid points on all component grids. In advancing the solution by one time step in the Runge–Kutta method, one computes the intermediate solutions $v_{ij}^{(k)}$ of Equation (5) in four stages, $k = 1 \cdots 4$.

In the following, we present a sequential algorithm for the implementation of stage k of the Runge–Kutta method in the context of overlapping grids.

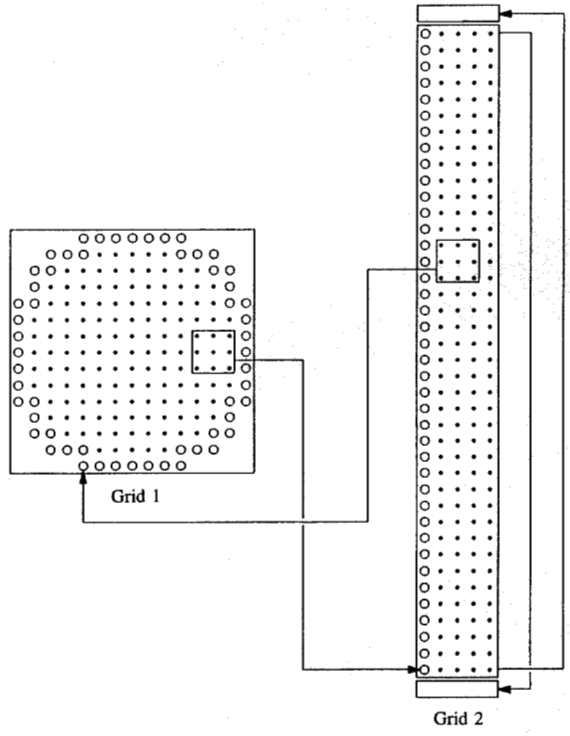

Figure 3

Interpolation and periodic-boundary update of a function defined on the composite grid of Figure 1. Cells marked with circles are interpolated from nine cells (shown bounded by a box) of the other grid. The one-cell-wide strips on opposite sides of Grid 2 are updated in the direction of periodicity.

Overlapping-grid Runge–Kutta algorithm for stage k

1. Interpolate $u^n + \alpha_{k-1}v^{(k-1)}$ (or u^n only, for stage $k = 1$) for the interpolated cells in the region of overlap between component grids, where $\alpha_1 = \alpha_2 = 1/2$ and $\alpha_3 = 1$.
2. Copy boundary values $u^n + \alpha_{k-1}v^{(k-1)}$ (or u^n) from opposite sides of the same component grid, for the cells at which the grid has periodic boundary conditions.
3. For each discretization cell, compute $v^{(k)}$ from $u^n + \alpha_{k-1}v^{(k-1)}$ (or u^n) [as specified in Equation (5)].
4. If $k < 4$, compute and save the sum $u^n + \alpha_k v^{(k)}$; if $k = 1$, set u^{n+1} to $u^n + \beta_1 v^{(1)}$; if $k > 1$, accumulate $\beta_k v^{(k)}$ into u^{n+1}, where $\beta_1 = \beta_4 = 1/6$ and $\beta_2 = \beta_3 = 1/3$.
5. Apply boundary conditions to $u^n + \alpha_k v^{(k)}$, if $k < 4$; otherwise to u^{n+1}.

Figure 3 illustrates the implementation of steps 1 and 2, the interpolation and periodic update of a function

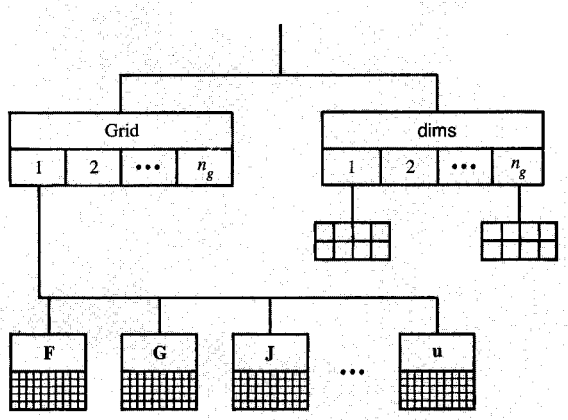

Figure 4

Composite-grid data structure (simplified).

defined on the composite grid shown in Figure 1. Grid 1 corresponds to the square grid (with some cells removed), and grid 2 corresponds to the annular grid, which has a periodic boundary condition in one direction. To interpolate u^n on grid 1 in the region of overlap (step 1), for each cell of grid 1 marked with a circle at its center, we compute u^n as the weighted sum of u^n in a 3×3 block of cells of grid 2. The cell in the center of this block is the nearest cell to the cell interpolated on grid 1. We use Lagrange interpolation to determine the weights in the sum. Step 2 primarily serves the purpose of implementing periodic boundary conditions in a convenient manner. Note that for the example of Figure 1, step 2 is applicable only to grid 2, which is periodic in one direction. For convenience in implementing the discretization given by Equation (4), we extend grid 2 by an extra row of cells (a one-cell-wide strip at the two opposite sides, in the direction of periodicity (in Figure 3, this is depicted by empty rectangular boxes at the top and bottom edges of grid 2), and replicate the values of u^n from the interior edge of grid 2 into the opposite edges of grid 2 into the extended strips (direction of arrows indicates the direction of replication).

- *Data structures*

One function of the DSK package [1] is to manage data structures. This package provides facilities for managing lists and arrays (including arrays of lists and lists of arrays) in FORTRAN programs. We use (but do not describe here) the database features of DSK, which allow us to access data structures created by the composite-grid-generation program CMPGRD. Later in this paper, we describe some features of DSK that enable us to implement an efficient parallel version of the solution of our model problem, with only small modifications to the algorithm and its implementation in FORTRAN.

