GRIDS—A Parallel Programming System for Grid-Based Algorithms

U. GEUDER, M. HÄRDTNER, A. REUTER, B. WÖRNER AND R. ZINK

Institute of Parallel and Distributed High Performance Systems, University of Stuttgart, Breitwiesenstraße 20-22, D-70565 Stuttgart, Germany

Received June 1993; revised September 1993

1. INTRODUCTION

There is general consensus that scientific progress is hampered by two problems: lack of funding money and shortage of compute power. The problem collections such as the 'grand challenges' (National Science Foundation, 1991) have been compiled in order to illustrate the pervasiveness of this phenomenon. There are many pressing problems in climatic research, high energy physics, aerodynamics, genetic engineering and drug design (Smith *et al.*, 1991; Simon, 1992) calling for much more compute power than current supercomputers can deliver. There is also a consensus—albeit on a smaller scale—that conventional scalar and vector processor architectures will never reach the levels of performance required, simply because of the speed of light. Therefore, the next quantum leap in computing performance can only be achieved by harnessing parallelism. However, parallel computing is still a 'niche' technology. Some special, isolated problems that have been implemented on a parallel machine (Ayakanat *et al.*, 1988; Lee *et al.*, 1988; Evans and Yousif, 1988)—usually with considerable effort—demonstrate both the feasibility and usefulness of the approach, but whenever somebody gets a significant program running on a parallel computer, this is still considered interesting news, which illustrates the point.

Why is this so? One reason is that parallel computers still are experimental vehicles rather than computational commodities. They are difficult to use, they have poor availability and, given the fact that they are not general purpose, they are pretty expensive. However, the real reason has to do with software. For all the large, interesting scientific problems, there exists a huge body of code, usually written in FORTRAN or some other classical, sequential programming language. In many cases, the performance-critical parts of this code have been carefully optimized for some specific machine architecture—for vector machines mostly. When this code is put on a parallel machine without further measures, it will use exactly one of its processors, resulting in terrible performance. So there are two options: (i) to parallelize the existing code and (ii) to completely rewrite the critical parts with a parallel machine as the target architecture.

This state of affairs has two major consequences. Programs designed to run on a parallel computer not only have to consider the algorithmic aspects, they also have to deal with all the operating system and communication issues implied by parallel execution. Since the functionality for controlling parallel execution is highly idiosyncratic for each type of parallel machine, such programs are strictly tied to the platform for which they have been developed.

Thus in order to make parallel programming more useful and more efficient, it will be necessary to consolidate some of the low level functionality. One such effort is the standardization of a message passing interface (MPIF, 1993). On such a platform, it will be easier to develop programming models that can be supported on a variety of platforms, rather than on just one type of machine.

The GRIDS system described in this paper is one such effort. Its goal is to offer the engineer or scientific user (as usual in scientific computing, no distinction is made between application programmer and user) access to efficient parallel computing without the need for explicit parallel programming. This is certainly less than the ultimate goal of fully automatic parallelization, but it is feasible with current technology, and it is very useful.

The paper is organized as follows. Section 2 presents an analysis of basic approaches to automatic parallelization. In Section 3, the GRIDS approach is introduced. Its programming model is described in detail in Section 4. In Section 5, we present a sample GRIDS program to give an idea of how the system is used. Section 6 discusses the key implementation aspects. Section 7 presents a summary and points to a number of extensions to the GRIDS system.

2. BASIC APPROACHES TO AUTOMATIC PARALLELIZATION

Parallel programming typically follows a 'holistic' approach. Algorithm design, data partitioning, message exchange and synchronization are considered explicitly and must be handled by the programmer. This is an unfortunate situation, because coding a, say, numerical algorithm is a different issue from controlling its parallel execution. Coping with parallelism is inherently complex and error-prone, and therefore should be done automatically whenever possible. The current status of parallel programming is very much like the early days of machine-language programming, where the programmer was responsible for everything, from the application down to the external storage devices.

A compiler capable of automatically parallelizing arbitrary sequential programs, written in conventional

programming languages, is infeasible for fundamental reasons, which we will not discuss here.

Given that the magic transformation of FORTRAN 'dusty decks' into efficient parallel programs is beyond reach, research into parallel programming focuses on two aspects:

- The first line of research assumes that the algorithms are coded in conventional sequential programming languages. One then attempts to augment these programs with additional information (expressed in a descriptive, non-procedural way) that assists a compiler in detecting independent portions in such programs so it can automatically generate the code for its parallel execution. This approach typically is called 'semi-automatic parallelization'. Its major advantage is that it is still as 'general purpose' as the programming language used; the price to be paid for this generality is a potentially low parallel efficiency of the resulting code.
- The second line of research does not strive for generality but for performance. It still uses conventional sequential programming languages, but the annotations are assumed to pertain to certain types of applications or algorithms. This restricts the number of parallel computations to be generated by a compiler to a well-known class and thus can be planned ahead. Usefulness and scope of this approach are largely determined by how general the application class is that the system is based on.

There is a third line of research aiming at fully automatic parallelization. In this case, one gives up the use of conventional programming languages and assumes other, non-procedural languages instead, for which a compilation into parallel units of execution is feasible. The most notable example of this category is SQL (Pirahesh *et al.*, 1990; DeWitt and Gray, 1992). However, since both the type of language and the application domain are radically different from what is typically found in scientific computing, we will ignore this approach for our purposes.

2.1. Semi-automatic parallelization

Semi-automatic parallelization based on hints from the programmer generally exploits the potential for data parallel execution of programs written in a conventional programming language. This language is moderately extended to allow for the control of parallelism, whenever it is required. The approach is designed to support medium to coarse-grained data parallelism. Typical units of parallel execution are loop bodies, where the set of values of the loop counter is distributed across the compute nodes.

