Multithreaded model for dynamic load balancing parallel adaptive PDE computations

Nikos Chrisochoides*
Advanced Computing Research Institute
Cornell Theory Center, Cornell University
Ithaca, NY 14853-3801

Abstract

We present a multithreaded model for the dynamic load-balancing of numerical, adaptive computations required for the solution of Partial Differential Equations (PDEs) on multiprocessors. Multithreading is used as a means of exploring concurrency in the processor level in order to tolerate synchronization costs inherent to traditional (non-threaded) parallel adaptive PDE solvers. Our preliminary analysis for parallel, adaptive PDE solvers indicates that multithreading can be used as a mechanism to mask overheads required for the dynamic balancing of processor workloads with computations required for the actual numerical solution of the PDEs. Also, multithreading can simplify the implementation of dynamic load-balancing algorithms, a task that is very difficult for traditional data parallel adaptive PDE computations. Unfortunately, multithreading does not always simplify program complexity, often makes code re-usability not an easy task, and increases software complexity.

*This work was supported by an Alex Nason Prize Award, by NSF ASC 93 18152/ PHY 93 18152 (ARPA supplemented), NASA Contract No. NAS1-19480, while the author was in residence at ICASE, NASA Langley Research Center, Hapton, VA 23681 and in part by the Cornell Theory Center.

1 Introduction

One of the difficulties in parallel programming is attaining good performance when solving problems with irregular and dynamic (adaptive) computations. Many of the early successes of parallel processing were obtained on relatively regular problems (e.g., structured, static grids). The need to solve real-life problems increased the necessity to address issues related to the parallelization of irregular and adaptive computations such as adaptive finite-element computations for fluid flows and structures.

Traditional load-balancing methods for non-threaded computations under-utilize multiprocessor resources such as CPU, memory, and network, since the load-balancing is carried out in sequential phases that require global synchronizations [1], [2]. As an alternative to the traditional load-balancing methods we propose a multithreaded model. According to this model processors execute many computational actions or threads of control; each thread is typically dependent on results of local or remote threads. Instead of scheduling these threads in a static and predefined way, we allow them to be dynamically scheduled based on availability of the data they depend on.

Multithreading here is used as a mechanism for concurrently executing actions or tasks required for load-balancing—such as information dissemination, decision making, and data migration—and tasks required for the computation of the actual application. Multithreading improves CPU and network utilization by masking the inherent synchronization delays involved in traditional non-threaded load-balancing methods. We prove that under certain values of the parameters (i.e., number of threads, and context switch time) of the model, multithreaded load-balancing systems are expected to perform better than existing non-threaded load-balancing software.

The rest of this paper is organized as follows: Section 2 provides an overview of exiting load-balancing methods and a brief introduction to lightweight threads. In Section 3 we describe a set of geometric abstractions that we use in the rest of the paper. Section 4 outlines the basic principles of our load-balancing algorithm based on the multithreaded model. Section 5 presents a comparison of the load-balancing algorithm with existing direct and incremental load-balancing methods. Finally, in Section 6 we outline a number of advantages and disadvantages of the multithreaded approach for parallel numerical computing and we conclude with future work and directions.

2 Background

Parallelism, for data-parallel PDE solvers, is achieved by decomposing the underline geometric (i.e., grids) or algebraic (i.e., linear systems of equations) data structures. The data decomposition of grids or sparse coefficient matrices is equivalent to a graph partitioning problem which is an NP-Complete problem. During the last 8 to 10 years many interesting heuristics have been proposed to compute sub-optimal data distributions for static PDE problems. In this section we first, present a brief overview of the most commonly used heuristics and introduce basic concepts of threads which we will be using throughout the paper.

2.1 Load-balancing: an overview

The time it takes to execute a program on a multiprocessor is equl to the time it takes for the slowest (overloaded) processor to complete its computation (calculations and communication). Therefore, the objective of load-balancing algorithms and software is to minimize the execution time of the slowest processor. Existing load-balancing algorithms and software systems assume no overlapping of communication with calculations and balance the processors'[1] workloads by minimizing the following objective function [48]:

$$OF \;=\max_{1\leq i\leq \mathbf{P}}\{\;\; W(m(D_i)) \;+\sum_{D_j\in\kappa_{D_i}} C(m(D_i),m(D_j)) \;\;\} \tag{1}$$

where, for data-parallel PDE solvers, $m:\{D_i\}_{i=1}^{\mathbf{P}}\rightarrow\{P_i\}_{i=1}^{\mathbf{P}}$ is a function that maps the grid points (grids) of submesh D_i to processor $P_i=m(D_i)$; $W(m(D_i))$ is the computational load of the processor $m(D_i)$, which is proportional to the number of grids in D_i; $C(m(D_i),m(D_j))$ is the communication cost required between the processors $m(D_i)$ *and* $m(D_j)$; and, finally, κ_{D_i} is the set of submeshes that are adjacent to D_i. These heuristics are classified into two classes: direct (or clustering) and iterative (or incremental). A list of very interesting results for direct and incremental methods, which is by no means complete, appears in [1], [13], [22], [23], [24], [25], [40], [41], [42], [43], [44], [45], [46], [48], and [49].

The data-clustering algorithms are based on grouping mesh points into clusters such that the points within a cluster have a high degree of "natural association" among themselves, while the clusters are "relatively distinct" from each other. In our case, "natural association" is expressed

[1] We assume homogeneous processors.

in terms of the locality properties of the finite element and finite difference stencils that are used to approximate a continuous PDE operator. The "relative distinction" is expressed in terms of the address space that is associated with the unknowns of the mesh or grid points of the same cluster. Most of these algorithms are computationally expensive and very successful in solving the load-balancing problem for static PDE computations [13], [47], [48]. They require a complete global knowledge of the graph (mesh) and, therefore, these methods are not suitable for adaptive methods in which the topology and/or geometry of the mesh change any time h-refinement is performed throughout the PDE solution process. In addition, some of these methods are sensitive to small perturbations (h-refinement) and often lead to heavy data migration [43], [44].

On the other hand, existing incremental (non-threaded) methods are not as expensive as clustering methods. Flaherty's group at RPI have shown in [1], [49] that these methods are very successful in load-balancing the computation of adaptive PDE methods on distributed-memory MIMD machines. Incremental methods—as is true for direct methods—decompose the parallel adaptive PDE computations into three phases: (*i*) *computation*, (*ii*) *balancing* and (*iii*) *data-migration*. The *computation* phase corresponds to the actual computation and inter-processor communication required by the PDE solver, while the *balancing* and *data migration* phases correspond to the calculation and inter-processor communication required to solve and enforce the solution of the minimization problem defined by equation (1). A global synchronization barrier guarantees that all processors reach the balancing and data-migration phases at the same time [1].