In general, it is useful to implement overlapping-grid PDE code in such a way that the number of component grids and their sizes need not be known in advance. To achieve this, we allocate memory for the data structures that describe the component grids and the discrete solution of the problem. **Figure 4** shows a simplified version of the hierarchical data structure used in the model problem. In this figure, data structure dims stores the dimensions of each of the n_g component grids. Each component grid contains some large arrays, like **u** (which holds the solution), whose dimensions are the number of grid cells in each direction. The boxes in Figure 4 labeled **F**, **G**, **J**, and **u** are the large arrays associated with grid 1. Associated with each component grid are also some small arrays (not shown in the figure) that contain information about the grid, such as its periodicity, and some intermediate-sized arrays containing interpolation data. This data structure is sufficiently flexible to describe a PDE computation on overlapping grids with any number of component grids, all of different sizes.

- *Parallel implementation*

A natural way to exploit the parallelism inherent in the solution of a PDE problem is to subdivide each component grid (in one or more directions) into blocks and to assign one or more blocks of the various component grids to each processor. In general, each processor may be assigned blocks of more than one component grid. Since there may be any number of component grids, each of a different size, it is not possible in general to partition the component grids so that all of the blocks are of the same size (i.e., have the same amount of work) and to assign exactly the same number of blocks to each processor. Thus, in any block-to-processor assignment scheme, the computational work load may not be balanced across processors. Moreover, the number of blocks into which each component grid should be divided in each direction is a compromise between load balance and communication complexity. The load balance may be improved by choosing the total number of blocks to be larger than the number of processors, estimating the amount of work associated with computations on each block, and assigning a set of blocks to each processor so that the total work from all of the assigned blocks is approximately the same for all processors. This strategy, however, may significantly increase the communication overhead. On the other hand, if the component grids are partitioned into a few large blocks, locality in computation can be exploited efficiently, leading to relatively less interprocessor communication. However, with this strategy, there may

not be enough blocks to evenly distribute the computational work among processors.

The first step in assigning blocks to processors is to assess the work associated with each block. A count of the number of discretization cells and interpolation cells in a block provides one estimate of the work associated with that block. When work per cell is uniform, this estimate is quite accurate. Quite often, however, the work associated with interior cells differs significantly from the work associated with the cells on the domain boundary, the cells where the values are interpolated. In such cases, the computational work per block may be measured at run time, after which the blocks may be assigned to different processors appropriately. With such a technique, computational work can be distributed evenly among processors, but this is possible only at the cost of significant run-time overhead in estimating the computational work and distribution of blocks among processors. Generally, this technique is more suitable for fine-tuning the computational workload among processors. The extended DSK package estimates the work per block to be proportional to the number of discretization cells and interpolation cells in the block. On this basis, the blocks are ordered in descending sequence of their computational work and then assigned to processors, using a bin-packing type of algorithm.

The control structure of our parallel implementation of one stage of the Runge–Kutta method on overlapping grid is almost identical to that in the sequential algorithm. The main differences are that (1) the interpolation (step 1 of the sequential algorithm) is split into two steps and (2) the update for periodicity (step 2) becomes an update of the overlaps between neighboring blocks of the same grid. In the distributed algorithm, each processor works on only the blocks that are assigned to it, and before each Runge–Kutta stage, we ensure that the data needed for the computations on each block are available on the processor to which the block is assigned. Specifically, at stage k of the distributed Runge–Kutta algorithm for overlapping grid, each processor performs the following operations on every block b of each component grid assigned to it.

Distributed Runge–Kutta algorithm for stage k

1. Sum the contributions from blocks of other grids to the interpolation of $u^n + \alpha_{k-1} v^{(k-1)}$ (u^n, if $k = 1$) at the interpolation cells (if there are any) of block b, where $\alpha_1 = \alpha_2 = 1/2$ and $\alpha_3 = 1$.
2. For each discretization cell, compute $v^{(k)}$ from $u^n + \alpha_{k-1} v^{(k-1)}$ (or u^n).
3. If $k < 4$, compute and save the sum $u^n + \alpha_k v^{(k)}$; if $k = 1$, set u^{n+1} to $u^n + \beta_1 v^{(1)}$; if $k > 1$, accumulate $\beta_k v^{(k)}$ into u^{n+1}, where $\beta_1 = \beta_4 = 1/6$ and $\beta_2 = \beta_3 = 1/3$.
4. Apply boundary conditions to $u^n + \alpha_k v^{(k)}$, if $k < 4$; otherwise to u^{n+1}.
5. For the overlapping parts of neighboring blocks of the same component grid, send updates to $u^n + \alpha_k v^{(k)}$, if $k < 4$; otherwise to u^{n+1}.
6. Compute and send contributions from block b to the interpolation of $u^n + \alpha_k v^{(k)}$ (or u^{n+1}) on blocks of other component grids.

Note that steps 1 and 6 in the above algorithm complete step 1 of the sequential algorithm, and step 5 replaces step 2. In the following, we explain these modifications in some detail.

For updating the solution at a discretization cell, values at the eight neighboring cells are necessary. When a component grid is partitioned into blocks, the neighboring cells of a discretization cell may belong to another block. Thus, the solution update at such cells requires fetching data from cells belonging to one or more other blocks. For implementation convenience, we extend the data structures associated with each block b so that they store the appropriate data, from the neighboring blocks, needed in the computations of the discretization cells of block b. This results in an "overlap" among blocks of the same component grid. These overlaps are shown as empty rectangular boxes in **Figure 5**. In that figure, for the sake of clarity, blocks are moved apart from one another, and the block extensions are indicated by narrow strips of empty rectangular and square boxes. The arrows indicate the relations between the boundary regions and the extensions from neighboring blocks. Thus, in step 5 of the distributed Runge–Kutta algorithm, the $u^n + \alpha_k v^{(k)}$ values computed at the discretization cells on the boundary of a block are copied into the extended parts of neighboring blocks of the same component grid. This copying may involve interprocessor communication if the two blocks reside on separate processors.