The programmer has to understand data partitioning, and has to make sure code pieces work correctly and are instantiated at different processors. This requires a certain coding discipline, in particular restricted use of pointers.

High performance FORTRAN (HPFF, 1993) is an example of this approach. Data partitions are described by annotations to a conventional FORTRAN program. Further annotations can be used to assert the independence of statements with respect to control and data flow. All such annotations are made by the programmer and serve as hints to the compiler in order to generate code for parallel execution.

An increasing number of tools support the programmer during the annotation process by analyzing data dependencies, creating usage statistics of critical statements and program sections, and so forth (Zima *et al.*, 1988; Levesque, 1990; Kennedy *et al.*, 1991).

2.2. Restricted automatic parallelization

A different approach is pursued by what could be called 'restricted automatic parallelization'. Rather than just creating the code that controls the parallel execution, the declaration, allocation and maintenance of the data structures containing the independent data partitions is also done automatically. This is as close to automatic parallelization as one can get with conventional programming languages—at the price of restricting applicability to certain types of algorithmic structures. Examples of this approach are: Paragrid (Poggi *et al.*, 1990) and GRIDS (Geuder *et al.*, 1993).

The compiler incorporates knowledge about the inherent parallelism for that class and thus can generate efficient parallel code for any given instantiation. In the ideal case, the programmer does not have to deal with any aspect of parallel execution. All that is required is a proper use of the descriptive language constructs that describe the overall computational structure.

3. THE GRIDS APPROACH

The key observation motivating the GRIDS approach is the following. Users (application programmers) in the scientific domain typically have a very good understanding of the structure of the problem they want to solve. However, they do not want to be bothered with mapping the structure onto a parallel runtime environment—which in many cases is much more complicated than solving the problem itself. Users do also have a good idea of what domain decomposition means and how it applies to their problem. Yet they do not want to manage the domains and map them onto processes, threads, etc.

This is exactly the gap the GRIDS system tries to bridge. It asks the user to program its iterative solvers in a sequential programming language, in the same fashion this would be done on a conventional machine. It also provides the user with an interface that allows him to describe the problem structure in terms of domain decomposition, from which independent units of parallel execution can be derived. Once this is done, the individual domains together with the iteration procedures can automatically be mapped onto processors and processes.

Of course, neighbouring domains will share certain variables and there will also be the need for global variables, but all this can be expressed by the user in a descriptive rather than a procedural way, and the GRIDS compiler can automatically generate code for synchronizing access to such shared variables and minimize the delay incurred by synchronization. The runtime system provides the functionality to manage and execute the precompiled GRIDS program on a parallel platform.

The key components of the GRIDS approach are shown in Figure 1. They will be described in detail in the rest of the paper. All three components, the application class, the programming model and the runtime system, must be adapted to the underlying algorithmic 'metaphor', i.e. iterations on grids, to achieve high parallel efficiency.

Grid-based applications can be characterized as follows:

- The problem has an underlying topology defining the 'grid', i.e. a set of (heterogeneous) elements among which neighbourhood relationships are defined.
- Iterative calculations are carried out on the constituents of the grid until the solution converges, or the last time step has been reached, or until it is decided that the iteration does not converge.
- Values of one iteration step are computed from the local variables of an element and (some of) its neighbours in the previous step.

As will be demonstrated later, the notion of 'element' is very general and should not be confused with the same word in the context of 'finite element' modeling (see Section 4.1). In the typical case, an element is a point, an edge, a surface, a cube, etc., but it can be associated with other constituents of a problem space.

Grid-based algorithms can be found in computational fluid dynamics and structure mechanics, just to mention a few. The multi-grid algorithm for solving partial differential equations is a generalization of this idea.

The components of the GRIDS system reflect this style of grid-based computation. The notions of elements, associated variables, topology, neighbourhood, etc., are part of the GRIDS programming model. The user expresses the computations using these application level terms. The complexity of parallel programming is hidden underneath the (topological) domain decomposition. For example, there is no way the user can talk about messages—and no need to do so on his part.

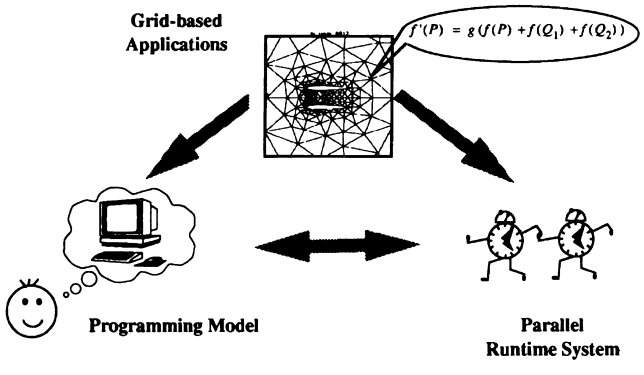

FIGURE 1. Key components of the GRIDS approach.

While the GRIDS programming model is geared towards general MIMD architectures, the runtime system must exploit the properties of the underlying hardware in order to achieve efficient resource utilization. It also has to map the units of parallel computations onto processors and processes, applying standard optimization techniques such as message bundling, send ahead, etc. In addition to that, the runtime system is responsible for load balancing. The GRIDS runtime system is presented in detail in Section 6.

4. THE GRIDS PROGRAMMING MODEL

This section describes the ideas behind the GRIDS programming model. For a more detailed explanation of the user interface, syntax, and the system structure see Geuder *et al.* (1993).

