AIM

Association for Information and Image Management

1100 Wayne Avenue, Suite 1100
Silver Spring, Maryland 20910

301/587-8202

Centimeter

Inches

MANUFACTURED TO AIM STANDARDS
BY APPLIED IMAGE, INC.

1 of 1

Experience with Automatic, Dynamic Load Balancing and Adaptive Finite Element Computation

Stephen R. Wheat¹
MP Computing Research Laboratory
Department 1424
Sandia National Laboratories
Albuquerque, NM 87185-5800
srwheat@cs.sandia.gov

Karen D. Devine^{1,2}
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180
kddevin@cs.sandia.gov

Arthur B. Maccabe^{1,3}
Department of Computer Science
The University of New Mexico
Albuquerque, NM 87131
maccabe@cs.unm.edu

Abstract

Distributed memory, Massively Parallel (MP), MIMD technology has enabled the development of applications requiring computational resources previously unobtainable. Structural mechanics and fluid dynamics applications, for example, are often solved by finite element methods (FEMs) requiring millions of degrees of freedom to accurately simulate physical phenomenon. Adaptive methods, which automatically refine or coarsen meshes and vary the order of accuracy of the numerical solution, offer greater robustness and computational efficiency than traditional FEMs by reducing the amount of computation required away from physical structures such as shock waves and boundary layers.

On MP computers, FEMs frequently result in distributed processor load imbalances. To overcome load imbalance, many MP FEMs use static load balancing as a preprocessor to the finite element calculation. Adaptive methods complicate the load imbalance problem since the work per element is not uniform across the solution domain and changes as the computation proceeds. Therefore, dynamic load balancing is required to maintain global load balance.

We describe a dynamic, fine-grained, element-based data migration system that maintains global load balance and is effective in the presence of changing work loads. Global load balance is achieved by overlapping neighborhoods of processors, where each neighborhood performs local load balancing. The method utilizes an automatic element management system library to which a programmer integrates the application's computational description. The library's flexibility supports a large class of finite element and finite difference based applications.

We test the system's effectiveness with an adaptive order (p -) refinement Discontinuous Galerkin FEM for the solution of hyperbolic conservation laws. We demonstrate the effectiveness of the data migration system on a 1024-processor nCUBE2 by comparing total execution times for the adaptive FEM with and without the migration system enabled. The results show the significant impact synergistically obtained by combining the adaptive FEM with the automatic data migration system.

I. Introduction

We have developed a fine-grained, data migration technique for intra-task, dynamic application load balancing [23]. Our approach targets Massively Parallel (MP) *Multiple-Instruction, Multiple-Data (MIMD)* computers -- machines with hundreds or thousands of processors. Our emphasis has been on MIMD machines because the latest introduction of hardware from various vendors has favored this approach, as is evidenced by the recent introduction of Intel's Paragon [9], Thinking Machines Corporation's CM-5 [2], and nCUBE's nCUBE 3 [14]. We present the results of our dynamic load balancing approach on *distributed memory* MIMD machines. Although our approach could

¹ This work was performed at Sandia National Laboratories, operated for the U.S. Department of Energy under contract #DE-AC04-76DP00789.

² This work was partially supported by Sandia National Laboratories under Research Agreement #67-8709.

³ On Faculty Sabbatical to Sandia National Laboratories.

be applied to *shared memory* MIMD machines, they are not considered in this work since current shared memory technology, unlike distributed memory technology, does not scale to hundreds or thousands of processors.

The primary goal of our work is to dynamically load balance applications to achieve high parallel efficiency. For each processor allocated to the execution of a program, the ideal improvement in execution time is obtained when the execution time satisfies

$$\text{time}(n) = \text{time}(1)/n$$

where $\text{time}(n)$ is the time it takes to execute the application on n processors. Applications whose performance behavior is close to this ideal have high parallel efficiency.

Achieving high parallel efficiency is a difficult task; additional processors require additional partitioning of the program and an associated increase in communication. In fact, application performance is bounded such that, at some point, the assignment of more processors yields no more increase in performance [1][10][15]. This effect has been attributed to the fact that communication overhead and necessarily serial computation eventually dominate parallel computation time.

While it may not be possible to maintain parallel efficiency when processors are added and the problem remains fixed, one may increase the size of a problem to better utilize additional processor resources, introducing the concept of *constant time applications* [8]. As MP machines become more powerful, the applications grow to consume the newly available resources. Each execution of the application uses approximately the same amount of time as before, but solves larger and/or more detailed problems.

By using application load balancing, one attempts to apply the maximum processing power of each processor to a given problem. Our approach to application load balancing differs from those that use tasks as the unit of migration to achieve load balancing [12][16][18][19]. For example, Lin and Keller developed a task oriented load balancing algorithm that utilizes a balancing scheme similar to our work [13]; however, they migrate whole tasks. Our work uses a much finer grained, data-element based technique for intra-application load balancing.

A secondary goal is to help application programmers achieve expected speed-ups while retaining an acceptable level of program abstraction. Our load balancing system enables the programmer to integrate the application into it by specifying the data communication patterns, per-element processing, and boundary element processing of the application. The run-time system automatically performs data migration to achieve global load balance.

