Electronic Transactions on Numerical Analysis.
Volume 21, pp. 66-80, 2005.

ISSN 1068-9613.

ETNA
Kent State University
etna@mcs.kent.edu

ADDRESSING THE STOCHASTIC NATURE OF SCIENTIFIC COMPUTATIONS VIA DYNAMIC LOOP SCHEDULING*

IOANA BANICESCU† AND RICOLINDO L. CARIÑO‡

Dedicated to Alan George on the occasion of his 60th birthday

Abstract. In general, scientific applications are large, computationally intensive, irregular and data-parallel. One of the main performance degradation factors when running scientific applications in parallel and distributed environments is load imbalance that may rise from a wide range of characteristics, due to problem, algorithmic, and systemic factors. Moreover, due to the stochastic behavior of many scientific applications, the load imbalance can unexpectedly occur at runtime, leading to difficulties in making decision about optimal partitioning, allocation and scheduling. In recent years, since parallel loops are prevalent and a major source of parallelism in scientific computations, many dynamic loop scheduling algorithms have been developed to improve performance via load balancing. These algorithms address the unpredictable behavior of computations, and many of these are based on probabilistic analyses. These have proven to be effective in improving performance of important applications used in scientific computing by addressing the entire spectrum of factors that lead to load imbalance. This paper proposes the use of a dynamic loop scheduling approach for repartitioning in scientific applications that contain computationally intensive parallel loops. A new partition is computed and data is redistributed during the execution of a parallel loop. This approach addresses load imbalance factors as they arise during a computational step, in contrast to the traditional load balancing approach of determining a new partition and migrating data after a computational step. Preliminary performance tests of an application that implements the proposed approach validate the feasibility and the effectiveness of the approach.

Key words. dynamic load balancing, data repartitioning, loop scheduling

AMS subject classifications. 68W10

1. Introduction.

Scientific applications are, in general, large, computationally intensive and data parallel. The efficient utilization of computing resources assigned to such applications requires two objectives to be achieved: the processors should be busy doing useful work, and the overhead of interprocessor communications should be kept small. For some applications, these objectives can be achieved by a simple, static partitioning of the work components among processors if the components have the same or predictable execution times, and the relative speeds of the processors are known. For many other applications, a dynamic partitioning of the work is necessary because the execution times of the work components inherently change as the application progresses, or the effective speeds of the processors vary due to unpredictable system induced effects. A number of algorithms have been proposed for this dynamic load balancing problem. See [20] for a review. Several implementations have been developed and incorporated in software libraries, such as PLUM [27], DRAMA [10] and Zoltan [9]. For example, the Zoltan library includes the following load balancing and parallel repartitioning methods: Recursive Coordinate Bisection [8], Recursive Inertial Bisection [31], Hilbert Space Filling Curve [35, 28], Refinement Tree Based Partitioning [26], ParMETIS [24], Jostle [33, 34], and Octree Partitioning [17].

In general, applications that require repartitioning may be described by a high level algorithm such as the one in Figure 1.1. Here, the load balancing approach could be best described

*Received August 19, 2004. Accepted for publication October 3, 2005. Recommended by S. Oliveira. This work was partially supported by the following National Science Foundation grants: CAREER #9984465, ITR/ACS #0081303, ITR/ACS #0085969, #0132618, and #0082979.

†Dept. of Computer Science and Engineering, Mississippi State University, PO Box 9637, Mississippi State MS 39762 (ioana@cse.msstate.edu). Also with the Center for Computational Sciences–ERC, Mississippi State University.

‡Center for Computational Sciences–ERC, Mississippi State University, PO Box 9627, Mississippi State MS 39762 (rlc@erc.msstate.edu).

ETNA
Kent State University
etna@mcs.kent.edu


```
Establish initial data partitioning;
do
   !   Computation step
   Perform some computations;
   if termination condition detected, then exit;
   !   Load balancing step
   Compute new partition;
   Perform data migration;
end do
```

FIG. 1.1. *A high level description of dynamic applications.*

as *iterative static repartitioning*. Typically, the load balancing is performed after one or more computation steps of the application are completed, or after an *a priori* established threshold of imbalance has been detected. Since load balancing is a separate step from the computations, processor work load differences during the computations are not addressed as they occur, contributing to application performance degradation.

A major source of parallelism in scientific applications is the *parallel loop,* a loop without dependencies among its iterations. A loop with certain kinds of dependencies among its iterations can also be converted into a parallel loop by eliminating the dependencies using methods such as loop unrolling. The iterations can be executed in any order, or even simultaneously, without affecting the correctness of the application. Typically, in Figure 1.1, the `computations` will involve a parallel loop over all application data items, like mesh points or elements, or particles. An application can have several parallel loops, each loop having unique characteristics. A partitioning of the application data which results in good load balancing for one loop may lead to severe load imbalance for other loops. Or, the application can have a single computationally intensive parallel loop whose characteristics change as the application progresses. Therefore, the partitioning has to be adapted during loop execution in order to achieve balanced processor work loads. This has motivated the development of dynamic loop scheduling techniques for load balancing parallel loops, such as factoring [23], fractiling [2], weighted factoring [22], adaptive weighted factoring [4, 5], including its variants [11], and adaptive factoring [6, 3]. Based on probabilistic analyses, these techniques schedule the execution of loop iterations in chunks with variable sizes. The chunk sizes are determined during the loop execution such that chunks have a high probability of being completed before the optimal time. Some of the applications in which the techniques have been successfully incorporated include Monte Carlo simulations [23], radar signal processing [22], N-body simulations [2, 4, 5, 3], computational fluid dynamics on unstructured grid [4, 5, 3], profiling of a quadrature routine [12], and wave packet simulations [15, 16]. Fractiling has also been incorporated into a parallel runtime system that combines data parallel load balancing with task parallel load balancing [30]. The techniques have been implemented in a load balancing tool for executing distributed parallel loops in MPI applications [13]. The techniques have also been incorporated in a load balancing library for executing parallel loops in applications using the DMCS/MOL system [1].