As was mentioned before, a GRIDS program consists of two parts, a description of the computation topology (the grid) and a description of the algorithm to be executed on the elements of the topology. Beside the advantages discussed below, the separation of topological and algorithmic aspects enables also the potential of code reuse, because the same algorithms can be used with different topologies at no extra cost on the programmer's part.

Since all topological aspects such as neighbourhood, variable sharing, synchronization, update propagation, etc., are mapped to simple language constructs that can be used in a conventional sequential manner. Separating the topological and the algorithmic part effectively means separating the handling of parallelism from the sequential algorithms. The topology can be managed by the GRIDS system instead of being coded into the application program, thus keeping programming free of irrelevant technical details.

Moreover, the topology declaration provides information about data dependencies that can be used by the precompiler and the runtime system to generate efficient parallel code and schedule it accordingly.

The algorithms to be executed for each topological element are typically nested iterative methods. These algorithms are represented by the problem-specific iteration procedures. Different procedures can be invoked as steps in different phases of the overall computation.

All this is specified in a descriptive manner when declaring the script. By script we denote a set of declarations saying which algorithms (step routines) are to be executed on which topological element under which circumstances. The roles of each component will become more obvious during the following discussions and examples.

4.1. Topology and variables

All computations in GRIDS are based on an underlying topological structure, the grid. A grid consists of grid elements of different types and of neighbourhood relations between the grid elements. GRIDS does not define any semantics, neither in the typing of grid elements nor in the typing of the neighbourhood relations. Figure 2 gives an example, using a simple geometry and its derived topology. Note that vertices and surfaces are just different types of topological elements in GRIDS, the interpretation of which is left to the application. Geometric information—if there is any—is not treated differently from any other values stored in the element variables. There are no restrictions imposed on the topology. The number of element types and types of neighbourhood relations is only limited by the machine resources.

4.1.1. Grid elements

A grid element, i.e. an element maintaining topological relations with other elements, has a user-specified set of local data. All variables in a GRIDS program are associated with a grid element and accessible only in the local context of this grid element. There is also a way of using global variables, but we will ignore this for the current.

An iteration algorithm typically refers to different types of grid elements. Of course, each type has different local variables, and different neighbourhood relations. Each variable is replicated for each grid element of the same type. All instances of the same variable are called a variable pool.

Each grid element may maintain different neighbourhood relationships to other grid elements of the same or any other type. In the simple three-dimensional example used in Figure 2 one can define the vertices and surfaces as different types of grid elements, containing different variables. One neighbourhood relationship connects vertices that have common edges, another one connects adjacent surfaces.

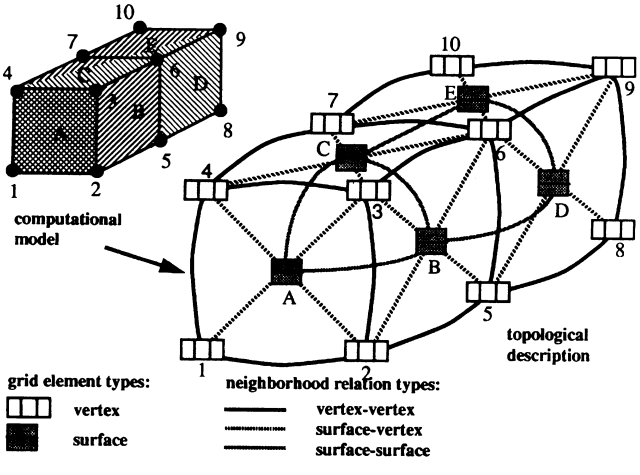

FIGURE 2. Topology example.

4.1.2. Neighbourhood relations

The neighbourhood relations specified in the topology describe the legal directions of data access 'out of' an element. Neighbourhood relations are typed in order to give additional information for selective use of data. According to their role as data access specifications, the relations are not necessarily symmetric nor transitive nor reflexive. In Figure 2 the neighbourhood relation *surface–vertex* connecting *surface* elements to *vertex* elements is unsymmetrical, since *vertex* elements cannot access variables of surfaces via this neighbourhood relation.

Each grid element may have several neighbours within one relation.

4.1.3. Sets of grid elements

Types of grid elements can be refined into sub-types, because often algorithms need to execute different operations on elements of the same type or the sequence of steps has to be changed. For example, one may have specified the type 'node', but a node is updated differently depending on whether it is a boundary node, or an internal node. Every 'node' has the same local variables, but the computations performed are different. Therefore the elements of a certain type can be partitioned into disjoint sets (sub-types). Algorithms (step routines) can be associated with either types or sub-types.

4.1.4. Global data

As mentioned before, all variables in GRIDS are associated with grid elements. In other words: no global variables accessible from everywhere exist. However, the effect of a global variable can be achieved by defining a special grid element that is connected to every other element. Since explicitly declaring such elements is rather tedious, GRIDS provides the type of a global element as part of the language. It also treats these elements as 'special citizens' by replicating their data across all nodes in order to speed up access to the global information.

Although global variables are not heavily used in grid-based computations, there is at least one common case where they are needed. Convergence of an iteration is a property of the entire computation, not of a single grid element. Therefore, the result of a convergence test has to be stored in a variable attached to a global element.

4.2. Algorithms associated with grid elements

The basic concept of the GRIDS system is a two-tier approach to coding the algorithms—which is why the topology description was introduced in the first place. At one level, the iterations are specified from the local view of a single grid element. At the next level, the iteration on the whole grid is described from a global perspective.

4.2.1. The local view: step routines and access functions

GRIDS executes steps in the context of one grid element. A step is a sequence of operations on the data of this element. The description of a step contains information about the variables used (and updated) by the step. It can use data values of neighbouring elements as input, but it will not refer to those variables again during its execution.

