Asynchronous Programming and Generation of Delaunay Triangulations in Parallel

Florian Sukup

Institute for Applied and Numerical Mathematics,
Technical University of Vienna,
Wiedner Hauptstrasse 8-10, A-1040 Vienna, Austria

Contents

Chapter 1

Introduction

Parallel computing widens the range of problems which can be solved today while uniprocessor machines will not be able to do so in the years to come. However, parallelism opens the door to a new dimension of difficulties a parallel programmer has to go through. Depending on the problem to solve, parallel computing may well be one of the most challenging fields in computer science.

Parallel Computing — A Controversial Topic

Parallel computing contains a huge potential of problem solving capacity. Computational performance can be increased to Tflop/s or Pflop/s by assembling many uniprocessors to a single parallel machine. This increase of performance does not only mean an increase of speed but an increase of memory as well. Missing availability of sufficient memory capacity on single processor machines often restricts the successful solution of problems. Parallel computing may provide help at this issue. Another topic is the cost of a computer system. High end computers are always expensive solutions to computationally intensive problems. Parallel computers provide the same computational power at much lower hardware costs.

On the other hand this increase of freedom — and it is a kind of computational freedom if you are able to direct the computational flow through several processors in various ways — takes its price. More complex concepts have to be developed for parallel programs, more code has to be written in order to implement these concepts, more and "more difficult to find" errors will be made and more time will be needed to debug the programs to find them.

The Usability of Parallel Computing

Does parallel computing pay off? The answer to this question depends on a combination of various parameters. First, it depends on the problem characteristics. Some algorithms are very easy to parallelize ("embarrassingly parallel"), some are inherently sequential etc. Second, which level of abstraction is used? Are parallelizing compilers able to do the work of parallelizing at the highest level of abstraction or is it necessary to dig deeper into parallelism? The more freedom programmers are given to design parallel programs the more they will feel the drawbacks mentioned above. And, third, how much support is provided at a chosen level of parallelism, are the available tools (for example, parallelizing compilers, communication tools, etc.) able to increase the efficiency of programmers?

The abstraction provided by various parallel programming models well reduces the effort for parallel programming. But abstraction of parallelism finds its limit especially in applications which are very data intensive. This means computational costs of operations are small compared to the amount of data which are needed to perform these operations. An efficient decomposition of data is crucial and normally has to be done by the user who knows about the locality of operations and data and has to step down to a lower level of abstraction to decide about how the data of a particular problem are to be distributed.

Contributions and Earlier Approaches

The research documented in this work focuses on one specific field of parallel computing. As one major contribution, the Bowyer-Watson algorithm, a widely used algorithm in computer science is selected to be parallelized. Due to its nature, the number and occurrence of operations are unpredictable, an efficient parallel algorithm has never been developed. In this work this sequential algorithm, the Bowyer-Watson algorithm, is explained and the differences between programming *synchronously* and *asynchronously* are discussed. Solutions of how to parallelize the Bowyer-Watson algorithm in the *asynchronous parallel programming* model are shown. Earlier attempts to solve this problem in the *synchronous parallel programming* model did not yield satisfying results.

The second major contribution of this work is the extraction of the problems faced at parallelizing this particular kind of algorithm. While asynchronous communication tools are available on a rather low level many problems remain unsolved or arise out of using these tools. Solutions to these problems are presented. These ideas are realized in a newly designed layer for asynchronous communication. This layer is evaluated and compared to implementations in the *synchronous parallel programming model*. Finally all the gained experience results in a model where data decomposition is to be implemented separately from regular computation which, in consequence, can be done at a higher level of abstraction.

Synopsis

The work is organized as follows: Chapter 2 explains the widely used, sequential Bowyer-Watson algorithm. In Chapter 3 the properties of two different parallel programming models, the *synchronous* and the *asynchronous* one, are discussed. These two chapters are the basis for Chapter 4 where two parallel algorithms are presented which can be seen as parallel versions of the sequential Bowyer-Watson algorithm of Chapter 2. Chapter 5 gives an overview about available asynchronous communication tools. Chapter 6 shows the problems faced by their users and presents a layer where these difficulties are taken away from the developer of asynchronous parallel programs. An evaluation of this layer can be found in Chapter 7. Finally Appendix C shows a new model for *asynchronous programming* which is useful for solving *out-of-core* problems, too.

Chapter 2

The Sequential Bowyer-Watson Algorithm

Triangulation is a widely used technique in various areas of scientific computing such as finite element analysis [5], solid modeling and shape representation [1], partial differential equation (PDE) solvers [25], computational fluid dynamics (CFD) codes [29, 36, 44] and in several other situations as described, for instance, in [6]. In many applications triangles need to be as equilateral as possible [5, 46] which makes Delaunay triangulations [27] very attractive. Delaunay triangulations partition areas into triangles using a given set of points satisfying the Delaunay constraint (Section 2.1). Each triangle embeds three angles. The size of the smallest angle in all triangles can be used to asses the quality of a triangulation. Delaunay triangulation maximizes the smallest angle for a given set of points [46].

As described in [8] and [10] the applications mentioned above need triangulations which satisfy certain quality constraints. Triangles have to be well shaped and well sized. Well shaped means that the embedded angles of a triangle must be within a lower and an upper bound. A triangle is well sized if its size does not exceed a given limit. If this is not the case the mesh has to be improved by point insertion. In this process infeasible triangles are deleted and better triangles are created.

In this work the input for mesh generation algorithms is always a mesh, where some triangles are infeasible. In the process of mesh refinement (Section 2.2) the given initial triangulation is improved to obtain a Delaunay triangulation which satisfies the given quality constraints, i. e., where the each triangle is well shaped and well sized. The quality improvement is achieved by inserting new points. Figure 2.1 shows an example of how the input and the output of the mesh refinement can look like.

