Submitted to IRREGULAR'98, Berkeley, CA, Aug. 9-11, 1998.

Using Multithreading for the Automatic Load Balancing of 2D Adaptive Finite Element Meshes*

Gerd Heber Rupak Biswas† Parimala Thulasiraman Guang R. Gao

Abstract

In this paper, we present a multithreaded approach for the automatic load balancing of adaptive finite element (FE) meshes. The platform of our choice is the EARTH multithreaded system which offers sufficient capabilities to tackle this problem. We implement the adaption phase of FE applications on triangular meshes, and exploit the EARTH token mechanism to automatically balance the resulting irregular and highly nonuniform workload. We discuss the results of our experiments on EARTH-SP2, an implementation of EARTH on the IBM SP2, with different load balancing strategies that are built into the runtime system.

Key words: multithreading, unstructured mesh adaption, indexing scheme, dynamic load balancing.

1 Introduction

In this paper, we examine a multithreaded approach for the automatic load balancing of adaptive finite element (FE) meshes. During a FE adaption phase, the unstructured mesh is refined/coarsened (according to some application-specific criteria), and the workload may become (seriously) unbalanced on a multiprocessor system. This significantly affects the overall efficiency of parallel adaptive FE calculations. Some of the difficulties encountered when using the traditional approach to resolve the load imbalance problem can be summarized as follows:

- It is necessary to assemble global mesh information to make an accurate analysis of the general load situation – a step that often creates serious bottlenecks in practice.
- Significant effort is usually required to preserve data locality while making the load balancing decisions – an optimization which is computationally difficult.
- To make matters worse, the evolution of the computational load and data locality requirements are dynamic, irregular, and unpredictable at compile time.

Multithreaded architectures, such as the EARTH (*Efficient Architecture for Running THreads*) system [6], offer new capabilities and opportunities to tackle this problem. They strive to hide long latency operations by overlapping computation and communication with the help of *threads*. A thread is a (small) sequence of instructions. EARTH provides mechanisms to enable automatic load balancing for applications that do not allow a good static (compile time) task distribution. The programmer can simply encapsulate a function invocation as a *token*. The advantage of this token mechanism is that they can flexibly migrate over processors, by means of a load balancer.

Based on our experience with EARTH, we decided to purse a novel approach to implement the dynamic mesh adaption procedure proposed in [1]. Our decisions were based on the following observations:

- EARTH provides a runtime load balancing mechanism at a very fine-grain level, i.e., the token can be an ultra light-weight function invocation. Therefore, as long as the computation generates a large number of such tokens, it is likely that the EARTH runtime system can automatically distribute them

*For additional information contact: CAPSL, University of Delaware, 140 Evans Hall, Newark, DE 19716; {heber,thulasir,ggao}@capsl.udel.edu

†NASA Ames Research Center, Moffett Field, CA 94035; rbiswas@nas.nasa.gov

to the processors to keep them usefully busy. It is our hypothesis that we can thus eliminate an explicit repartitioning/remapping phase (in the first order) and the gathering of global mesh information.

- Locality is always an important issue for efficiency. However, since EARTH has the ability to tolerate latency using multithreading, its performance is less sensitive to the amount of non-local accesses by each processor. In other words, the optimality of locality or data distribution is less critical. Instead of optimally trying to repartition/remap the data to minimize communication, the user should take advantage of the EARTH token mechanism and structure the algorithm/code such that the input arguments to a token can be migrated easily with the token itself. The token should thus have good *mobility*.
- The difficulty of handling the irregular and unpredictable load evolution in traditional systems is partly due to the limitations of the data parallel programming model and the SPMD execution paradigm where the computation in each processor should be loosely synchronous at a coarse-grain level. The EARTH execution model is fundamentally asynchronous and does not rely on the SPMD model. Function level parallelism can be naturally expressed at a fine-grain level in the EARTH programming paradigm. The possible association of a parallel function invocation with a token at such a fine grain enables the runtime load balancing mechanism to work smoothly with the evolving computational load.

We implement the adaption phase of a simulated FE calculation and exploit the EARTH token mechanism to automatically balance the highly nonuniform workload. To achieve good token mobility, we apply a new indexing methodology for FE meshes that does not assume a particular architecture or programming environment. During mesh adaption, tokens that migrate to another processor to be executed leave the processed data in that processor instead of transferring the data back to the processor that generated the token. This is a novel approach in solving this problem: we avoid doing work twice and do a load-driven remapping without explicit user intervention. A major part of this paper is devoted to a discussion of the *quality* of such an approach which can be achieved using different load balancing strategies (built into the EARTH runtime system).

2 The EARTH System

2.1 The EARTH Platform(s)

EARTH (*Efficient Architecture for Running THreads*) [10] supports a multithreaded program execution model in which user code is divided into threads that are scheduled atomically using dataflow-like synchronization operations. These "EARTH operations" comprise a rich set of primitives, including remote loads and stores, synchronization operations, block data transfers, remote function calls, and dynamic load balancing. EARTH operations are initiated by the threads themselves. Once a thread is started, it runs to completion, and instructions within it are executed in sequential order.[1] Therefore, a conventional processor can execute a thread efficiently, even when the thread is purely sequential. For this reason, it is possible to obtain single-node performance close to that of a purely sequential implementation, as shown in our earlier work [12].

