Parallel Programming Environment for Mesh Generation †

Andrey Chernikov, Nikos Chrisochoides, and Kevin Barker

Computer Science Department
College of William and Mary
Williamsburg, VA 23185
{ancher,nikos,kbarker1}@cs.wm.edu

Abstract

This paper presents a parallel programming environment for mesh generation. Our approach is based on *overdecomposition*. The programming environment supports: *low-latency one-sided communication, global address space in the context of data/object mobility, automatic message forwarding and dynamic load balancing*. These are the minimum requirements for developing efficient adaptive parallel mesh generation codes on distributed memory machines and clusters of workstations and PCs. Performance data from a 3-dimensional advancing front mesh generation code designed for crack propagation simulations suggest that the flexibility and general nature of our parallel programming environment does not cause undue overhead.

Introduction

This paper presents a parallel programming environment for developing parallel mesh generation codes. Our experience during the last 12 years on designing and implementing problem specific environments [1, 2] for parallel field solvers suggests that application developers do not like "black box" solutions. They prefer the clean and very flexible RISC approach from computer architectures (i.e., a simple "instruction set") which in this case translates to a simple functionality for implementing building blocks for parallel mesh generation. Contrary, existing systems [3, 4, 5] follow the CISC approach (i.e., complex and efficient instructions) which in the case of problem specific environments for adaptive applications like mesh generation and refinement translates to complex libraries of routines for difficult problems like dynamic load balancing.

Our programming environment like RISC architectures supports few, but

†This work was supported by the NSF Career Award #CCR-0049086, ITR #ACI-0085969, and Research Infrastructure #EIA-9972853.

flexible primitives like *get_op and put_op* and *remote service request* [6] for simple communication operations; *object install and uninstall*, *user-defined pack/unpack functionality* and *mobile messages* for data migration [7], and call-back functions for querying object attributes like granularity and data-dependencies for information dissemination and decision making. Then the decision making and data migration for dynamic load balancing can be built on top of these primitives.

Moreover, our design philosophy is based on *separation of concerns or requirements* of applications. Researchers from sciences and engineers do not use software tools and libraries in the critical parts of their applications, they want to understand them in order to be able to maintain and sometimes even customize them for their needs. Our programming environment is designed so that different users can use different tools based on their requirements, competence, and understanding of the software system.

The programming environment consist of three layers: (i) *Data Movement and Control Substrate* (DMCS) that implements low-latency and one-sided message passing, (ii) *Mobile Object Layer* (MOL) that implements global address space and automatic message forwarding on top of DMCS, and (iii) *Load Balancing Layer* (LBL) that provides support for dynamic load balancing on top of MOL. Notice that the application can access directly any of the underlying software layers including MPI. The interoperability of our software with MPI allows the re-use of libraries and parallel codes that have been developed independently of our programming environment.

The rest of the paper is organized as follows. First we describe the message passing functionality we provide for adaptive applications that require frequent migration due to load balancing. Then we describe our approach to dynamic load balancing and how it differs from the traditional approach followed by existing application specific libraries. An example that demonstrates the use of the programming environment for a parallel advancing front (PAF) method is given. The PAF method is based on a sequential code developed at Cornell [8] and an approach to parallel mesh generation described in [9]. However, in order to eliminate potential problems with the scalability due to the master/worker model utilized in [9] we modify this approach by substituting the master/worker model with a decentralized scheduling approach. Finally, we conclude with performance evaluation data and a summary of our contributions.

Message Passing

Our programming environment supports one-sided message passing and remote method invocation or remote procedure call communication paradigm

and it has been implemented on top of different communication substrates that support either the binary (*send/recv*) or the one-sided communication protocol. Our objective is to provide a *constant* one-sided application programming interface (API) across different parallel platforms and at the same time take advantage of the best message passing implementation possible. We also noticed that different MPI implementations make different assumptions and optimizations that impose constrains which for some extreme cases affect the behavior of parallel programs. Our programming environment isolates the variations of MPI implementations from the rest of the application.