2.1 Delaunay Triangulation

Assume a given set of points in a plane. Then the area within the convex hull can be triangulated such that the circumcircle of each triangle does not contain any other of the given points. This restriction is called the *Delaunay constraint*. The case where another point lies exactly on the circumcircle, a degeneracy where four or more points are cocircular is mentioned in the next paragraph. Figures 2.2 and 2.3 show the two possible triangulations of four points. In Figure 2.2 the

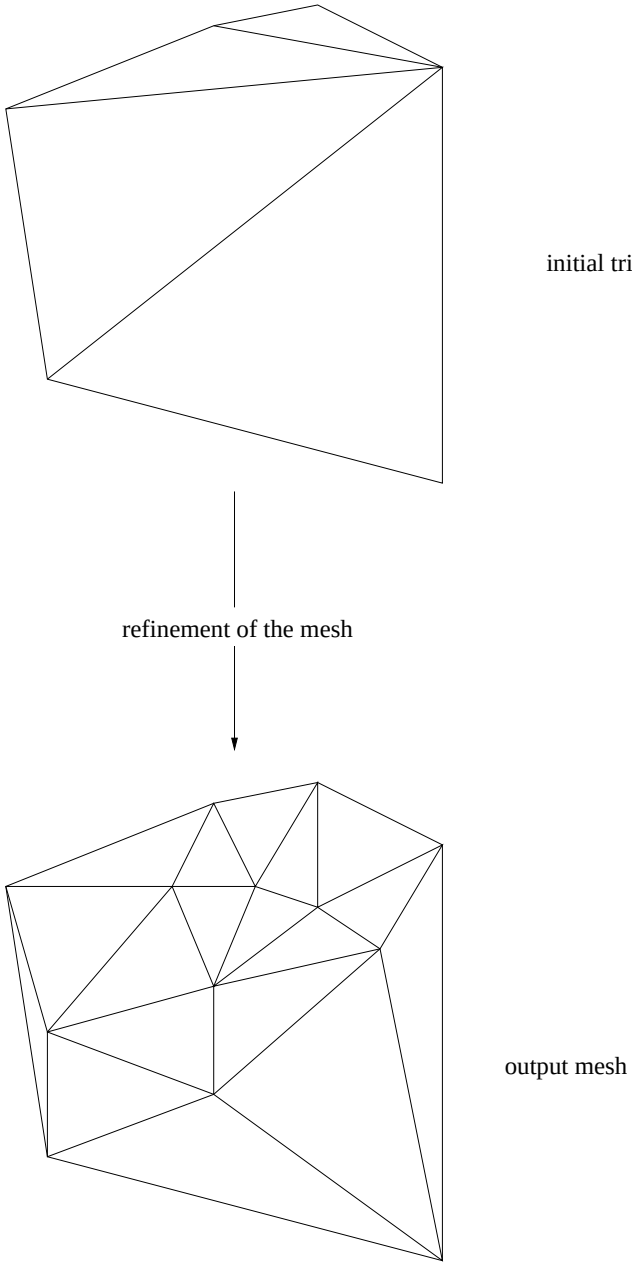

Figure 2.1: An example of mesh refinement.

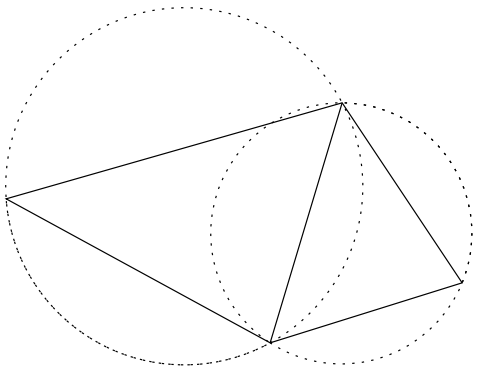

Figure 2.2: The circumcircles of triangles in a Delaunay triangulation must not contain any other mesh points.

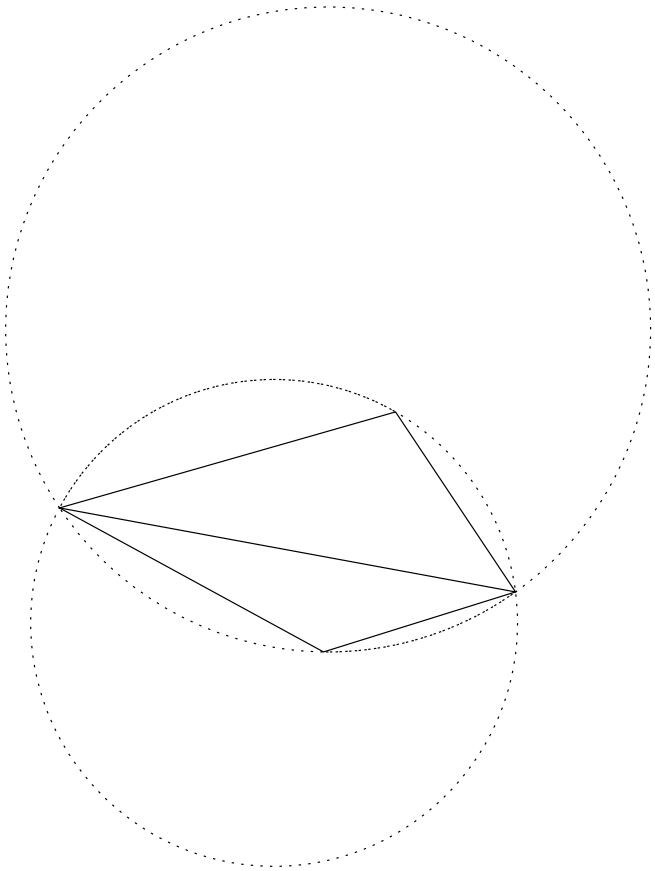

Figure 2.3: This triangulation is not Delaunay.

triangulation satisfies the Delaunay constraint, the triangulation in Figure 2.3 does not. A Delaunay triangulation of a set of 7 points is shown in Figure 2.4.