We have developed a fine-grained data driven load balancing technique for finite-element and finite-difference based applications, a specific, yet very large, class of applications. The processor load characteristics of these applications are determined by the per-element processing of the finite-element/difference data. In general,

the more elements that are assigned to a given processor, the more load that processor will have. However, per-element processing requirements are not always constant and can vary greatly, making a static balance difficult to determine. Our research is based on the concept of *tiling*, which provides an automatic, element-based balancing facility for applications.

We test the tiling system with an *adaptive* finite element method. In the adaptive method, the work per element can differ from element to element, and can change from time step to time step. The non-uniform and changing work loads present a significant challenge to a dynamic load balancing system; however, the tiling system successfully recovers much of the parallel performance lost to the adaptive scheme.

II. A tiling approach to load balancing

Tiling integrates dynamic load balancing with a mechanism to describe the communication requirements of a given application code segment. Many applications use data exchanges at natural synchronization points. For example, an iterative solver might have each processor perform work on its data, exchange inter-processor boundary data with neighboring processors, perform boundary processing, and then exchange data to determine if another iteration is needed. The pattern is so common that many programmers simply borrow data exchange routines from other applications or utilize library codes. Intermixing these common routines with load balancing becomes quite complicated. In most cases, the programmer must abandon the common routines in order to integrate the data exchange with the load balancing code. The programming sophistication required for dynamic load balancing is so great that either the programmer does not include it in the application or codes it uniquely for each application.

With tiling, the programmer only needs to communicate the data exchange pattern, the per-element processing, and boundary processing to the run-time environment. Dynamic migration of elements from processor to processor is automatically performed by the run-time system.

Figure 1 illustrates the dynamic balancing provided by tiling. Without *a priori* knowledge, the data set is divided evenly among 16 tiles. After some period of processing where processors (0,1) and (3,2) were more utilized than their neighbors, processor (0,0) receives some of the data originally allocated to processor (0,1), and processor (3,2) gives processor (3,3) some of its data, as shown in Step 1. Processors (0,0) and (0,1) are now equally balanced yet out of balance with other processors. Thus, in Step 2, some data is migrated from processor (0,1) to processor (1,1). The ripple effect continues to move through processors (2,1) and (3,1) during subsequent balancing steps.

In considering this example, note that the migration causes tiles to take on irregular shapes -- leading to additional communication. With per-processor memory sizes increasing, tile boundaries can be allowed to become quite complex since communication costs become less of an issue. Thus, as a programmer increases the data set size, more processing is required when compared to the additional communication requirements. Therefore, there is some

Figure 1. Migration Example

room to allow for more complicated communication patterns, especially if the resulting reduction in application execution time outweighs the additional communication costs. While these arguments illustrate that communication costs should not be a primary concern in load balancing, our algorithms attempt to maintain reasonable tile interfaces in an effort to control communication costs.

Tiling is a modified version of the global load balancing technique developed by Leiss and Reddy[12][16]. The Leiss/Reddy load balancing technique uses local balancing performed within overlapping processor neighborhoods to achieve global load balance. A neighborhood is defined as a processor at the center of a circle of some predefined radius and all other processors within the circle. Each processor can be a neighborhood center. An example, showing 25 processors in 10 neighborhoods, is shown in Figure 2. Processors within a given neighborhood are balanced with respect to each other using local (as opposed to global) performance measurements. Individual processors may belong to several neighborhoods. In the Leiss/Reddy technique, work can be migrated from a processor to any other processor within the same neighborhood. Our technique restricts work migration so that elements are migrated only to neighboring processors following the element inter-connections.

Figure 2. Neighborhood Example - 25 processors in 10 neighborhoods

In tiling, a processor's load depends on the number of elements that are assigned to its local mesh and the per-element processing cost. When each processor has the same number of elements and all element costs are equal, global load balance results. When one processor is assigned elements having higher per-element processing costs than the elements assigned to another processor, as in the adaptive finite element method, processing imbalance is created.

III. Definitions

We refer to *elements* as those data structures that form the basis of finite-difference or finite-element numerical techniques. Elements are separated into two classes, *local* and *ghost*. Local elements are further divided into two sub-classes, *internal* and *boundary*.

Internal local element -- an element that is assigned to a processor and has all its element neighbors local to the same processor.

Boundary local element -- an element that is assigned to a processor and has at least one element neighbor assigned to another processor.

Ghost element -- an element that represents the element data values and element interconnection for an element that is assigned to another processor.

Figure 3 shows the basic element classes for a given processor.

Figure 3. Element Classes Example

A *processor neighborhood* is defined as a processor (the neighborhood center) and all other processors for which the center has ghost cells. Therefore, every processor is the center of one neighborhood, and may belong to many neighborhoods. Note that neighborhoods are not related to the hardware interconnection of the processors as they were in the work by Leiss and Reddy. A neighborhood may consist of many processors, all of which may not be nearest neighbors with regard to the processor's hardware interconnections.

For the purposes of element migration, there are three classes of processors: *exporters*, *importers*, and *concerned partners*.

Exporter Class -- Consists of all processors that will send elements to other processors for the purpose of developing a better distribution of work between the processors.

Importer Class -- Consists of all processors that will receive elements from other processors for the purpose of increasing their work loads to better balance the neighborhood's work load.

Concerned Partners Class -- Consists of all processors owning an element adjacent to an element that is being migrated from a processor in the exporter class to a processor in the importer class.

During a migration phase, a processor may belong to zero, one, two, or all three of the processor classes.