This paper proposes an approach to repartitioning in scientific applications with computationally intensive parallel loops characterized by stochastic and irregular behavior. Based on dynamic loop scheduling, this approach aims to address the entire spectrum of factors arising from application-specific, algorithmic, and systemic characteristics that induce load imbalance, leading to performance degradation. One previous indicator for the feasibility of this approach is the successful implementation of fractiling in N-body simulations [2]. Here,

ETNA
Kent State University
etna@mcs.kent.edu

the factoring technique determines how much work should be migrated between processors in order to achieve load balance while they cooperatively execute the computationally intensive loop over the leaves of a tree of particle data. The tiling technique identifies which work pieces will be migrated. Subsequent experiments have shown that adaptive techniques based on factoring further improved the performance of the simulation [4, 5, 3, 1]. This example highlights the feasibility of repartitioning during the `computations` phase in Figure 1.1.

The rest of the paper is organized as follows. Section 2 reviews non-adaptive and adaptive techniques that have been developed to dynamically execute parallel loop iterations. The adaptive techniques originally assume a centralized work queue of iterations from which idle processors obtain chunks to execute, an assumption that ideally maps to a shared memory architecture. Section 3 gives a high level description of a dynamic loop scheduling algorithm with a distributed work queue of iterations on a message passing environment. This algorithm is the basis of the proposed approach to dynamic repartitioning in applications with computationally intensive parallel loops. Section 4 describes the preliminary implementation of the proposed approach in the context of a real application — a framework for the simultaneous analysis of multiple datasets on general-purpose Linux clusters. Specifically for this paper, the framework is configured for datasets of gamma-ray burst (GRB) time profiles, to be fitted with nonlinear multivariate time series models. The framework can be configured for other datasets and their corresponding analysis routines as well. Section 5 presents performance results of the framework in the analysis of GRB datasets, highlighting the effectiveness of the repartitioning strategy. Section 6 summarizes previous efforts that have influenced the development of the loop scheduling approach to repartitioning and the design of the framework for simultaneous analysis of multiple datasets. Section 7 concludes with remarks on ongoing related efforts.

2. Overview of Dynamic Loop Scheduling. Assume that in a parallel application, a loop with N independent iterations is to be executed by P processors. The iterations are stored in a work queue from which idle processors obtain chunks. The sizes of the chunks are determined according to a scheduling technique that attempts to minimize the overall loop execution time. The technique is classified as *non-adaptive* when the chunk sizes are predictable from information that is available or assumed *before* loop execution, or *adaptive* when the chunk sizes depend on information available only *during* loop execution.

Non-adaptive loop scheduling techniques generate equal size chunks of iterates or predictable decreasing size chunks. Equal size chunks are generated by *static chunking* (STAT) where all the chunks are of size N/P, *self scheduling* (SS) where all the chunks are unit size, and *fixed size chunking* [25] (FSC) which requires the following parameters to be known *a priori*: μ, σ - the mean and the standard deviation of the iteration execution times, and h - the overhead of scheduling. Methods that generate predictably decreasing size chunks have also been implemented. The idea underlying these techniques is to initially allocate large chunks and later use the smaller chunks to smoothen the unevenness of the execution times of the initial larger chunks. The chunk sizes decrease geometrically in *Guided Self Scheduling* [29] (GSS), and linearly in *Trapezoid Self Scheduling* [32] (TSS). In *Factoring* [23] (FAC), iterations are scheduled in batches, where the size of a batch is a fixed ratio of the unscheduled iterations, and the batch is divided into P equal size chunks. The ratio is determined from a probabilistic analysis such that the resulting chunks have a high probability of finishing before the optimal time. For N-body simulations, the combination of factoring and *tiling*, a technique for organizing data to maintain locality and cache reuse, is known as *fractiling* [2] (FRAC). *Weighted factoring* [22] (WF) incorporates information on relative processor speeds in computing chunk sizes, where these speeds are assumed to be fixed throughout the execution of the loop.

ETNA
Kent State University
etna@mcs.kent.edu

A number of techniques that generate adaptive size chunks have evolved from factoring and weighted factoring. The requirement for the processor speeds in weighted factoring is relaxed in *adaptive weighted factoring* (AWF), a method developed to be utilized in time stepping applications [4, 5]. The processor weights are initially set to unity for the first time step. The execution times of chunks during a time step are recorded, and the data is used to adapt the processor weights at the end of the time step. The AWF, however, does not adapt to any load imbalance that occurs during the current step. A variant of the AWF [11] to address this shortcoming is to utilize the rates of execution of iterations in earlier chunks to adapt the processor weights for the succeeding chunks within the time step. The requirement in factoring that the mean and standard deviation of the iteration execution times are known *a priori* and are the same on all processors, is relaxed in *adaptive factoring* (AF) [6, 3]. These quantities are dynamically estimated during runtime from earlier chunks. The sizes of succeeding chunks are then computed using these estimates, and these estimates are refined by using more information from recently executed chunks. Although based on FAC, AF does not need to schedule iterations in batches; the size of an AF chunk depends only on the remaining iterations, and on the mean and standard deviation of the iteration execution times of the most recent chunk executed by each processor. AF incurs higher overhead due to the necessity of timing each iteration in order to estimate the mean and standard deviation of the execution times of iterates belonging to the new chunks.

Assuming that the scheduling operation has the same cost regardless of the size of the chunk, analytical expressions for the scheduling overhead can be derived for most of the techniques. The overhead depends on the number of chunks generated by the technique. Each chunk triggers some arithmetic operations to compute the chunk size, and bookkeeping operations for information about the chunk. The fixed sized techniques (STAT, SS, FSC) generate N/P, N, and N/(FSC size) chunks, respectively, while GSS, FAC and AWF generate $O(P \log(N/P))$ chunks. For the adaptive factoring (AF), the number chunks is unknown since it depends on measurements taken during runtime; however, experience indicates that it is no more than twice that of FAC. Since the number of chunks does not asymptotically exceed the problem size N, loop scheduling is theoretically considered scalable.

Loop scheduling techniques were originally developed assuming a centralized work queue of iterations where idle processors obtain chunks to execute. This assumption ideally maps to a shared memory architecture, requiring the processors to synchronize on a small set of variables. On a message passing environment, loop scheduling is typically implemented using a scheduler/worker algorithm. The work queue may be *centralized* in the scheduler, *replicated* in all processors, or *distributed* among the processors. Workers trigger scheduling events by sending requests to the scheduler. If the work queue is centralized, the scheduler responds with the data for a chunk of iterates. If the work queue is replicated, the scheduler sends only the chunk size. In the case of a distributed work queue, if the requesting worker is done with its local queue, the scheduler initiates the transfer of work queue elements from a "slow" worker to the requesting worker. The scheduling function may not require a dedicated processor; hence, the scheduler can also participate in executing iterations as another worker. The communication of control information and the migration of data in the message-passing implementation of loop scheduling contribute to the overall loop execution cost, which hopefully is offset by the performance gains due to load balancing.