The program code to carry out the computation of a single step is contained in a step routine. Technically speaking, step routines are parameterless subroutines implemented in say FORTRAN77, or C, or whatever. They use the GRIDS constructs to access the element variables and to refer to the topology.

GRIDS automatically grants access to all variables needed. A step routine has access to variables belonging to either the current element or to its neighbours. The step routine can read and write all variables of the local element; in fact, the task of a step routine is to iterate the contents of some variables of the current element. Access to variables of neighbouring elements is restricted to read-only.

The variables of the current element can be accessed using the function *Own_Value* (variable). This expression can be used anywhere in the code. The read access to variables of other grid elements is achieved using the function *Neighbour_Value* (neighbour_relation, neighbour, variable). Both functions are generic; the type is determined by the GRIDS preprocessor. Topological information can be determined by ancillary functions, such as *get_neighbours* (neighbour-relation), which returns the number of neighbours in a neighbourhood relation.

4.2.2. The global view: script

The script is an implicit description of the data flow using GRIDS constructs and describes the structure of the iteration from a global perspective. It defines the order of computations to be carried out on a grid. From this global perspective, the computation is a sequence of step specifications together with rules that say when an iterated value of an element can be made accessible to its neighbours (*grid_update*). Each step specification names a step routine and a set of grid elements, on which the routine is to be executed.

Each step has its well-defined position within the sequence of grid updates. This allows the precompiler and the runtime system to determine the relative positions of all steps with respect to each other. On the other hand, the programmer knows precisely in which 'global' state a step is executed.

Iteration. Grid computations are iterative computations. Putting it simply, each iteration (used to denote one step of an iterative method) consists of two phases. First, there is the local computation of new values, followed by the propagation of these values to the neighbours. It is important to understand that variables of neighbour elements do not change their values as visual from the outside during the first phase. Changes become visible only when the next round of iteration starts. After phase two of each iteration, all variables are in a globally consistent state. Their values can now be used for convergence tests and output operations. It should be noted that the 'globally consistent state' does not imply the need for a global synchronization of the system.

In the general case, many algorithms iterate on different variables. These variables may depend on each other. In such cases, computations have to be carried out in a specified sequence. This requires new values to be propagated to neighbour elements within an iteration. This is supported by so-called grid updates, which allow for propagating of selected variables to neighbouring elements at arbitrary (user-defined) points in time.

Grid update. Within a single iteration, propagation of updated variables is explicitly controlled by grid updates. Step computations change the values of variables local to the current element. A grid update makes these changes visible to neighbouring elements. A grid update is executed in virtual synchrony for all variables in the same variable pool. Again, no actual global synchronization is performed by the system, but the user can write his code as though global synchronization would be enforced.

Each step routine on a grid element accesses all non-local variables (those belonging to neighbours) in the state after the last grid update for these variables. If a local variable of a grid element is modified by a step, the effect is local until the next grid update for this variable has occurred. The access to variables in neighbouring grid elements always returns the 'old' value, independent of the state of the calculation on other grid elements. This is essential for being able to freely schedule computations on different grid elements in parallel.

Convergence test. Determining convergence of the overall computation involves global information. In a parallel environment, it can be expensive to explicitly create a globally consistent state. In GRIDS, the transition to the next iteration is an occasion to gather information pertaining to such a state.

The convergence test is performed using a set of functions that implement global operations such as global sum or average for a variable pool. Because global operations are expensive, the system provides them in optimized form. There exists the possibility to skip the execution of the convergence test for some iterations.

Order of execution. The GRIDS programming model does not define a strict execution order for the step routines. The only ordering that is relevant for execution control results from the data dependencies incurred by interplay of data access and grid update among the neighbouring elements.

Therefore the runtime system will rearrange the execution of steps quite liberally to make sure the maximum

level of parallelism is achieved. This holds for data parallelism as well as for function parallelism. For example, steps not participating in a certain grid update may change order with this update.

5. AN EXTENDED EXAMPLE

This section demonstrates the use of all components of the GRIDS system. As a sample application we have chosen the solution of a two-dimensional steady flow problem using the Euler equations. A simple finite volume method with linear convection is implemented. For a complete syntax description see Geuder *et al.* (1993).

5.1. The declarations

In this example three different grid element types are defined; triangles, edges and nodes. Triangles are associated with all the conservation variables and, of course, represent the finite volumes. The flow variables are associated with the edges and nodes carry the geometric information (coordinates). A global element is used to execute the convergence test and to control the iterations.

There are several sub-types defined for the three grid element types. Edges are distinguished into the sub-types:

EdgEl	Internal edge with two adjacent triangles, default.
EInBnd	Boundary edge with one adjacent triangle, wall boundary condition.
EExBnd	Boundary edge with one adjacent triangle, free flow boundary condition.

For triangles there is only one type *TriEl*, because the flow is computed along the edges, and integration is done accessing the values associated with the edges.

The neighbourhood relations in this example are defined according to the access requirements from one grid element type to the variables of elements of the same or different element types. To calculate the triangle area, the triangles need to access the coordinates of their three bounding nodes. Therefore, a neighbourhood relation *TriNod* is defined, linking triangles to nodes. An *Edge* grid element accesses the coordinates of its end nodes via the neighbourhood relation *EdgNod* binding edges to nodes.