IV. Algorithm overview

The set of applications addressed by our approach are those that *are two-dimensional, regularly gridded, element based, with or without periodic boundary conditions, and having potentially varying per-element processing time requirements*. To be integrated into the tiling environment, application programs are partitioned into two phases: a *computation phase* and a *balancing phase*. The computation phase corresponds to the application's implementation without load balancing. Each processor operates on its local data, exchanges inter-processor boundary data via ghost cells, and then processes the inter-processor boundary data. These actions are repeated until the application meets its termination condition.

The tiling algorithm introduces dynamic load balancing in the inter-processor boundary data exchange operation. Some number of complete computation phases are performed followed by the balancing phase. Once the balancing phase has completed, another set of computation phases is performed. Each balancing phase consists of the following operations:

Determine work loads. Each processor determines its work load, the accumulated time taken to process its own local element data. This time is the time since the previous balancing phase less the time taken to exchange the inter-processor boundary data during the computation phase. Neighborhood average work loads are also calculated.

Determine from which processor to request work. Each processor compares its own work load to the work load of the other processors in its neighborhood. It looks for processors that have greater work loads than its own. If any are found, it selects the one with the greatest work load (ties are broken arbitrarily) and sends a request for work to that processor. Each processor may send only one work request, yet a single processor may receive several work requests. Figure 4 shows an example of the work load requests a neighborhood center would send given the work loads shown.

Determine which processor(s) to grant work to. Each processor prioritizes the work requests it receives based on the request sizes. However, before prioritizing the requests, each request is modified to be at most the number of ghost cells the exporting processor has for the requesting processor. This limit allows for smooth edge transformations of cell exports. Requests are satisfied until the processor would have its work load fall below the neighborhood average.

Select export elements. Each exporting processor determines which elements to export. Details of the selection algorithm are given below.

Average Work Load = 73.7

Figure 4. Neighborhood Work Request Example: Processor 4 requests work from its most heavily loaded neighbor.

Notification of and transfer of elements. Once the elements to be exported have been selected, the importers and concerned partners are notified. Importing processors allocate space for the incoming elements, and the elements are transferred.

Each processor knows the number of application iterations to perform before entering the balancing phase. Furthermore, the synchronous nature of the applications guarantees that each processor will enter the balancing phase at the same iteration as the other processors. This synchronous nature is a direct result of the most popular form of programming these machines, *Single-Program, Multiple-Data (SPMD)* [6]. Although SPMD programming leads to a natural measure of relative processor load (computation time), tiling does not require the application to be SPMD. For non-SPMD applications, another measure of relative loads, such as processor idle times, must be used.

The technique for selecting elements for export results in a "peeling" of elements on the processor boundary, preventing the creation of narrow, deep holes in the element structures. Elements are assigned priorities based upon the locality of their element neighbors. An element's priority is initially zero. For each element neighbor local to the element's processor, the element's priority is decreased by one. For each element neighbor local to the importing processor, the element's priority is increased by two. For each element neighbor local to some processor other than the importing processor, the element's priority is decreased by two. Thus, elements whose neighbors are already in the importing processor are more likely to be exported to that processor than elements whose neighbors are local to the exporting processor or are local to some other processor. A special case exists when an element becomes "orphaned";

that is, when it has no neighboring elements in its local processor. To reduce communication costs, it is advantageous to export an orphaned element to any processor whose elements are neighbors of the orphan. Thus, orphaned elements are given a priority of ten so they are always selected for export.

The following example illustrates exporting four of sixteen elements to a neighbor to the east. Initially, as shown in Figure 5, elements 4, 8, 12, and 16 are the only boundary local elements that are eligible. Element 4 has priority -2 since there are 2 local neighbors (-2), one ghost neighbor that does not reside on the importing processor (-2), and one ghost neighbor that does reside on the importing processor (+2). The priorities for the other three elements are determined in the same manner. Our priority scheme will cause either element 8 or element 12 to be selected. Assuming element 8 is selected, the priorities for the remaining elements are adjusted accordingly, as shown in Figure 6. Element 7 is now eligible for export, but its priority is low because it has three local neighbors. Element 12 becomes the clear choice. Selecting element 12, as shown in Figure 7, causes the remaining elements to have their priorities adjusted, along with adding element 11 to the candidates. An arbitrary decision is made as to which of elements 4 and 16 are selected. Selecting element 4, as shown in Figure 8, then makes element 3 a candidate and element 16 the clear choice for the next selection. Selecting element 16 completes the process of migrating the four elements. The selection process results in an edge similar to that of the original edge (compare Figure 5 to Figure 9).

V. Data structures

Application elements are managed by element control data structures that maintain element connectivity and data position information, as shown in Figure 10. Ghost elements point only to local elements, not to each other. Ghost element data are contiguous, allowing processors to receive ghost element data from each neighbor in a single message. A data gather operation is needed to send boundary local element data to neighboring processors.

Local elements are maintained in an AVL tree to allow for efficient manipulation of the element interconnections during migration. Each node in the tree is an element assigned to the local processor. In practice, the local elements are organized in the local element tree to allow for in-order traversal visits of the elements by element ID numbers.

Ghost elements are also maintained in an AVL tree. Each node in the tree represents an element assigned to another processor. As with the local element tree, the ghost element tree is organized to allow for in-order traversals by element ID numbers to allow the assignment of the ghost element data segments in the correct order.