Early implementations of FAC, WF, AWF and AF on a message passing architecture utilize a replicated work queue of iterations. This setup has the advantage of using only very short control messages to and from the scheduler. Data movement during loop computation is not necessary, since each processor has a local copy of the work queue; however, additional communications are needed in order to make the results of the computations by one processor

ETNA
Kent State University
etna@mcs.kent.edu

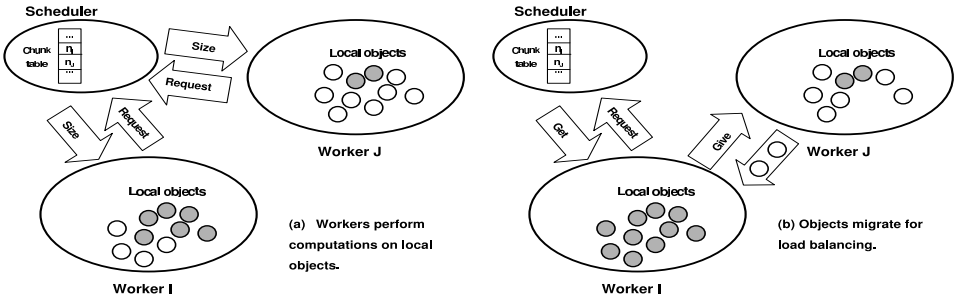

FIG. 3.1. *Dynamic loop scheduling approach to repartitioning.*

available on other processors. The disadvantages of this setup include the possibility of the scheduler becoming a bottleneck when the number of processors becomes large, and more significantly, the problem size must be limited such that the work queue will fit into the memory available on any of the processors. In contrast, a *distributed* work queue is utilized in the implementations of FRAC for the N-body simulations [2], where only a subset of the application data is stored in a processor. This setup accommodates larger problem sizes; however, it incurs higher communication overhead due to the need to redistribute data for load balancing.

3. Loop Scheduling Driven Repartitioning. For scientific applications that adhere to the abstraction provided by Figure 1.1, the computations and the load balancing can be performed in a single step if the computations involve one or more CPU intensive parallel loops. This is typical in many data parallel applications. Employing dynamic scheduling to execute a parallel loop causes data to be redistributed in order to balance processor loads. The repartitioning is *active* since it addresses load imbalance detected *during* the computations. In contrast, the repartitioning in Figure 1.1 is *reactive*, occurring only *after* computations that are imbalanced have already contributed to performance degradation. In terms of overheads, the dynamic approach incurs scheduling and data migration costs, while the reactive approach incurs penalty for load imbalance, the cost of computing a new partition, and cost of data migration.

Figures 3.1(a)–(b) illustrate the proposed dynamic loop scheduling approach to repartitioning in applications that contain computationally intensive parallel loops. The application objects are assumed to be distributed among the workers before loop execution. The scheduling function is the responsibility of the scheduler, which may or may not participate in the computations as another worker. As a simplification, the scheduler is assumed to be dedicated. The scheduler maintains a `chunk table` to keep track of which objects are stored by each worker.

Figure 3.1(a) illustrates the phase during loop scheduling when workers perform computations on local objects. A worker sends a `Request` message to the scheduler to request a chunk size. When received, this message triggers a scheduling event; the scheduler determines a chunk size according to the loop scheduling technique and responds with a `Size` message containing this size to the worker. The `Request` message contains the performance data of the worker based on the computation times of objects from previous chunks. These timings provide the scheduler with a continuously updated global view of worker performance during loop execution. These timings are used by the scheduler to determine subsequent chunk sizes according to the loop scheduling technique. During this phase, very small control messages are communicated, and the computations on objects may be interleaved with

ETNA
Kent State University
etna@mcs.kent.edu

the `Request` message to reduce waiting time by workers for chunk sizes. With interleaving, a worker sends the request before executing the last few iterations of the current chunk; hopefully, the next chunk size would have arrived from the scheduler before the computation for the last object of the current chunk is finished. To establish the number of iterations left in the current chunk before sending the request for a new chunk size, precise measurements of send/receive message latencies would have to be conducted.

Figure 3.1(b) describes the phase during loop scheduling when actual repartitioning takes place. This phase commences when a worker, say Worker I, has finished the computations on its local objects and sends a `Request` message. The scheduler recognizes the situation from the chunk table; it determines the next chunk size and the slowest worker, say Worker J, and sends these information to Worker I through a `Get` message. Upon receipt of this message, Worker I prepares a receive buffer and sends a `Give` message containing the size to Worker J. Worker J then sends a chunk of objects to Worker I. In computing the size, the scheduler may consider many factors such as the availability of space for the objects in Worker I, estimates of the cost of moving the objects from Worker J to Worker I, and the penalty of load imbalance if the objects were to remain in Worker J. On a message passing system that supports one sided communications, an alternative strategy is for Worker I to perform a one sided get operation for a chunk of objects from Worker J.

4. Sample Application. As a preliminary investigation of its effectiveness, the proposed dynamic loop scheduling approach to repartitioning was implemented in a framework for the simultaneous statistical analysis of multiple datasets on a general-purpose cluster. A parameterized statistical model is to be fitted on multiple datasets. A "one processor per dataset" parallel strategy is not suitable due to wide differences in dataset analysis times, ranging from a few seconds to several hours, depending on the number of observations contained in a dataset. A processor assigned to a large dataset will finish long after those assigned to smaller datasets. An "all processors working on one dataset at a time" strategy precludes the exploitation of the large number of processors available on typical clusters because of the limited degree of concurrency in the analysis procedure. Thus, the framework is based on a "processor groups" strategy, where a large number of processors is organized into groups, each group responsible for a number of datasets. The load imbalance factors to be addressed by the framework arise from the differences in the sizes of datasets, differences in the computational powers of the processor groups, and the unpredictable network latencies or operating system interferences inherent in a cluster environment.