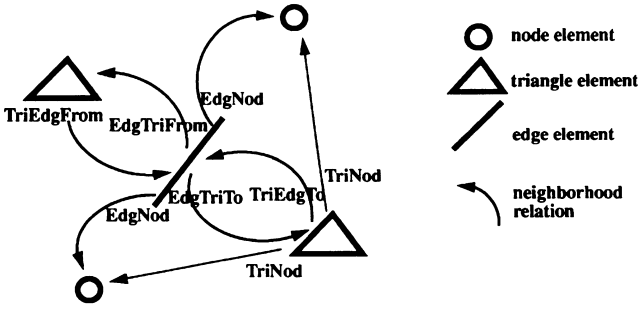

FIGURE 3. Grid element types and neighbour relations.

```
..
NodEl
Values X: double precision, /* X coordinate */
Y: double precision, /* Y coordinate */

EdgEl
  Values  FlRho: double precision,          /* Flow across the edge          */
          FlRhoU: double precision,         /* Flow across the edge          */
          ...
  Relations EdgNod with NodEl,              /* Access of coordinate values   */
            EdgTriFrom with TriaEl,         /* Flow directed "From -> To"    */
            EdgTriTo with TriaEl
  Tags EExBnd, EInBnd

TriaEl
  Values  Area: double precision,           /* Area of the triangle          */
          Rho: double precision,            /* Density                       */
          ...
          TDT: double precision,            /* difference to last iteration  */
          Bander: integer                   /* Used for local boundary info  */
  Relations TriNod with NodEl,              /* Access of coordinate values   */
            TriEdgFrom with EdgEl,          /* Flow is directed "From -> To" */
            TriEdgTo with EdgEl

Global CntrEl
  Values  Iter: integer,                    /* Number of iteration           */
          Res: double precision,            /* Residual                      */
...
```

FIGURE 4. Sample GRIDS program (declarations part).

Two neighbourhood relations are defined for the *Edge* type to associate edges with triangles. They are called *EdgTriFrom* and *EdgTriTo*, respectively, thus establishing a direction for the flow. The same is described from the triangles perspective by the relations *TriEdgFrom* and *TriEdgTo*, respectively. The topological elements for the two-dimensional flow problem are illustrated in Figure 3. To give an idea of the descriptive elements of the GRIDS language, an excerpt of the topology declaration is shown in Figure 4.

5.2. The script

This section illustrates the description of the iteration process from the global perspective. Since the computation is very simple, only one type of iteration without nesting is sufficient. The first phase of the iteration computes new approximation of the flow for all edges and propagates the new values to neighbours (Grid_Update). The second phase integrates the conservation variables for all triangles and makes them accessible to the neighbours. Finally, a test for convergence is done. This strategy is specified by code as illustrated in Figure 5.

```
...
BEGIN ITERATION
  FlowEdges(EdgEl,EExBnd,EInBnd)            /* calculate flow                */
  Grid_Update(EdgEl, FlRho, EdgEl, FlRhoU, ... ) /* make the new values visible to */
                                            /* neighbor elements             */
  Integrate(TriaEl)                         /* calculate new state           */
  Grid_Update(TriaEl, Rho, ... )            /* make the new values visible to */
                                            /* neighbor elements             */
END ITERATION
CONVERGENCE Ctest(CntrEl)                   /* gather global state           */
...
```

FIGURE 5. Sample GRIDS program (script part).

5.3. The step routine

This section shows the use of generic GRIDS functions accessing both local variables and data of neighbour elements. These functions allow the user to write sequential code that can automatically be executed in parallel.

In the sample script, there are two step routines and one convergence test. The iteration step routines FlowEdges() and Integrate() calculate the new values for the flow, conservation variables, respectively. In order to illustrate the concept, the code for the step routine Integrate() is shown in Figure 6 in some detail.

5.4. Generating a specific topology

The specifications made so far describe the type of the problem to be solved, they declare the type of topology to be used, etc. For solving any given problem, however, we must instantiate this type description by generating specific triangles, specific edges, and so on. In addition, some initial values of local variables must be set. There is a set of functions to instantiate a pre-declared topology. Of course the size of any realistic topology description is too large to be presented here completely. So we restrict ourselves in Figure 7 to show some sample calls that can cover the idea. Note that this sequence was generated by a simple filter program from the output of a public domain net generator for two-dimensional CFD problems (Bank and Chan, 1986).

```
C-----------------------------------------------------------
      (TriaEl, Area IN, Rho INOUT, ... )
      2 step_routine Integrate (
      3 TriEdgFrom, FlRho, TriEdgTo, FlRho,
      ... ,
      7 TriNod, X, TriNod, Y
      8 CntEl, DT)
C
      integer n, i
      double precision OldRho, DTS
...
C the integration is controlled by a variable at the global element
C so first we get the actual time step size from global element
C and compute the local integration factor by dividing through the triangle area
      DTS=neighbor_value(CntrEl,1,DTS) / own_value(Area)
C now we do the actual integration in two loops
C first we loop over all edges with flow to the current triangle
C get the number of neighbor edges with flow into the triangle
      n=get_neighbors(TriEdgFrom)
      do 10 i=1,n
        own_value(Rho) = own_value(Rho) +
     1          DTS * neighbor_value(TriEdgFrom,i,FlRho)
        ...
 10   continue
C get the number of neighbor edges with flow pointing out of the triangle
      n=get_neighbors(TriEdgTo)
      do 20 i=1,n
        own_value(Rho) = own_value(Rho) -
     1          DTS * neighbor_value(TriEdgTo,i,FlRho)
        ...
 20   continue
...
 END
C-----------------------------------------------------------
```

FIGURE 6. Sample GRIDS program (excerpt from step routines part).

```
...
Insert_Element(NodEl, 1, X, -5.000000e-01, Y, 0.000000e+00)
Insert_Element(NodEl, 2, X, 5.000000e-01, Y, 0.000000e+00)
...
Insert_Element(TriaEl, 1)
...
Insert_Neighbor(TriaEl, 1, TriNod, 7)
Insert_Neighbor(TriaEl, 1, TriNod, 43)
...
Insert_Element(EdgEl, 1)
Insert_Neighbor(EdgEl, 1, EdgNod, 1)
Insert_Neighbor(EdgEl, 1, EdgNod, 52)
...
Set_Tag(NodEl, 3, NExBnd)
Set_Tag(NodEl, 1, NInBnd)
...
```

FIGURE 7. Sample GRIDS program (topology instantiation).