Delaunay triangulations are unique for a given set of points [27]. This means

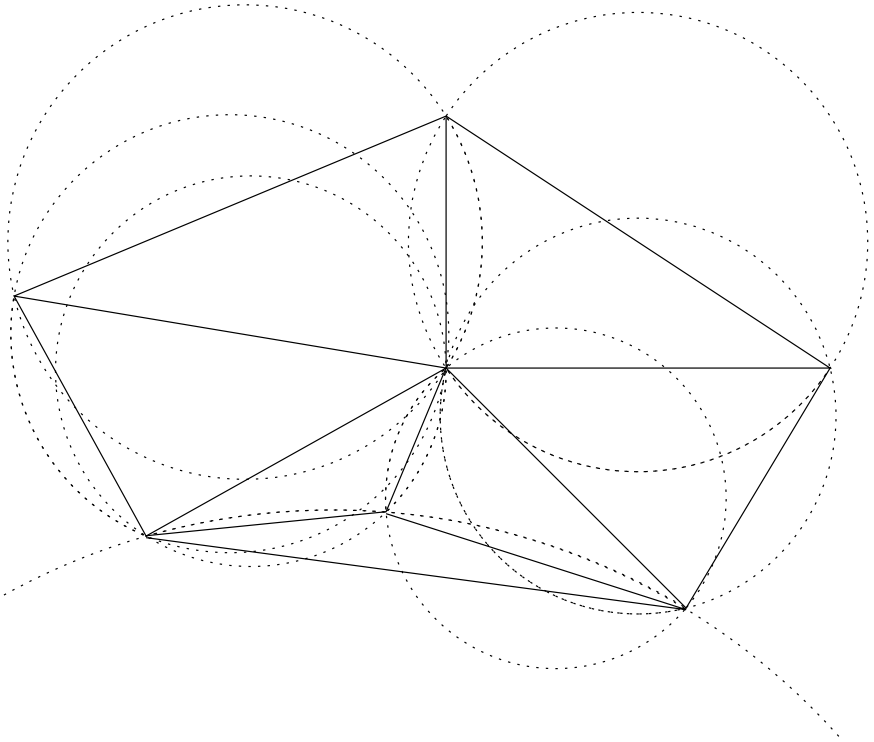

Figure 2.4: A Delaunay triangulation with the circumcircles of the triangles.

there is exact one possibility to triangulate this set of points while satisfying the Delaunay constraint. An exception to this uniqueness of the Delaunay triangulation happens in the case of degeneracies. This means four or more points are lying on a circle. Then more than one solution is possible (see Figure 2.5). Fortunately this fact does not have influence on the Bowyer-Watson algorithm which is used to generate Delaunay triangulations in this work (Section 2.2.2). Proofs, more properties and applications of Delaunay triangulations can be found, for instance, in [27]. Basic material on Delaunay triangulations can be found in [21, 42, 46].

2.2 Mesh Refinement

As mentioned in Section 2 the input for the mesh refinement algorithm is a Delaunay triangulation which has to be improved. The initial mesh, produced by any of the algorithms mentioned in Section 2.2.2, is assumed to contain triangles which are not well shaped or not well sized. These infeasible triangles are removed by inserting new points into the mesh. The output will be a feasible mesh (Figure 2.1). Section 2.2.1 shows how to compute the new points to be inserted in order to improve mesh quality. In Section 2.2.2 the Bowyer-Watson algorithm for inserting new points into meshes is explained. Section 2.2.3 presents the algorithm to solve

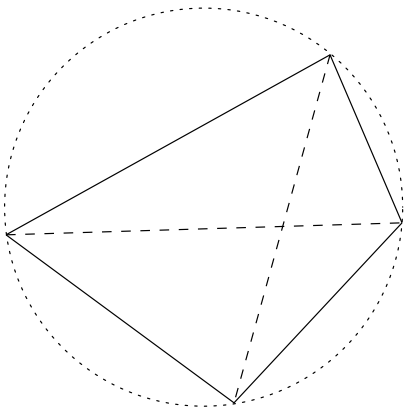

Figure 2.5: If four points are lying on a circle then there are two possibilities (dashed lines) to make a Delaunay triangulation.

the given problem, namely the refinement of an initial mesh to achieve a quality mesh as output. Finally the used data structure is described in Section 2.2.4.

2.2.1 Point Insertion to Achieve Feasible Meshes

As mentioned above a Delaunay triangulation has to satisfy certain quality constraints. Triangles have to be well shaped and well sized. If this is not the case the mesh has to be improved by point insertion. In this process infeasible triangles are removed and better ones are created. Chew [8, 10] developed the following algorithm for finding a new point to be inserted which yields a feasible mesh (see also [47]).

First, one infeasible triangle is chosen and its circumcenter is calculated which is tried to be inserted into the mesh as new point. But this new point may lie too close or outside the boundary. This happens if the angle between imaginary edges to the end points of the boundary edge is greater than a given limit (see Figure 2.6). This limit is needed to guarantee the quality of triangles [8, 10], the smaller the angle has to be the higher becomes the mesh quality, or, used at parallel mesh generation, to keep constrained distributed meshes entirely Delaunay as explained in Section 4.3.5.

In the case that a new point lies too close to a boundary edge, the boundary edge has to be split and the center of this boundary edge is inserted into the mesh as new point instead of the original point.

There is one restriction for the user providing the initial mesh as input. The angle between two boundary edges must not be smaller than the size of the minimum angle of a well shaped triangle.

The reason is that at point insertion after two iterations (both edges are split) a similar triangle still exists with the same angle between the boundary edges, which are just half as long as before (Figure 2.7).

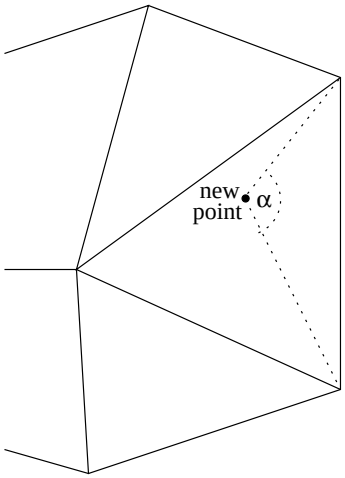

Figure 2.6: If α is greater than a given limit then the new point lies too close to the boundary edge. In this case the respective edge has to be split and its center inserted into the mesh.