2.2 Threads

Threads were used for many years in disciplines such as real-time systems, and distributed operating systems very successfully. A thread is an independent set of instructions that executes within the context of a UNIX process (see [8], [9], [10], [11], and [12]). Threads in multithreaded programs run logically concurrently. For multicomputers, the decomposition of a coarse-grained computation into finer grained, logically concurrent executable threads is needed for three reasons: (1) to overlap in time logically separate tasks that use different resources (i.e., network, CPU, disks), (2) to simplify parallel programming, and (3) to load balance computations. Typical examples for the logically concurrent execution of threads are (i) the overlapping of calculation with communication [14], and (ii) the overlapping of load balancing and actual computation phases [15].

During execution each thread can be in one of the following states: *new, ready, running, blocked, dead.* The state of a thread is defined by its current activity. When a thread is created it is given a function to run and it is set to a *new* state. A thread in the *new* state consists of various data structures that describe its context. Once the data structures are allocated and the thread is registered with the system, the thread moves to the ready queue and its state is set to *ready.* If a thread is selected to execute, then its state changes to *running.* While a thread is in the running state it may decide[2] to wait for a condition or for outstanding receives to be signaled, in which case its state changes to *blocked.* Finally, after a thread completes its execution or decides to terminate its state, it changes to *dead.* The use of these states will become clear in Section 4.

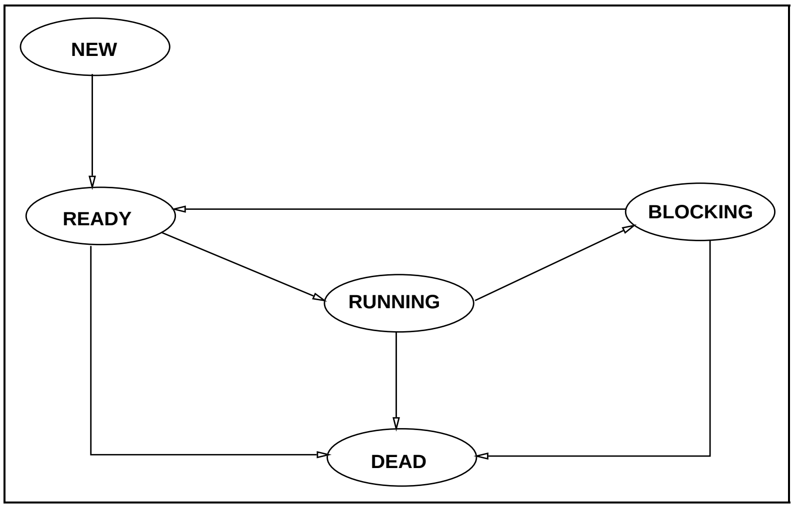

Figure 1: Thread state diagram.

For the type of parallel computations we address in this paper we need non-preemptive scheduling of the threads (i.e., threads run to completion or voluntarily yield the CPU). An advantage of such a scheduling strategy is the reduction of overhead by keeping context-switches[3] to a minimum. Threads are classified into *heavy- or light-weight* threads based on the amount of context (weight) that needs to be saved/restored when a thread is removed or reinstated from/to CPU.

[2]In the case of non-preemptive systems.

[3]Switching from one thread to another requires a certain amount of time for administration (saving and loading registers and memory maps, updating various tables and lists).

In this paper we use light-weight (or user-defined) threads.

The computation of multithreaded data-parallel programs consists of two phases, namely, *computation* and *thread-scheduling* phases. During the computation phase, the processor concurrently performs the operations needed to execute the actual computation: (1) it initializes *send/recv* operations, (2) it polls and notifies threads with outstanding *recvs*, (3) it performs actual calculations, and (4) it provides remote service requests such as check/change status of remote threads. Finally, during the thread-scheduling phase the processor does the bookkeeping required for the administration and the execution of threads. This is an additional overhead that does not appear in the non-threaded, data-parallel paradigm.

3 Basic abstractions

We break the original load-balancing problem into many simpler problems by defining a hierarchy of geometric abstractions: *domains, blocks, subdomains and regions.* Regions are mapped to scalar objects called threads. Threads execute in a loosely synchronous mode. Based on computation and synchronization requirements, threads are grouped into distributed objects: *strings, ropes and nets.* Threads that correspond to regions of the same subdomain belong in the same string (see Figure 2). Threads that belong in the same string execute the same code on different data (SPMD model). Strings that correspond to the subdomains of the same block belong in the same rope. Threads on different ropes might compute the solution for different PDEs, use different grid types, apply different solution methods, and they may therefore have different computational requirements and synchronization points (MPMD model). Finally, ropes that correspond to blocks of the same domain and handle the computation associated with the same application belong in the same net (see Figure 2).

Processor workload is balanced by: (i) migrating threads from overloaded processors to underloaded ones that handle strings from the same rope, and (ii) by migrating strings from overloaded processors to underloaded ones that handle ropes from the same net. The thread and string migration is non-preemptive and, therefore, instead of thread migration we perform data-migration. The data are migrated so that subsequent communication for the actual parallel computation of the PDE solver is minimized.

For each subdomain, for example S_{j,B_i} of Figure 3, we identify two types of regions: *interface* regions and *interior* regions. A region is considered to be interface if there is a grid point in the

