Task Parallel Implementation of the Bowyer-Watson Algorithm

Nikos Chrisochoides[1]
Florian Sukup[2]

Advanced Computing Research Institute[3]
Cornell University
Ithaca, NY, 18450, USA

Abstract

In this paper we present a parallel implementation of the Bowyer-Watson (BW) algorithm using the task-parallel programming model. The BW algorithm constitutes an ideal mesh refinement strategy for implementing a large class of unstructured mesh generation techniques on both sequential and parallel computers, by preventing the need for global mesh refinement. Its implementation on distributed memory multicomputes using the traditional data-parallel model has been proven very inefficient due to excessive synchronization needed among processors.

In this paper we demonstrate that with the task-parallel model we can tolerate synchronization costs inherent to data-parallel methods by exploring concurrency in the processor level. Our preliminary performance data indicate that the task-parallel approach: (i) is almost four times faster than the existing data-parallel methods, (ii) scales linearly, and (iii) introduces minimum overheads compared to the "best" sequential implementation of the BW algorithm.

Introduction

Bowyer-Watson algorithm [1] and [2] has been used successfully for a class of

[1]This work supported by the Cornell Theory Center which receives major fundings from the National Science Foundation, IBM corporation, New York State and members of the its Corporate Research Institute, and in part by the Alex Nason Prize Award and NASA Contract No. NAS1-19480, while the author was in residence at ICASE, NASA Langley Research Center, Hapton, VA 23681.

[2]This work supported by the Kurt-Göedel fellowship from Austrian Ministry of Science and Research and in part by the Cornell Theory Center.

Delaunay triangulation methods for unstructured grid generation [3], [4], and [5] on sequential machines. It generates a Delaunay triangulation of an arbitrary set of points by sequentially adding new points and modifying the existing triangulation by means of purely local operations. The BW algorithm is an iterative procedure; each iteration performs two basic operations: (i) *point insertion*, where a new point is inserted using an appropriate spatial distribution technique; and (ii) *element creation*, where existing triangles which violate the circumcircle property are removed and new triangles are built by properly connecting the newly inserted point with old points, so that the resulting triangulation satisfies certain geometric properties [6]. Figure 1 depicts two iterations performed by the BW algorithm; for each new point we find the element which contains it, and then we identify all neighbor triangles that are in conflict (i.e., new point lies within the circle which circumscribes a neighbor triangle); all triangles in conflict are contiguous and form a connected cavity. New triangles are created by joining the new point with the vertices of the cavity. The resulting triangulation is always a Delaunay triangulation [1] and [2].

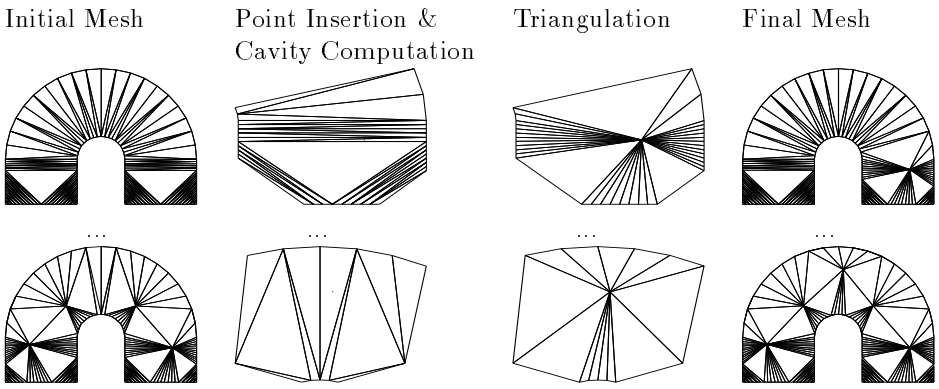

Figure 1: Point insertion and element creation steps of the BW algorithm; initial and final meshes at each iteration shown at the leftmost and the rightmost columns respectively.