If there is such a spike in the boundary it has to be cut off and meshed separately. This work will not cover the special treatment of spikes produced by boundary angles smaller than the given minimum angle. It is the user's responsibility not to provide such input.

2.2.2 The Bowyer-Watson Algorithm

Various algorithms have been developed for generating Delaunay triangulations. The most important ones are the *Flipping* [24, 28], the *Incremental* [23], the *Random Incremental* [2, 16, 17, 18, 32], the *Divide-and-Conquer* [32, 49] and the *Plane Sweep* [26, 33] algorithm.

The *Divide-and-Conquer* and the *Plane Sweep* algorithms are designed to compute triangulations from scratch and cannot be used for mesh refinement. The *Flipping* algorithm is significantly worse in performance than the *Incremental* one [27].

In this work the *Incremental* algorithm is used due to its simplicity in its sequential version. This algorithm was first mentioned in [31] and then generalized by Bowyer [3] and Watson [53]. Standard references in computational geometry and, therefore, also in the field of Delaunay triangulation algorithms are [23, 46].

Description of the Algorithm

The Bowyer-Watson algorithm inserts an additional point into an existing triangulation:

1. Find a triangle t_0 which is in conflict with the new point. This means the new point has to lie in its circumcircle (Figure 2.8a).

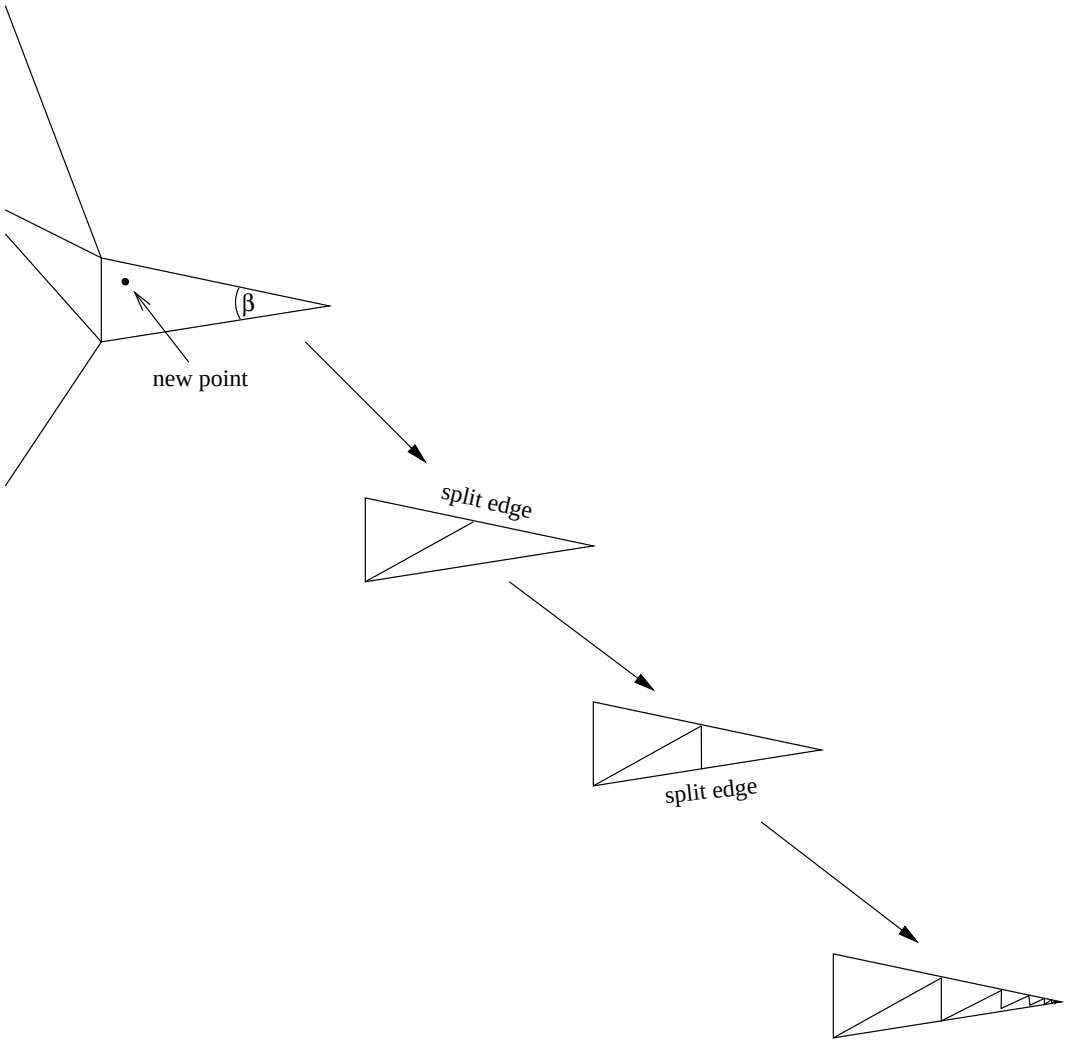

Figure 2.7: β is smaller than the limit for minimum angles. To replace the triangle by better ones a new point has to be inserted. However, if the new point is too close to its edges one of them has to be split. To improve the remaining triangles the other edge has to be split resulting in a triangle which is has to be removed as well .... This leads to an infinite loop, since, β never is changed.

2. Starting from t_0 find all other triangles in conflict with the new point to obtain the set of all these triangles called cavity (*cavity expansion*).

   Go from triangle t_0 to its adjacent triangles, three at most. Check each one if the new point lies in its circumcircle. If yes, go to its other adjacent triangles (two at most) and repeat this procedure as long as triangles, which are in conflict with the new point, can be found.