Boundary local elements are maintained in multiple binary trees -- one for each neighboring processor. In practice the boundary element trees are organized to allow for an in-order traversal of the boundary elements based on their element ID numbers, facilitating proper ordering of the boundary element data during an element gather operation for boundary element exchanges

1	2	3	4
pri(x)	pri(x)	pri(x)	pri(x)
5	6	7	8
pri(x)	pri(x)	pri(x)	pri(x)
9	10	11	12
pri(x)	pri(x)	pri(x)	pri(x)
13	14	15	16
pri(x)	pri(x)	pri(x)	pri(x)

Figure 5. Initial Export Priorities:
Elements 4, 8, 12, and 16 are
eligible for export.

1	2	3	4
pri(x)	pri(x)	pri(x)	pri(x)
5	6	7	
pri(x)	pri(x)	pri(x)	
9	10	11	
pri(x)	pri(x)	pri(x)	
13	14	15	
pri(x)	pri(x)	pri(x)	

8

Figure 6. Export Priorities After First
Element Selection: Element 8 is exported;
element 7 becomes an export candidate.

1	2	3	
pri(x)	pri(x)	pri(x)	
5	6		
pri(x)	pri(x)		
9	10		
pri(x)	pri(x)		
13	14	15	
pri(x)	pri(x)	pri(x)	

8
12

Figure 7. Export Priorities After Second
Element Selection

1	2		4
pri(x)	pri(x)		
5	6		8
pri(x)	pri(x)		
9	10		12
pri(x)	pri(x)		
13	14	15	
pri(x)	pri(x)	pri(x)	

Figure 8. Export Priorities After Third
Element Selection

1	2		4
pri(x)	pri(x)		
5	6		8
pri(x)	pri(x)		
9	10		12
pri(x)	pri(x)		
13	14		16
pri(x)	pri(x)		

Figure 9. After Completing Export
Element Selection

Figure 10. Element Interconnection Diagram

A heap is used to maintain a priority list of elements that are candidates for export. Elements are inserted at the end of the heap and rise in the heap to their appropriate locations according to their priority. Each heap entry has two data items, the element's priority and a pointer to the element itself, to enable dynamic update of an element's priority.

VI. Migration convergence

In this section, we prove that our load balancing technique leads to global load balance, even though only local information is used for migration decisions. Leiss and Reddy provide a proof of global load balance convergence, similar to the one that follows, for their task oriented load balancing algorithm.

Assume an imbalance exists and the processing costs of individual elements do not differ by significant amounts.

Define $L(p)$ as the processor load of processor p .

Define two processor classes, P_H , the most utilized processors, and P_L , those that are not most utilized. That is, P_x is in P_H if and only if $L(P_x) \geq L(p)$ for all processors p . By our definition of processor neighborhoods, all processors are neighborhood centers.

For a given migration iteration t , each neighborhood N_x with center P_x in P_H is in one of the following two classes:

- (i) all neighbors, P_j , of P_x are in P_H , or
- (ii) at least one neighbor, P_j , of P_x is in P_L .

The migration iteration t does not change the load of neighborhoods in class (i).

For N_x in class (ii), we show that the migration reduces the load of P_x so that P_x is no longer a member of P_H . In such an instance, some P_j , a neighbor of P_x , makes a request for up to $(L(P_x) - L(P_j))/2$ units of work, which P_x honors. Thus, $L(P_x)_{t+1} < L(P_x)_t$, and P_x moves from P_H to P_L .

Each migration step reduces the cardinality of P_H . Eventually, all P_x in P_H fall into class (ii); i.e., they have neighbors not in P_H . The migration step t then reduces the total iteration cost, and $P_{H,t+1}$, a new P_H set, is created with $L(P_x) < L(P_y)$, for all P_x in $P_{H,t+1}$ and in P_y in $P_{H,t}$.

The iteration process and reduction of $L(P_x)$, where P_x is in P_H , continues until $P_H = \emptyset$, implying all processors have equal loads and thus, the global load is balanced.

It is important to note that only when a new P_H is defined is the benefit of migration realized. Having distributed local balance does not result in a faster iteration; only the reduction of the most costly processor's load will result in a faster iteration. It does not matter if P_1 and P_2 are out of balance as long as the work load of some P_3 is driving the iteration cost. Therefore, the goal of load balancing is not to obtain a perfectly balanced load, but to reduce the load of the most heavily utilized processor. Thus, we tolerate intermediate imbalances to achieve faster iterations sooner.

VII. Application interface

The tiling system is designed to be independent of the application. As shown in Figure 11, the Element Management System (EMS) uses three application interface routines: *App_preproc()*, *App_compute()*, and

App_postproc(). The application programmer provides these routines, using the data structures for the element mesh provided by the EMS.

Figure 11. Element Management System to Application Interface

App_preproc() is called to create elements and assign them initial values. The EMS provides three major routines to the application pre-processing code: *create_ghost_element()*, *create_local_element()*, and *convert_links()*. The two element creation routines allocate element control structures and insert them into the appropriate element management trees. During element creation, element pointers are initialized to the neighboring elements' identification numbers, since the addresses of the elements may not be known yet. However, once all of the local and ghost elements have been created, the application code calls the *convert_links()* routine to replace the element numbers with actual pointers. *App_preproc()* also initializes application specific parameters and element data. The element trees are visible to the application programmer for this purpose.