Steps 1 and 6 of the distributed algorithm complete the interpolation part. This is illustrated in **Figure 6**. To understand the interpolation step, it is convenient to think of interpolation as consisting of step 6 followed by step 1, since these steps follow each other in going from one Runge–Kutta stage to the next, or from one time step to the next. When a processor reaches step 6 of stage k, it has finished computing $u^n + \alpha_k v^{(k)}$ for the current block, so it has the information needed to compute the contributions from this block to the interpolation of cells of other grids. It computes these contributions as weighted sums and stores them temporarily in a local array, sorted according to which blocks of other grids will need them. These contributions must be made available to the processors where they will be needed later for step 1 of the next stage or the next time step. This data movement is made possible by interfacing with the DSK package environment, as explained in the following subsection. The interpolation in step 1 is completed on each processor,

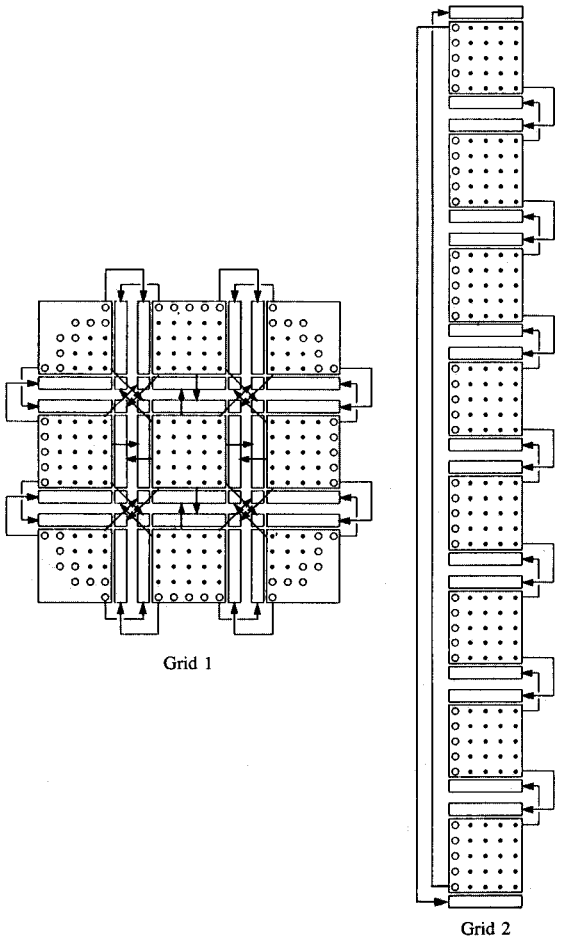

Figure 5

Block boundary update of a distributed composite-grid function, for the composite grid shown in Figure 1.

once all required contributions from blocks of other grids are made available.

• *Interface to DSK*

As mentioned earlier, the DSK package provides a convenient means for managing data structures that are typically associated with overlapping-grid computations. In addition, with the extended DSK package, the distributed computations described above can be implemented efficiently and with significantly reduced programming efforts. Using the model problem described above as an example, we briefly describe the DSK interface to user programs.

Shown in **Figure 7** is an outline of the outer loop of the parallel Runge–Kutta algorithm, where the solution is advanced by one time step in each loop iterate. The calls to subroutine RKStage perform the computations corresponding to the Runge–Kutta stages, which are outlined in **Figure 8**. We set a scheduling point before performing the computations for each Runge–Kutta stage and at the end of each time step. Such markings in the control flow of the computations indicate to DSK the progression of computations on each processor.

Setting a SCHEDULE_POINT essentially translates into calling dsksch of the DSK package with the appropriate key word, BEGIN, MIDDLE, or END. We explain the details of dsksch in the following section. For the current discussion, it is sufficient to note that the interprocessor dataflow remains the same from one time step to the next and that the schedule points are meant to take advantage of this repetitive pattern in scheduling computations on a processor as well as in scheduling interprocessor

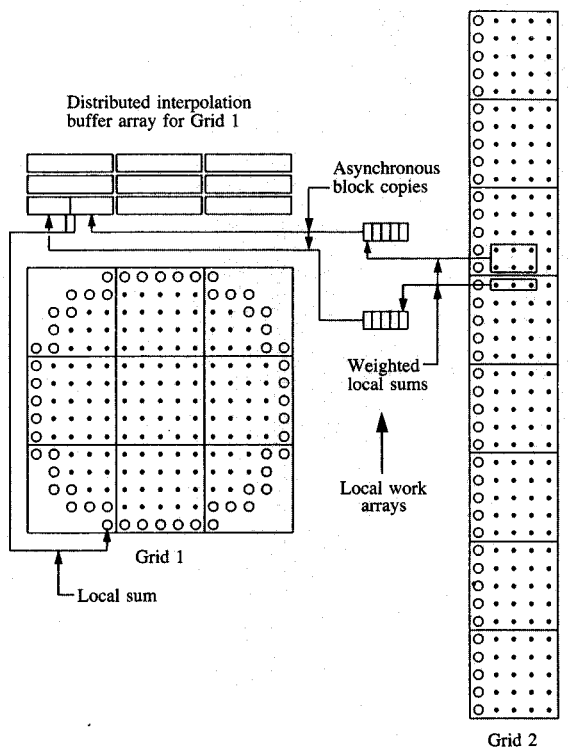

Figure 6

Interpolation of a distributed composite-grid function, for the composite grid of Figure 1. Contributions from each block of Grid 2 to the interpolation of cells in a block of Grid 1 are accumulated in a work array local to the processor assigned to the block. The local array is copied asynchronously into a distributed interpolation buffer for Grid 1. Later, the partial sums in the interpolation buffer are added together into the interpolated cells of the block of Grid 1.

communication. The schedule points marked as BEGIN and END demarcate this repetitive pattern. [Since a BEGIN mark follows an END mark of the previous time step, it is possible to replace these two with a single mark and achieve the same functionality. (This will, of course, require an additional marking, either before entering the loop or upon exiting the loop.) For maximum flexibility and convenience, however, we use both BEGIN and END markings.] The interprocessor communication repeats itself even within a time step. In fact, the dataflow is analogous for each stage of the Runge–Kutta algorithm. The completion of one stage and the beginning of the next stage are indicated to DSK by the MIDDLE marking.