4.1. Processor Groups Strategy. Figure 4.1 illustrates the strategy for the analysis framework. The analysis job is submitted to a cluster, and the cluster scheduler commits the number of processors requested by the job. The framework designates one of the processors as a dedicated scheduler S, which is responsible for: (1) organizing the rest of the processors into groups of crew members C and appointing a foreman F in each group; (2) retrieving the datasets from disk and distributing these to the groups; and (3) scheduling the analysis of the datasets by the groups. The scheduling proceeds as outlined in Figure 3.1, where a group is considered collectively as a single worker, and a dataset corresponds to an object.

The cluster is usually organized into racks that are connected by a cluster switch, each rack consisting of a number of nodes connected by a rack switch, and each node containing one or more processors. Obviously, the communications between processors assigned to a job will be more efficient if the processors reside in a single rack instead of being spread across several racks. A message between two processors p1 and p2 located in the same rack requires at most two hops (p1→ rack switch→p2), as opposed to four hops for a message between two processors p3 and p4 located on different racks (p3→rack switch→cluster switch→rack

ETNA
Kent State University
etna@mcs.kent.edu

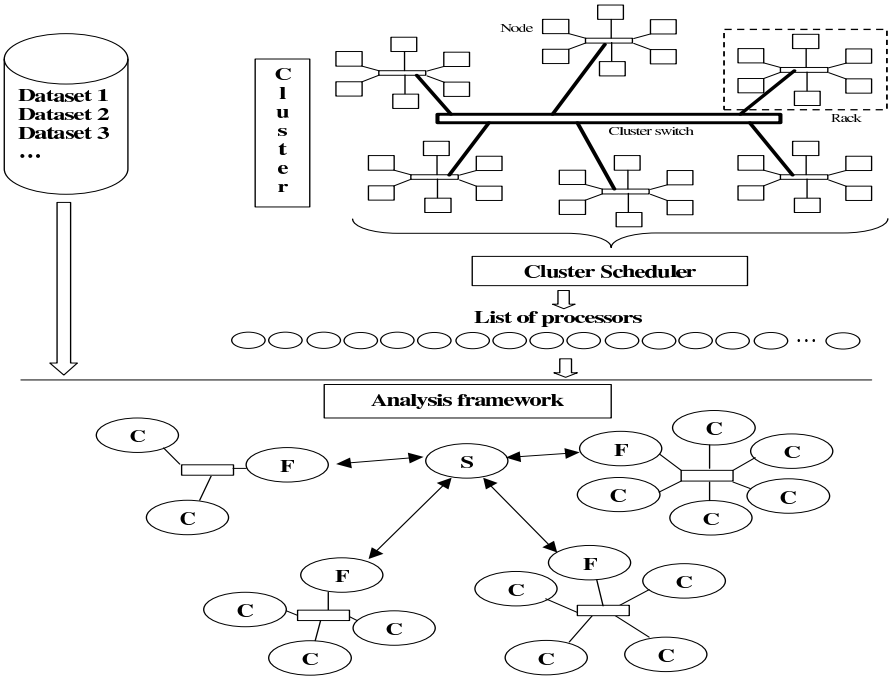

FIG. 4.1. *A processor groups strategy for the simultaneous analysis of multiple datasets on a cluster: C=crew, F=group foreman (also a crew), S=Scheduler.*

switch→p4). Typically, the job scheduler for the cluster attempts to assign nodes from a single rack to a job for efficient communications. Even with excellent job scheduling algorithms, fragmentation of processors across a number of racks occurs with a high probability, especially for jobs that request large numbers of processors. This fragmentation is conveniently exploited by the framework to match the degree of concurrency in the analysis procedure with the appropriate number of processors. The scheduler S forms the processors residing in a rack into a processor group acting as single worker, to carry out the analysis procedure on one dataset at a time. A very large number of processors in a rack (relative to the degree of concurrency in the analysis procedure) can formed into two or more workers. However, if there are racks that contribute tiny numbers of processors, then the processors are combined together to form a single more powerful worker in order to avoid the possibility of "tiny" workers being assigned very large datasets. This manner of organizing processors by racks enables two levels of concurrency: *1)* the simultaneous processing of datasets, and *2)* the parallel execution of the analysis procedure for a single dataset. The organization also exploits the efficiency of communications among processors residing in a single rack. Load balancing can be employed on both concurrency levels, that is, among the processor groups and within a group.

The typical naming convention for the nodes in Linux clusters make it easy to determine which processors belong to the same physical rack. The processors of a cluster are usually named `<name>-<rn>-<nn>`, where `<name>` is the cluster name, `<rn>` is the rack number and `<nn>` is the node number. Thus, nodes with the same `<rn>` belong to the same rack. In MPI, the routine MPI_GETHOSTNAME() may be invoked with appropriate arguments to determine the full name of the node containing the processor executing the routine.

ETNA
Kent State University
etna@mcs.kent.edu

4.2. Initial Distribution of Datasets. After the processors assigned to the analysis job are identified and the processor groupings are established, the datasets are retrieved from disk for distribution to the processor groups. Keeping the datasets in memory offers some advantages over "on demand" retrieval of datasets from disk. If the datasets are not massive, moving a dataset from one processor to another within the cluster is at least an order of magnitude faster than retrieving the dataset from a filesystem outside the cluster.

Given only the number of observations in each dataset and the sizes of the processor groups, the framework uses the following heuristic to initially distribute the datasets among the groups. Let D denote the number of datasets, G the number of processor groups, n_i the number of observations in dataset i, and s_j the size of group j. Therefore, the total number of observations is $W = \sum_{i=1}^{D} n_i$ and the number of processors in the groups is $P = \sum_{j=1}^{G} s_j$. Then, the datasets are distributed such that group j has a total of approximately $s_j \times W/P$ observations. Thus, the number of observations in a group is proportional to the number of processors in the group. These observations would come from a number of datasets selected to avoid the situation where all the big datasets are lumped together into one processor group. The distribution procedure is as follows. The datasets are first sorted according to decreasing n_i. Then, for each dataset in sorted order, the group which is farthest from its quota of observations is identified and the processor with the minimum number of observations in that group will store the dataset. This ensures that the big datasets are effectively scattered among the groups, and that the processors in a group store comparable numbers of observations.

4.3. Redistribution of Datasets. The proportionality of the total number of observations stored by a group to the number of processors of the group is not a guarantee for good load balance among groups. This is because the number of observations in a group may not be an appropriate measure of the computational load of the datasets of that group. The dynamic nature of a cluster environment also induces other types of load imbalance that must be addressed during the actual analysis of the datasets, necessitating their redistribution, as outlined in Figure 3.1(b).