The efficiency of the grid generation methods based on the BW algorithm depends on: (i) the efficiency of the search algorithm that is used to identify the first triangle in conflict for each new point insertion, and (ii) the position in which the new points are inserted —certain point insertion orders avoid the formation of short edges and thus the creation of "bad" triangles that would need to be improved again. Parallel methods for addressing such efficiency issues as

well as methods for parallel edge smoothing and swapping are not within the scope of this paper and will appear elsewhere. In this paper we concern about the efficient implementation of the element creation step on distributed memory multicomputers.

The element creation step of the BW algorithm takes place into two phases: (i) cavity computation, for the newly inserted point, and (ii) reconnection of the new point with the vertices of the cavity. Two new points might not be added concurrently, if their corresponding cavities overlap. In such a case synchronization is required among the processors that insert points in conflict. The synchronization is necessary to produce a unique and valid Delaunay triangulation. The cost of synchronizing the participating processors is the main source of inefficiency in the parallel BW algorithm. An example of two cavities in conflict is given in Figure 2; where the initial mesh (Figure 1, top left) is distributed into processors P0 and P1 and the top center part of this mesh (Figure 2a) is the interface region for both processors P0 and P1. In this case the independent and simultaneous insertion of points O_0 and O_1 will result into the generation of two different and non-conforming triangulations (Figure 2b and 2c).

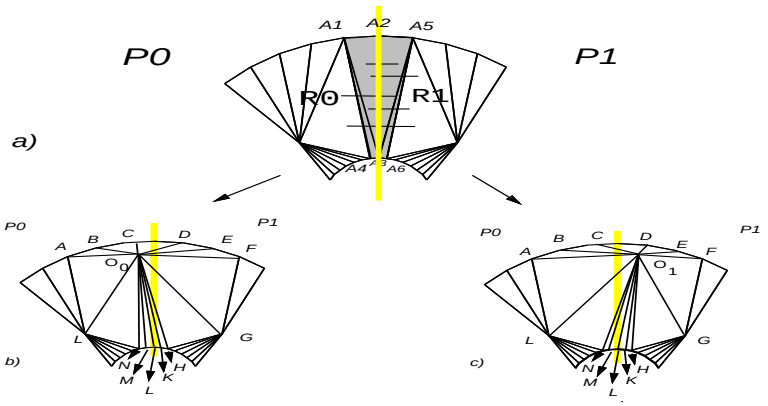

Figure 2: The cavity of a point, O_i, inserted in the region R_i of processor P_i, expands across the processors boundary towards region R_j of processor P_j, where $i, j = 0, 1$ and $i \neq j$.

The rest of the paper is organized as follows: in Section 2 we examine closely only those data-parallel approaches which are relevant to the proposed method. In Section 3 we present our new approach which is essentially a hybrid method that combines both the data- and task-parallel programming models and finally, in Section 4 we conclude with a number of observations and future directions.

Overview

The literature on parallel implementations of Delaunay triangulation is limited, and the few papers that have been published to date are based on the data-parallel programming model [7], [8] and [9]. Data-parallel methods distribute the computation to processors by decomposing the domain to be gridded into a number of subdomains which are handed to different processors for farther remeshing. The data-parallel methods presented in [8] and [9] can be described from the following five step procedure: (i) decompose the domain to be gridded into N subdomains, S_i, (ii) subdivide each subdomain into the interior region, τ_i, and the inter-subdomain region, $\tau_{i,j}$, (iii) generate a grid in each interior region separately (iv) generate the grids for the inter-subdomain regions and finally (v) assemble the final grid. Steps (iii) and (iv) can be interchanged [8], between these two steps (in whichever order they are executed) a global synchronization barrier among processors is assumed. New points can not be inserted in the inter-subdomain regions before all adjacent interior regions are completed.

Note that the pre-computed[4] inter-subdomain regions create buffer zones that guarantee the uncoupling of the interior regions of the subdomains. All processors before they enter the gridding phase of inter-subdomain (or interior) regions, need to synchronize. In the case of data-parallel Delaunay triangulation these synchronization overheads are exacerbated by the fact that the computation cost associated to Delaunay triangulation of a region is variable and unpredictable. Therefore although the domain is decomposed into subdomains with equal number of points or elements, the computational load of processors might not be evenly distributed. This type of imbalances forces a number of processors to wait idle till all processors reach the synchronization point between steps (iii) and (iv). This problem is even worse in the case of adaptive triangulations.