6. IMPLEMENTATION OF THE GRIDS SYSTEM

In this section we present some basic issues of the GRIDS prototype implementation. One version of the prototype runs on a network of RS/6000 workstations using TCP/IP. Another version has been implemented on the Intel Paragon. There are only few system dependent components other than the communication subsystem. Communication is implemented in a package called SPPL (Zink, 1993), so porting effort is low and concentrated within SPPL. User programs are independent of the target system, and only need to be recompiled when moving from one platform to the next.

The approach of how to parallelize grid-based computations in GRIDS is simple. The grid is partitioned into disjoint sets of domains and each processor works on one of the resulting partitions.

Preparing a GRIDS program for execution consists of three phases. First the preprocessor processes the GRIDS source program. The declaration file and the script are converted into an internal representation. The script is translated into a data dependency graph, as explained below. The step routines are processed by both a GRIDS-specific precompiler and a standard FORTRAN compiler. The resulting FORTRAN routines are compiled and linked to the GRIDS runtime kernel using standard system utilities. The steps mentioned so far are denoted as GRIDS preprocessing. See Figure 8.

The remaining two phases are handled by the runtime system. Execution starts with a setup phase. First, the topology description is read and the initial partitioning of grid topological elements across processors is determined. Each physical processor is assigned a single partition of the grid. Each processor runs an operating system process called 'worker' that executes the step routines for elements in its partition. After receiving its partition, each worker adapts the data dependency graph produced by the script preprocessor to its local partition and to the partitions its elements are neighbours of.

The final phase is the execution of the iterative step

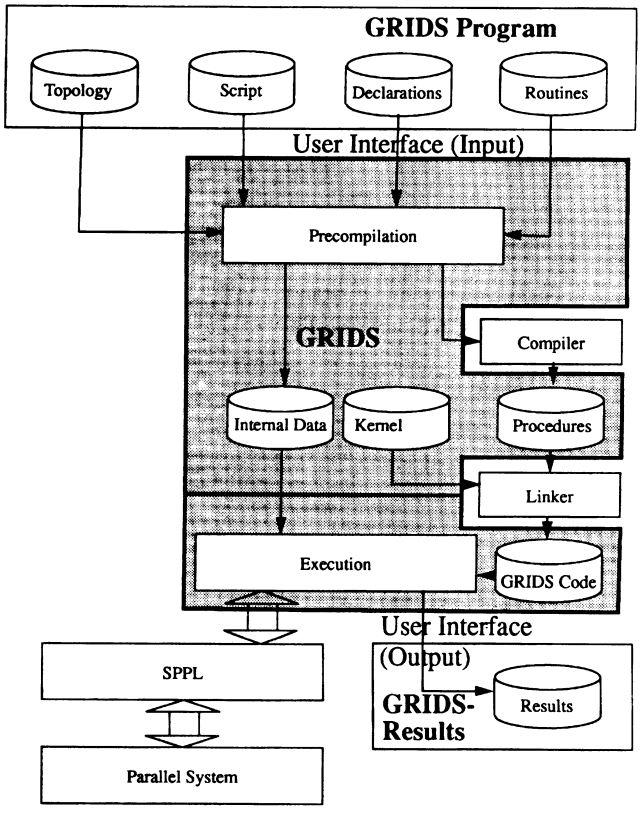

FIGURE 8. Implementation overview.

routines. Within the routines each worker accesses local data only. Data of neighbour elements assigned to another partition are transparently cached in so-called wrap guard elements.

6.1. Partitioning the grid across processors

The basic prerequisite for parallel computing is the distribution of independent units of work among the available processors. Considering the computational grid as a graph, distributing the work is equivalent to graph partitioning, an NP-complete problem. Finding an optimal partitioning depends on the cost measure. However, costs of the computation in a grid element are usually not known in advance. Moreover, they may vary over time. As a consequence, a distribution of work that is (near) optimal in the beginning might not be adequate for the entire calculation.

To support efficient heuristic work load distribution, we need approximate solutions for the following partial problems:

- What is the average work load per grid element?
- What is the trade-off between an unbalanced work load distribution and the overhead for finding a better one and actually redistributing the work?
- What is an efficient heuristic method to solve the graph partitioning problem?

6.1.1. The cost function for the step routines

GRIDS uses the following cost model to estimate the work load on a single partition; it includes both CPU cost and communication overhead.

$$T(P_i) = T_{\text{startup}} N_{P_i} + \sum_{e \in S(P_i)} T_{\text{send}}(e) + \sum_{e \in R(P_i)} T_{\text{rec}}(e) + \sum_{e \in (P_i)} T_{\text{cal}}(e)$$

P_i denotes the set of grid elements of partition i. $S(P_i)$ and $R(P_i)$ denote the set of grid elements that have neighbourhood relations to other partitions, to which they send data or from which they access data. $T_{\text{cal}}(e)$ is the CPU load of the grid element e in time units. $T_{\text{send}}(e)$ and $T_{\text{rec}}(e)$ are the accumulated costs of send and receive operations to and from neighbouring elements, again in units of time. These costs can be estimated based on the amount of data to be transferred and the performance of the communication system. N_{P_i} is the number of neighbouring partitions to P_i, i.e. partitions containing neighbouring elements. T_{startup} is the communication system's start-up cost for a single message.