Conceptually, each EARTH node consists of an *Execution Unit* (EU), which executes the threads and a *Synchronization Unit* (SU), which performs the EARTH operations requested by the threads (cf. Fig. 1). The EARTH runtime system is currently available on the MANNA architecture [9], the IBM SP2 multiprocessor [3], and the Beowulf workstation clusters.

2.2 The EARTH Threaded-C Language

Currently, programs running on EARTH are written in Threaded-C, a C extension containing multithreading instructions. It is clean and powerful enough to be used as a user-level, explicitly parallel programming language. Figure 2(a) shows a simple example of a Threaded-C function that computes

[1]Instructions may be executed out of order, as on a superscalar machine, as long as the semantics of the sequential ordering are obeyed.

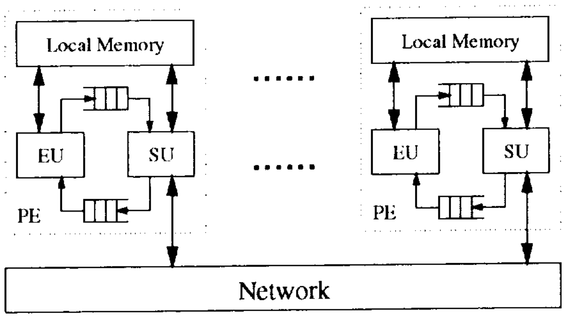

Figure 1: The EARTH architecture.

Fibonacci numbers using binary recursion. The keyword THREADED specifies that this function's frame should be allocated from the heap and that the function may contain multiple threads.[2] This function contains two threads. Each thread, except for the first, begins with a label THREAD_*n*; the first thread begins at the start of the function, and is automatically executed when the function is invoked using an INVOKE or TOKEN instruction. Each thread ends with either END_THREAD(), which simply executes a **fetch_next** instruction, or END_FUNCTION(), which executes an **end_function** instruction to deallocate the frame before executing a **fetch_next**.

```
THREADED fib(SLOT *done, long n, long *result)
{
  SLOTS SYNC_SLOTS[1];
  long sum1, sum2;
  if (n<2) {
      DATA_SYNC_L(1, result, done);
  } else {
     INIT_SYNC(0, 2, 2, 1);
     TOKEN(fib, SLOT_ADR(0), n-1, &sum1);
     TOKEN(fib, SLOT_ADR(0), n-2, &sum2);
     END_THREAD();
 THREAD_1:
     DATA_SYNC_L(sum1 + sum2, result, done);
  }
 END_FUNCTION();
}
```

(a) Threaded-C example of fib program

```
THREADED vadd(SLOT *done, int size, double *a, double *b, double *res)
{
   SLOTS SYNC_SLOTS[2];
   int i;  double la, lb;
   INIT_SYNC(0, 2, 2, 1);
   INIT_SYNC(1, size, size, 2);
   for  (i=0; i<size; i++)  {
      GET_SYNC_D(a++, &la, 0);     GET_SYNC_D(b++, &lb, 0);
      END_THREAD ();
 THREAD_1:
      DATA_SYNC_D(la+lb, res++, 1);
    }
   END_THREAD();
 THREAD_2:
   RSYNC(done);
   END_FUNCTION();
}
```

(b) Example of split-phase transactions in Threaded-C

Figure 2: Two examples of EARTH Threaded-C programs.

The *fib* function has a single sync slot. Since *fib* uses simple binary recursion, it needs to invoke itself twice (if n>2) and add the results of both children. Therefore, the INIT_SYNC command sets the sync count to 2. The other arguments to this command are the slot number (0 in this case), the reset count (2), and the thread number to execute when the sync count reaches 0 (THREAD_1). Note that the sync count and the reset count are the same. This is often the case, but not required. The sync count could be initialized to a lower value to indicate that some initial data is already available, or to a higher value to force the thread to wait for additional events to occur during initialization.

The function *fib* takes two arguments in addition to n. These are pointers to the address where the result should be sent (result), and to the sync slot that receives the sync signal (done). For coding efficiency, data and sync slot locations are represented in Threaded-C as single addresses rather than as <*fp*, *offset*> pairs. If n<2, the function sends a 1 to *result and signals the sync slot *done (atomically, using a DATA_SYNC).[3] Otherwise, it invokes two instantiations of itself, telling each to signal sync slot 0 (the macro SLOT_ADR(0) generates a pointer to the correct slot). The TOKEN command is used

[2]Non-threaded functions are specified as regular C functions.

[3]The suffix indicates the type of value being transferred, e.g., _L for long or _D for double.

to invoke the load-balancing mechanism. The initial thread then terminates, and THREAD_1 will not be executed until both functions return their values and signal slot 0. Other threads in the ready queue can be executed in the interim. When THREAD_1 is activated, the addition is performed, the data is sent to this function's caller, and the frame is deallocated. Note that thread boundaries can be located inside any compound statement, e.g., the if-statement. In this example, if n<2, the program jumps to the middle of THREAD_1 and executes END_FUNCTION().

Another example is shown in Fig. 2(b). This shows a function *vadd* that fetches the elements of vectors stored on a remote processor using GET_SYNC instructions. This code effectively performs split-phase transactions, because THREAD_1 will not start until both values have returned (assuming that slot 0 has sync and reset counts of 2). This code is purely sequential, but the time spent waiting for remote accesses can be used to execute another thread.