In [9] Cingoni et. al, in order to avoid such synchronization costs assigns the inter-subdomain regions of two subdomains to both processors that handle the corresponding interior regions. With this approach the synchronization cost is minimized but it is not eliminated. Unfortunately this approach introduces two other drawbacks: (i) dramatic increase of space complexity (i.e., number of triangles that are duplicated — each processor P_i has to deal with points in $S_i \cup_j (\tau_{i,j} \cup \overline{\tau}_{i,j})$, where $\overline{\tau}_{i,j} = \{t, t \in S_j \ and \ cavity(t) \ \cap \ S_i \neq \emptyset\}$) and (ii) additional computation is required per processor due to the duplication of the work for the gridding of the $\cup_j \overline{\tau}_{i,j}$ regions. Also, in [9] Cingoni et. al presents a master/workers (MWs) approach in generating parallel Delaunay triangulations. This is a centralized method that eliminates excessive synchronization costs but

[4] It is impossible to pre-compute the inter-subdomain regions without computing the influence regions (cavities) of the points to be inserted. In [8] inter-subdomain regions are defined by the union of elements that share either edges or vertices with elements of another subdomain, as Figure 3 (left bottom) depicts this is not enough for the parallel Delaunay triangulation.

is not scalable. Performance data on the implementation of the MWs approach on NCUBE 2 and on a cluster of workstations using Linda reported in [9], suggest that the optimum number (for maximum efficiency) of worker processes is three plus one processor for the master process.

Finally, Wiliams et. al. in [7] copes with the synchronization problem by building a very efficient tree-like structure for the communication between processors that have to synchronize. He is using this structure to compute the influence region (cavity) for each newly inserted point. The insertion of new points is based on an arbitration procedure; the new point is either inserted or is stored on a temporary queue so that it can be added later. The speedups that are reported in [7] for this implementation of parallel Delaunay triangulation on nCUBE 2 are of the order 0.1 to 0.3 on 16 processors, using the execution time of the best sequential algorithm, for the time of a single processor.

Parallel Bowyer-Watson Algorithm

The parallel Bowyer-Watson (PBW) algorithm presented here it is essentially based on a combination of the data-parallel and the task-parallel model; data-parallel because the mesh is treated as a distributed object that is build upon a hierarchy of data objects which by themselves are distributed with one exception, the vertices. Vertices are scalar objects entirely "owned" by a processor. The task-parallelism comes from the fact that we treat the computation as a collection of computational tasks that can be *created, deleted, suspended and scheduled* at runtime. These tasks are classified with respect to their state as *blocking* and *ready*. Blocking tasks are waiting for the completion of pending requests from other processors —they do not block the processor's CPU. Instead they are stored in the blocking-queue and free the processor's resources for the execution of other tasks. A suspended task is moved from the blocking queue to the ready-queue when all of its remote requests have been serviced. The ready queue is organized in FIFO order.

The input to the PBW algorithm is a boundary conforming mesh which is distributed over P processors, $\Pi_N = \{P_0, P_1, ..., P_N\}$. The data distribution method used here is based on block data distribution methods like $P \times Q$ [10], such a distribution is depicted in Figure 3 (top left). It has been shown, in [10], that this data distribution method is very fast and as effective as other complex and computational expensive heuristics, for 2-dimensional complex geometries. Moreover, this data distribution is very convenient for: (i) implementing very fast search techniques required for the cavity computation[5], (ii) developing memory management methods for exploring the memory hierarchy of multicomputers, and (iii)implementing efficient load-sharing algorithms that minimize synchronization

[5]For each newly inserted point we need identify the first triangle that contains it.

costs due to load imbalances. Due to space considerations, issues regarding fast parallel search techniques, memory management methods as well as load balancing for PBW algorithm appear elsewhere [11]. In the rest of this section we focus on the task-parallel implementation of the element creation step of the PBW.