In conventional loop scheduling, a parallel loop with N iterations is to be executed on P processors. Chunks of iterations are assigned to processors with the objective of minimizing the loop completion time. The sizes of chunks are determined according to a loop scheduling technique. Mapped to the present context of simultaneous analysis of multiple datasets, the N loop iterations correspond to the W observations, and the P processors correspond to the G groups, a group being a single worker. A chunk of iterations is essentially a fraction f of the total N iterations; this chunk corresponds to a *collection* of datasets whose cumulative size is approximately $f \times W$ observations. Using these correspondences, the dynamic loop scheduling techniques are applicable in the present context, with the possible exception of techniques like adaptive factoring (AF) which require measurement of individual iterate execution times. The correspondence between a single iteration and a single observation point may not be valid because the execution of an iteration can timed, while observations are not analyzed individually, but only collectively in a dataset.

The analysis of a chunk of datasets by a processor group proceeds as follows. The group foreman F receives information about the chunk from the scheduler S and broadcasts this to the crew members C. If the scheduler S sent a `Size` message, then the processors in the group examine the list of datasets that came with the information in lock step, the owner of the dataset under examination broadcasts it to the rest of the group, and the group collectively invokes the analysis routine on the dataset. Otherwise, if the scheduler S sent a `Get` message, then the foreman will send `Give` messages to the owners of the datasets, the crew members C will post receives for the incoming datasets, the group waits until all the datasets have been received, and then the group proceeds as if the scheduler had sent a `Size` message. The

ETNA
Kent State University
etna@mcs.kent.edu

broadcast of datasets within a group will be very efficient if the group resides in the same physical rack.

4.4. Performance Analysis. The high level control flow for the framework is given by the following pseudocode:

```
L1. Organize processors into groups;
L2. Retrieve datasets and distribute to processor groups;
L3. For parameter in {pval_1, pval_2, ..., pval_N} do
L4.    For all datasets in parallel do
L5.      Analyze dataset using current value of parameter;
L6.    end parallel do
L7. end do.
```

The execution time of the procedure to analyze a dataset typically changes with the value of `parameter`, which can be a simple value or a vector of values. Due to load imbalance factors, redistribution of datasets among groups is expected to occur during the execution of the parallel loop.

Of particular interest is the performance of the framework in executing the parallel loop in lines `L4`–`L6`. To aid in performance analysis, the framework records the following information for each value of `parameter`:

- I1 - the number of groups, their sizes, and names of processors;
- I2 - the initial and final distribution of datasets;
- I3 - the total time spent by each processor in the analysis routine only; and
- I4 - the completion time of each processor for the parallel loop, i.e., the elapsed time for lines `L4`–`L6`.

Note that I1 does not change across `parameter` values, and that in I2, the final distribution of datasets for one `parameter` value will be the same as the initial distribution of datasets for the next `parameter` value.

For a given `parameter` value, a basic performance metric for the parallel loop is the parallel time T_P, which is the *maximum* of the I4 values. The cost metric is $P \times T_P$. If the processors are homogeneous and equally loaded, then an estimate of the sequential loop time T_1 is given by the *sum* of the I3 values, i.e., the total time in the analysis routine only. If this estimate for T_1 is accurate, then the speedup and efficiency metrics are T_1/T_P and $T_1/(P \times T_P)$, respectively.

It is expected that the bulk of the cost of dataset redistribution will be incurred by the first iterate of loop `L3` (i.e., for the first `parameter` value). The reason for this is that no work load information is available to guide the initial distribution of datasets. Therefore, the initial distribution is very likely to induce load imbalance, which will be corrected at some cost during the analysis with the first `parameter` value. If a change in the `parameter` value affects the analysis of all datasets in the same manner (for example, the analysis time is increased or decreased by the same percentage), then the first redistribution cost is amortized during the analysis with the remaining `parameter` values, when relatively fewer datasets have to be moved to balance group work loads, because the bulk of the transfers happened with the first `parameter` value.

5. Experimental Results. For this paper, the framework described above was configured for the analysis of gamma-ray burst (GRB) datasets using vector functional coefficient autoregressive (VFCAR) time series models. GRBs are cosmic explosions that occurred in distant galaxies and are thought to be excellent "beacons" for tracing the structure of the early universe. Scientific satellites have been recording GRB time profiles for a number of years [21, 7], and the analysis of the collected datasets is currently an "open" problem in

ETNA
Kent State University
etna@mcs.kent.edu

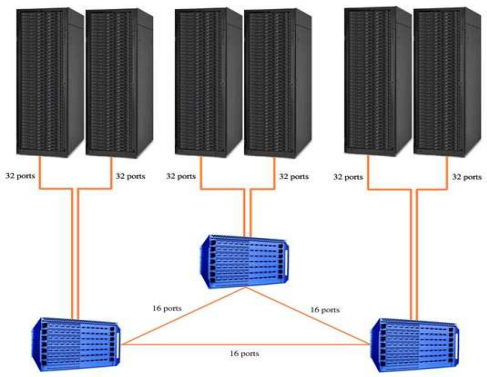

FIG. 5.1. *Maverick cluster diagram. Each rack contains 32 nodes, and each node contains dual 3.06GHz Xeon processors and 2.5GB RAM. Each node is connected via 10Gb/s InfiniBand to one of three switches, and a switch is directly connected to the other two. There is a supplementary 100Mb/s Ethernet network used for PXE, NFS, NIS, etc. Installed software include RedHat Linux and PBS/Pro. According to the Top 500 Supercomputer Sites list published in June 2004, Maverick was the 158th fastest computer in the world. (Source:* `http://www.erc.msstate.edu/computing/maverick/`*)*

the astrophysics community. The small-scale analysis mentioned by this paper is a preliminary investigation of the usefulness of VFCAR models [18, 19] in describing features of GRBs. More thorough investigations are planned for the near future, and the astrophysics and statistics findings will be reported elsewhere. The parallel performance of the framework configured for a small-scale analysis on a general-purpose cluster is presented in the remainder of this section as initial verification of the effectiveness of the proposed data redistribution approach.