In addition, for those applications for which a good task distribution cannot be determined statically by the programmer and communicated to the compiler, EARTH provides an automatic load balancing mechanism. The programmer can simply encapsulate a function invocation as a token. A token is sent to the SU, which puts it on top of the local token queue. When there are no more threads in the ready queue, the SU removes a token from the top of the token queue and invokes the function as specified locally. This load balancing technique is derived from the method of token management used in the ADAM architecture [11]. Note that putting locally-generated tokens on top of the queue and then removing tokens from the top results in a *depth-first* traversal of the call-graph. This generally leads to better control of functional parallelism, i.e., it diminishes the likelihood of parallelism explosion that can exhaust the memory resources of a node.

When both the ready queue and the token queue are empty, the SU sends a message to a neighboring processor requesting a token, in effect performing *work stealing*. The neighboring processor, if it has tokens in its queue, extracts one from the bottom of its queue and sends it back to the requester. In this manner, a *breadth-first* traversal of the call-graph is implemented across processors, hopefully resulting in a better distribution of tasks. If the neighboring processor does not have any tokens to satisfy a request, the neighbor's neighbor is queried, and so on. Either a token will be found, or the request cannot be fulfilled. Idle processors periodically query their neighbors for work.

3 An Indexing Technique for FE Meshes

Dynamic mesh adaption is a common and powerful technique for efficiently solving FE problems. The adaption of a mesh is achieved by *coarsening* and/or *refining* some elements of the computational mesh. In 2D FE meshes, triangular elements are the most popular. Once a triangle is targeted for refinement, it is generally subdivided by bisecting its three edges to form four congruent triangles as shown in Fig. 3. This type of subdivision is called *isotropic* (or 1:4) and the resulting triangles are referred to as being *red*. A problem is that the refined mesh will be *nonconforming* unless all the triangles are isotropically subdivided. To get a consistent triangulation without global refinement, a second type of subdivision is allowed. For example, a triangle may be subdivided into two smaller triangles by bisecting only one edge as shown in Fig. 3. This type of subdivision is called *anisotropic* (or 1:2 in this case) and the resulting triangles are referred to as being *green*. The process of creating a consistent triangulation is defined as

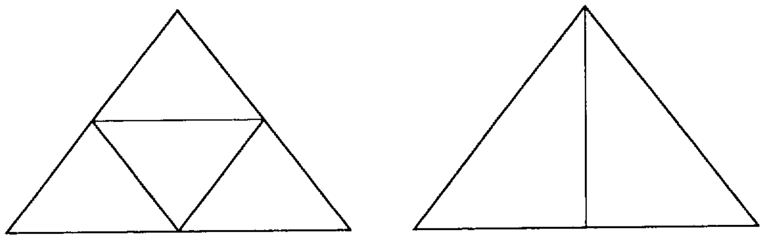

Figure 3: The left picture shows the isotropic subdivision of a triangle. The right one gives an example of anisotropic subdivision.

a *closure* operation. Note that several iterations may be necessary to achieve closure [1].

A couple of additional rules are applied, primarily to assure that the quality of the adapted mesh does not deteriorate drastically with repeated refinement:

1. All triangles with exactly two bisected edges have their third edge also bisected. Thus, such triangles are isotropically refined.
2. A green triangle cannot be further subdivided. Instead, the previous subdivision is discarded and isotropic subdivision is applied to the (red) ancestor triangle [2].

It is the task of an *index scheme* to properly name or label the various objects (vertices, edges, triangles) of a mesh. We prefer the term index scheme instead of *numbering* to stress that the use of natural numbers as indices is not sufficient to meet the naming requirements of the FE objects on parallel architectures.

We give here a brief description of our indexing technique for the sake of completeness; for a detailed discussion, refer to [7]. Note that our technique is intended to be used for hierarchical meshes. Our index scheme is a combination of *coarse* and *local* schemes. The coarse scheme labels the objects of the coarse mesh in such a way that the incidence relations can be easily derived from the labels (cf. Fig. 4). The

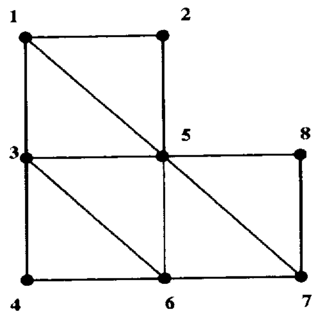

Figure 4: An L-shaped domain and its coarse triangulation.

vertices are enumerated starting from 1. Then the set of vertices for the coarse triangulation consists of the following numbers:

$$\text{vertices} = \{1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8\}.$$

The edges of the coarse triangulation are indexed by ordered pairs of integers that correspond to the endpoints of the edges. The ordering is chosen so that the first index is less than the second one. For the example in Fig. 4, the set of coarse edges consists of the following pairs:

$$\begin{aligned}\text{edges} = \{&(1,2),\ (1,3),\ (1,5),\ (2,5),\ (3,4),\ (3,5),\ (3,6),\\ &(4,6),\ (5,6),\ (5,7),\ (5,8),\ (6,7),\ (7,8)\}.\end{aligned}$$

The same principles are applied to index the coarse triangles. They are denoted by the triple consisting of their vertex numbers in ascending order. Thus, the set of coarse triangles reads:

$$\text{triangles} = \{(1,2,5),\ (1,3,5),\ (3,4,6),\ (3,5,6),\ (5,6,7),\ (5,7,8)\}.$$

Note that this index scheme can be applied to elements with curved boundaries as well.