The processing performed by *App_preproc()* is expected to be quite unbalanced. For example, typically only one processor accesses host files to get run-time initialization information and broadcasts the information to the rest of the processor array. Therefore, migration processing is not performed during the application initialization stage.

App_compute() performs the application's computations. *App_compute()* is called iteratively until it returns a completion flag indicating application termination. The application programmer can schedule any processing desired

in *App_compute()*; however, any communication done during *App_compute()* will affect the processor load calculation. The *App_compute()* processing time determines the processors' loads for the balancing phase.

As in *App_preproc()*, the element tree structures are visible to *App_compute()*. *App_compute()* accesses elements via tree traversal. In Figure 12, we show an example *App_compute()* routine that does not take any parameters, and returns a BOOLEAN completion flag that indicates to the EMS whether the application has completed. This example of *App_compute()* uses two variables to control its iteration, *App_iteration* and

```

BOOLEAN App_compute(void)
{
    BOOLEAN rc;
    extern UINT32 App_iteration;
    extern UINT32 App_iteration_max;

    update_tree(Local_tree_root);

    App_iteration++;
    if (App_iteration == App_iteration_max) {
        rc = FALSE;
    } else {
        rc = TRUE;
    }

    return(rc);
}

```

Figure 12. Example *App_compute()*

App_iteration_max. It is assumed that the corresponding *App_preproc()* routine initialized these two variables. *App_compute()* also uses the *Local_tree_root* variable, which points to the AVL tree managing the processor's local elements. Figure 13 shows example code for *update_tree()*. This version of *update_tree()* is a simple, in-order traversal of the local element AVL tree. For some applications, it is useful to perform this in-order traversal, gaining a horizontal, bottom-up *scan-line* visit of the local elements.

App_postproc() is provided to separate the application's computation from the application's post-processing. Using this routine, the application may perform operations, such as transferring results to the host processor, without interference from the migration processing. Once *App_postproc()* returns, the application is terminated.

VIII. Results

The computational results presented here were obtained by applying our load balancing technique to a massively parallel, adaptive finite element method for hyperbolic conservation laws of the form


```

inline void
update_tree(ELEMENT_TREE_TYPE *p_root)
{

    ELTYPE *p_me, *p_n, *p_e, *p_s, *p_w;
    ELEMENT_TYPE *p_el;

    if (p_root) {
        update_tree(p_root->left);

        p_el = p_root->p_el;
        p_me = (ELTYPE *) (p_el->p_eldata);
        p_n = (ELTYPE *) (p_el->nabor[NORTH].p->p_eldata);
        p_e = (ELTYPE *) (p_el->nabor[EAST].p->p_eldata);
        p_s = (ELTYPE *) (p_el->nabor[SOUTH].p->p_eldata);
        p_w = (ELTYPE *) (p_el->nabor[WEST].p->p_eldata);

        update_element(p_me, p_n, p_e, p_s, p_w);

        update_tree(p_root->right);
    }
}

```

Figure 13. Example Tree Traversal For Updating Local Elements

$$\mathbf{u}_t + \mathbf{f}(\mathbf{u})_x + \mathbf{g}(\mathbf{u})_y = 0, \quad (x, y) \in \Omega, \quad t > 0, \quad (1a)$$

with initial conditions

$$\mathbf{u}(x, y, 0) = \mathbf{u}^0(x, y), \quad (x, y) \in \Omega \cup \partial\Omega, \quad (1b)$$

where $\mathbf{u}$ is the system of conservation variables modelled by the finite element discretization, $\mathbf{f}$ and $\mathbf{g}$ are flux functions, Ω is the physical problem domain, $\partial\Omega$ is the boundary of the domain, and subscripts t , x , and y denote differentiation with respect to time and space in the x - and y - directions. The Euler equations, used in computational fluid dynamics, and the shallow water equations, used in global climate modelling, are examples of hyperbolic conservation laws.

We use a local discontinuous Galerkin finite element method [3][4]. Restricting Ω to be rectangular, partition it into rectangular elements

$$\Omega_{ij} = \{ (x, y) \mid x_{i-1} \leq x \leq x_i, y_{j-1} \leq y \leq y_j \}, \quad i = 1, 2, \dots, I, j = 1, 2, \dots, J.$$

Approximate the solution $\mathbf{u}(x, y, t)$ on Ω_{ij} by a basis of tensor products of Legendre polynomials on the canonical element $\Omega_C = \{ (\xi, \eta) \mid -1 \leq \xi, \eta \leq 1 \}$ that are continuous on the element, but may have discontinuities at inter-element boundaries:

$$\mathbf{u}(\xi, \eta, t) = \mathbf{U}_{ij}(\xi, \eta, t) = \sum_{k=0}^p \sum_{m=0}^p c_{ijkm}(t) P_k(\xi) P_m(\eta), \quad (\xi, \eta) \in \Omega_C. \quad (2)$$