An experiment was conducted involving the analysis of 555 GRB datasets the sizes of which ranged from 46 to 9037 observations. The Maverick cluster of the Mississippi State University Engineering Research Center (see Figure 5.1 for a diagram) was used as the computational platform. The analysis called for fitting the datasets under four scenarios, specified by the `parameter` values 400, 600, 800, 1000 (see §4.4). Briefly, for this analysis, the parameter represents the number of replications in the statistical test for model misspecification. The general effect of an increase in the `parameter` value is to lengthen the analysis time for a dataset by a factor depending on the square of the number of observations in the dataset. The experiment was submitted as a single 64-processor job so that the same set of processors is used for the four analysis scenarios. The 64 processors were spread across racks 2, 4 and 5 of the cluster, with 4, 32 and 28 processors, respectively. The maximum group size was hard coded in the framework to be 16 processors, so the scheduler S formed four groups: two 16-processor groups in rack 4, one 15-processor group in rack 5 (the scheduler S resided here), and one 16-processor group split between racks 2 and 5. Load imbalance was expected to be induced by the high variability of dataset analysis times, the differences in processor group sizes and communication latencies, and the contention for network resources since other jobs were executing on the cluster side by side with the experiment.

The analysis was executed without data redistribution (STAT) and with data redistribution using the following loop scheduling techniques: modified fixed size chunking (mFSC), guided loop scheduling (GSS), and a variant of adaptive weighted factoring (AWFC). Recall that these techniques generate fixed size chunks, predictable decreasing size chunks, and adaptive size chunks of iterations, respectively. In mFSC, the chunk size is chosen such that the number of chunks generated is the same as the number generated by factoring (FAC); that

ETNA
Kent State University
etna@mcs.kent.edu

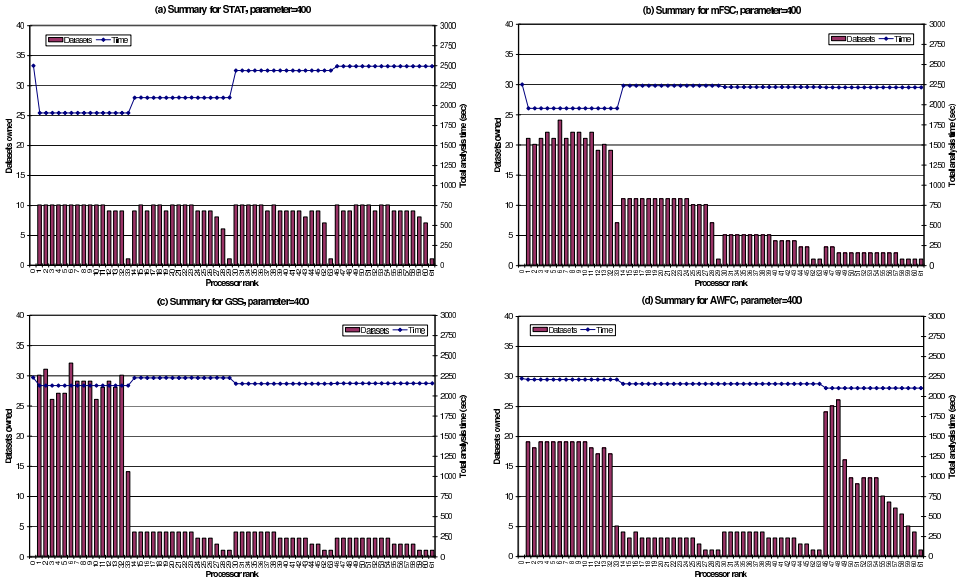

FIG. 5.2. *Number of datasets owned and cumulative analysis time for each processor for parameter=400: (a) without dataset redistribution (STAT), and (b)–(d) with dataset redistribution using mFSC, GSS and AWFC loop scheduling techniques.*

TABLE 5.1
Number of datasets transferred (Xfer) and estimated efficiency (Eff).

Dataset redistribution scheme	Parameter value							
	400		600		800		1000	
	Xfer	Eff	Xfer	Eff	Xfer	Eff	Xfer	Eff
STAT	0	0.88	0	0.87	0	0.86	0	0.87
mFSC	184	0.94	127	0.96	40	0.97	19	0.95
GSS	277	0.96	8	0.95	0	0.95	111	0.95
AWFC	186	0.95	112	0.96	20	0.98	82	0.98

is, mFSC and FAC both generate the same number of scheduling events.

Figures 5.2(a)–(d) summarize the number of datasets owned and the cumulative time spent in the analysis routine by each processor using the STAT, mFSC, GSS and AWFC dataset redistribution schemes for parameter=400. Note that in each chart, the processor groupings are ranks 1–33, ranks 14–29, ranks 30–63 and ranks 46–61, and that the analysis time plotted for processor rank 0 is the parallel time T_P.

Table 5.1 summarizes the number of datasets transferred during the analysis for each `parameter` value, as well as the estimated efficiency (see § 4.4).

The results summarized by Figure 5.2 and Table 5.1 support the following preliminary conclusions regarding the framework. The total number of observations stored in a processor group is not a good measure of the work load for the datasets owned by the group. Thus, the heuristic utilized in the initial distribution of datasets among the processor groups induces load imbalance; however, this imbalance is not severe since 86-88% estimated efficiency is achieved without dataset redistribution. This imbalance is effectively corrected, and the performance of the framework is improved when data redistribution is employed, using any of loop scheduling techniques tested. If the framework is used to carry out a sequence of

ETNA
Kent State University
etna@mcs.kent.edu

related analyses such as a parametric study, then the majority of the dataset transfers will likely occur early in the sequence, barring drastic changes in dataset execution times due to the parameters or due to irregularities in the execution environment. In general, the high cost of the initial dataset transfers is compensated by the higher performance levels achieved by the analyses with the later parameters in the sequence.

6. Related work. In the last years, performance gains of scientific applications using the factoring-based dynamic loop scheduling algorithms over other techniques for load balancing have been obtained, and reported in the literature. Experimental testing of running applications using these algorithms involved a wide range of problem and environmental characteristics, and a wide mixture of problems sizes and number of processors. The results of this extensive testing revealed the benefits of using the factoring-based dynamic loop scheduling algorithms for improving performance of scientific applications via load balancing. In general, performance gains of applications using these algorithms over using straightforward parallelization were often up to 50% and sometimes even 74% cost improvements, while over other competitive load balancing techniques reported in the literature were often in the range of 15% to 35% [1, 11, 12, 13, 14]. Earlier experience with factoring-based dynamic loop scheduling algorithms proved they are extremely robust and effective for a wide range of scientific applications, and that their use is essential in applications whose computations exhibit a stochastic and irregular behavior.