With the help of the schedule points, DSK manages data integrity across the system and schedules interprocessor communications in an efficient manner. In the next section, we describe the details of the optimizations accomplished with this scheduling mechanism. In the section on performance results, below, we present experimental results showing the performance gains obtained by this type of scheduling.

As indicated above, Figure 8 is an outline of the parallel algorithm for performing a single stage of the Runge–Kutta method. There, B is a list of blocks assigned to a processor (also referred to as LocalBlocks), maintained by DSK for each processor. On each processor, the single-stage

Initialize arrays
Begin *solution for time step:*

Set SCHEDULE_POINT *to* BEGIN
call RKStage (k = 1)

Set SCHEDULE_POINT *to* MIDDLE
call RKStage (k = 2)

Set SCHEDULE_POINT *to* MIDDLE
call RKStage (k = 3)

Set SCHEDULE_POINT *to* MIDDLE
call RKStage (k = 4)

Set SCHEDULE_POINT *to* END

Continue *to next time step.*

Figure 7

An outline of the parallel Runge–Kutta time-stepping algorithm, showing the scheduling interface to the DSK environment.

Algorithm RKStage (k)

$B \leftarrow$ list of LocalBlocks
foreach $b \in B$ **do:**
  if $k = 1$
    GET_ARRAYS u^n and interp_buffers *for b*
    Interpolate u^n *at interpolation cells*
    Compute $v^{(1)}$ *from* u^n
    Initialize u^{n+1} *to* u^n
    Accumulate $\beta_1 v^{(1)}$ *into* u^{n+1}
  else
    GET_ARRAYS u^n, u^{n+1}, $(u^n + \alpha_{k-1} v^{(k-1)})$
      and interp_buffers *for b*
    Interpolate $u^n + \alpha_{k-1} v^{(k-1)}$ *at interpolation cells*
    Compute $v^{(k)}$ *from* $u^n + \alpha_{k-1} v^{(k-1)}$
    Accumulate $\beta_k v^{(k)}$ *into* u^{n+1}
  endif

  if $k < 4$
    Compute $u^n + \alpha_k v^{(k)}$
    Apply boundary conditions to $u^n + \alpha_k v^{(k)}$
    Compute local values for interpolation of $u^n + \alpha_k v^{(k)}$ *on block of other grids*
    SAVE_ARRAYS u^{n+1} *and* $u^n + \alpha_k v^{(k)}$
  else
    Apply boundary conditions to u^{n+1}
    Compute local values for interpolation of u^{n+1} *on block of other grids*
    SAVE_ARRAYS u^{n+1}
  endif
end foreach

end Algorithm RKStage

Figure 8

An outline of the parallel algorithm for performing a single stage of the Runge–Kutta method. GET_ARRAYS and SAVE_ARRAYS interface with the DSK environment.

computations are performed on each local block, one after another. GET_ARRAYS is an interface to DSK, which accomplishes retrieving the arrays necessary in the computations of the stage. This interface makes calls to DSK package routines dskdsw. Before returning control to the user program, DSK ensures that the variables associated with the boundary overlaps of these arrays are appropriately updated. With the values in the interpolation buffers, interpolations are performed at the interpolated cells of block b. Following this, $v^{(k)}$ is computed at

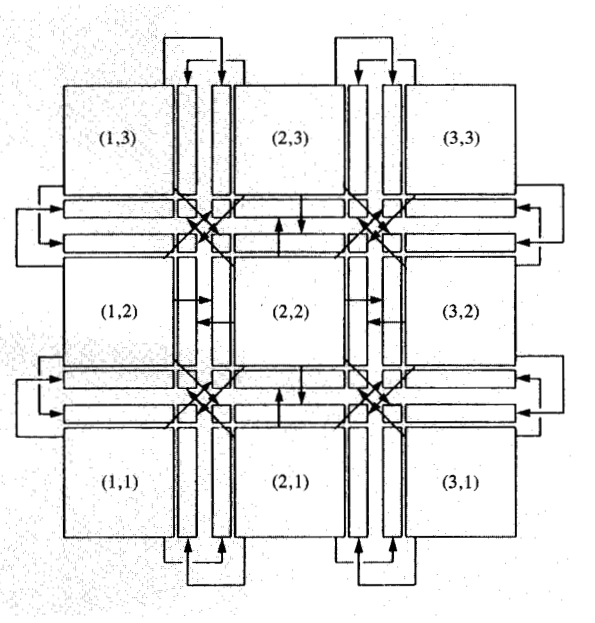

Figure 9

Overlapping blocks of an array.

the discretization cells of block b and appropriately accumulated into array u^{n+1}. Also computed are an array with values $u^n + \alpha_k v^{(k)}$ (which is used in the computations of the next stage of the same time step), the values at the grid boundaries according to the specified boundary conditions, and the interpolation values that may be needed for blocks belonging to other grids.

SAVE_ARRAYS forms another interface to DSK. In performing this function, DSK uses appropriate values from the arrays just computed to update the boundary overlaps of neighboring blocks, as well as to update the interpolation buffers of blocks belonging to other component grids. Note that some blocks may not have any data that are needed in interpolation. In that case, no interpolation data are retrieved. When there are boundary update data or data for interpolation, DSK copies the data to the proper arrays of the appropriate blocks. This is accomplished by a call to dskdcb. If the block to be updated is assigned to a remote processor, a message is sent, transparent to the user, and the DSK environment on the remote processor retrieves that message and copies the data into the proper array of the appropriate block. Details of dskdsw and dskdcb are presented in the next section.

Parallelization with the DSK package

In this section, we describe the implementation in the DSK package of array partitioning and mapping, as well as the interprocessor communication involved in accessing distributed arrays. We discuss in some detail the communication-optimization techniques embedded in DSK.

• *Array partitioning*

As mentioned earlier, with the DSK package, the individual arrays used by an application may be further partitioned into a user-specified number of array blocks. The partitioning can be specified along each of the array dimensions. DSK ensures that the array elements are divided as evenly as possible along the direction(s) of partitioning. In the following discussion, when there is no ambiguity, we use the term *block* to mean an array block. Partitioning an array associated with a single grid results in *adjacent* blocks. **Figure 9** shows a two-dimensional array partitioned into three blocks in each dimension. The blank boxes shown between pairs of adjacent large blocks and the arrows in that figure have the same meanings as in Figure 5. The pair (i, j) at the center of each array block is the coordinate of that block (assuming the lower left corner as the origin). The DSK package can handle higher-dimensional arrays and their partitioning along multiple dimensions.