   The result is the set of all triangles which are in conflict with the new point. This set is called cavity (Figure 2.8b).

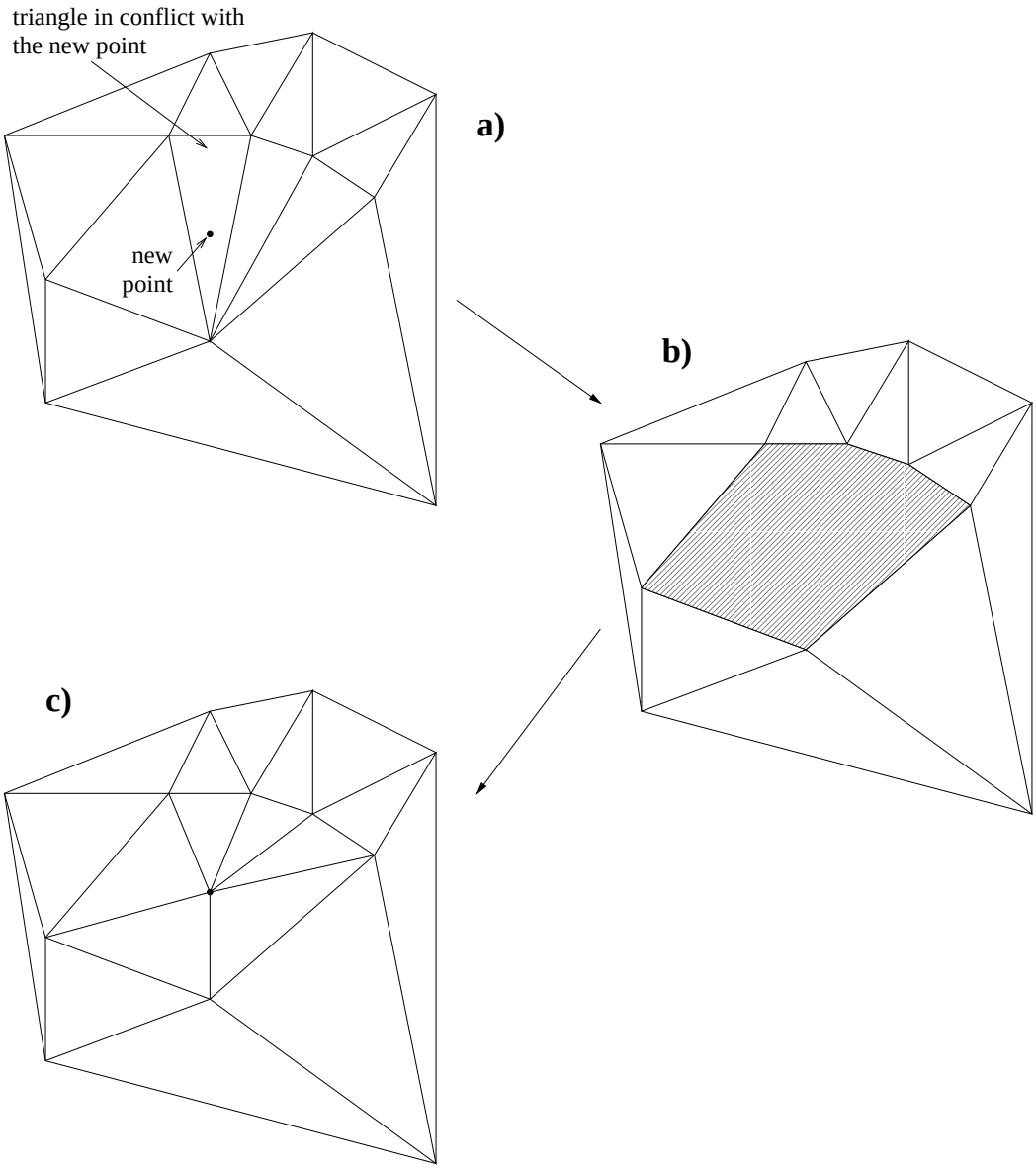

Figure 2.8: Insertion of a new point into an existing mesh: Beginning with a triangle which is in conflict with the new point (a) the cavity has to be found. The cavity contains all triangles which are in conflict with the new point (b). Then the triangles of the cavity are deleted and new triangles are created by introducing edges from the new point to the ones at the border of the cavity (c).

3. Remove the triangles of the cavity and create new ones (*element creation*). This is done by deleting all the interior edges. Interior edges are shared edges of two triangles of the cavity. Afterwards new triangles are created by introducing edges from the new point to every point of the removed triangles (Figure 2.8c).

2.2.3 The Mesh Refinement Algorithm

Summarizing the last sections results in the following algorithm which starts with a given initial Delaunay triangulation and refines this mesh to obtain a quality Delaunay triangulation.

PROCEDURE *refine_delaunay_mesh*

Get the initial triangulation of the object.

Search the mesh for infeasible triangles.

Do while an infeasible triangle exists

Generate the circumcenter of the infeasible triangle as the new point to be inserted.

Expand the cavity: Start at the infeasible triangle, which, by definition, is in conflict with its circumcenter. In a recursive way check the neighbors of the triangle whether they are in conflict with the new point. If the boundary of the mesh is hit also check whether the new point lies outside the mesh or too close to a boundary edge.

If the circumcenter of the new triangle turned out to be outside the mesh or too close to a boundary edge

Then release triangles of the cavity.

Split the boundary edge where the circumcenter of the infeasible triangle lies closest to.

Expand cavity with the center of the split boundary edge as new point: Start with the triangle which includes the split boundary edge. In a recursive way check the neighbors of the triangle whether they are in conflict with the new point.

End if

Remove the triangles of the cavity and create new ones.

Search the mesh for other infeasible triangles.