Multiplying (1) by a test function proportional to $P_k(\xi) P_m(\eta)$, integrating the result on Ω_{ij} while integrating the flux terms by parts, mapping Ω_{ij} to Ω_C , and applying the orthogonality properties of Legendre polynomials [21], we construct a weak form of (1):

$$\frac{d}{dt} c_{ijkm} = -\frac{(2k+1)(2m+1)}{2\Delta x_i \Delta y_j} (I_1 + I_2 + I_3), \quad k, m = 0, 1, \dots, p, \quad (3a)$$

where

$$I_1 = -\int_{-1}^1 \int_{-1}^1 [\Delta y_j P_k'(\xi) P_m(\eta) f(\mathbf{U}_{ij}) + \Delta x_i P_k(\xi) P_m'(\eta) g(\mathbf{U}_{ij})] d\xi d\eta, \quad (3b)$$

$$I_2 = \Delta y_j \int_{-1}^1 [P_m(\eta) f(\mathbf{U}_{ij}(1, \eta, t)) - (-1)^k P_m(\eta) f(\mathbf{U}_{ij}(-1, \eta, t))] d\eta, \quad (3c)$$

$$I_3 = \Delta x_i \int_{-1}^1 [P_k(\xi) g(\mathbf{U}_{ij}(\xi, 1, t)) - (-1)^m P_k(\xi) g(\mathbf{U}_{ij}(\xi, -1, t))] d\xi, \quad (3d)$$

and $\Delta x_i = x_i - x_{i-1}$, $\Delta y_j = y_j - y_{j-1}$.

The system of ordinary differential equations (3) is integrated in time using a Runge-Kutta method of order $p+1$. The boundary fluxes $f(\mathbf{U}_{ij}(\pm 1, \eta, t))$ and $g(\mathbf{U}_{ij}(\xi, \pm 1, t))$ are computed by solving an approximate Riemann problem [17][20], requiring an exchange of information across neighboring elements' boundaries. Integral terms are evaluated exactly for linear problems ($\mathbf{f}(\mathbf{u}) = \mathbf{A}(t)\mathbf{u}$) using the orthogonality properties of Legendre polynomials, or numerically for non-linear problems using Gauss-Legendre integration. A projection limiter [3][22] is applied to the numerical solution to remove oscillations that develop near discontinuities.

We apply an adaptive order enrichment (p -refinement) strategy to the two-dimensional method (3). The degree p of $\mathbf{U}_{ij}$ is adjusted separately for each $i = 1, 2, \dots, I$, $j = 1, 2, \dots, J$, so that the estimated local error $E_{ij}(t)$ on Ω_{ij} is less than a specified tolerance TOL_L . For these experiments, we use a p -refinement spatial error estimate:

$$E_{ij}(t) = \left\| \int_{-1}^1 \int_{-1}^1 |\mathbf{U}_{ij}^{p+1}(\xi, \eta, t) - \mathbf{U}_{ij}^p(\xi, \eta, t)| d\xi d\eta \right\|_{\infty} \quad (4)$$

where $\mathbf{U}_{ij}^p$ is the p^{th} -degree approximation of $\mathbf{u}$. When $E_{ij} > \text{TOL}_L$, the degree of the approximation on the element is increased, and the solution for the time step is recomputed. When a time step is accepted, the adaptive strategy

attempts to predict the polynomial degree of the approximation for the next time step on each element by increasing p on Ω_{ij} when $E_{ij} > 0.8\text{TOL}_L$ and decreasing p on Ω_{ij} when $E_{ij} < 0.05\text{TOL}_L$. An outline of the adaptive refinement algorithm is shown in Figure 14.

```

void adaptive_p_refinement()
{
    while (t < t_final) {
        perform_runge_kutta_time_step(all_elements);
        do
            Solution_Accepted = TRUE;
            for each element {
                error_estimate = compute_error_estimate(element);
                if error_estimate > TOL_L {
                    mark_element_as_unacceptable();
                    increase_polynomial_degree_on_the_element();
                    Solution_Accepted = FALSE;
                }
            }
            if (!Solution_Accepted)
                recompute_solution_on_unacceptable_elements();
        } while (!Solution_Accepted);
        accept_solution(all_elements);
        predict_polynomial_degrees_for_next_time_step(all_elements);
        t = t + delta_t;
    }
}

```

Figure 14. Algorithm for the adaptive order refinement strategy.

The non-adaptive Discontinuous Galerkin method is well suited to parallelization on massively parallel computers. The computational stencil involves only nearest-neighbor communication regardless of the degree of the piecewise polynomial approximation and the spatial dimension. Additional storage is needed for only one row of “ghost” elements along each edge of the processor’s uniform subdomain. Thus, the size of the problems solved can be scaled easily with the number of processors. Indeed, scaled parallel efficiencies of over 97% have been demonstrated on an nCUBE/2 parallel computer [3].

Due to processor load imbalance, parallel performance substantially degrades when adaptive p -refinement is incorporated into the method. To determine an efficient domain decomposition and processor mapping, many massively parallel finite element methods use *static* load balancing [5][7][11] as a preprocessor to the finite element computation. Adaptive methods, however, require dynamic load balancing to adjust changing processor loads as the computation proceeds, making them ideal test applications for the tiling load balancing system.

Example 1. We solve

$$u_t + 2u_x + 2u_y = 0, \quad 0 \leq x, y \leq 1, \quad t > 0, \quad (5a)$$

using the adaptive p -refinement method with initial and Dirichlet boundary conditions specified so that the exact solution is

$$u(x, y, t) = \frac{1}{2} (1 - \tanh(20x - 10y - 20t + 5)), \quad 0 \leq x, y \leq 1. \quad (5b)$$

In Figure 15, we show the exact solution of (5) after ten time steps and the adaptive 32×32 -element mesh generated to satisfy $TOL_L = 2.5 \times 10^{-5}$.