The DSK package subroutine dskdef allows the user to define a distributed-array data structure by specifying its dimensions, partitioning parameters (e.g., number of partitions or size of each partition along each dimension), periodicity along any of the dimensions, and the width of the overlaps (extensions may be more than one cell wide) between adjacent array blocks. This package then partitions the array into a number of blocks that depends upon parameters such as the size of the array, the number of processors available, the desired granularity of computation, and the type of data dependencies among the array elements. Since there are many choices, and users typically want to experiment with various possibilities for performance, the choice for the number of blocks into which an array is to be divided is left up to the user. As noted earlier, load balancing among processors is easier when the number of blocks is larger than the number of processors; however, this may result in significantly higher communication and bookkeeping costs, as well as the costs of maintaining the overlaps associated with these blocks.

In many applications, including those with multiple, overlapping grids (see the example of the previous section), more than one array data structure may be involved. Moreover, the array dimensions may be considerably different. Each array may be divided into a different number of block arrays, and the block sizes may differ from array to array. Such heterogeneous, multiple, distributed-array partitionings are managed by the DSK package with minimal user involvement.

•• *Mapping of array blocks to processors*

DSK automatically distributes the blocks from all of the

arrays among the available processors so as to divide the total computational work as evenly as possible. For this, the total computational work associated with each block is first determined. With computational work considered as the weight, all blocks are sorted in descending order of weight and then mapped to processors according to a bin-packing algorithm. Entire blocks are assigned to processors; they are not further divided. Each processor may be assigned more than one block, and the blocks assigned to a processor may belong to one or more arrays. As in the case of array partitioning, the DSK package provides routines that facilitate mapping array blocks to processors.

- *Update and copy operations*

The DSK package provides facilities for "get" and "put" types of communication operations. For example, with a get type of operation, it is possible to update values in an array block with values from the adjacent array blocks; with a put type of operation, the values in the neighboring blocks can be updated with the values in the array block. Both of these operations are handled by making a call to dskdsw. Similarly, a call to dskdcb allows the user to specify a copy operation between a local array (e.g., a temporary work array) and an array managed by DSK.

For the model application described earlier, values may be needed from neighboring blocks prior to performing a computation on a block. In this case, a call is made to dskdsw, to update the boundary of the block, for locations where there is an overlap with the adjacent blocks. Using the internally stored tables describing the block partitioning, the DSK package performs the update operation at all the boundary elements of a block. Similarly, after an array block is updated at the end of a Runge–Kutta stage or at the end of a time step, dskdsw can be called to update the boundaries of the adjacent blocks. Note that a call to dskdsw is made both to update the boundaries of a block with the values from adjacent blocks and to update the boundaries of adjacent blocks with the values from a specified block. We refer to the former type of update as the *fan-in* boundary update and to the latter type as the *fan-out* boundary update. The direction of update is specified by flags passed to dskdsw. These flags can be used to stipulate a fan-in or a fan-out update, or even both types of updates. Furthermore, the updates on a block may be performed using values corresponding to the same block or from another block. We used a fan-out update in the model application.

Figure 9 shows an example in which values from blocks (1,2), (2,1), and (2,2) are needed for the computations to proceed on block (1,1). Similarly, values computed in block (1,1) are used to update the boundaries of blocks (2,1), (1,2), and (2,2). As mentioned earlier, the buffers shown between neighboring blocks in Figure 9 indicate extensions to the array blocks needed to hold values from the neighboring blocks, and the arrow tails indicate the array blocks from which the values are copied into the buffers. The DSK package maintains this information and performs the appropriate updates when dskdsw is invoked. This avoids explicit copying of data or buffer management by the user. Also, there is no need for explicit specification of which values are to be copied or communicated among processors.

During the interpolation phase (steps 1 and 6) of the model application, a copy operation involving two arrays is required. One of the arrays may be a local work array, and the other is an array managed by the DSK package. This operation is accomplished by calling dskdcb. For example, Figure 6 shows the steps involved in interpolating the boundary values of an array associated with grid 1 using the values from an array associated with grid 2. First the contributions from each block of the array of grid 2 are accumulated in local work arrays of the processors or assigned to *handling* the blocks. The contents of the local work arrays are then copied into the interpolation buffer arrays associated with the blocks of grid 1. This is accomplished by calling dskdcb. Note that the interpolation buffer itself may be partitioned into blocks, as shown in Figure 6; however, the user need not be concerned about the distributed nature of these arrays, since DSK handles the details of the copy operations transparently.

- *Interprocessor communication*

As described above, calls to dskdsw and dskdcb involve accessing one or more array blocks managed by the DSK package. If all of these array blocks are mapped onto the same processor that calls dskdsw or dskdcb, DSK satisfies these requests by performing local memory-to-memory copy operations; i.e., communication messages are eliminated when they are not necessary. If one or more blocks are stored on remote processors, interprocessor communication is required. The DSK handling of these communication steps is transparent to the user. In this case, a call to dskdsw or dskdcb returns control to the calling program only after the communication has been successfully completed.

The interprocessor communication is handled through use of the tables that specify the mapping of blocks to processors. These tables are created during the setup phase and are maintained by the DSK package for all blocks. From the definitions of the blocks, DSK can also determine the blocks from which the boundary values are required, for any given block. If a remote block is involved, a message is sent to the processor to which that block is assigned. Note that interprocessor communication may be involved in both the fan-in and fan-out updates; in both cases the communication is handled by DSK and is

transparent to the user program. In the case of a *fan-in* type of update involving a remote block, requests for information are sent to the appropriate processors, and the information is received in response to these explicit requests. In the case of a *fan-out* type of update, the information is sent out to the appropriate processors on the basis of *a priori* knowledge of the need at a remote block for the appropriate local information. With such a scheme, there is no need to issue explicit messages requesting specific information from other processors.