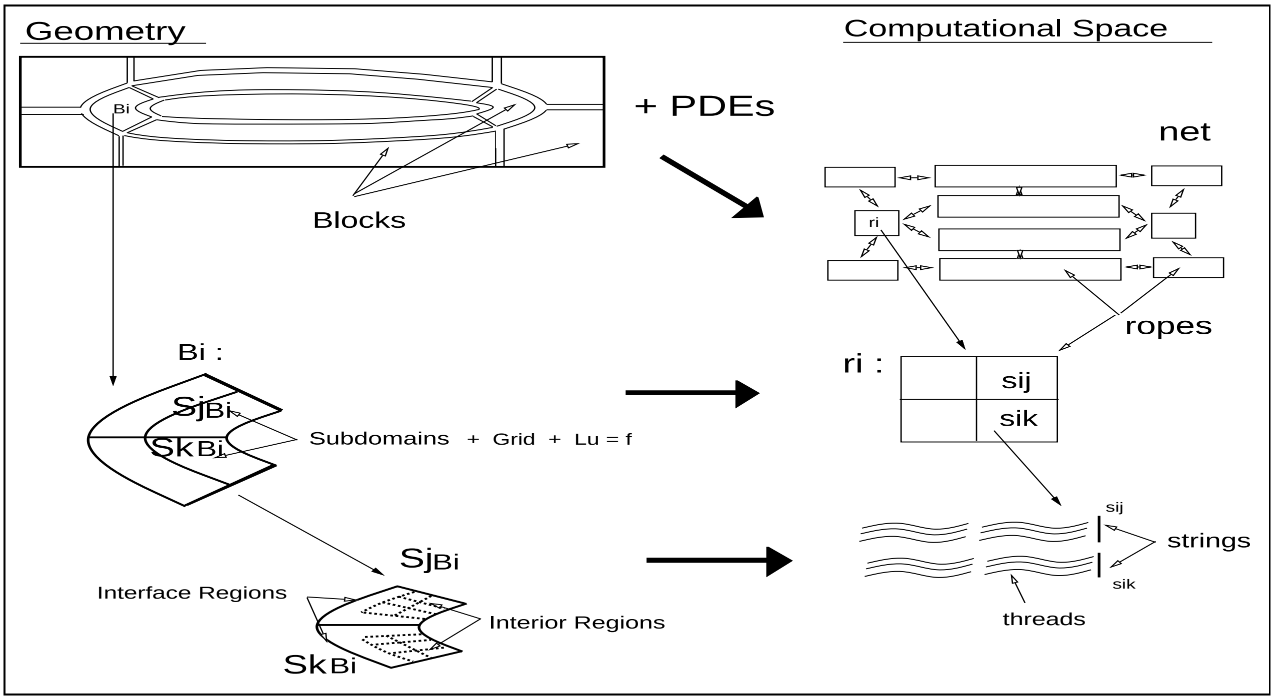

Figure 2: Geometric and parallel abstractions.

region that has at least one of its adjacent neighbor points residing in a different context (traditionally non-local memory). Non-interface regions are considered to be interior. Computations on interior regions of different subdomains can be performed independently. To each interface or interior region we correspond (create) one *thread*, t (see Figure 3). The size, $|t|$, of a thread is analogous to the number of grid points of the corresponding region—many times in this paper we denote by t the mesh that corresponds to the thread t. All threads for h-refinement PDE methods are of the same size. The size of a thread can change during the computation in order to achieve better balancing of processors' work-loads (i.e., each thread can be split into two or more threads, depending on the required load-balancing resolution; such a resolution can be achieved within a small number—order of $\log_2 |t|$ or $\log_4 |t|$— of iterations compared to the number of iterations required by incremental load-balancing algorithms [1]).

Interior threads execute exclusively on data residing in the memory of the processor on which the threads execute, while interface threads require the access of non-local data. Threads corresponding to regions of the same subdomain belong in the same process (context) and communicate using the shared-memory (user-address space for the process) model, while interface threads that correspond to regions from different subdomains communicate through message

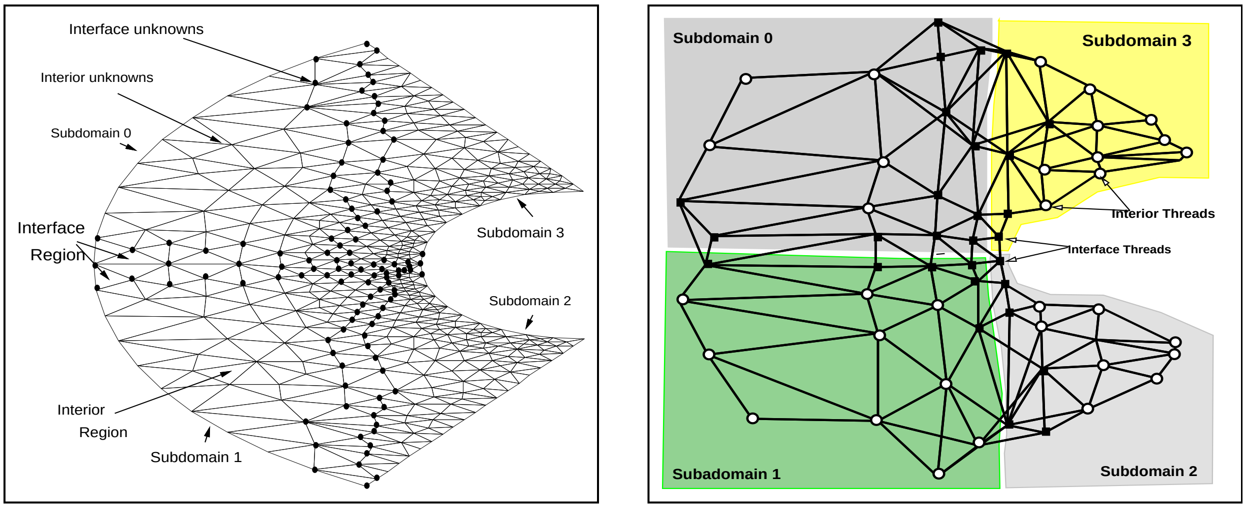

Figure 3: Left) Block, B_i, in middle row second from the left of Figure 2; B_i is partitioned into four subdomains, $\{S_{i,B_i}\}_{i=1}^{4}$. Right) Global thread graph and its partition into four subgraphs that correspond to $\{S_{i,B_i}\}_{i=1}^{4}$. Threads with data dependencies (edges) that cross the internal boundary of the subdomains are interface threads, otherwise they are interior threads.

passing (for more details, see [17]). Two types of local communication are identified: *inter-block (or inter-group)* and *intra-block (or intra-group)*. Network trafic can be reduced by using a scheduling communication mechanism between threads with different types of data flow needs. Such schedulings can be achieved by grouping threads of the same block into *ropes* [26], [27], [28]. Ropes can use group synchronization and collective communication mechanisms and more than one rope can share resources such as CPU, network, and memory. For more details on ropes, see [27] and [26].

4 Multithreaded approach for load-balancing

In this section we describe a multithreaded model for developing load-balancing (sharing) algorithms. The traditional non-threaded approach for load-balancing of PDE computations leads to (1) under-utilization of multiprocessor resources such as the CPU and network and (2) in some cases intensification of problems like network contention —due to the fact that all processors perform data migration simultaneously. In this section we propose a new approach for load-balancing that explores concurrency within the processor in order to maximize utilization of multiprocessor resources without sacrificing program complexity. Our approach, in contrast to the direct and incremental, non-threaded, load-balancing methods, attempts to ensure that no processor is waiting idle while more than one thread remains to be executed on other processors. Each processor, when the need arises, requests work (threads) from a subset (neighborhood) of processors that are overloaded or slow.

Independent of the approach (traditional or multithreaded) we use to load-balance PDE computations, we should focus on about three fundamental issues, namely: *(1) memory latency, (2) synchronization cost, and (3) convergence rate.* In this paper we address the memory latency and synchronization cost; it is difficult to uncouple these issues and therefore we have to consider the convergence rate of PDE solvers whenever we deal with message passing and scheduling latencies. In the rest of the paper we address issues related to convergence rate of PDE solvers only when it is absolutely necessary.

4.1 Memory latency

Parallel computers introduce a new level in the storage hierarchy; in addition to registers, cache and memory, there is remote memory that is accessed across an interconnection network. In this paper our objective is to distribute the computation and data so that we not only balance processors' workloads but also minimize overheads due to message passing (i.e., memory latencies). In order to achieve our objective: (1) we minimize the access of non-local unknowns by minimizing the number of grid points that reside on the interfaces of the subdomains (see Figure 3) and (2) we mask memory latencies due to access of non-local data by overlapping calculations with communication.