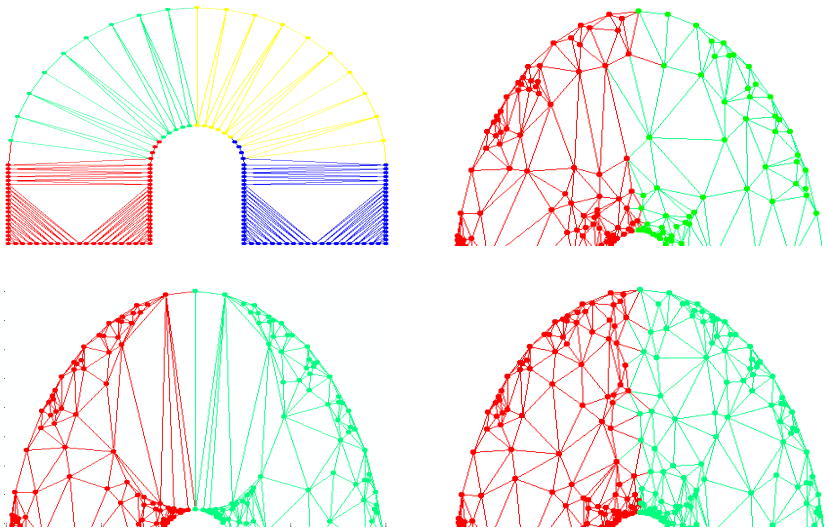

Figure 3: Top: distribution of the initial boundary conforming mesh into four processors and top center part (interface) of the final mesh using task-parallel approach. Bottom: the same top center part of the mesh after the completion of the first phase of the data-parallel approach and the final mesh after the gridding of the inter-subdomain region.

Element Creation: Task-Parallel Approach

Contrary to the data-parallel approach the computation associated to the element creation step of the parallel BW algorithm is decomposed into a number of tasks. These tasks are classified into two groups. The first group of tasks is used to compute the cavity of a newly inserted point, to collect all local and remote data required for the element creation and to reconnect the new point with the vertices of the cavity. The second group of tasks is used to maintain the hierarchy of the distributed data-objects of the mesh. The above tasks are *created, deleted, suspended and scheduled* at runtime based on the presence of remote service requests (RSRs) and/or local computation. In general is hard to

predict the state of these tasks and schedule them in a rigid and fixed manner.

The runtime data distribution of the newly created objects to processors is guided by a set of distribution rules (D-rules). D-rules uniquely assign the newly created data-objects to processors so that access to non-local data can be minimized for future tasks. The D-rules are expressed in terms of the geometric and topologic attributes of the data-objects and a distribution function m, $m : \Omega \ \ni o \rightarrow P_i \in \Pi_N$, where $\Omega \in \{V,\ E,\ T,\ C\}$, and V is the set of vertex data-objects, E is the set of edge data-objects, T is the set of triangle (or element) data-objects and finally C is the set of cavity data-objects. The vertices of an edge, $e \in E$, are ordered into first vertex, $v_f(e)$ and second vertex $v_s(e)$. Throughout this paper we denote any new vertex, edge, and cavity by v_n , e_n, t_n, and $c_n(v_n)$ respectively. Also, the area included in a triangle, t, and cavity, c, are denoted by $t(area)$ and $c(area)$, respectively. Using the above notation, next we outline the D-rules:

1. $\forall e \in E \ \Rightarrow \ m(e) = m(v_f(e))$.
2. $\forall v \in t(area) \ \Rightarrow \ m(v) = m(t)$.
3. $\forall t = t(e_1, e_2, e_3) \in T \ \Rightarrow \ m(t) = m(e_i) = m(e_j) = m(e_i \cap e_j) = m(v_f(e_i)) = m(v_f(e_j))$, where $i \neq j$ and $i, j \in \{1, 2, 3\}$
4. $\forall e_n$ created by connecting v_n with a vertex of the cavity $c = c(v_n)$ then $v_f(e_n) = v_n$.
5. $\forall c \in C \ \Rightarrow \ m(c) = m(v_n) = m(e_{n,i})$, where $v_n \in c(area)$ and $e_{n,i}$ edges created by connecting v_n with the vertices of the cavity c.
6. $\forall t_n \in T \ \Rightarrow \ m(t_n) = m(v_n) = m(c_n(v_n))$.