• *Communication optimization*

The above-described interprocessor communication in the calls to dskdsw and dskdcb may involve extra communication and synchronization overheads. For instance, when a call is made to dskdsw with a fan-in type of boundary update for a particular block and the update information is not available locally, messages requesting the necessary information are sent to the "owner" processors. We refer to such messages as *request-type messages*. Whenever a processor receives a request for data from another processor (by means of such a request-type message), it satisfies the request by sending back the requested data. This is carried out by the DSK system running on the owner processor, which performs this task when the user transfers control of execution to DSK via one of the calls to the DSK package. (In other words, the requests do not generate interrupts.) Also, since the storage holding the data generated in one iteration is reused in the next iteration, it is necessary to ensure that all request-type messages are satisfied before the computation on the next iteration begins. A global synchronization at the end of a time step ensures that all requests are satisfied, even if these requests arrive asynchronously. Although the above-described procedure ensures correct execution, there are two major types of overhead that affect performance: the request messages that must be issued for the needed data, and the global synchronization that must be performed whenever storage reuse results in loss of data that may be needed elsewhere. These costs considerably reduce the efficiency of parallel computation. In the worst case, they can render unscalable a perfectly scalable application.

We now describe certain optimization steps incorporated into the DSK package that minimize the effects of the above-mentioned overheads, without weakening any of the capabilities. These optimizations take advantage of the repetitive nature of the computations observed in the iterative solution of PDEs.

For the class of problems we are considering (steady-state solution to PDEs), the interblock dependencies do not change from time step to time step; therefore, one can reduce the overheads by collecting information on the data-request and data-delivery patterns among the processors by inspecting the execution of the first time step. This information can then be used in scheduling the messages. With such an arrangement, owner processors can send appropriate values to "consumer" processors in the most efficient manner, without being prompted for those values. Thus, in subsequent time steps, no global synchronization is needed.

Broadly speaking, there are two ways in which a program can be monitored for recording communication requirements. One way is to perform a preprocessing step prior to commencing the numerical computations. At the end of the preprocessing step, the monitoring or inspection step is complete, and all of the iterations in the computation can be executed using the schedules established in the preprocessing phase. This approach is used by the multiblock PARTI library [7]. One advantage of this approach is that the schedules can be used in all iterations. A disadvantage is that an explicit preprocessing step must be introduced, adding some computation and communication overhead.

The second approach is to gather the necessary information during the actual execution of the first time step and create a schedule of communication based on these observations. The DSK package uses this approach. An advantage of this approach is that there is no need for an explicit preprocessing step. A second advantage is that the schedules can be tuned by taking the system behavior into account during the actual numerical computations of an iteration. The only overhead associated with this approach is in keeping a record of the communication events in the first time step. As is seen in the following section, this overhead is small and is amortized over the rest of the time steps. The gains in each subsequent time step are substantial.

To realize these optimizations, a call to dsksch must be made at each schedule point in the user code. A schedule point is a state in the program at which all pending messages must be processed, in order for the program to proceed with correct execution of the code. Note that global synchronization can be used to accomplish this objective, by forcing all processors into this state at the same time. This is an expensive and often nonscalable means of achieving the result. For that reason, we use such a global synchronization only at the end of the first time step. During that time step, at the very beginning of stage 1, a schedule point is set by making a call to dsksch. This starts the process of recording the message traffic on each processor. Another schedule point is set at the beginning of each subsequent stage of the first time step. Finally, the end of the first time step is marked by the last schedule point. At this schedule point, all outstanding request-type messages are satisfied, the message recording is terminated, and an explicit global synchronization is performed to guarantee that all outstanding messages

are satisfied. The schedule sequence (consisting of five schedule points) is repeated in subsequent iterations without the global synchronization at the end of each iteration. At each schedule point, on a processor, the DSK system waits until all the messages posted at the previous schedule point are satisfied; new messages are then posted for the next phase of computation. Note that there is no need to issue request messages in the subsequent iterations, since the DSK package maintains a log, created at the first time step, of processors that need locally computed data. In other words, a tightly synchronous computation is transformed into loosely synchronous computation.

Performance results

In this section, we present performance results from two parallel systems at the IBM Thomas J. Watson Research Center: a 32-processor IBM POWER Visualization System™ (PVS) and an eight-processor experimental system called Power Parallel Prototype (PPP).

The PVS is a bus-based hierarchical-memory system. Each PVS processor is based on the i860™ microprocessor with 8KB data cache and 40-MHz clock speed. Each processor has 16 MB of local memory, of which about 13.5 MB is available to the user. The code, data, and stack for each processor are kept in its local memory. In addition, the processors are connected to 256 MB of global memory via a high-speed bus and communicate with one another using that shared memory.

The PPP is a distributed-memory system consisting of eight IBM RISC System/6000® (RS/6000) Model 550 processors, each with 64 KB of cache and 42.5-MHz clock speed. The processing element has 32 MB of physical memory. Each processor supports the IBM proprietary AIX® operating system and can function independently as a full-fledged workstation with virtual memory. The processors are connected with one another by a high-speed switch based on the same high-performance switch technology as that of the switch used in the IBM Scalable POWERparallel™ 1 (SP1) system [12].

Both systems we consider are of multiple-instruction–multiple-data (MIMD) type parallel architectures, and for both, a separate copy of the code is loaded into the local memory of every processor. On both systems, we used a message-passing paradigm for implementing the extended DSK package. At the lowest level of DSK package implementation on the PVS, we used the EUIm message-passing environment [13]. This environment emulates the IBM Extended User Interface (EUI) message-passing protocol [14], using the shared memory and semaphores provided by the PVS. On the PPP, we implemented the extended DSK package on top of the EUIH communication protocol, developed at the Watson Research Center.* Note that some of the low-level implementation details differ from system to system; however, the user program, such as our model application, remains the same.