For multithreaded parallel PDE computations we identify two types of communications: first, the communication between interior and interface threads that reside in the same context (local threads), and second, the communication between interface threads that belong to different contexts (remote threads). The efficient communication of interior and interface threads is critical for the overall performance and success of the multithreaded approach. Next we briefly describe the different communication mechanisms between local and remote threads. For more details on the implementation of these mechanisms, see [35], [36], [37], [20], [38], and [39].

The communication between local threads can be implemented using one of the following three approaches: (1) *message-passing*, (2) *shared variables*, or (3) *$\mathcal{W}$-buffer scheme* [35]. An advantage of the first approach is that it treats the communication of both interior and interface threads in a uniform way, thereby simplifying programming. A disadvantage is that its implementation (with some exceptions e.g., Mach) requires copies of thread-specific data-structures from one thread to another; recall that (a) the data-structures reside in the same user-address space and (b) memory copy operations introduce an additional overhead that does not appear in the non-threaded approach. The second approach eliminates additional copy operations by using shared global data-structures. A disadvantage of this approach is that it requires the use of synchronization mechanisms (mutexes) to protect unwanted reads/writes from/to shared variables. The implementation of this approach increases program complexity and requires drastic code restructuring of existing non-threaded programs.

Finally, the third approach we present in [35] is a communication scheme specific to PDE computations[4] and is based on $\mathcal{W}$ (≥ 2) copies of the shared-variables. The idea of this scheme is to use "rondeau" memory locations in such a way that a thread (destination in the case of message passing) always reads the correct values that its partner (source) just wrote. Read and write operations between threads that share these copies of variables are interleaved (odd/even —*mod* $\mathcal{W}$— iterations in the case $\mathcal{W} = 2$) in a way that the overwriting (by one thread) of useful values (to another thread) is prevented. Therefore, this scheme preserves the integrity of shared variables without substantially increasing program complexity (in the case of non-copy shared-variables) and without introducing overheads of unnecessary and expensive copy operations (in the case of message passing) on local data structures. This approach can be implemented on top of the existing non-threaded data-parallel codes with minimum modifications. A disadvantage of this communication scheme is that the storage complexity is increased by $O\left(\mathcal{W} \cdot \sqrt{|t|}\right)$.

The communication between remote threads (i.e., threads that reside in the memory of different processors) can be implemented on top of existing message-passing such as MPI [19], p4 [29], or PVM [30]. Unfortunately, most of the currently available message-passing software do not provide support for sending/receiving messages to (from) a specific entity (function) of a process. To provide thread-to-thread communication mechanism we use an idea similar to the AMs which is described next. For more details on the implementation of the thread-to-thread communication mechanism, see [36], [37], [20] and [39].

An interface thread that executes for the first time sends its messages to other threads, then

[4] However it can be generalized for many other similar computations.

posts all its receives[5] and voluntarily yields the CPU to the dispatcher. The dispatcher does the proper bookkeeping and schedules the next interface thread—if any. After all the interface threads from the ready queue are exhausted the dispatcher schedules the first interior thread. Interior threads require only local data and therefore execute until completion. This process is repeated until all interior threads are also exhausted from the ready queue.

During the time interior threads perform their own computations, non-local data is arriving from the network (see Figure 4). The non-local data is stored in user-address space and in memory locations that the user provides to the OS. In [36] we describe a mechanism where a specific function (message-handler) is activated (on message arrival the process is interrupted by a hardware signal) and performs the following operations:

- Parses the header of the message and identifies the destination thread.
- Decrements thread counter that indicates the number of outstanding receives that correspond to this thread; if the counter of outstanding receives becomes zero then the state of the thread changes from *blocked* to *ready*, and the thread moves from the blocking stack to the ready stack (see Figure 4). At this point, interface threads have all the data (local and non-local) they need to perform their computations. Notice that while interface threads are waiting for incoming messages (through the network), the CPU is utilized by the interior threads, and thus calculations and communication are overlapped.

4.2 Synchronization cost

Load-balancing operation is a special case of the *producer-consumer* operation. Consumer-producer operations like forks and joins and mutual exclusion in parallel programming require synchronization. In parallel adaptive PDE computations the synchronization cost appears in the form of waiting time due to unbalanced processor workloads. Our objective is to minimize this time and mask if possible inherent delays involved in the traditional load-balancing methods without increasing program complexity. Our approach, in contrast to the direct and incremental non-threaded load-balancing methods, attempts to ensure that no processor is idle while more

[5] If a non-blocking receive operation is available, it first posts its receives and then sends its messages to other threads, which increases the probability of saving a local copy of the message from the system buffer to user address space.

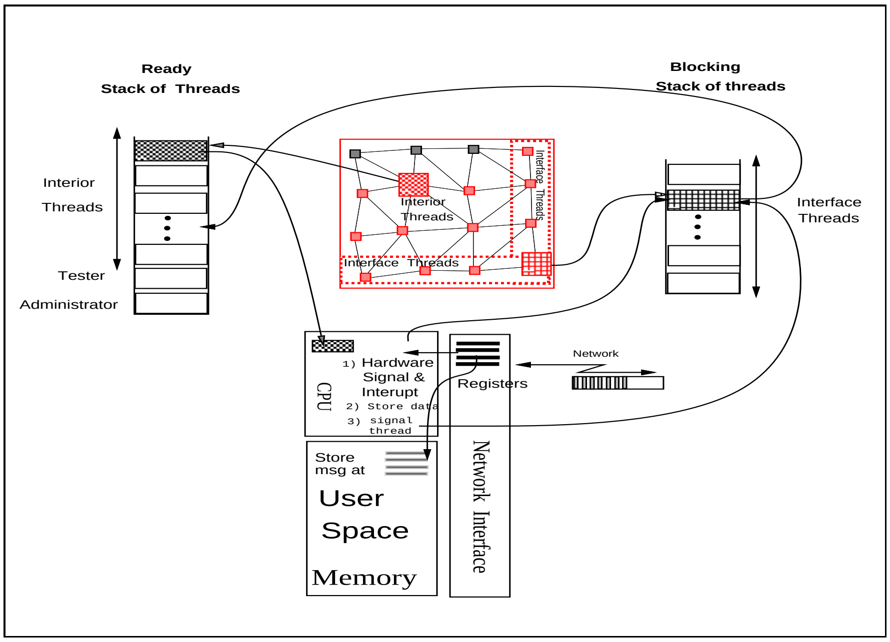

Figure 4: Overlapping of computation with communication; while interface threads are blocked in the *blocking stack* waiting for the arrival of non-local data, interior threads—from the ready stack—utilize the CPU, performing computations on local data.

than one thread remains to be executed on other processors. Each processor, when the need arises, requests work (threads) from a subset (neighborhood) of processors that are overloaded or slow.