The local scheme exploits the regularity (and the finiteness) of the refinement rules to produce names for the objects at subsequent refinement levels [7]. We use (scaled) natural coordinates as indices in the local scheme. Again, this is done in a way such that the incidence relations are *encoded* in the indices of the objects. For example, the set of vertices at level k in the local model is given by:

$$V_k = \{(a,b,c) \in \mathbb{N}^3 \mid a+b+c = 2^k\}.$$

We do not know which of these will actually be present; however, we already have names for them. Figure 5 shows the local indices for the vertices and the triangles on the first two refinement levels (for

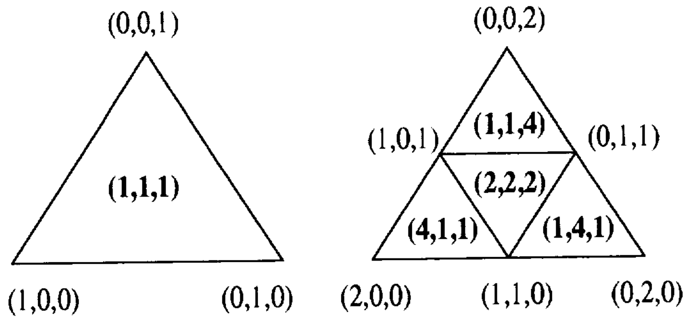

Figure 5: Examples of the local index scheme for triangular elements. The vertices and triangles are denoted by integer triples (triangles by bold face).

isotropic subdivision). The coarse and local schemes are combined by taking the union of the Cartesian products of the coarse mesh objects with their corresponding local schemes. Ambiguities are resolved by using a *normal* form of the index.

The key features of such a scheme are:

- Each object is assigned a ***global name*** that is independent of any architectural considerations or implementation choices.
- Combinatorial information is translated into simple arithmetic.
- It is *well-behaved* under (adaptive) refinement. No artificial synchronization/serialization is introduced.
- It can be extended (with appropriate modifications) to three dimensions [7].

4 Description of the Test Problems

4.1 Mesh, Partitioning, and Mapping

As the initial mesh for our experiments, we choose a rectangular mesh similar to, but much larger than, the one shown in Fig. 6. Mesh generation simplicity was an overriding concern. However, we never, neither in the algorithms nor in the implementation, exploit this simple connectivity. Recall that geometry information is irrelevant for our index scheme. Mesh adaption on even a simple case like this can fully exercise several aspects of the EARTH multithreaded system and its dynamic load balancing capabilities.

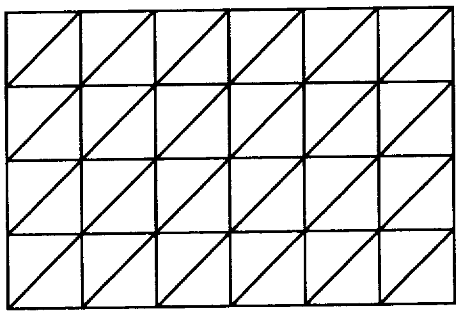

Figure 6: A small sample of our test mesh.

Fine-grain multithreading would allow us to have one thread per element. However, to control granularity and to prevent *fragmentation*[4] of the triangulation, we choose partitions as our smallest migratable unit. Reference [4] gives a good overview of common partitioning techniques and mapping strategies. Based on their observations, we decided to use a $P \times Q$ block partitioning.

[4]Currently, the EARTH runtime system does not support a restricted migration of tokens.

Initially, the entire mesh completely resides on node 0. Then a marking procedure (an *error estimator/indicator* in real applications) marks edges/elements for refinement or coarsening (on higher levels). Based on this marking, tokens – one for each partition, are generated that perform the actual refinement. This situation immediately serves as a ***stress*** test for the EARTH token mechanism to automatically balance the highly nonuniform workload. If a token migrates to and is executed on another processor, it transfers the partition and refinement information to the executing node. The actual refinement/coarsening is done there. At this point, it is therefore absolutely crucial to ***compress*** the ***metacontext*** as much as possible. Here the property of our index scheme to provide global indices becomes extremely useful: it is not necessary to transform names if we cross processor/memory boundaries. Furthermore, only the indices of the elements have to be transferred since the indices of all other objects (edges, vertices) can be easily recomputed [7]. (This can be further optimized since a parent element can, in conjunction with the refinement information, rapidly calculate the indices of its children.)

For mesh refinement, one can distinguish between two strategies. In the ***first*** case, a parent element is replaced by its children, i.e., no ***history*** information is preserved. In the ***second*** case, the refinement history is maintained. The advantage of the second strategy is that parent objects do not have to be reconstructed from scratch during mesh coarsening. We have implemented this second strategy [1].

When all the elements in a FE mesh are subdivided isotropically, we refer to it as ***uniform*** refinement. This procedure automatically leads to a conforming triangulation. In the case of ***nonuniform*** refinement, a (iterative) consistency procedure is required since adaptive coarsening/refinement generally leads to inconsistent triangulations. Moreover, to guarantee mesh quality using our adaption algorithm, the parents of green triangles are reinstated before further refinement [2].

In real applications, the mesh adaption is driven by an error indicator that is obtained either geometrically or from the numerical solution. The computational mesh is refined in regions where the indicator is high, and coarsened in regions where it is low. In our experiments, we completely ignore the physics and/or the geometry of the application, and use a random number generator to select elements for adaption. We can mimic the behavior of an error indicator by suitably adjusting the distribution pattern of the random numbers.