Figure 15. Exact solution of (5) after 10 time steps and adaptive p -refinement mesh generated for local error tolerance 0.000025.

Example 2. We solve (5) using the adaptive p -refinement method and tiling on 32×32 -mesh on 16 processors of the nCUBE/2 hypercube. In Figure 16, we show the processor domain decomposition with and without balancing after 10 time steps. The shaded elements have higher-degree approximations and thus, higher work loads. The tiling algorithm redistributes the work so that processors with high-order elements have fewer elements than those processors with low-order elements. The total processing time was reduced 25.9% from 29.49 seconds to 21.86 seconds by balancing once each time step. The average/maximum processor work ratio without balancing is 0.353; with balancing, it is 0.609.

Example 3. We solve (5) for 229 time steps on 1024 processors of the nCUBE/2 without balancing and with balancing once each time step. With balancing, the maximum computation time (not including communication or balancing time) was reduced by 60.0% (see Table 1). The irregular subdomain boundaries created by the tiling

Figure 16. Processor domain decomposition after 10 time steps for Example 2 using adaptive p -refinement on 16 processors without load balancing (left) and with load balancing (right). Shaded squares are high-order elements, and dark lines represent processor subdomain boundaries.

algorithm increased the average communication time by 31.0%. Despite the extra communication time and the load balancing time, however, we see a 35.8% improvement in the total execution time.

	Without Balancing	With Balancing
Maximum Execution Time	698.71 secs.	448.47 secs.
Maximum Computation Time	659.59 secs.	264.04 secs.
Average Communication Time	74.07 secs.	97.04 secs.
Maximum Balancing Time	0.0 secs.	41.28 secs.
Average/Maximum Work Ratio	0.369	0.924
Parallel Efficiency	34.8%	54.4%

Table 1. Performance comparison for Example 3 without and with load balancing once each time step.

In Figures 17-20, we show the convergence of the processor work loads from uniform domain decomposition toward global balance. In Figure 17, we divide the difference of the maximum and minimum processor work loads at time $t = 0$ into 32 bins. As time progresses and the processors approach global balance, the processor distribution should converge toward the bin representing the average computation time. The dashed line represents the number of processors in each bin at time $t = 0$. The solid line represents the number of processors in each bin after 20 balancing steps. At time $t = 0$, the maximum number of processors in a single bin is 904. After 20 time steps, the maximum number of processors in a single bin drops to 859 as work from more heavily loaded processors is migrated to the less heavily loaded processors. After 80 time steps, 773 processors have converged to a single bin with fewer processors in the heavily loaded bins, as shown by the dashed line of Figure 18. Twenty time steps later, the processors have

converged even further toward global balance, with 760 processors in a single bin and almost no processors in the heavily loaded bins, as shown by the solid line.

Figure 17. Processor work load distributions initially (dashed line) and after 20 time steps (solid line).

In Figure 19, we show the +1 and -1 standard deviation curves of the maximum computation time for each time step. Initially, the deviation is large, indicating the processors are far from global balance. The deviations quickly become smaller, indicating the processors rapidly approach global balance. In Figure 20, we show the 5th, 35th, median, 65th, and 95th-percentile processor loads. The steep downward slope of the 95th-percentile curve indicates significant improvement in the load balance early in the program's execution.

In Figure 21, we show the maximum processing costs per time step, including the computation time and the balancing time. The dashed line represents the maximum cost per time step without balancing; the solid line represents the maximum cost with balancing. Even including the load balancing time, the balanced computation's maximum cost per time step is significantly lower than without balancing. Variations in the cost per time step without balancing illustrate the changing computational effort caused by the adaptive finite element method.

In Figure 22, we show the cumulative maximum processing times with and without balancing. The immediate and sustained improvement of the application performance is shown.

Figure 18. Processor work distributions after 80 time steps (dashed line) and after 100 time steps (solid line).

IX. Future directions.

The extension of this work to 3D regular grids is straightforward. The element interconnections remain simple; pointers in the Up and Down directions are included with the 2D pointers in the North, South, East, and West directions. The balancing overhead is slightly higher, as there are more neighbors to track and notify of changes. The concept of processors neighborhood does not change except for the initialization to 6 neighbors instead of the 4 used in the 2D domain.

Irregular grids constantly challenge application programmers. Effective representations are complicated, requiring non-trivial data structures. For tiling, the added challenge resides mostly in the inverse mapping techniques: how can element A look up the reference to element B given element B knows how to reference element A. For 2D and 3D regular grids, the mapping is easy: if A is B's north neighbor, then B must be A's south neighbor. Although determining an efficient reverse mapping for irregular grids is not crucial to using tiling with irregular grids, having one would certainly improve the tiling performance on such grids. One possibility is to use an algorithm that *colors* elements so that an element is assigned a color that is different from all of its neighbors and no two neighbors of any given element would have the same color. Then knowing A is RED tells B how to look up its reference to A. For fixed sized elements, one must have connectivity space for the worst case interconnection requirement. To avoid the need

Figure 19. Processor work load mean and standard deviation for each time step.

Figure 20. Sampled processor loads for each time step. The steep descent of the 95th-percentile curve indicates the processors are approaching global balance rapidly.

Figure 21. Maximum work load in each time step with balancing (solid line) and without balancing (dashed line).