Let us assume that our processors are homogeneous and our PDE solver does not share the resources of the system with any other application. We define as *computational graph* $G_C(E_C, V_C)$, and denote as G_C, the graph whose vertices, v_i, correspond to the submeshes D_i, and the edges $e_{i,j}$ connect two vertices (v_i, v_j) if $D_i \cap D_j \neq \emptyset$. The weights, w_i, on the vertices v_i of the graph correspond to the computation associated with the submesh D_i and are analogous to the number of mesh points, $|D_i|$. Figure 5–right depicts the computational graph of the refined mesh (Figure 5–left); the weights w_i indicate the number of threads per subdomains (i.e., context or processor).

During the computation of adaptive PDE methods, the mesh is refined in areas where the resolution of the solution is larger than a given tolerance (see Figure 5). After the mesh refinement is completed, new threads are created (or old ones are destroyed) at runtime. All threads are approximately of the same size (h-refinement). Processor computation is balanced by migrating interface threads. The thread migration is non-preemptive (i.e., threads migrate before they

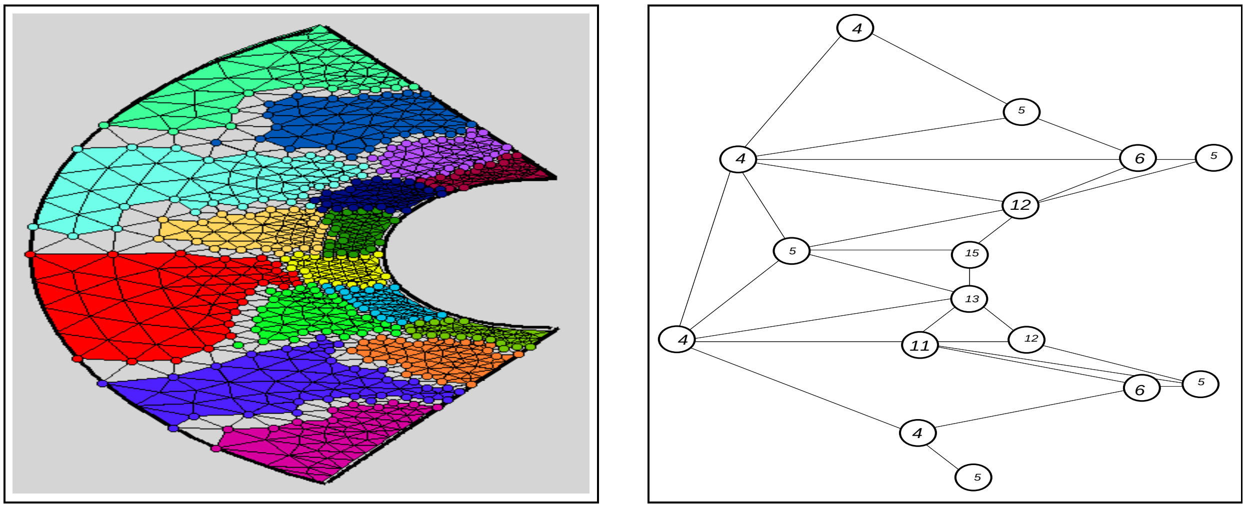

Figure 5: Left) h-refinement and a 16-way partition of the block B_i of domain defined in Figure 2. Right) Computational graph with uneven number of threads per subdomain (i.e., context or processor).

start execution) and, therefore, instead of thread migration, (save-and-migrate threads context, registers-values, etc.) we perform data-migration of mesh-points only. The data is migrated so that subsequent communication for the actual parallel computation of the PDE solver is minimized. In contrast to traditional incremental methods, load sharing of processors is completed within the first iteration (i.e., before any global reduction operation is required for error checking or update of global variables).

The policy for thread migration is based on a *consumer-initiated consumer/producer (C/P) paradigm*. That is, every processor $P_i = m(D_i)$ (consumer), after it completes its computation (when counter of ready and blocking stacks is zero), searches its neighborhood

$$N(P_i, l) = \{P_j, P_j = m(D_j) \; and \; D_j \in N(D_i, l) \; , l = 1, ..., \; diameter(G_C) \; \}$$

to identify neighboring processors that are overloaded.[6] Since our model attempts to assure that no processor remains idle, the consumer sends interrupt-driven messages to its neighbors (see [36] for implementation details) and requests the migration of one or more threads.

[6] Due to changes on the demand of computation, in the case of adaptive methods, or due to external loads of the processors in the case of time-sharing heterogeneous workstations.

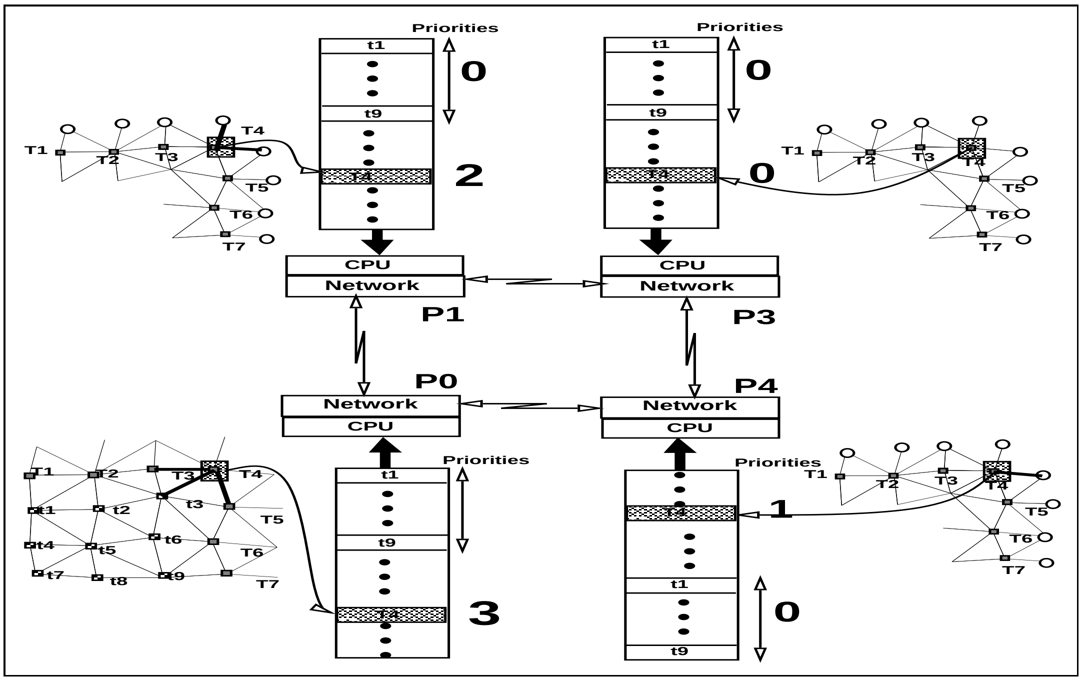

Figure 6: Threads T_4 migrates from processor P_0 to processor P_1; T_4 has most of its data dependencies with the threads of processor P_1. The thread migration is based on the principle of minimizing the communication of the actual computation.