4.2 Load Balancing Strategies

For a detailed description of the load balancing strategies that have been implemented in the EARTH-SP[5] runtime system, we refer the reader to [3]. This is a comprehensive study that provides an excellent overview of the implemented strategies and their qualitative behavior under several stress tests.

Some of the questions we would like to answer for our particular simulation are:

- What portion of the total number of elements are finally located on nodes other than node 0? How do the numbers of nodes and tokens generated per node affect this behavior? Is the load acceptably balanced outside of node 0?
- How does the token generation process affect the runtime? Is there a measurable system overhead?
- Does an increased number of tokens decrease the *variance* of the load balancer (as one would expect)?
- How does the system behave if there are less tokens than processors?

5 Results

The data structure used to keep the indices of our FE mesh objects is a red-black tree [5], as it is used in the implementation of the set container in the C++ Standard Template Library. Reference [8] discusses some related issues. The code for the uniform refinement case consists of about 2000 lines of Threaded-C code. (This includes the code for the red-black tree.) In the nonuniform case, closure of the triangulation process is added, and the code is about 2700 lines.

The program contains certain optimizations for the case when a token is executed on the node it originates. Executing the program on one node implies that all tokens are both generated and executed on the same node. In that case, no remote operations or tree compression/expansion are necessary.

[5]This refers to the portable version of Threaded-C available for the IBM SP2 and SP3.

The initial mesh is the same for all the experiments and consists of 2048 vertices, 5953 edges, and 3906 triangular elements. At the beginning of each experiment, the entire mesh resides on node 0. It is the responsibility of the EARTH runtime system to scatter partitions across the available nodes.

Currently, there are eight different load balancers available with EARTH-SP, and we tested all of them. As a representative for the discussions in this section, we chose the DUAL load balancer. It is one of the simplest load balancers (a ***virtual ring*** [3]), and does not provide the best quantitative results for our experiments. However, its general behavior is comparable with the other load balancers in EARTH-SP. This indicates a certain ***robustness*** of the whole system. For the sake of simplicity and space restrictions, we analyze only the uniform refinement case. A representative nonuniform example will be discussed in the final presentation.

A total of four mesh refinement steps were performed in the uniform case. The final mesh contained 999,936 triangles since each triangle was refined isotropically. We measured the execution time upon completion of these four steps.

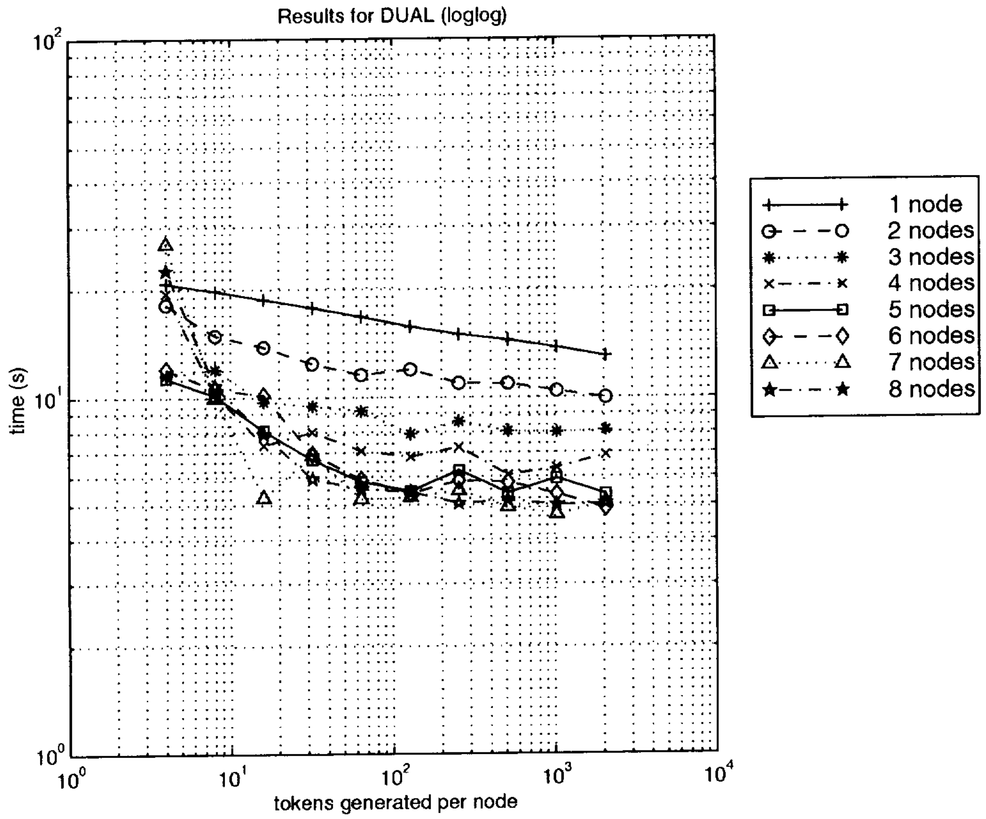

Figure 7: Execution time versus the number of tokens generated per node (in *loglog* representation).