We coded our model application, discussed earlier in the model application section, in FORTRAN 77. The overlapping grid used as an example in these experiments consists of two component grids, one with dimensions 144×144 (grid 1) and the other with dimensions 360×240 (grid 2). We partitioned the first grid into eight blocks, each with dimensions 36×72, by dividing the grid into four slices along one dimension and two slices along the other dimension. We partitioned grid 2 into 24 blocks by making six slices along the longer dimension and four along the shorter dimension, so that the resulting blocks had dimensions 60×60. While these partitioning parameters are somewhat arbitrary, they bring out the characteristics common to real-life applications in which the component grids have different sizes and the blocks, after partitioning, may not have the same amounts of computational work.

We performed the experiments in two modes: synchronous and asynchronous. Synchronous implies that no communication optimizations were performed; instead, a global synchronization was performed at the end of each time step. Asynchronous implies that during the first time step, a record was made of the communication pattern and of the computation sequence. As described above, a global synchronization was performed at the end of the first time step. In the subsequent time steps, information gathered from the first time step was used to schedule messages sent and received.

Table 1 shows the performance results obtained on the PVS. The average execution time, in seconds per time step, and the corresponding speedups in the synchronous mode are shown under the heading *Synchronous*. The execution times are averages over 50 time steps. We have shown results for 1, 2, 4, 8, 16, and 32 processors. Although not shown, other numbers of processors are possible. The performance of the asynchronous mode is shown in the remaining columns. The execution times for the first step and for subsequent time steps are shown separately. For the latter, we have taken the average over time steps two through fifty. Notice that the first time step under the asynchronous mode has an overhead of up to nine percent compared to an average time step in the synchronous mode; the overhead generally increases with the number of processors. However, the gains over the synchronous mode for the subsequent time steps in the asynchronous mode, which we define as (*Synchronous time per time step* − *Asynchronous time per time step*)/*Synchronous time per time step*, range from 26% to

*P. Hochschild, "EUIH: An Experimental EUI Implementation," IBM internal report, IBM Research Division, Yorktown Heights, NY, 1993.

Table 1 Performance of PVS on model problem.

Number of processors	*Synchronous*		*Asynchronous*				
	Average		*First step*		*Average*		
	Solution time per time step (s)	*Speedup*	*Solution time per time step* (s)	*Overhead* (%)	*Solution time per time step* (s)	*Gain* (%)	*Speedup*
1	3.55	1.00					
2	2.78	1.28	2.85	2	2.06	26	1.72
4	2.19	1.62	2.23	2	1.12	49	3.17
8	1.26	2.82	1.29	2	0.66	48	5.38
16	0.58	6.12	0.48	—	0.28	52	12.68
32	0.34	10.44	0.37	9	0.21	38	16.90

Table 2 Performance of PPP on model problem.

Number of processors	*Synchronous*		*Asynchronous*				
	Average		*First step*		*Average*		
	Solution time per time step (s)	*Speedup*	*Solution time per time step* (s)	*Overhead* (%)	*Solution time per time step* (s)	*Gain* (%)	*Speedup*
1	1.67	1.00					
2	1.30	1.28	1.34	3	0.91	30	1.82
4	0.88	1.90	0.89	1	0.53	40	3.13
8	0.48	3.48	0.50	4	0.33	31	5.08

52%. The gains of the asynchronous mode are relatively small with two processors; however, with more than two processors, the asynchronous mode results in substantial gains (about 50%), except with 32 processors, where the gains drop to 38%. At 32 processors, the computation per processor is relatively small, and other overheads, such as those due to load imbalance, bookkeeping, and boundary overlap manipulations, tend to dominate. Thus, with 32 processors, the effect of not having to synchronize and issue request messages does not reduce the total execution by the same factor as that observed with smaller numbers of processors. Finally, we compare the speedups in Table 1 for the synchronous and asynchronous modes. The speedup figure for p processors under the synchronous mode is the ratio of the execution time of an average time step on p processors to the execution time of an average time step on one processor. The same is computed under the asynchronous mode, with the execution time of an average time step not including the first time step. The rationale for this is that, in this type of computation, typically hundreds and even thousands of iterations or time steps are computed. As a result, the execution time averaged over all time steps is almost the same as the execution time averaged over all time steps except for the first. The improvements in performance are clear from the speedup metric.

Similar performance results for the PPP are shown in **Table 2**. The overhead in the first time step is 1% to 4%, while the gain in the subsequent time steps is in the range of 30% to 40%. As in PVS, with a larger number of processors, the gain in the total execution time due to asynchronous communication drops off. The RS/6000 processors are relatively more powerful than the i860 processors; as a result, overheads other than those in global synchronization start becoming dominant even at eight processors. Note that we solved the same problem on both systems, and the problem size we used was small, as is evident from the total execution time. The two speedup columns in Table 2 provide another measure for observing the advantages of the asynchronous mode and its effect as the number of processors is increased.

It is instructive to compare the performance of PVS and PPP systems in order to see which system benefits more by the asynchronous communication strategy, which avoids global synchronization. Such a comparison is shown in **Table 3**. Since PPP has only eight processors, we restrict our comparison to runs with eight or fewer processors.

Shown in the second column of Table 3 are ratios of the synchronous execution times on PVS to those on PPP with the same number of processors. The ratio for the single-processor case is an indicator of the relative speeds of the i860 and RS/6000 Model 550 processors. Note that as the number of processors is increased from two to eight, the ratio increases, indicating that the PVS becomes progressively slower. In other words, while the useful work per processor remains the same, the overhead of parallel implementation in the synchronous mode is higher on the PVS. The third column in Table 3 compares the performance of the first step of the asynchronous mode on the two systems. These ratios are similar to those in second column, indicating that the two systems behave relative to each other in a similar manner as in the synchronous mode. The last column, which gives the ratios of the execution times for time steps 2 and onward, shows a completely different trend. Note that these ratios are all approximately the same as the ratio for the one-processor case (approximately 2.1), indicating that under the asynchronous mode, the overheads of parallel implementation for both systems grow at a similar rate. This is evident when we compare the speedup columns under asynchronous mode in Tables 1 and 2.