After an overloaded processor $P_j \in N(P_i, l)$ is identified, P_j (producer) interrupts its computation and sends a thread (data) to P_i. The thread that is migrated from the producer to consumer is likely to have data dependencies with other threads that already reside on the consumer's side. The producer's decision to migrate an interface thread, t_i, is based on priorities, Π_{t_i}, that are computed at runtime using the following equation:

$$\Pi_{t_i} = \sum_{t_j \in N_{t_i}} (1 - \chi(t_i, t_j))$$

where

$$\chi(t_i, t_j) = \begin{cases} 0 & \text{if } m(t_i) \;=\; m(t_j) \\ 1 & \text{if } m(t_i) \;\neq\; m(t_j) \end{cases}$$

and N_{t_i} is the set of all threads t_j with data dependencies on t_i. Figure 6 depicts the different priorities for an interface thread T_4 of processor P_0.

A consumer (processor) might get data from more than one producer. In this case it creates and executes remote threads one at a time, using a FIFO policy. Before a remote thread,t_{rmt}, is created by the consumer, the consumer uses a remote service request, Check_Thread_State, to find out the current state of thread t_{rmt} on the processor (producer) from which thread t_{rmt} come from. If the current state of t_{rmt} thread is *ready*, then t_{rmt} is created and scheduled for execution on the consumer; its state on the producer changes from *ready* to *dead*. The Check_Thread_State is also an interrupt-driven RSR (see [36] for implementation details). Notice that in contrast to existing load-balancing methods, no global synchronization is required for load sharing between producer and consumer processors. Also, the consumer performs most of the computation required for load sharing decision. Since underloaded processor (consumers) are anyway idle, we can use them for the extra work required for solving the load-balancing problem. Also, notice that we use interrupt-driven remote service requests so that the consumer can get data and schedule as soon as possible the remote threads that have migrated to its context. Existing load-balancing methods are based on global synchronization barriers that have to be reached by all processors. In the case of non-threaded incremental methods, this implies that some processors have to wait, since the processors' loads are balanced gradually.

5 Analysis

In this section we compare the multithreaded approach with traditional incremental methods only, since direct methods are very expensive to be used for the load balancing of parallel adaptive computations. Consider the computation graph of Figure 5 and let T_{st} be the total execution time required by the PDE solver—whose computation is balanced by an incremental algorithm—that performs N iterations until the next mesh refinement occurs. An incremental method will balance the computation in K iterations. For each iteration i, with $1 < i < K$, let W^i_{max} denote the maximum workload over all processors. Let T_{slb} be the summation of time needed for the decision making, communication, and packing data to be migrated from an overloaded processor to an underloaded one. Once the processors decide on the data to be migrated, they send/receive the additional data, we denote this time as T_{migr}.

Taking into account that the slowest (most overloaded) processor dominates the execution time at each iteration, we can compute the total execution time between two mesh refinements by:

$$T_{st} = \sum_{i=1}^{K} W^i_{max} + (N-K)W^K_{max} + K \cdot (T_{slb} + T_{migr}) \tag{2}$$

Let $W^i_{max} = W^K_{max} + \Delta_i$ where Δ_i depends on the application and effectiveness of the incremental algorithm and let $C_1 = \sum_{i=1}^{K} \Delta_i$, where Δ_i, and subsequently of C_1, are relatively large constants.[7] Then (2) can be written as :

$$T_{st} = K \cdot W_{max}^K + C_1 + N \cdot W_{max}^K - K \cdot W_{max}^K + K \cdot (T_{slb} + T_{migr}) \Rightarrow$$

$$T_{st} = N \cdot W_{max}^K + K \cdot (T_{slb} + T_{migr}) + C_1 \tag{3}$$

Now, consider again the same computation (Figure 5) as above and let T_{mt} be the total execution time required by the PDE solver—whose computation is load balanced by a multithreaded load-sharing algorithm—that performs N iterations until the next mesh refinement occurs. Then T_{mt} is equal to :

$$T_{mt} = \sum_{i=1}^{N} (W_{avr}^t + N_t \cdot T_{ctxt}) + L \cdot (T_{mlb} + T_{put}) \Rightarrow$$

$$T_{mt} = N \cdot W_{avr}^t + N \cdot N_t \cdot T_{ctxt} + L \cdot (T_{mlb} + T_{put}) \tag{4}$$

where W_{avr}^t is the average load using threads, L is the number of times the slowest processor has to migrate data, and N_t is the maximum number of threads. Since we adjust (reduce) the size of the threads in order to get better load resolution (unit-wise, where a unit can be an element for FEM or a grid point for FD), we can say that after a small number of iterations, M, that depends on the size of the thread, we can have $W_{avr}^t \simeq W_{avr}$, which is very close to perfect load balance, unit-wise. For example, for threads with 100's of units (elements of grid points), by reducing the thread size, $|t|$, each time by half, we can achieve perfect balance in fewer than ten iterations. Also, let $\delta_i = W_{avr}^t - W_{avr}$ for $i < M$ then $C_0 = \sum_{i=1}^{M} \delta_i$ is a small constant. Then (4) can be written as :

$$T_{mt} = N \cdot W_{avr} + N \cdot N_t \cdot T_{ctxt} + L \cdot (T_{mlb} + T_{put}) + C_0 \tag{5}$$

From equations (3) and (5) and the fact that $W_{max}^K = W_{avr} + \alpha \cdot t_{unit}$ (usually $\alpha >> 1$ [1]), we see that a multithreaded load-sharing algorithm can be more efficient than any non-threaded incremental algorithm if:

$$N_t \leq \frac{N \cdot (W_{max}^K - W_{avr}) + K \cdot (T_{slb} + T_{migr}) + C_2 - L \cdot (T_{mlb} + T_{put})}{N \cdot T_{ctxt}} \tag{6}$$

For light-weight threads, T_{ctxt} as well as T_{mlb} and T_{put} is of an order of tens of micro-seconds [32]. Therefore $N \cdot T_{ctxt}$ and $L \cdot (T_{mlb} + T_{put})$ are very small numbers compared to $K \cdot (T_{slb} + T_{migr})$,

[7] For the examples that appear in [50], Δ_i varies from 10 to 50 units, and K varies from a few tens of iterations to a few 100's of iterations.

$N \cdot \alpha \cdot t_{unit}$, and $C_2 = (C_1 - C_0)$ [8] [1] that are in the order of seconds. Therefore, theoretically, a light-weight multithread system with a reasonably large number of threads, N_t, is capable of improving the performance of parallel adaptive PDE methods even further. A careful and very efficient implementation of such a model will be able to realize the above expectations. This is easier when high-order schemes are used or problems with many degrees of freedom per grid point since $W^K_{max} - W_{avr} >> 1$.