The programming environment supports global pointers which can be used to build efficient distributed data structures. Other runtime software systems like *Chaos++* [10] also support global address space. However, for dynamic applications like adaptive mesh generation, the static global address space these systems support has limited use. Our programming environment supports global name space in the context of data migration. We extend the concept of a *global pointer* which is defined by a pair of *(local pointer, processor id)* to *mobile pointer* which is defined as a pair of 16 bit processor number and a 32 bit index number; the processor number is the processor where the object was originally allocated, which is called the object's "home node". Mobile pointers are like global pointers but remain valid even if objects migrate to different processors.

Also, we go one step further and implement a correct and efficient message forwarding and communication mechanism for sending messages to mobile objects. Message to data or objects can reach their destination independent off the location of data or objects. Moreover, even if the data or objects are in the process of migrating due to dynamic load balancing the messages will still be received at zero programming effort from the application.

Dynamic Load Balancing

Other than communication the dynamic load balancing is another important issue that programmers for adaptive applications face. Large number of parallel libraries have been developed to ease the burden of implementing load balancing for adaptive applications on distributed memory machines. These libraries are classified into two groups. The first group of libraries couples dynamic load balancing with a specific mesh refinement algorithm. For example, application specific libraries like DAGH [11, 5] support efficient dynamic load balancing methods, but share the CISC approach. In most of the application-specific libraries the data distribution choice and dynamic load balancing methods are tightly coupled for improving perfor-

mance and this limits usage and flexibility. The second group of libraries like Parallel Metis [12] support generic data partitioning methods. These libraries provide efficient data distribution and re-partitioning methods. Our experience with Parallel Metis suggests that the overheads due to data conversion of mesh data to the CSR format that Metis and most of these libraries use and the overhead due to synchronization and data migration are high for adaptive mesh generation.

In contrast to the traditional approach to load balancing problem our programming environment supports the means to develop new load balancing policies or fine tune existing load balancing methods. This is achieved by: (1) encapsulating application-defined data objects (which we call schedulable objects) as well as any pending work that has arrived in the form of application-defined message handlers. (2) plug in different *scheduler* modules which encapsulate the decision making and data migration functionality into a single module. The scheduler is isolated from the rest of the load balancer infrastructure by a well defined and simple interface. Its primary job is to schedule the execution and migration of Schedulable Objects during application runtime. The exact policies used to make these decisions are, however, left to the individual Scheduler implementation.

Schedulable Objects (SOs) are the units of granularity recognized by the LBL. Migrating work from overloaded to underloaded processors involves migrating SOs. SOs are migrated from processor to processor in response to LBL's detected work-load imbalances. Therefore LBL's effectiveness depends on how well work-load imbalances are measured and detected. This is not an easy task. However, schedulable objects work so closely with the Scheduler module and they must be able to provide some information that can be used by the Scheduler to make load balancing decisions. Specifically, Schedulable Objects can provide granularity and dependency information that the Scheduler can use to make scheduling decisions.

Also, in order to migrate Schedulable Objects, LBL must pack them into a single contiguous buffer so that they may be transported using either the DMCS or other data movement operations provided by lower-level communication subsystems. Because the structure of the data objects associated with each Schedulable Object is known only to the application the LBL provides support so that application-specific *packing* and *unpacking* functions can be called only by LBL and only for object migration.

The Scheduler module provides an interface (for lack of space we describe its implementation elsewhere) which individual Scheduler implementations must conform, but they may provide any desired scheduling behavior. The Scheduler interface is as simple and generic as possible in order to allow the

widest possible range of scheduling policies to be implemented. Because the Scheduler is isolated from the application by the API of LBL, minimal changes are required to application source code in order to make use of new Scheduler implementations.

Parallel Mesh Generation

Most of the mesh parallelization techniques explore concurrency in coarse-grain level using *domain decomposition* methods [13]. Some of the early parallelization methods are based on the efficient but globally synchronous and restrictive *data-parallel* programming paradigm [14]. Some recent parallelization methods are based on the asynchronous and flexible, but less efficient *task-parallel* programming paradigm [15]. Some of the task-parallel approaches simplify software complexity by using the less scalable *master-worker* programming paradigm [9].

In this paper we use a *data-centric task-parallel* approach i.e., explore concurrency using domain decomposition, but view computation as tasks that can migrate from processor to processor and along with them they can carry their corresponding data. The task migration is taking place to load balance the computation. The load balancing problem is formulated as a problem of *minimizing the idle cycles in the parallel system* rather than the traditional formulation as an *equidistribution problem*.