Discussion of Figure 7: This figure shows the runtime versus the number of tokens generated per node. (*loglog*)

- On one node, the program runs faster as the number of tokens increase! Recall that the number of tokens corresponds to the number of partitions. Hence, for a fixed mesh size, more partitions imply less elements per partition. Since the elements in a partition are organized in a tree, it follows that the trees will be shallower. This, in turn, accelerates the tree operations. The curve for one node in the *loglog* representation confirms that the underlying data structure must be of logarithmic complexity.
- A reasonable speedup is obtained for up to five nodes. However, we must consider the additional quality information (discussed below) for our evaluation, since quality has rather strong implications

for other parts (solver/preconditioner) of a complete FE application. It might therefore be useful to invest some more resources that do not necessarily increase the speedup but improve quality.

- We observe a certain instability in the region where the number of tokens is not much greater than the number of processors.

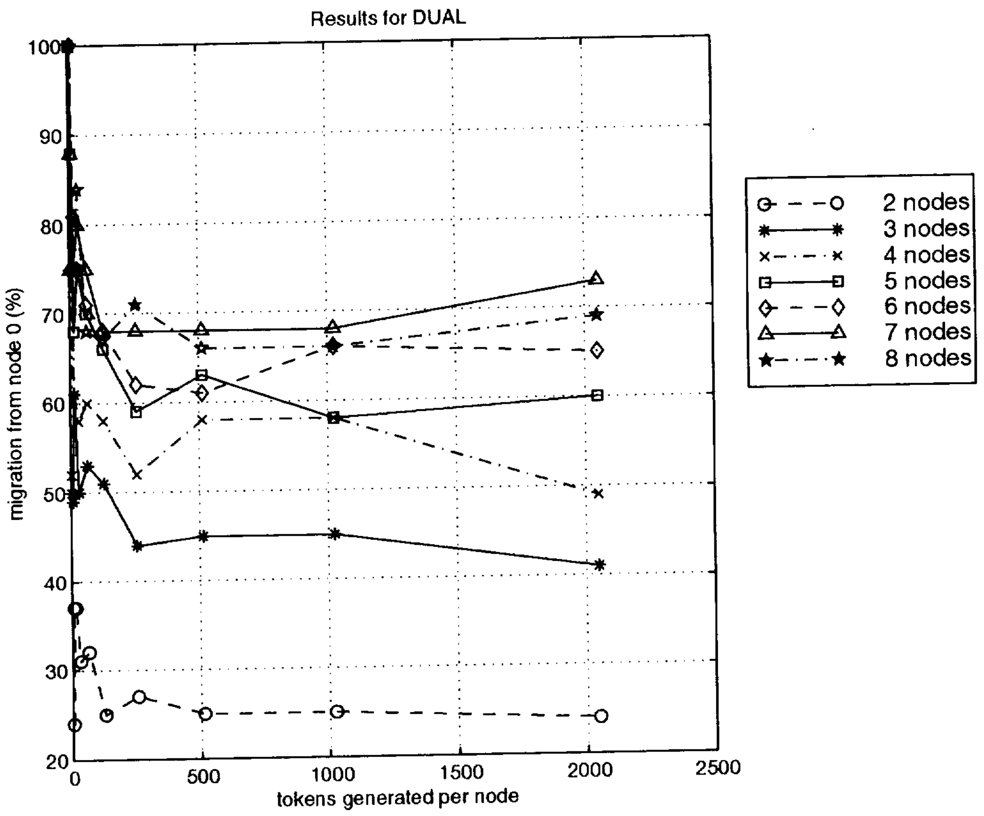

Figure 8: Total migration from node 0 versus the number of tokens generated per node.

Discussion of Figure 8: This figure shows the migration from node zero versus the number of tokens generated per node.

- We measure the *migration* as the percentage of all triangles that are, at the end, owned by nodes other than node 0. This measure is somewhat imprecise, since a partition may migrate back and forth.[6]
- The migration increases with the number of nodes, but might decrease with the number of tokens if not enough processors are available. We believe that this is because the DUAL load balancer is unable to handle the load within these resource limitations.
- It seems that there is a region of stabilization at about 70 percent of migration. Asymptotically, the percentage of migration from node 0 when using P nodes is $100 \times (P-1)/P$. From our experience, migration can be increased toward this asymptotic by using more processors or more tokens. Generating more tokens implies downsizing partitions, which in the extreme case, consist of only one triangle. However, we would then inevitably be faced with a fragmentation problem (and a total loss of locality).
- Once again, we observe a region of instability when the number of tokens is not much greater than the number of processors.

Discussion of Figures 9 and 10: These figures show the *total variance* and the *unbiased variance* outside of node 0 versus the number of tokens generated per node.

[6]Such *fluctuations* actually occur, especially if there are only a few processors and/or tokens.