6 Discussion - Conclusions

Existing load-balancing algorithms require that all processors enter the balancing phase at the same time—guaranteed by global synchronization barriers. This requirement leads to: (1) the under-utilization of resources such as the CPU and network because many processors may wait until the overloaded or slow processors reach the global barrier, and (2) the intensification of problems like network contention due to exclusive use either of the network (data-migration) or of the CPU (decision-making). Concurrent execution of tasks required for load-balancing with tasks required for the actual computation is the key ingredient for developing efficient load-balancing algorithms. Here threads are used as a mechanism to explore concurrency at the processor level in order to tolerate memory latency and mask synchronization costs inherent in traditional load-balancing methods. It is important that threads tolerate memory and scheduling latencies without sacrificing program simplicity and portability.

Our preliminary experimental data using CHANT [37] and NEXUS [20] indicate that up to 64 threads, the overhead introduced due to context switch is very small compared to the execution time required for the computations required for the numerical solution of the PDE. Moreover, for applications with a large number of coarse-grain threads (up to a point) can minimize cache misses and improve performance (of course the same performance can be achieved by the re-structuring of non-threaded programs —an error prone process). Also, our preliminary data indicate that with up to 32 threads on SP2 one can overlap computation with communication and improve processor and network utilization.

Finally, further research is required in a number of directions: (1) evaluation of multithreaded approach with respect to numerical stability of PDE solvers, (2) evaluation and calibration of

[8] $C_0 \leq C_1$, since M usually is of the order of 10 ($log_2 100$), while K can be from a few tens to a few hundreds of iterations [1] and δ_i decreases each time by half, while Δ_i can be between a few units to tens of units.

the model using real applications -we invistigate the use of this approach for dealing with load balancing problems in numerical relativity codes [55], (3) generalization of the model to handle hp-refinement methods, (4) development of transformation mechanisms for converting off-the-shelf vast number FORTRAN libraries for PDEs from non-threaded to multithreaded programming paradigm, (5) creation of multithreaded runtime support systems for parallel compilers -we investigate this within the Bernoulli compiler [56], and problems solving environments -we investigate this within the Parallel ELLPACK environment [57].

Acknowledgements

I would like to thank Mike del Rosario, Matthew Haines, Thomas Fahringer, Piyush Mehrotra Geoffrey Fox, John Rice, David Keyes and Janusz Niemic for interesting and very helpful discussions on threads and dynamic load-balancing of adaptive computations.

References

[1] S. R. Wheat, K. D. Devine, and A. B. Maccabe, Experience with automatic, dynamic load balancing and Adaptive Finite Element Computation, *Proceedings of the 27th Hawaii International Conference on Systems Sciences*, January, 1997.

[2] Ravi Ponnusamy, Yuan-Shin Hwang, Joel Saltz, Alok Choudhary, Geoffrey Fox, Supporting Irregular Distributions in FORTRAN 90D/HPF Compilers University of Maryland, Department of Computer Science and UMIACS Technical Reports CS-TR-3268, UMIACS-TR-94-57, 1994.

[3] R. Das, Y. Hwang, M. Uysal, J. Saltz, A. Sussman, Applying the CHAOS/PARTI Library to Irregular Problems in Computational Chemistry and Computational Aerodynamics *Proceedings of the Scalable Parallel Libraries Conference, Mississippi State University, Starkville, MS pp: 45–46*, October 6–8, 1993.

[4] High Performance Fortran Forum, High Performance Fortran Language Specification, *Scientific Programming, Vol.2 No.1*, July 1993. Also available by anonymous ftp from ftp.npac.syr.edu.

[5] Fox, G., S. Hiranadani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. FortranD Language Specification. Technical Report SCCS-42c, Rice COMP TR90-141, 37p, 1991.

[6] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, "Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers: Design, Implementation, and Performance Results, *Proceedings of Supercomputing '93*, Partand, OR, November 1993;

[7] B. Chapman, P. Mehrotra, H. Zima, Vienna Fortran — A Fortran language extensions for distributed memory multiprocessors, *NASA Contractor Report* 187634, *ICASE Report* No. 91-72.

[8] J. Boykin, D. Kirschen, A. Langerman, and S. LoVoerso, Programming under Mach. Unix and Open Systems Series, Addison-Wesley, 1993, p500.

[9] H. Lockhart, Jr., OSF DCE, Guide to Developing Distributed Applications, J. Ranade Workstation Series, McGraw-Hill, Inc. 1994.

[10] B. Marsh, M. Scott, T. LeBlanc, and E. Markatos, First-class user-level threads. *Proceedings of the Thirteenth SOSP, Pacific Grove, CA*, October 1991

[11] F. Mueller, Implementing POSIX threads under UNIX: Description of work in progress, *Proceedings of the 2nd software engineering research forum, Melbourne, Florida*, Nov. 1992.

[12] F. Mueller, A library implementation of POSIX threads under UNIX, *1993 winter USENIX, San Diengo, CA*, January 25-29, 1993.

[13] D. Horst Simon. Partitioning of unstructured problems for parallel processing. Technical Report RNR-91-008, NASA Ames Research Center, Moffet Field, CA, 94035, 1990.

[14] E. Felten and D. McNamee, Improving the Performance of Message-Passing Applications by Multithreading, *Proceedings of the Scalable High Performance Computing Conference pp 84–89*, 1992.

[15] N.P. Chrisochoides, Multithread PDE solving systems for distributed address space parallel machines, *Proceedings of the IMACS World Congress on Computational and Applied Mathematics, pp 93-96, Atlanta*, July 11-15, 1994.

[16] D. Keppel, Tools and techniques for building fast portable threads packages, University of Washington, Department of Computer Science and Engineering, Technical Report UWCSE93-05-06, 1993.

[17] N.P. Chrisochoides and Mike del Rosario Evaluation of Remote Service Protocols for Distributed Multithreaded Runtime Support Systems *Poster paper presented in Frontiers'95.*

[18] N. P. Chrisochoides, M. Haines and P. Mehrotra An Evaluation of Distributed Multithreaded Primitives for PDE Computations, In preparation ICASE Report.

[19] MPI Forum, Message-Passing Interface Standard, April 15, 1994.

[20] I. Foster, Carl Kesselman, R. Olson, and Steve Tuecke, Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems, Argonne National Laboratory, ANL/MCS-TM-189, May 1994.

[21] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr, Implementing a Parallel C++ Runtime System for Scalable Parallel Systems, *Proceedings of the Supercomputing'93 Conference, Portland, Oregon, Nov. 15-19*, 1993.

[22] Fox, G., R. Williams and P. Messina *Parallel Computing Works!* Morgan Kaufmann Publishers, Inc. San Francisco, California, 1994.

[23] N. P. Chrisochoides, Elias Houstis and John Rice, Mapping Algorithms and Software Environment for Data Parallel PDE Iterative Solvers, *Special issue of the Journal of Parallel and Distributed Computing on Data-Parallel Algorithms and Programming*, Vol 21, No 1, pp 75–95, April, 1994.