Domain decomposition approach initially was proposed by Lohner in [16] and extended for handling adaptivity and load balancing in [9]. We do not use the concept of interior and interface regions to uncouple the subdomains as in [16]. Moreover, instead of the master/worker model [9] we implement a decentralized scheduling of parallel computation. The skeleton of the parallel advancing front method we implement is described as follows: (1) decompose the initial volume grid into N ($N >> P$, where P is the number of processors) subdomains (i.e., apply *overdecomposition*) (2) generate the dual graph of the subdivision and partition it into P sets of subdomains, (3) find the representation of each subdomain as a set of triangular faces and orient the faces, (4) load each set of subdomains in parallel into the corresponding processor and create a Schedulable Object for each of the subdomains, (5) apply mesh generation subroutine [8] on each processor for every local Schedulable Object (i.e., subdomain), while executing a dynamic load balancing algorithm in between, so that some of the Schedulable Object are moved from one processor to another in the case of load imbalance due to different levels of refinement in the geometry, (6) If necessary use ParMetis [12] on the dual graph of the subdivision in order to reduce the surface to volume ratio, and (7) glue the adjacent sub-

domains on each processor in parallel. At the end of the execution of the above steps the mesh is ready for parallel finite element analysis.

Evaluation of Overdecomposition

The use of overdecomposition and introduction of Schedulable Objects (one additional level of indirection) has its advantages and disadvantages. Figure 1 depicts the reduction of total execution time per processor for the pipe geometry (right) viewed by Medit from INRIA. The reduction in the total execution time is due to better memory utilization and nonlinear computational complexity of the sequential mesh generator. As N increases and the final mesh size remains fixed the working set per processor is decreasing. The improvement in the utilization of virtual memory overcomes the overhead introduced by the runtime system.

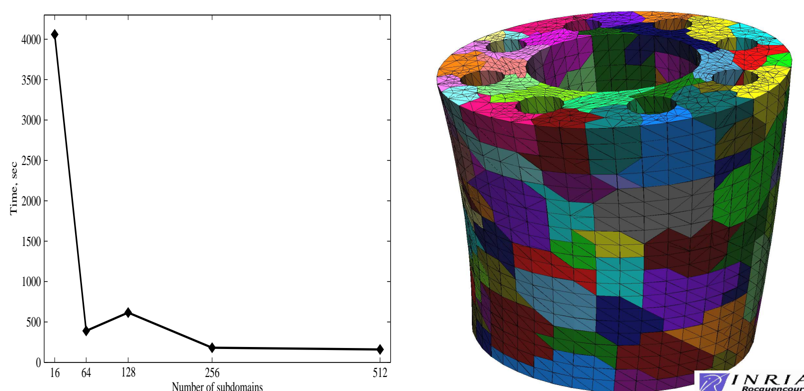

Figure 1. The influence of overdecomposition on the meshing time for the pipe and the gear models.

Figure 2 depicts the breakdown of the execution time for the parallel meshing of the pipe divided into: (left) 512 subdomains with 32 of them refined and (right) 256 subdomains, 16 of which are refined. The final mesh size in the first case is 194,231 elements and the final mesh in the second case is 154,802. Despite the fact that a larger mesh is generated when we use 512 regions, the total execution time is much lower. There are two reasons, one, the memory utilization is much better with larger number of regions and second, the idle time is significantly less due to higher flexibility for load balancing i.e., when the number of objects per processor is decreased, the load balancing methods become less effective.

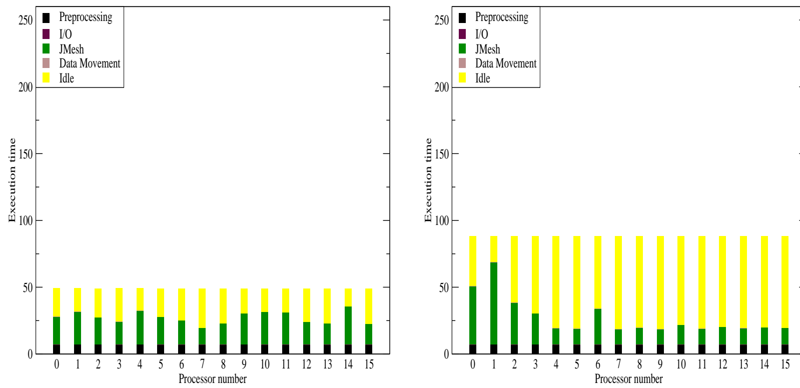

Figure 2. Breakdown of execution time (left) 512 subdomains with 32 of them refined and (right) 256 subdomains, 16 of which are refined. Explicit Dynamic Work-Stealing Load Balancing policy implemented within the parallel programming environment for the pipe.