The successful integration of fractiling and adaptive weighted factoring in N-body simulations [2, 4], and similarly for adaptive weighted factoring and adaptive factoring in computational field simulation on unstructured grids [4, 3], highlight the competitiveness of the loop scheduling approach to load balancing in large, irregular and data parallel scientific applications. However, the tight coupling of these loop scheduling techniques into the application codes precluded the ease of reuse of the implementation of the techniques in other applications.

Loose coupling between the loop scheduling implementation and the application code is illustrated in the simulation of wave packets using the quantum trajectory method [15, 16]. The simulation has three separate computationally intensive parallel loops, with the data stored in simple arrays. A load balancing routine based on loop scheduling techniques is invoked to dynamically schedule the three loops, each invocation having a different computational routine as one of the arguments. The load balancing routine assumes a replicated work queue, implemented as an array in each processor. This setup is sufficient for the given application due to its small memory requirements. Results of computations are sent to the scheduler during load balancing, for broadcast to the rest of the processors after loop completion.

A loop scheduling routine for MPI applications has been proposed [13]. The routine is designed for parallel loops like `y(i) = f(x(i)), i=1,2,...,N`, where `x()` and `y()` are arrays of possibly complex types, and are partitioned among the processors. The loop scheduling routine takes as arguments the routine that encapsulates `f()` for a chunk of iterations, a pair of complementary routines for sending and receiving chunks of `x()`, a similar pair of routines for chunks of `y()`, and the partitioning of the `N` data items among the processors. The routine performs load balancing only: a chunk of `x()` is sent, for example, from Worker J to Worker I, and Worker I returns a chunk of `y()`, hence no repartitioning takes place. This loop scheduling routine was used as the base code for the development of the framework described in Section 4.

A well known disadvantage of the scheduler/worker parallel programming strategy, especially in message passing systems, is its limited practical scalability. When the number of processors is large, the scheduler becomes a communication bottleneck, resulting in in-

ETNA
Kent State University
etna@mcs.kent.edu

creased idle times for some workers. To address the bottleneck, a useful modification of the strategy is to utilize multiple foremen. A setup for dynamic loop scheduling using processor groups is described in [14]. The work queue of application objects is partitioned among the processor groups. Initially, each group performs loop scheduling on its own partition. However, the groups may be reorganized by the transfer of processors between groups in order to balance the load among groups. An opposite approach to load balancing was implemented for the framework described in Section 4 — the processor groups are fixed and objects are redistributed.

7. Summary and ongoing work. This paper proposes a dynamic loop scheduling approach for data redistribution in large data parallel scientific applications characterized by irregular and stochastic behavior. The traditional approach to load balancing, where a repartitioning step is made *after* one or more computational steps, suffers from performance degradation because load imbalance present during a computational step is not immediately addressed. The traditional approach assumes that the load imbalance will persist in the succeeding computational steps; hence a repartitioning step and data migration step are taken to preempt future imbalance. In addition to the the costs of computing a new partition and data migration, this traditional approach incurs the cost of load imbalance that is not immediately addressed. In the proposed approach, load balancing occurs *within* a computational step, specifically during the execution of a computationally intensive parallel loop via dynamic loop scheduling. A new partitioning is generated as a byproduct of load balancing, not by an additional step in the application. The costs include the loop scheduling costs, mostly from control messages, and data migration costs.

Previous work on the integration of loop scheduling techniques in large scientific applications such as N-body simulations and field simulations on unstructured grids provide evidence that substantial performance gains can be obtained by using the proposed approach for data redistribution. As further validation, this approach was incorporated into a framework for the simultaneous analysis of multiple datasets on clusters. The framework successfully integrates this approach with lessons learned from previous efforts to implement general-purpose loop scheduling routines and to exploit the availability of large numbers of processors on clusters through processor groups. Preliminary performance tests of the framework configured for a small-scale analysis of gamma-ray burst datasets indicate that the approach driving the framework is effective for achieving high parallel performance. Extensive analyses of gamma-ray burst datasets, as well as datasets from other disciplines, have been planned to be conducted using the framework. Efforts are also under way for the integration of the proposed repartitioning approach into other applications.

Acknowledgments. The authors thank Jane Harvill and John Patrick Lestrade for providing the test problem – the VFCAR analysis of GRB datasets.

REFERENCES

[1] M. BALASUBRAMANIAM, K. BARKER, I. BANICESCU, N. CHRISOCHOIDES, J. P. PABICO AND R. L. CARINO, *A Novel Dynamic Load Balancing Library for Cluster Computing*, in Proceedings of the 3rd International Symposium on Parallel and Distributed Computing, in association with the International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (ISPDC/HeteroPar'04), IEEE Computer Society Press, 2004, pp. 346-353.

[2] I. BANICESCU AND S. F. HUMMEL, *Balancing processor loads and exploiting data locality in N-body simulations*, in Proceedings of the 1995 ACM/IEEE Conference on Supercomputing, ACM Press, 1995, on CDROM.

[3] I. BANICESCU AND V. VELUSAMY, *Load balancing highly irregular computations with the adaptive factoring*, in Proceedings of the 16th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2002 - HCW), IEEE Computer Society Press, 2002, on CDROM.

ETNA
Kent State University
etna@mcs.kent.edu

[4] I. BANICESCU AND V. VELUSAMY, *Performance of scheduling scientific applications with adaptive weighted factoring*, in Proceedings of the 15th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2001 - HCW), IEEE Computer Society Press, 2001, on CDROM.

[5] I. BANICESCU, V. VELUSAMY AND J. DEVAPRASAD, *On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring*, Cluster Computing: The Journal of Networks, Software Tools and Applications, 6 (2003), pp. 215–226.

[6] I. BANICESCU AND Z. LIU, *Adaptive factoring: A dynamic scheduling method tuned to the rate of weight changes*, in Proceedings of the High Performance Computing Symposium (HPC) 2000, The Society for Modeling and Simulation International, 2000, pp. 122–129.

[7] *Burst And Transient Source Experiment (BATSE)*, http://www.batse.msfc.nasa.gov/batse.