[24] B. Hendrickson and R. Leland, The Chaco User's Guide, SAND93-2339

[25] B. Hendrickson and R. Leland, A Multilevel Algorithm for Partitioning Graphs, SAND93-1301

[26] Piyush Mehrotra and Matthew Haines, An overview of the OPUS language and runtime system, NASA CR-194921 ICASE Report No. 94-39 , Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23681-0001, May 1994.

[27] N. Sundaresan and L. Lee, An object-oriented thread model for parallel numerical applications. *Proceedings of the 2n Annual Object-Oriented Numerics Conference - OONSKI 94, Sunriver, Oregon, pp 291-308*, April 24-27 1994.

[28] Gannon, D. S. Yang, and P. Beckman. User Guide for a portable Parallel C++ Programming System pC++. Department of Computer Science and CICA, Indiana University, January 1994.

[29] Ralph M. Butler, and Ewing L. Lusk, *User's Guide to p4 Parallel Programming System* Oct 1992, Mathematics and Computer Science division, Argonne National Laboratory.

[30] A. Belguelin, J. Dongarra, A. Geist, R. Manchek, S. Otto, and J. Walpore, PVM: Experiences, current status and future direction. Supercomputing'93 Proceedings, pp 765–6.

[31] R. Konuru, J. Casa, R. Prouty, S. Otto and J. Walpole, A user-level process package for PVM, Proceedings of the Scalable High Performance Computing Conference, Knoxville Tennessee, May 23-25, pp 48–55, 1994.

[32] Thorsten von Eicken, Davin E. Culler, Seth Cooper Goldstein, and Klaus Erik Schauser, Active Messages: a mechanism for integrated communication and computation *Proceedings of the 19th International Symposium on Computer Architecture, ACM Press*, May 1992.

[33] E. Brewer and B. Kuszmaul, How to get good performance from CM-5 data network, *Proceedings of the International Parallel Processing Symposium*, 1994.

[34] Thorsten von Eicken, Personal Communication.

[35] N.P. Chrisochoides, An Efficient thread-to-thread communication for hybrid shared/distributed address space programming paradigms, In preparation, to be submitted to IEEE Trans. Parallel and Distributed Computing.

[36] Juan Miguel del Rosario and N.P. Chrisochoides An interrupt driven implementation of thread-to-thread communication for distributed address space machines, To be submitted to IEEE Trans. Parallel and Distributed Computing.

[37] Matthew Haines, David Cronk, and Piyush Mehrotra, On the design of Chant : A talking threads package, NASA CR-194903 ICASE Report No. 94-25, Institute for Computer

Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23681-0001, April 1994.

[38] M. Feeley, J. Chase, and E. Lazowska, User-level threads and interprocess communication, University of Washington, Department of Computer Science and Engineering, Technical Report 93-02-03, 1993.

[39] I. Kala, E. Arjomandi, G. Gao, and B. Farrell, FTL: A multithreaded environment for parallel computation, *Proceedings CASCON'94, pp 292–303*, 1994.

[40] Farhat, C. A simple and efficient automatic fem domain decomposer. *Computers and Structures*, 28:579–602, 1988.

[41] Nashat Mansour and Geoffrey Fox. Allocating Data to Multicomputer Nodes by Physical Optimization Algorithms for Loosely Synchronous Computations. *Concurrency: Practice and Experience*, Vol. 4, Number 7, pp 557-574, October 1992.

[42] M. Berger, S. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Computers*, C-36, 5 (May), pp. 570–580, 1987.

[43] R. D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency Practice and Experience*, 3(5), 457-481, 1991.

[44] C. Walshaw and M. Berzins, Dynamic load balancing for PDE solvers an adaptive unstructured meshes, University of Leeds, School of Computer Studies, Report 92.32, 1992.

[45] M. Jones and P. Plassman, Parallel algorithms for adaptive refinement and partitioning of unstructured meshes. *Proceedings of the Scalable High Performance Computing Conference, Knoxvile Tennessee, May 23-25, pp 478–485*, 1994.

[46] A. Vidwans and Y. Kallinderis, A parallel dynamic load balancing algorithm for 3-D adaptive unstructured grids, *In 11th AIAA Computational Fluid Dynamics Conference, AIAA-93-3313-CP, Orlando, FL*, July 1993.

[47] N. P. Chrisochoides, J. R. Rice. Partitioning heuristics for PDE computations based on parallel hardware and geometry characteristics. In *Advances in Computer Methods for Partial Differential Equations VII*, (R. Vichnevetsky. D. Knight and G. Richter, eds) IMACS, New Brunswick, NJ, pages 127-133, 1992.

[48] N. P. Chrisochoides, Nashat Mansour and Geoffrey Fox. A Comparison of data mapping algorithms for parallel iterative PDE solvers *Journal of Concurrency Practice and Experience*, 1995.

[49] H. L. deCougny, K. D. Devine, J.E. Flaherty, R. M. Loy, C. Ozturan, and M. S. Spephard, Load Balancing of Parallel Adaptive Solution of Partial Differential Equations, Rensselaer Polytechnic Institute, Department of Computer Science, Technical Report, TR94-8, 1994.

[50] C. Ozturan , H.L. deCougny , M.S. Shephard , J.E. Flaherty, Parallel Adaptive Mesh Refinement and Redistribution on Distributed Memory Computers, Rensselaer Polytechnic Institute, Department of Computer Science, Technical Report, TR93-26, 1993.

[51] N. P. Chrisochoides, G. C. Fox and J. F. Thompson, MENUS-PGG : Mapping Environment for Numerical Unstructured & Structured - Parallel Grid Generation, *Proceedings of the Seventh International Conference on Domain Decomposition Methods in Scientific and engineering computing,* 1995.

[52] J. Holm, A. Lain, and P. Banerjee, Compilation of scientific programs into multithreaded and message driven computation, *Proceedings of the Scalable High Performance Computing Conference, Knoxville Tennessee, May 23-25, pp 518–525*, 1994.

[53] I. Foster, Carl Kesselman, Steve Tuecke, Portable Mechanisms for Multithreaded Distributed Computations Argonne National Laboratory, MCS-P494-0195.

[54] Berger, J.B. and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations, *Journal of Computational Physics.*, 53: 484-512, 1984.

[55] Grand Challange Alliance: Binary Black Holes: http://www.npac.syr.edu/projects/bbh/more.html.

[56] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill, Automatic Parallelization of Sparse Conjugate Gradient Code, Department of Computer Science, Cornell University, 1995.

[57] E.N. Houstis, J.R. Rice, N.P. Chrisochoides, H.C. Karathanases, P.N. Papachiou, M.K. Samartzis, E.A. Vavalis, Ko Yang Wang and S. Weerawarana, Parallel ELLPACK: A numerical Simulation Programming Environment for Parallel MIMD Machines, *Proceedings of the* 4^{th} *International Conference on Supercomputing, ACM publications, pages 96-107*, 1990.