Figure 22. Cumulative maximum loads with and without balancing.

for fixed sized elements, an approach using variable reference counts and one more level of indirection could be implemented to provide a mapping of what RED means to B as opposed to what RED may mean to other elements. Although this extra level of indirection requires slightly more processing, the reduced memory requirements may be worth the extra processing.

X. Conclusion

We have demonstrated the effectiveness of the tiling run-time system as a dynamic load balancer for a wide class of finite element and finite difference applications with both fixed and dynamic per-element processing costs. Even when integrated with an adaptive finite element method, which provides a "moving target" for global load balancing, the tiling system improved the application's parallel efficiency by over 20% and reduced total execution time by over 35%. The application interface to the tiling system is modularized so application programmers who are not specialists in parallel computing science can easily improve the performance of their programs. Finally, the tiling system communication software is isolated to very few routines, leading to simple porting of the system from one MP MIMD architecture to another.

XI. References

- [1] Amdahl, G. "Validity of the Single-Processor Approach to Achieving Large-Scale Computer Capabilities." *AFIPS Conference Proceedings* 30, 1967, 483-485.
- [2] CM-5 Technical Summary, Thinking Machines Corporation, 245 First Street, Cambridge, Massachusetts, 02142-1264, January 1992.
- [3] Biswas, R., K. Devine, and J. Flaherty, "Parallel, Adaptive Finite Element Methods for Conservation Laws." *Applied Numerical Mathematics*, to appear.
- [4] Cockburn, B., S. Hou, and C.-W. Shu. "The Runge-Kutta Local Projection Discontinuous Galerkin Finite Element Method for Conservation Laws IV: The Multidimensional Case." *Math. Comp.*, 54 (1990), 545-581.
- [5] Hammond, S. *Mapping Unstructured Grid Computations to Massively Parallel Computers*. Ph.D. thesis, Rensselaer Polytechnic Institute, Dept. of Computer Science, Troy, NY, 1992.
- [6] Heath, M.T. "The Hypercube: A Tutorial Overview," *Proceedings of the First Conference on Hypercube Multiprocessor*, pp. 7-10, August 26-27 1985, SIAM Philadelphia 1986.
- [7] Hendrickson, B., and R. Leland. "Multidimensional Spectral Load Balancing." Sandia National Laboratories Tech. Rep. SAND93-0074.
- [8] Gustafson, J.L., G.R. Montry, and R.E. Benner. "Development and Analysis of Scientific Applications Programs on a 1024-Processor Hypercube." *SIAM Jnl. Sci. Stat. Comp.* 9 (1988), 609-638.
- [9] Intel Paragon Supercomputers Sales Literature. Order Number 203/6/92/10K/GA, Intel Corporation, Supercomputer Systems Division, 15201 NW Greenbrier Parkway, Beaverton, OR 97006, 1992.
- [10] Kao, S.T. and E.L. Leiss. "An Experimental Implementation of Migration Algorithms on the Intel Hypercube," *The International Journal of Supercomputer Applications.*, Vol. 1, No. 2, pp. 75-99, 1987.
- [11] Kernighan, B. and S. Lin. "An Efficient Heuristic Procedure for Partitioning Graphs." *Bell Systems Tech. Jnl.*, 29 (1970), 291-307.
- [12] Leiss, B. and H. Reddy. "Distributed Load Balancing: Design and Performance Analysis." *W.M. Keck Research Computation Laboratory*. 5 (1989) 205-270.
- [13] Lin, F.C.H. and R.M. Keller. "The Gradient Model Load Balancing Method," *IEEE Transactions on Software Engineering*, July 1987.

- [14] nCUBE Corporation, nCUBE 3 Technical Overview, nCUBE, 919 East Hillsdale Blvd., Foster City, CA 94404, 1992.
- [15] Rau, B.R. "Cydra 5 Directed Dataflow Architecture," *COMPCON Spring 88, Thirty-Third IEEE Computer Society International Conference*, pp. 106-113, March 1988.
- [16] Reddy, H.N. "On Load Balancing," Ph. D. dissertation, Department of Computer Science, University of Houston, 1989.
- [17] Roe, P.L. "Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes." *Jrnl. of Comp. Phys.*, 43 (1981), 357-372.
- [18] Suen, T.T.Y. and J. S. K. Wong. "Efficient Task Migration Algorithm for Distributed Systems," *IEEE Transactions on Parallel and Distributed Systems*, pp. 488-499, July 1992.
- [19] Schwederski, T., H. J. Siegel and T. L. Casavant. "A Model of Task Migration in Partitionable Parallel Processing Systems," *IEEE Second Symposium On The Frontiers of Massively Parallel Computations*, pp. 211-214, 1988.
- [20] Sweby, P.K. "High Resolution Schemes Using Flux Limiters for Hyperbolic Conservation Laws." *SIAM J. Numer. Anal.*, 21 (1984), 995-1011.
- [21] Szabo, B. and I. Babuska. *Introduction to Finite Element Analysis*, Wiley, New York, 1990.
- [22] Van Leer, B. "Towards the Ultimate Conservative Difference Scheme. IV. A New Approach to Numerical Convection." *Jrnl. of Comp. Phys.*, 23 (1977), 276-299.
- [23] Wheat, S. *A Fine Grained Data Migration Approach to Application Load Balancing on MP MIMD Machines*. Ph.D. Thesis. Univ. of New Mexico, Dept. of Computer Science, Albuquerque, NM, 1992.

DATE

FILMED

6 / 14 / 94

END