End do

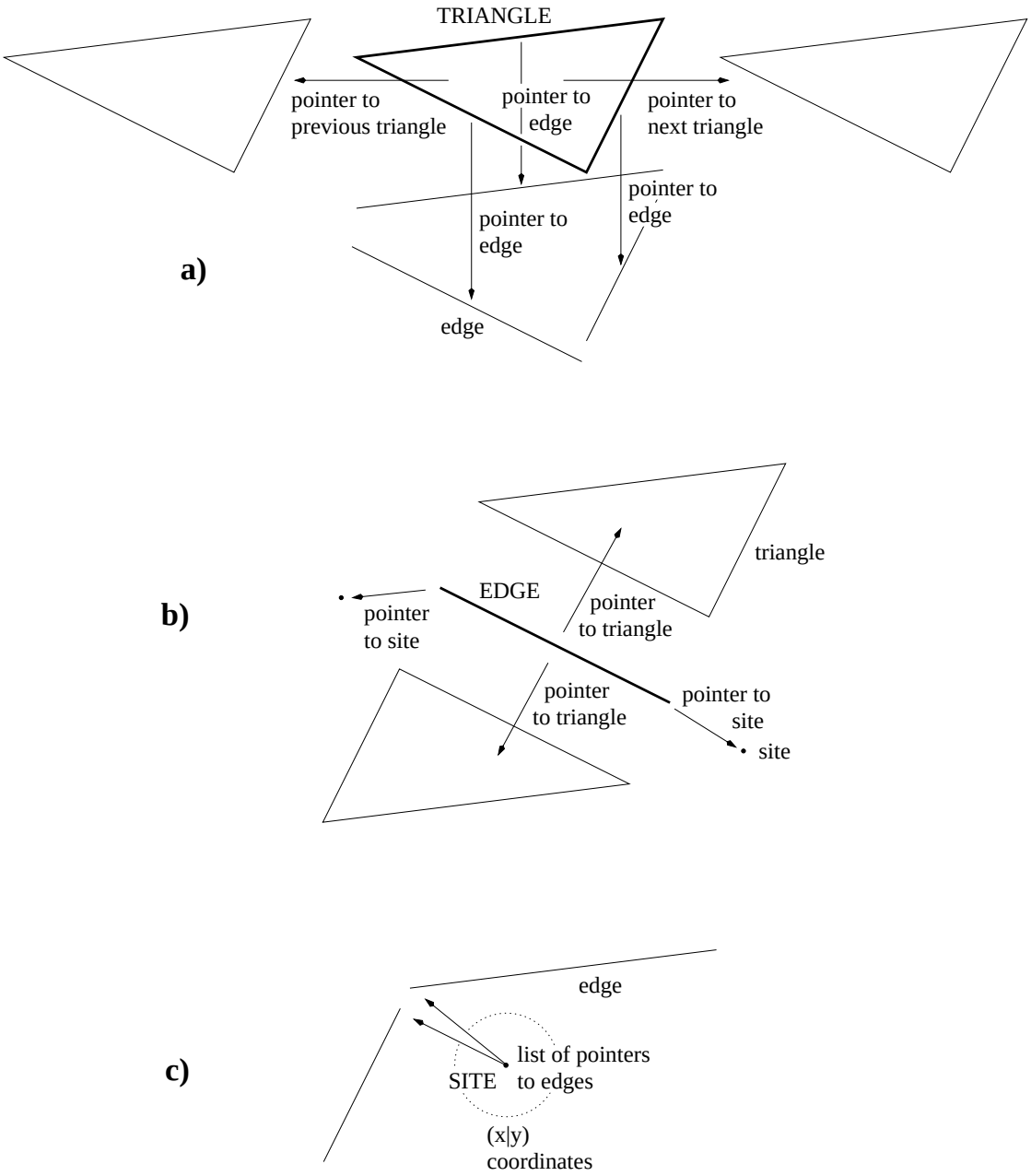

Figure 2.9: A triangle contains pointers to three edges, a pointer to the next triangle and to the previous triangle in the double connected list (a). An edge connects two sites and has pointers to two triangles (b). A site owns a list of pointers to edges and its coordinates (c).

2.2.4 The Data Structure

A generic data structure is used in this work. As shown in Figure 2.9 a triangle consists of three edges and an edge connects two sites[1]. Therefore, each triangle has three edge pointers, each edge two triangle pointers and two site pointers and each site has a list of pointers to edges. Section A.1 shows the source code of one particular data structure. A major reason for using this data structure is consistency. If there is a change at the coordinates of a site, the edges and the triangles do not have to be updated, since, they only contain pointers to sites and edges, respectively. Another reason is the extendibility to higher dimensions. To extend from dimension n to $n+1$ only the $(n+1)$-simplex has to be put on top of the n-simplex. For example: To extend to the third dimension tetrahedra only have to be built on triangles without any influence on the relationship between triangles and edges or edges and sites. Then sites contain three coordinates.

Furthermore all triangles are organized in a doubly connected list. This data structure guarantees a quick access to all elements without much redundancy. This list is used when the quality of the triangles is evaluated that means each triangle has to be checked whether it is well sized and well shaped. The double connection between the triangles in the list is required for efficient deletion of triangles at the process of point insertion.

[1] Commonly used expression for vertices in a mesh.

Chapter 3

Synchronous vs. Asynchronous Parallel Programming

One of the goals of this work was to develop a parallel version of the Bowyer-Watson algorithm. As it will be shown in this chapter, the *asynchronous parallel programming* model fits to the nature of this algorithm.

3.1 Overview

Processor units have to communicate with each other in order to solve problems in parallel. In the *synchronous parallel programming* model program runs consist of parts without communication while communication is performed between each part. These messages represent a kind of barrier which synchronizes processes. Figure 3.1 shows the run of parallel synchronous processes.

Often nonblocking *send* and blocking *receive* commands indicate explicitly synchronous communication. Some synchronous communication tools provide global operations, too. *PVM* and *MPI* are well known representatives of synchronous communication tools.