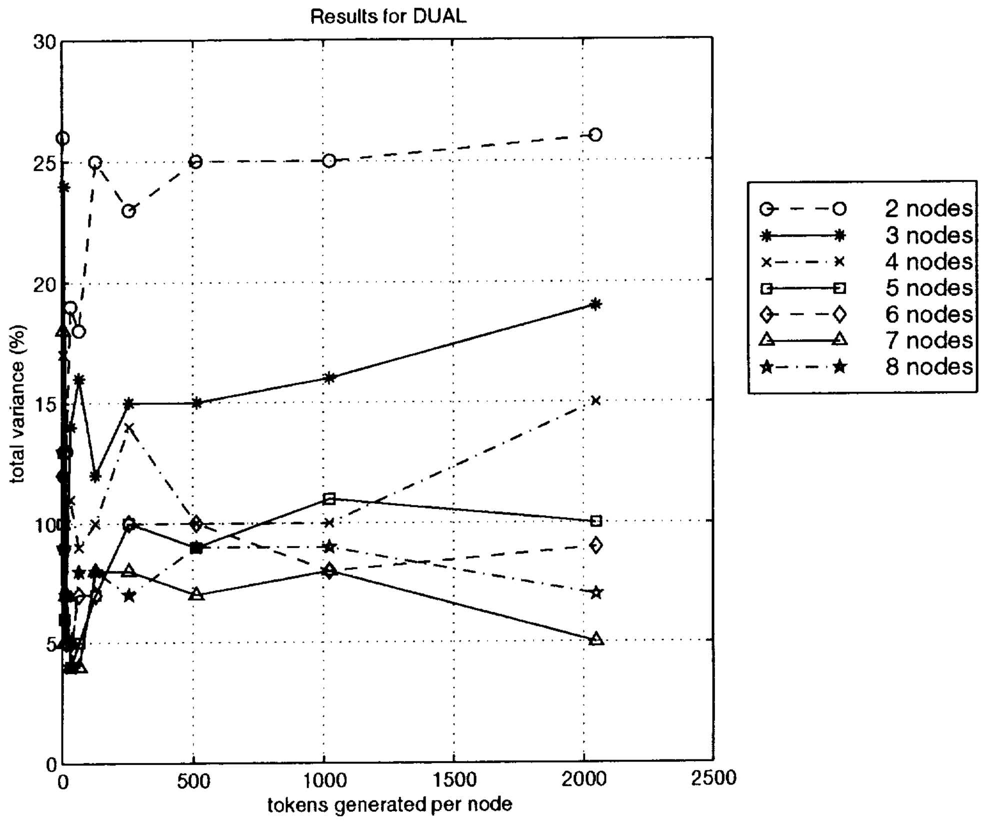

Figure 9: Total variance versus the number of tokens generated per node.

- We define total variance as the expected deviation from the mean value of the number of triangles per node measured as a percentage of the total number of triangles. Since this measure is biased towards node 0, we also compute the variance outside of node 0 to evaluate the uniformity of the migration (cf. Fig. 10).
- The variances oscillate rapidly if there are not enough tokens available. If not enough processors are available, increasing the number of tokens tends to overload the load balancer.
- A large number of tokens seems to guarantee the stability of, and possibly a slight improvement in, the variances.
- Perhaps the most important result is the quality of load balancing. Experimental values of the unbiased variance presented in Fig. 10 show that the load is extremely well balanced (between 3 and 6 percent). This is generally acceptable for actual adaptive FE calculations.

5.1 Summary of Results

We summarize our results by addressing each of the three observations made in Sec. 1:

- The experimental results confirm our hypothesis outlined in the introduction: although we did not have an explicit repartitioning/remapping phase, the processors appear to have a good and balanced utilization as long as there are enough tokens being processed in the system (cf. Figs. 8 and 9).
- The experiments also show that the natural indexing method used to enhance token mobility does appear to work; i.e., the remote communication due to token migration does not seem to have a major impact on the overall performance (see the region of good speedup and balanced workload implied in the curves in Fig. 7). Of course, the FE solution phase is usually more sensitive to data locality which is not included in the scope of this paper.

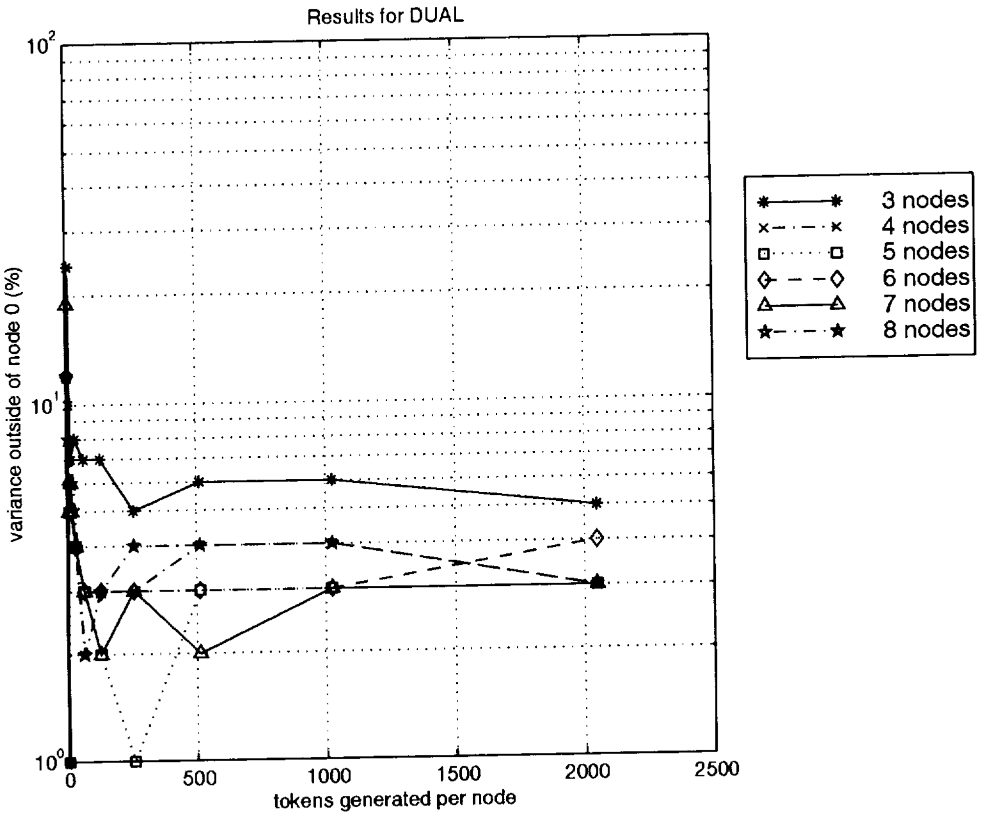

Figure 10: Unbiased variance outside of node 0 versus the number of tokens generated per node (in *semilog* representation). Note that results are not shown for two nodes since node 0 is excluded when computing this variance.