Evaluation of Parallel Programming Environment

Three decentralized load balancing strategies have been tested: implicit work-stealing implemented withing the parallel programming environment, explicit work-stealing method hardwired for better performance within the application, and the use of Parallel Metis library. The explicit work-stealing method exhibits the best performance. Its advantage is that it has been tuned for the needs of this specific application and doesn't suffer the overhead of the generic parallel programming environment. The use of the implicit work-stealing method has the advantage of very low implementation costs, and it achieves three to four times better performance compared to no load balancing. The use of generic graph partitioning methods like Parallel Metis requires a very good a priori estimation of the work load needed to refine each subdomain. Moreover, the re-partitioning methods have the disadvantage of global synchronization, which is disastrous in the case of asynchronous adaptive applications like parallel mesh generation and refinement.

Figure 3 depicts the breakdown of execution time for 16 processors for two cases; on the left side is the execution time when no load balancing is used and on the right side is the execution time when the work load is

balanced by the parallel programming environment, implementing Work-Stealing Load Balancing policy. Finally, Figure 4 compares the speed up of the case when no load balancing is used, the case where Parallel Metis is used for re-partitioning, the case where the load balancing is hardwired within the application (explicit work stealing) and the case where the parallel programming environment (implicit dynamic work stealing) is used.

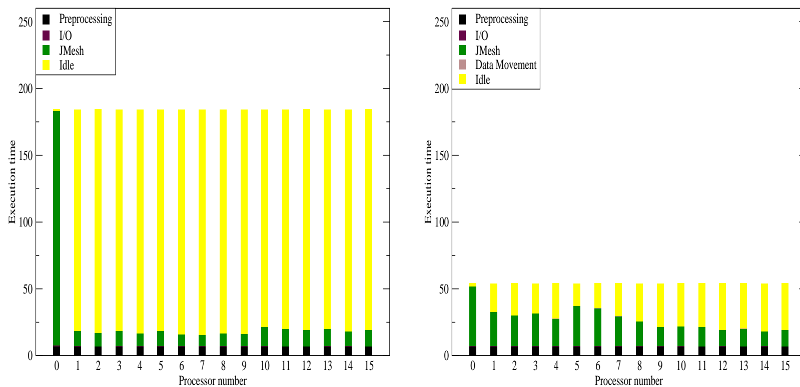

Figure 3. Breakdown of execution time with no load balancing (left), and load balancing (right) performed with Implicit Dynamic Work-Stealing policy implemented within the parallel programming environment for the pipe which is decomposed into 512 subdomains from which 32 subdomains in processor zero were refined by an additional 75%.

Summary

The design objectives of our parallel programming environment for mesh generation and refinement are: (1) assist mesh practitioners to design and implement high-performance adaptive mesh generation applications in an *evolutionary way* i.e., easily transform static (in terms of data migration) message passing codes into dynamic and efficient (i.e., introduce low communication overhead) adaptive applications (2) perform quick prototyping, testing, and evaluation of different dynamic load balancing methods with zero or minimum modification of the application code, (3) ease tasks like portability and maintenance of long life-time adaptive applications. Our preliminary performance data suggest that it is possible to achieve flexibility and ease of implementation without sacrificing performance.

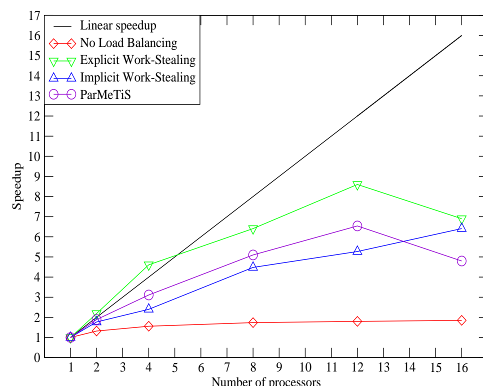

Figure 4. Speedup data for different Load Balancing algorithms. The pipe is decomposed into 512 subdomains 32 of which are refined more than the others.