In the *asynchronous parallel programming* model processes are running uncoupled and *remote service requests*, *put* and *get* operations are mainly used for communication. The asynchronous programmer's philosophy is "send and forget". A process sends a *remote service request* and keeps performing local operations. Therefore, there is no waiting for a response or any other message from other processes. Figure 3.2 shows the run of two asynchronous processes.

In case a *remote service request* arrives from a remote process it is served by the local processor at certain points of time described by the programmer. This means that a *remote service request* is not allowed to interrupt the local process all the time but is put in a queue to be executed at the right point of time, i.e., to avoid inconsistencies.

The *asynchronous parallel programming* model fits especially well to applications where operations which access remote data are unpredictable in at least one of the following terms:

- The time consumption of such operations is not known in advance. Waiting for the end of these operations, as done in the *synchronous parallel programming* model at communication steps, would yield idle time at remote processors.

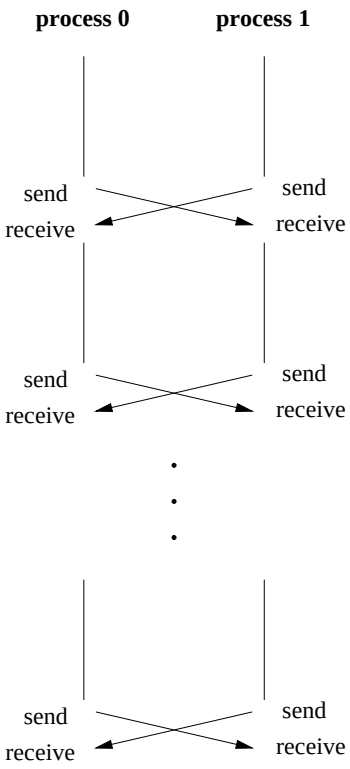

Figure 3.1: The run of two parallel synchronous processes which consist of n parts separated by communication as nonblocking *send* and blocking *receive* or global operations.

- The point of time when these operations need access to remote data cannot be predicted. A remote processor having the required data locally cannot just wait to receive a request for these data as done in the synchronous world, since, it does not know when and whether a request will come.
- It is unknown what data are exactly needed at such operations. Depending on the location of these data requests are sent to different processors.
- The number of such operations and their requests is not known in advance. This would mean in the *synchronous parallel programming* model to wait to receive requests in an unknown number or maybe not even a single one.

3.2 Polling

Asynchronous processes frequently need to poll for incoming requests. Then the computation is interrupted, the network is checked for requests and short routines may be executed. The user has to determine the places where to poll by calling the corresponding function provided by the used asynchronous communication tool. Such tools are described in Section 5.

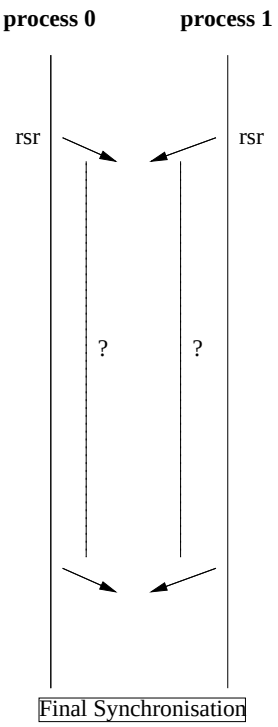

Figure 3.2: The run of two parallel asynchronous processes which communicate via *remote service requests* (rsr). The number, point of time and the size of these *remote service requests* are not known before they are sent.

3.3 Examples

The cause for choosing the one or the other programming model is the differences in nature of the algorithm to be parallelized. This section shows two tasks where one is an example for better using the *synchronous parallel programming* model and the other one fits better to the *asynchronous parallel programming* model.

3.3.1 Synchronous Programming

A list of names, equally distributed over two processors with distributed memory, has to be sorted in parallel such that the first processor finishes with a sorted list of all names A to M, the second processor with a sorted list of all names N to Z. Let's assume names are equally distributed over all letters. Therefore, no sophisticated load balancing algorithm is necessary. An algorithm solving this problem may look like that:

1. Locally separating the names beginning with letters A to M from the ones beginning with N to Z. The result are two lists, one containing all local names A to M, the other containing all local names N to Z.

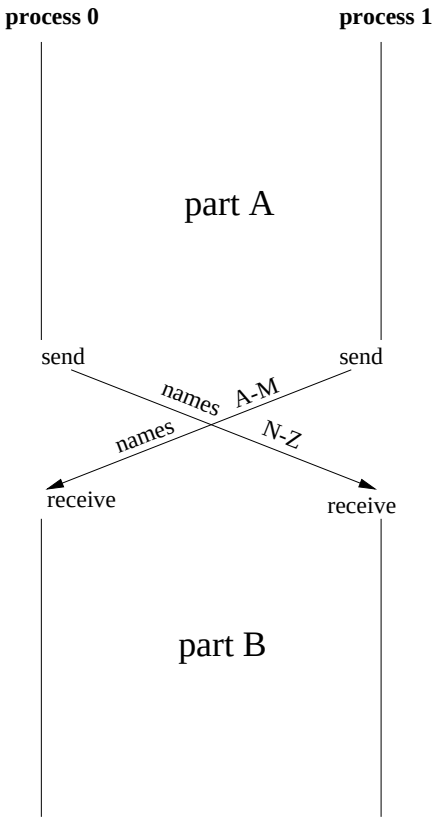

Figure 3.3: Two processes are sorting names in parallel. At part *A* names beginning with A to M are separated from names beginning with N to Z. Then communication is performed by sending one part of the data to the other process. Then, at part *B*, the local and the received part of the names are merged and sorted.

2. *Sending* the list with the names which belongs to the other processor.

3. *Receiving* the list from the other processor.

4. Locally merging the two lists and sorting the merged list.

Steps 1 and 4 represent the two parts of the program where computation is performed. They are separated by communication performed in Steps 2 and 3. Figure 3.3 shows the run of these two processes of this example.