- Finally, the experimental results have shown that the load balancing works smoothly as long as there are enough computational resources available, even when the workload evolution is irregular. This can be observed in Fig. 10, where we see a stable and small variance in the expected workload (measured by the number of triangles).

6 Conclusions

The ability to dynamically adapt a finite element mesh is a powerful tool for solving computational problems with evolving physical features; however, an efficient parallel implementation is rather difficult. To address this problem, we have examined a multithreaded approach for performing the load balancing automatically.

Preliminary results from experiments with 2D triangular meshes using the EARTH multithreaded system are extremely promising. The EARTH token mechanism was used to balance the highly irregular and nonuniform workload. To achieve good token mobility, a new indexing methodology was used. During mesh adaption, tokens that migrate to another processor to be executed leave the processed data in the destination processor. This is a novel approach that allows simultaneous load balancing and load-driven data remapping. Results showed that the load is extremely well balanced (3–6% of ideal) when a sufficient number of tokens is generated (at least 512 tokens per node). Token generation (up to 2048 tokens per node) does not significantly increase system overhead or degrade performance, independent of the number of nodes. Having a large number of tokens decreases the total variance of the load balancer and stabilizes the uniformity of the migration. Finally, an underloaded system (one that has few tokens per node) causes instability and leads to unpredictable behavior by the load balancer.

These conclusions do not seem very surprising, but qualitatively confirm what was expected. Given the fact that very minimal programming effort (on the user level) was necessary to obtain these results and that this is one of the first multithreaded approaches to tackle unstructured mesh adaption, our findings and observations become extremely valuable. More extensive experiments will be done in the future, and the results compared critically with an explicit message-passing implementation [14] and a global load balancer [13]. One must remember however that dynamic mesh adaption comprises a small though significant part of a complete application. Further investigations are needed to determine whether the functionality of such an approach is viable for real applications.

References

[1] R. Biswas, R.C. Strawn: A new procedure for dynamic adaption of three-dimensional unstructured grids, *Appl. Num. Math.* **13** (1994) 437–452.

[2] R. Biswas, R.C. Strawn: Mesh quality control for multiply-refined tetrahedral grids, *Appl. Num. Math.* **20** (1996) 337–348.

[3] H. Cai, O. Maquelin, G.R. Gao: Design and evaluation of dynamic load balancing schemes under a multithreaded execution model, *ACAPS Tech. Rep.*, McGill Univ., Montreal, Canada, in preparation.

[4] N. Chrisochoides, E. Houstis, J. Rice: Mapping algorithms and software environment for data parallel PDE iterative solvers, *J. Par. Dist. Comput.* **21** (1994) 75–95.

[5] C.H. Cormen, C.E. Leiserson, R.L. Rivest: *Introduction to Algorithms*, MIT Press, 1990.

[6] `http://www.capsl.udel.edu/EARTH/`

[7] J. Gerlach, G. Heber: Fundamentals of natural indexing for simplex finite elements in two and three dimensions, *RWCP Tech. Rep.*, Real World Computing Partnership, Tsukuba-shi, Japan, in preparation.

[8] J. Gerlach, M. Sato, Y. Ishikawa: A framework for parallel adaptive finite element methods and its template based implementation in C++, *1st Intl. Sci. Comput. in Obj. Oriented Par. Env. Conf.*, Marina del Rey, CA (1997) to appear.

[9] H.J. Hum, O.C. Maquelin, K.B. Theobald, X. Tian, G.R. Gao, and L.J. Hendren: A study of the EARTH-MANNA multithreaded system, *Intl. J. of Par. Prog.* **24** (1996) 319–347.

[10] H.J. Hum, K.B. Theobald, and G.R. Gao: Building multithreaded architectures with off-the-shelf microprocessors, *8th Intl. Par. Proc. Symp.*, Cancun, Mexico (1994) 288–297.

[11] O.C. Maquelin: Load balancing and resource management in the ADAM machine, *Advanced Topics in Dataflow Computing and Multithreading*, IEEE Press (1995) 307–323.

[12] O.C. Maquelin, H.J. Hum, and G.R. Gao: Costs and benefits of multithreading with off-the-shelf RISC processors, *Euro-Par'95 Parallel Processing*, Springer-Verlag LNCS **966** (1995) 117–128.

[13] L. Oliker, R. Biswas: PLUM: Parallel load balancing for adaptive unstructured meshes, *NAS Tech. Rep. 97-020*, NASA Ames Research Center, Moffett Field, CA, 1997.

[14] L. Oliker, R. Biswas, R.C. Strawn: Parallel implementation of an adaptive scheme for 3D unstructured grids on the SP2, *Parallel Algorithms for Irregularly Structured Problems*, Springer-Verlag LNCS **1117** (1996) 35–47.