The optimization goal is to find a partitioning with the lowest $max(T(P_i))$. Other criteria, such as the lowest variance are conceivable, but our choice has proven to be efficient and easy to approximate.

6.1.2. The partitioning algorithm

The partitioning algorithm has few dependencies on the other components of the runtime system. So the design decision for the first prototype was to implement a rather simple solution. This version, in addition to using simple heuristics throughout, is based on a number of simplifying assumptions, which are briefly listed in the following:

- T_{cal}, T_{send} and T_{rec} are based on user estimates for each element type. Constant values are provided as default.
- There is no repartitioning. (This is mostly a shortcut in order to have a running prototype earlier rather than a design decision. The next release will be able to do repartitioning. Other runtime system components can cope with repartitioning. Cost data for non-initial partitions will be estimated from figures measured in the previous interval.)
- The partitioning algorithm is executed sequentially. It consists of three steps: Greedy partitioning, communication reduction and load balance correction.

The initial partitioning is determined by a greedy algorithm (Farhat and Lesoinne, 1993). It generates partitions with roughly the same $T(P_i)$. This is done by adding neighbours to a partition until $1/p$ (where p is the number of processors) of the total cost is reached. This algorithm has linear complexity. It usually yields $p-1$ good partitions with respect to communication cost. However, the remaining partition is typically scat-

tered across the grid and this causes high communication cost.

Communication reduction is done next. In order to reduce the communication load, the algorithm examines whether it is advantageous to exchange grid elements between adjacent partitions. It is linear in the number of neighbourhood relationships crossing partitioning boundaries. Finally load imbalances introduced by the communication reduction can be compensated by moving single elements from overloaded partitions to a neighbouring partition.

6.2. Execution control

Another important component of the GRIDS runtime system is execution control. Computations on neighbouring partitions must be synchronized periodically for data to be transmitted among adjacent elements.

The execution control component exploits three types of information:

- Specification of the global computation structures as provided by the script.
- The actual grid topology.
- The current partitioning and mapping of partitions to processors.

The script contains all computation steps to be executed and defines a partial order on them. The partial order is induced by the data dependencies specified in the procedure headers. The data dependencies must be 'interpreted' from the perspective of the actual topology to decide which steps can be executed in parallel.

6.2.1. Concepts and strategies

In order to reach efficient execution in a distributed environment it is important to minimize delays due to synchronization and communication. Such delays can arise if data needed by a step routine are not computed in time by a neighbour, if a send operation gets deferred, or if there is a message delay in the communication system.

Besides trying to minimize the overall number of messages sent, the GRIDS system applies the following tricks inside each partition in order to avoid delays whenever possible:

- Use of wrap guards.
- Data exchange on availability (in contrast to exchange on request).
- Data-send-ahead.
- Use of asynchronous communication.

Storing values owned by a neighbouring partition in local wrap guards avoids multiple transfers of identical data. The other methods try to move ready computed data to the process where they are needed as soon as possible. Remember that the topology description says precisely where they will be needed. Data of grid elements at a partition boundary are computed before internal elements whenever possible. Such data are sent to the receiving partitions after completion. On the other hand, as long as no boundary element can be computed because of outstanding messages, computation can continue on local grid elements that do not depend on neighbouring partitions. With compact partitioning, there are many such internal elements. This strategy allows one to overcome temporary load imbalances and accidental message delays.

6.2.2. Task-internal load balancing

To support these strategies, the GRIDS execution control component uses local refinement on each partition. The key idea is to use tasks with priorities. Each grid partition is locally divided into smaller pieces, called GRIDS tasks. A task in GRIDS is a subset of grid elements of same type inside a single partition, together with the operations to be executed on these elements. (The term 'task' should not be confused with the operating system task. However, a GRIDS task could be mapped to an operating system task, given an operating system that supports those lightweight processes.) The code comprises the step routines, a convergence test or the grid update operations on the tasks elements. Tasks are the basic units of control. From a logical point of view, execution of a piece of code on all elements of a task appears to be atomic, i.e. any communication occurs either as the first (receive) or as the last (send) activity of task execution. The precedence relations among tasks are derived from the partial order defined for their member grid elements.

6.2.3. Data structures for execution control

The key data structure of the GRIDS execution control is a data dependency graph (DDG), as shown in Figure 9. The nodes of the GRIDS-DDG contain all information needed to perform a specified task. It includes:

- The action to be executed (step routine, convergence test, grid_update).
- Priority and current state of the task.
- Description of the data accessed.
- Grid elements belonging to the task.

The edges of the GRIDS-DDG describe the partial order among the tasks imposed by data dependencies. Loops in the GRIDS-DDG reflect the fact that overall computations are repeated iteratively.

The DDG implicitly describes all valid execution orders for the specified computations. Additionally the tasks are assigned priorities according to their possible effect on the parallel execution. Based on the information in the DDG and the priorities the execution order is determined dynamically at each node. By executing the ready task with the highest priority next the most efficient execution order can be easily detected at each moment. There is no need to evaluate all possible execution orders explicitly at the beginning of the run.