For example, let us assume that processor P_i identifies a "bad" triangle, t, with $m(t) = P_i$, and let v_n be the centroid of t which lies in the triangle $\hat{t}$, with $m(\hat{t}) = P_j$. In this case processor P_i sends a remote request to processor P_j requesting the expansion of the cavity $c(v_n)$, for v_n. Processor P_i then continues its own work by checking its local FIFO *ready*-queue of tasks and remote service requests; if these queues are empty, then it looks for another "bad" triangle. If there is no such a triangle it waits either to service new remote requests or to terminate. In the mean time processor P_j adds the request, $local\_cavity\_expan(v_n)$, to its ready-queue of tasks and performs the same type of checkings as processor P_i.

Let us assume that processor P_j services a number of remote requests and completes a number of tasks that were ready, before it gets to the execution of $local\_cavity\_expan(v_n)$. Let us also assume that the cavity $c(v_n)$ needs to expand in regions handed to more than one processor, say P_k and P_m. Then processor P_j sends a remote service request, $remote\_cavity\_expan(v_n)$, to P_k and P_m and stores the suspended task $local\_cavity\_expan(v_n)$ to its local blocking queue of tasks. Processor P_j does not block waiting for the completion of $local\_cavity\_expan(v_n)$ task. It schedules the next ready task or remote request for execution, if any, and

continues to add new tasks and requests to its ready and blocking queues as well as checks for possible "bad" triangles.

Upon successful completion of the $remote\_cavity\_expan(v_n)$, processors P_k and P_m send their results and data back to P_j. The moment all pending requests to $local\_cavity\_expan(v_n)$ have been serviced, processor P_j moves the task $local\_cavity\_expan(v_n)$ to the ready queue. The $local\_cavity\_expan(v_n)$ task, right before it yields the CPU to another task, creates the $data\_collection\_process$ task, which stores all data related to $c(v_n)$ in the proper data structures. These data are used by $reconnection\_process$ task to create the new triangles. The new edges and triangles are assigned to processors according to D-rules we've stated above. However, if one of the processors, say P_k, returns unsuccessfully from the task $remote\_cavity\_expan(v_n)$ (i.e., processor P_k had to include in the cavity $c(v_n)$ a triangle that is already included in another cavity which is not yet completed) then processor P_k releases all its marked triangles that included in $c(v_n)$ and sends a proper response to processor P_j. Then processor P_j sends remote requests, $release\_remote\_data$, to all processors that involved into the cavity expansion $c(v_n)$ and releases all its local triangles that included into the cavity expansion $c(v_n)$.
The complete parallel Bowyer-Watson algorithms is given below:

```
REPEAT
  Complete suspended local tasks whose pending remote requests
  have been serviced; and service the remaining remote requests
  WHILE (stack of ''bad'' triangles is non-empty) DO
     Insert new point which is associated to next ''bad'' triangle
     Identify triangle which contains the new point
     Compute the cavity for the newly inserted point and add
     uncompleted tasks to the blocking queue
     Create new triangles by connecting the new point with all
     the vertices of the cavity
     Poll from the network new remote request that need to be
     serviced here and add them to the ready queue
     Complete suspended local tasks whose pending remote requests
     have been serviced and service the remaining remote requests
  ENDWHILE
  Check termination conditions for the processor and set DONE
  to true if all local work is completed or no more remote
  requests need to be serviced
UNTIL DONE
```

A detailed description, implementation and performance analysis of the task-parallel algorithm for the element creation is given in [11]. The PBW algorithm has been implemented on the IBM 9076 SP2 machine. For message passing we've

used Active Messages, an asynchronous communication mechanism with the purpose of exploiting the full hardware at the level of the application while minimizing the operating system overheads as much as possible. A very efficient implementation of Active Messages on SP2 is presented in [12].