References

[1] N. Chrisochoides, E. N. Houstis and J. J. Rice. Mapping algorithms and software environment for data parallel pde iterative solvers. *Journal of Par. and Distr. Comp.*, 21(1):75–95, April 1994.

[2] Bruce Carter, Chuin-Shan Chen, L. Paul Chew, Nikos Chrisochoides, Guang R. Gao, Gerd Heber, Antony R. Ingraffea, Roland Krause, Chris Myers, Démian Nave, Keshav Pingali, Paul Stodghill, Stephen Vavasis, Paul A. Wawrzynek. Parallel FEM Simulation of Crack Propagation – Challenges, Status, *Lecture Notes in Computer Science 1800, pp. 443-449, Springer-Verlag* 2000.

[3] A. Basermann, J. Clinckemaillie, T. Coupez, J. Fingberg, H. Digonnet, R. Ducloux, J.-M. Gratien, U. Hartmann, G. Lonsdale, B. Maerten, D. Roose, and C. Walshaw. Dynamic load-balancing of finite element applications with the drama library. *Appl. Math. Modelling*, 25:83–98, 2000.

[4] L. Oliker and R. Biswas. Plum: Parallel load balancing for adaptive unstructured meshes. *Journal of Par. and Dist. Comp.*, 52(2):150–177, 1998.

[5] Karen Devine, Bruce Hendrickson, Erik Boman, Matthew St. John, and Courenay Vaughan. Zoltan: A dynamic load-balancing library for parallel applications: Developer's guide. Technical Report SAND99-1376, Sandia National Labs, 1999.

[6] K. Barker, N. Chrisochoides, J. Dobbelaere, D. Nave and K. Pingali. Data movement and control substrate for parallel adaptive applications. *Concurrency Practice and Experience*, Vol 14, pp 1-15, 2002.

[7] N. Chrisochoides, K. Barker, D. Nave, and C. Hawblitzel. Mobile Object Layer: A Runtime System for Parallel Adaptive Applications. *Adv. in Engin. Software*, Vol 31 (8-9), pp. 621-637, August, 2000.

[8] Cavalcante Neto, J.B., Wawrzynek, P.A., Carvalho, M.T.M., Martha, L.F., and Ingraffea, A.R., "An Algorithm for Three-Dimensional Mesh Generation for Arbitrary Regions with Cracks," Engin. with Computers, 17: 75-91, 2001.

[9] R. Said, N. Weatherill, K. Morgan & N. Verhoeven Distributed Parallel Delauany Mesh Generation *Comp. Methods Appl. Mech. Engrg. 177(1999), pp 109–125*

[10] C. Chang, A. Sussman, and J. Saltz. *Parallel Programming Using C++*, chapter CHAOS++. MIT Press, 1998.

[11] Manish Parashar and James C. Browne. Distributed dynamic data-structures for parallel adaptive mesh refinement. In *HiPC*, 1995.

[12] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel diffusion schemes for repartitioning of adaptive meshes. Technical Report 97-014, University of Minnesota, 1997.

[13] H. de Cougny, & M. Shephard Parallel unstructured grid generation CRC Handbook of Grid Generation, J.F. Thompson, B.K. Soni and N.P. Weatherill, Eds. CRC Press, Inc., Boca Raton, pp. 24.1-24.18, 1999.

[14] Y. Ansel Teng, Francis Sullivan, Isabel Beichl, and Enrico Puppo. A data-parallel algorithm for three-dimensional Delaunay triangulation and its implementation. In SC'93, pp 112–121. ACM, 1993.

[15] P. Chew, N. Chrisochoides, and F. Sukup. Parallel constrained Delaunay meshing. In *Proceedings of the 1997 ASME/ASCE/SES Summer Meeting, Special Symposium on Trends in Unstructured Mesh Generation*, Northwestern University, Evanston, IL, June 29–July 2, 1997.

[16] R. Löhner, J. Camberos, & M. Marshal Parallel Unstructured Grid Generation, *Unstructured Scientific Computation on Scalable Multiprocessors (Eds. Piyush Mehrotra and Joel Saltz)*, MIT Press, pp 31–64, 1990.