Table 3 A comparison of PVS performance with PPP performance.

Number of processors	*Ratios of synchronous computation times* (t_{PVS}/t_{PPP})	*Ratios of asynchronous computation times* (t_{PVS}/t_{PPP})	
		First step	*Subsequent steps*
1	2.13	—	—
2	2.14	2.13	2.26
4	2.49	2.51	2.11
8	2.63	2.58	2.00

Conclusions

For the important class of irregularly coupled regular-mesh problems, the complex data structures and dependencies make the task of manual parallel implementation on scalable architectures very difficult. This is because the communication primitives available on most scalable parallel systems present a very low-level programming interface. Parallel implementation using these low-level primitives tends to be tedious and error-prone, even when the data dependencies are somewhat irregular. The state of compiler technology has not advanced sufficiently to handle this class of problem. The environment made available by DSK helps in overcoming some of these difficulties. DSK provides a portable parallel-programming environment, managing distributed data structures and dependencies. By hiding communication details from the user, the DSK environment allows the user to focus on the problem to be solved. Compared to a compiler, a library package such as DSK gives a higher level of abstraction and control to the user for performing optimizations. Together with its database features, this makes DSK a powerful tool for the class of applications considered here.

Acknowledgments

We thank Ronald Mraz for helping us with the EUIm message-passing environment on PVS. We also thank Rolf Hempel of GMD for his help in developing the array-block–processor mapping algorithm used in the DSK package.

References

1. G. Chesshire and W. D. Henshaw, "The DSK Package, a Data Structure for Efficient FORTRAN Array Storage" (reference guide for the DSK package), *Research Report RC-14353*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1988.
2. G. Chesshire and W. D. Henshaw, "Composite Overlapping Meshes for the Solution of Partial Differential Equations," *J. Comp. Phys.* **90**, 1–64 (1990).
3. A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and J. Saltz, "Software Support for Irregular and Loosely Synchronous Problems," *Computing Syst. in Eng.* **3**, 43–52 (1992).
4. M. Lemke and D. Quinlan, "P++, a C++ Virtual Shared Grid Based Programming Environment for Architecture-Independent Development of Structured Grid Applications," *Technical Report 611*, GMD, St. Augustin, Germany, 1992.
5. S. R. Kohn and S. B. Baden, "An Implementation of the LPAR Parallel Programming Model for Scientific Computations," *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, Society for Industrial and Applied Mathematics, Philadelphia, 1993, pp. 759–766.
6. H. Berryman, J. Saltz, and J. Scroggs, "Execution Time Support for Adaptive Scientific Algorithms on Distributed Memory Architectures," *Concurrency: Pract. & Exper.* **3**, 159–178 (1991).
7. A. Sussman, J. Saltz, R. Das, S. Gupta, D. Mavriplis, R. Ponnusamy, and K. Crowley, "PARTI Primitives for Unstructured and Block Structured Problems," *Computing Syst. in Eng.* **3**, 73–86 (1992).
8. A. Gagan, A. Sussman, and J. Saltz, "Compiler and Runtime Support for Structured and Block Structured Applications," *Technical Report CS-TR-3052*, Computer Science Department, University of Maryland, College Park, 1993.
9. G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, "FORTRAN D Language Specification," *Technical Report CRPC-TR90079*, Center for Research on Parallel Computation, Rice University, Houston, TX, 1990.

10. K. M. Chandy and I. Foster, "Deterministic Parallel FORTRAN," *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, Society for Industrial and Applied Mathematics, Philadelphia, 1993, pp. 798–805.
11. D. L. Brown, G. Chesshire, and W. D. Henshaw, "Getting Started with CMPGRD, Introductory User's Guide and Reference Manual," *Report LA-UR-89-1294*, Los Alamos National Laboratory, Los Alamos, NM, 1989.
12. C. Stunkel, D. Shea, D. Grice, P. Hochschild, and M. Tsao, "The SP1 High-Performance Switch," *Proceedings of the 1994 Scalable High Performance Computing Conference*, Institute of Electrical and Electronics Engineers, Knoxville, TN, 1994.
13. R. Mraz, "EUIm: A Message Passing Library for the IBM Power Visualization System," *Research Report RC-19125*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1993.
14. V. Bala, J. Bruck, R. Bryant, R. Cypher, P. deJong, P. Elustondo, D. Frye, A. Ho, C. Ho, G. Irwin, S. Kipnis, R. Lawrence, and M. Snir, "The IBM External User Interface for Scalable Parallel Systems," *Research Report RC-19048*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1993.

Received June 24, 1993; accepted for publication February 10, 1994

Geoff Chesshire *Los Alamos National Laboratory, MS B265, Los Alamos, New Mexico 87545 (geoff@c3serve.c3.lanl.gov).* Dr. Chesshire graduated in 1986 with a Ph.D. degree in applied mathematics from the California Institute of Technology, where he worked on parallel solution techniques for PDEs. He then joined the Mathematical Sciences Department at the IBM Thomas J. Watson Research Center, where he worked on grid generation, numerical solution of PDEs, and database tools for the support of parallel scientific computation. He now works in the Computer Research and Applications group at Los Alamos National Laboratory.

Vijay K. Naik *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (VKN at YKTVMH, vkn@watson.ibm.com).* Dr. Naik is a Research Staff Member in the Parallel Applications Methods and Analysis group at the IBM Thomas J. Watson Research Center. Prior to joining IBM, he was a staff scientist at ICASE, NASA Langley Research Center. He received the Ph.D. and A.M. degrees in computer science in 1988 and 1984, respectively, from Duke University. In 1982, he received the M.S. degree from the University of Miami, and he received the B.Tech. in 1980, from the Indian Institute of Technology, Madras, both in mechanical engineering. Dr. Naik's current research interests include characterization of scientific applications and algorithms for high-performance computing. In particular, he is interested in applications in the area of computational fluid dynamics and algorithms for sparse-matrix and unstructured-grid computations. He is also interested in the development of systems and architectures for scalable high-performance computing that are tuned to the characteristics of these applications and algorithms.