The access of non-local data-objects introduces two sources of overhead in the task-parallel implementation: (i) the creation and release of RSRs from the CPU to the network interface, and (ii) the retrieval of RSR from the network and their storage to the ready queue. Both cases require only point-to-point communication. The overhead due to creation and push of RSRs to the network interface is very close to AMs message latency (i.e., 51 μs). The overhead cost for retriving the RSRs from the network and storing them to the ready queue is close to the AMs overhead for polled messages (i.e., 2.1 μs) plus the total time for unsuccessful polls. The cost for each unsuccessful poll is equal to a cache miss.

Summary and Future Work

From the above observations and the fact that each processor (for initial boundary conforming 2-dimensional meshes) has to communicate with at most 8 other processors (in average 2-3 processor) we conclude that the task-parallel implementation of BW algorithm should be scalable. Indeed preliminary performance data indicate linear speedups. A comparison between the data-parallel and task-parallel approaches indicate that for two processors the task-parallel approach is 3.17 times faster than the data-parallel and for three processors is 4.23 times faster. Finally, the task-parallel code on a single SP2 node is 1.6 time slower than the "best" sequential implementation of the BW algorithm. The slow down occurs mainly due to 28.7 % in polling for RSRs and 39 % in testing for global pointers.

The future plans for the next version of the task-parallel implementation of the PBW algorithm are to develop decentralized, load-sharing algorithms. These algorithms are based on the consumer initiated consumer-producer model [13] and their implementation is be based on PREMA, a multithreaded runtime support system that is under development at Cornell [14].

References

[1] Bowyer, A., Computing Dirichlet Tessellations, *The Computer Journal*, Vol. 24, No. 2, pp 162–166, 1981.

[2] Watson, D., Computing the n-dimensional Delaunay tessellation with applications to Vornoi polytopes, *The Computer Journal*, Vol. 24, No. 2, pp 167–172, 1981.

[3] Baker, T, Delaunay Triangulation for Three Dimensional Mesh Generation, Princeton University, MAE Report 1733, 1985.

[4] Mavriplis, D., Turbulent Flow Calculations Using Unstructured and Adaptive Meshes, ICASE Report, No. 90-61, 1990.

[5] Weatherill, N., Delaunay Triangulation in Computational Fluid Dynamics, Computers, Math. Applic. Vol24, No. 5/6, pp 129–150, 1992.

[6] Preparata, F. and M. Shamos, Computational Geometry, An Introduction, Springer-Verlag, pp 398, 1985.

[7] Wiliams, R. and E. Felten, Distributed Processing of an Irregular Tetrahedral Mesh, Caltech Report C3P793, 1989.

[8] Löhner Rainald, Camberos Jose and Merriam Marshal, Parallel Unstructured Grid Generation, *Unstructured Scientific Computation on Scalable multiprocessors, ed. Piyush Mehrotra and Joel Saltz, MIT Press, 1990.*

[9] Cignoni P., Laforenza D., Montani C., Perego R., Scopigno R., Evaluation of Parallelization Strategies for an Incremental Delaunay Triangulator in E^3, *unpublished manuscript, 1993.*

[10] Chrisochoides, N., N. Mansour, and G. C. Fox, A Comparison of Optimization Heuristics for the Data Mapping Problem, To appear in the *Journal of Concurrency Practice and Experience,* 1996.

[11] Chrisochoides, N. and F. Sukup, Parallel Bowyer-Watson Algorithm, Cornell University, CS Technical Report to appear, 1996.

[12] Chi-Chao Chang, Grzegorz Czajkowski, Thorsten von Eicken, Design and Performance of Active Messages on the IBM SP-2, Cornell University, CS Technical Report to appear, 1996.

[13] Chrisochoides, N., Multithreaded Model For Load Balancing Parallel Adaptive Computations On Multicomputers, *To appear in Applied Numerical Mathematics Journal*, 1996.

[14] Chrisochoides, N., PREMA: Portable Runtime Environment for Multicomputer Architectures, http:
www.cs.cornell.edu/Info/People/nikosc/projects/prema/index.html.