[8] M. BERGER AND S. BOKHARI, *A partitioning strategy for nonuniform problems on multiprocessors*, IEEE Trans. Comput., C-36 (1987), pp. 570–580.

[9] E. BOMAN, K. DEVINE, R. HEAPHY, B. HENDRICKSON, W. F. MITCHELL, M. ST. JOHN AND C. VAUGHAN, *Zoltan: Data-Management Services for Parallel Applications*, http://www.cs.sandia.gov/Zoltan.

[10] C&C RESEARCH LABORATORIES, NEC EUROPE LTD., *DRAMA: Dynamic Load Balancing for Parallel Mesh-based Applications*, http://www.ccrl-nece.de/drama/.

[11] R. CARIÑO AND I. BANICESCU, *Dynamic scheduling parallel loops with variable iterate execution times*, in Proceedings of the 16th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2002 - PDSECA), IEEE Computer Society Press, 2002, on CDROM.

[12] R. L. CARIÑO AND I. BANICESCU, *Load balancing parallel loops on message-passing systems*, in Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems, S.G. Akl and T. Gonzales, eds., ACTA Press, 2002, pp. 362–367.

[13] R. L. CARIÑO AND I. BANICESCU, *A load balancing tool for distributed parallel loops*, in Proceedings of the International Workshop on Challenges of Large Applications in Distributed Environments (CLADE 2003), IEEE Computer Society Press, 2003, pp. 39–46.

[14] R. L. CARIÑO, I. BANICESCU, T. RAUBER AND G. RÜNGER, *Dynamic loop scheduling with processor groups*, in Proceedings of the 17th ISCA International Conference on Parallel and Distributed Computing Systems (PDCS 2004), 2004, pp. 78–84.

[15] R. L. CARIÑO, I. BANICESCU, R. K. VADAPALLI, C. A. WEATHERFORD AND J. ZHU, *Parallel adaptive quantum trajectory method for wavepacket simulations*, in Proceedings of the 17th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2003 - PDSECA), IEEE Computer Society Press, 2003, on CDROM.

[16] R. L. CARIÑO, I. BANICESCU, R. K. VADAPALLI, C. A. WEATHERFORD AND J. ZHU, *Message-passing parallel adaptive quantum trajectory method*, in High performance Scientific and Engineering Computing: Hardware/Software Support, L. T. Yang and Y. Pan, eds., Kluwer Academic Publishers, 2004, pp. 127–139.

[17] J. FLAHERTY, R. LOY, M. SHEPHARD, B. SZYMANSKI, J. TERESCO AND L. ZIANTZ, *Adaptive local refinement with octree load-balancing for the parallel solution of three-dimensional conservation laws*, J. Parallel Distrib. Comput., 47 (1998), pp. 139–152.

[18] J. HARVILL AND B. RAY, *Functional coefficient autoregressive models for vector time series*, Comput. Statist. Data Anal., 2005, to appear.

[19] J. HARVILL AND B. RAY, *A note on multi-step forecasting with functional coefficient autoregressive models*, International Journal of Forecasting, 2005, to appear.

[20] B. HENDRICKSON AND K. DEVINE, *Dynamic Load Balancing in Computational Mechanics*, Comput. Methods Appl. Mech. Engrg., 184 (2000), pp. 485–500.

[21] *High Energy Transient Explorer (HETE)*, http://space.mit.edu/HETE.

[22] S. F. HUMMEL, J. SCHMIDT, R. N. UMA AND J. WEIN, *Load-sharing in heterogeneous systems via weighted factoring*, in Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, 1996, pp. 318–328.

[23] S. F. HUMMEL, E. SCHOENBERG AND L. E. FLYNN, *Factoring: A method for scheduling parallel loops*, Comm. ACM, 35 (1992), pp. 90–101.

[24] G. KARYPIS AND V. KUMAR, *ParMETIS: Parallel graph partitioning and sparse matrix ordering library*, Tech. Rep. 97-060, Department of Computer Science, Univ. of Minnesota, 1997, http://www-users.cs.umn.edu/~karypis/metis/parmetis.

[25] C. P. KRUSKAL AND A. WEISS, *Allocating independent subtasks on parallel processors*, IEEE Trans. Software Engineering, 11 (1985), pp. 1001–1016.

[26] S. MITCHELL AND S. VAVASIS, *Quality mesh generation in three dimensions*, in Proceedings of the 8th ACM Symposium on Computational Geometry, ACM Press, New York, NY, 1992, pp. 212–221.

[27] L. OLIKER AND R. BISWAS, *PLUM: Parallel Load Balancing for Adaptive Unstructured Meshes*, Journal of Parallel and Distributed Computing, 52 (1998), pp. 150–177.

[28] J. PILKINGTON AND S. BADEN, *Partitioning with space-filling curves*, Tech. Rep. CS94-349, Dept. of Com-

ETNA
Kent State University
etna@mcs.kent.edu

puter Science and Engineering, Univ. of California, San Diego, CA, 1994.

[29] C. D. POLYCHRONOPOULOS AND D. J. KUCK, *Guided self-scheduling: A practical scheduling scheme for parallel supercomputers*, IEEE Trans. Comput., C-36 (1987), pp. 1425–1439.

[30] S. H. RUSS, I. BANICESCU, S. GHAFOOR, B. JANAPAREDDI, J. ROBINSON AND R. LU, *Hectiling: An Integration of Fine and Coarse-Grained Load-Balancing Strategies*, in Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing, IEEE Computer Society Press, 1998, pp. 106–113.

[31] V. E. TAYLOR AND B.NOUR-OMID,*A Study of the Factorization Fill-in for a Parallel Implementation of the Finite Element Method*, Intl. J. Numer. Methods Engrg., 37 (1994), pp. 3809–3823.

[32] T. H. TZEN AND L. M. NI, *Trapezoid self scheduling: A practical scheduling scheme for parallel compilers*, IEEE Trans. Parallel Distr. Sys., 4 (1993), pp. 87–98.

[33] C. WALSHAW, *JOSTLE mesh partitioning software*, http://www.gre.ac.uk/jostle/.

[34] C. WALSHAW, M. CROSS AND M. EVERETT, *Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes*, J. Par. Dist. Comp., 47 (1997), pp. 102–108.

[35] M. WARREN AND J. SALMON, *A Parallel Hashed Oct Tree N-body Algorithm*, Proceedings of the Supercomputing '93, Portland, USA, 1993, pp. 12–21.