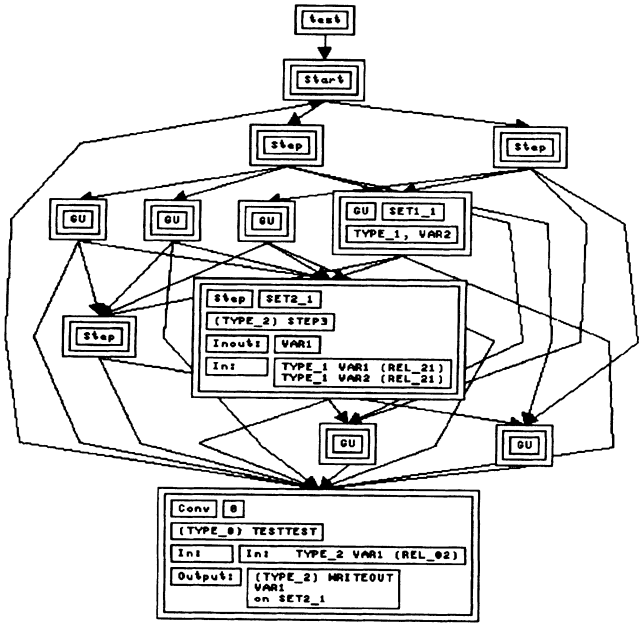

FIGURE 9. Data dependency graph for GRIDS execution control.

7. SUMMARY AND OUTLOOK

In this paper we have presented the GRIDS programming system for parallel computations. It has been argued that defining problem-oriented programming models allows us to handle all aspects of parallelization automatically. Grid-based methods are quite important in the field of scientific computing. This class has been selected for GRIDS. In contrast to many so-called general purpose approaches to parallel programming, it supports unstructured grids. The main part of the GRIDS programming interface offers a sequential language slightly enhanced by grid constructs. Most of the problem-specific parallelism is present in the programming model. Detailed information on specific dependencies are captured by the grid specific constructs at the programming interface. Keeping the overall parallel structure of the algorithm by means of a well-suited programming model makes automatic parallelization feasible. The concepts integrated in the programming interface to the GRIDS system have been discussed in detail and an example of a GRIDS program has been given.

Predictions of performance-related figures such as the optimal number of processors for a given problem and a given set of performance parameters of the parallel system need further work. A taxonomy of algorithms with respect to achievable parallel efficiency is a related field of research.

Future extensions of the GRIDS system are needed in two areas. The first group of extensions contains features that can be included without changing the programming model. The most attractive enhancements are: dynamic load balancing, intelligent checkpointing and other fault-tolerance measures supporting for long-running computations, and parallel execution of the partitioning. The second group of enhancements require extensions to the programming model and, consequently, to the user interface. Such extensions are: support of multi-grid methods, modification of the grid at runtime to support adaptive solvers and general modification of the topology at runtime as a result of the computations.

It might be interesting to see how far beyond the scope of scientific computing the idea of topology-based programming can be made to carry.

REFERENCES

Ayakanat, C., Özgüner, F., Ercal, F. and Sadayappan, P. (1988) Iterative algorithms for solution of sparse systems of linear equations on hypercubes. *IEEE Trans. Comp.*, **37**, 1554–1568.

Bank, R. E. and Chan, T. F. (1986) PLTMGC: a multi grid continuation program for parametrized nonlinear elliptic systems. *SIAM J. Sci. Stat. Comput.*, **7**, 540–559.

DeWitt, D. and Gray, J. (1992) Parallel database systems: the future of high performance database systems. *Commun. ACM*, **35**, 85–98.

Evans, D. J. and Yousif, W. S. (1990) The implementation of the explicit block iterative methods on the Balance 8000 parallel computer. *Parallel Computing*, **16**, 81–97.

Farhat, C. and Lesoinne, M. (1993) Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *Int. J. Numerical Methods Eng.*, **36**, 745–764.

Geuder, U., Härdtner, M., Wörner, B. and Zink, R. (1993) *GRIDS User's Guide.* Computer Science Report 4/1993, Department of Computer Science, University of Stuttgart.

National Science Foundation (1991) *Grand Challenges: High Performance Computing and Communication.* Committee on Physical, Mathematical, and Engineering Sciences, National Science Foundation, Washington DC.

High Performance Fortran Forum (1993) *High Performance Fortran Language Specification.* Rice University, Houston, TX.

Levesque, J. M. (1990) *FORGE90: A Parallel Programming Environment.* IEEE Parallel Programming, pp. 291–294.

Lee, J. Shragowitz, E. and Sahni, S. (1988) A hypercube algorithm for the 0/1 knapsack problem. *J. Parallel Distributed Comp.*, **5**, 438–456.

Kennedy, K., McKinley, K. S. and Tseng, C.-W. (1991) Interactive parallel programming using the parascope editor. *IEEE Trans. Parallel Distributed Syst.*

Message Passing Interface Forum (1993) *DRAFT Document for a Standard Message-Passing Interface.* MPIF.

Poggi, F., Dellagiacoma, F., Paoletti, S. and Vitaletti, M. (1990) *Paragrid: A Parallel Multi-block Environment for Distributed Memory Systems—Application Programmer's Guide.* IBM ECSEC Technical Report, Rome.

Pirahesh, H., Mohan, C., Cheng, J., Liu, T. S. and Selinger, P. (1990) Parallelism in relational data base systems: architectural issues and design approaches. In *Proc. DPDS*, IEEE Computer Society Press, Los Alomitos, pp. 4–29.

Simon, H. D., Van Dalsem, W. R. and Dagum, L. (eds) (1992) Parallel CFD: current status and future requirements. In *Parallel Computational Fluid Dynamics: Implementation and Results.* MIT Press, Cambridge, MA.

Smith, R. D., Dukowicz, J. K. and Malone, R. C. (1991) *Massively Parallel Global Ocean Modelling.* Technical Report LA-UR-91-2583, Los Alamos National Laboratory, Los Alamos, NM.

Zima, H. P., Bast, H.-J. and Gerndt, M. (1989) SUPERB: a tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, **6**, 1–18.

Zink, R. (1993) The Stuttgart parallel processing library SPPL and the X windows parallel debugger XPDB. In *Proc. Parallel Systems Fair, IPPS '93*, IPPS '93, Newport Beach, pp. 50–55.