3.3.2 Asynchronous Programming

A company stores data about its employees in a database. The record of each employee includes the name and the income of the parents and, for all male employees, the name of the wife. Furthermore, for statistical reasons, the record of a male employee also contains the sum of his parents' income and, if his wife

is also working at the company, the income of the parents of his wife. This entry has to be updated on a regular basis.

For logistical reasons the company is split into two parts and each one gets its own computer where data about its employees are stored. The computers can work independently from each other except the update of the entry in the male employee's record which denotes the sum of the income of his and his wife's parents, if his wife is working at the other part of the company. Then information is needed from the remote processor. This communication is unpredictable because the computers cannot know when and if the other computer needs data from its database. An asynchronous parallel algorithm may look like that (let's denote the field in the male employees' records where the accumulated income of his parents and the ones in law has to be written to as `both_parents_income`:

PROCEDURE *main*

Do for each male employee

Write his parents' income to field `both_parents_income`.

If he has a wife, employed by the company

Then

If his wife works at this part of the company

Then increase field `both_parents_income` in his record by her parents' income.

Else send a remote service request (*check_for_wife_s_parents*) to the computer of the other part of the company.

End if

End if

End do

PROCEDURE *check_for_wife_s_parents*

If his wife's parents have any income: send the income of his wife's parents in a remote service request (*add_wife_s_parents_income*) to the computer of the other part of the company.

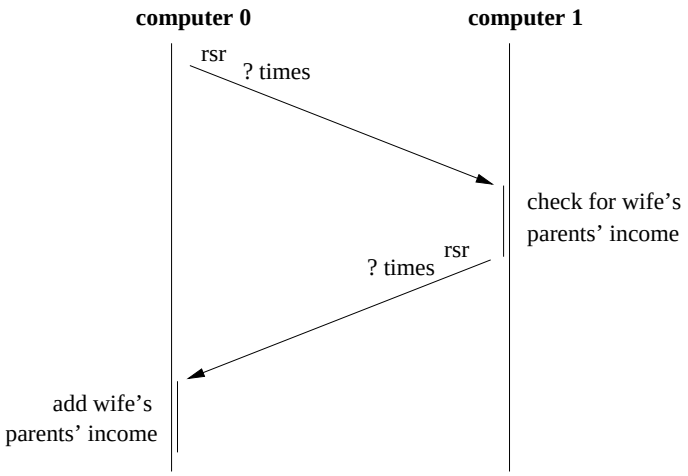

Figure 3.4: The two computer of the company are adding the specified incomes. The question mark at the *remote service requests* (rsr) indicates that their number depend on the data and cannot be told in advance.

PROCEDURE *add_wife_s_parents_income*

> Increase entry `both_parents_income` of the male employee's record by the income of his wife's parents.

This algorithm takes into account that it is unpredictable whether a male employee's wife is working at the other part of the company and, if this is the case, if her parents have any income. If they do not, a remote service request can be saved. Figure 3.4 shows the behavior of the two computers of this company.

3.4 The Asynchronous Nature of the Bowyer-Watson Algorithm

As explained in Chapter 2 the operation of inserting a new point is very unpredictable. Such an operation takes place wherever an infeasible triangle is detected. This implies that the number and location of these operations are not predictable. The number of triangles which are involved in such an operation are not known in advance. The impact on communication between processes arises when the involved triangles are located at the border between two processes and yields the conclusion that the *asynchronous parallel programming* model is the suiting one for developing a parallel version of the Bowyer-Watson algorithm.

Chapter 4

The Parallel Bowyer-Watson Algorithm

4.1 Parallel Mesh Generation

The literature on parallel Delaunay triangulation is limited, and the few papers that have been published to date are based on data-parallelism using the *synchronous parallel programming* model [14, 15, 39, 40, 41, 54]. Data-parallel methods distribute the computation to processors by decomposing the domain to be meshed into a number of subdomains which are handed to different processors for further remeshing. The data-parallel methods presented in [14, 15, 39] can be described in the following five step procedure:

1. Decompose the domain to be meshed into subdomains.

2. Subdivide each subdomain into an interior region and an inter-subdomain region. The interior region is defined as area where point insertion do not have any influence on other processors, i.e., do not have to access remote data. The inter-subdomain regions is the other part of the subdomain where the cavity at point insertion also expands to remote processors.

3. Generate the mesh in each interior region separately. This step can be done without communication to other processors

4. Generate the mesh for the inter-subdomain regions.

5. Finally assemble the mesh.

Between Steps 3 and 4 a global synchronization barrier among processors is assumed. New points can not be inserted in the inter-subdomain regions before all interior regions are completed. This yields blocking the adjacent processor and worsen speedup and performance. It also has to be considered that all communication is performed at Step 4 which induces heavy traffic.

Williams et. al. in [54] cope with the synchronization problem by building a very efficient tree-like structure for the communication between processors that have to be synchronized. They are 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 in case remote data are accessed. Therefore, communication is done at the end and all at once as well.

Blocking remote procedure calls are used in [40, 41] to expand cavities to remote processors to keep consistency in the mesh. Due to this inefficiency following improvement was done: All remote requests are stored in a queue to be sent at once to minimize the time for waiting for the answer of a request. But, again, it pushes all the communication to the very end.

In this work the *asynchronous parallel programming* model is used for parallel Delaunay meshing [13]. Two approaches were developed: The first (Section 4.2) deals with the mesh as entire graph, the second (Section 4.3) marks the edges along processor borders as constrained and each processor computes its own part of the mesh. Depending on certain parameters the result of the second approach might not be Delaunay but still a high quality mesh.

4.2 Parallel Delaunay Triangulation

4.2.1 The Data Structure

As described in Section 2.2.4 a generic data structure is used for generating the mesh. Parallelizing the algorithm for generating the mesh also means to distribute the data over all processors.

In order to refine a mesh in parallel the triangulation has to be decomposed into regions which are distributed to processors later on. If, during the algorithm, a new point is inserted into the mesh the region has to be identified where the new point is located. To minimize the effort in locating the new point in this work the plane where the triangulation lies in, is divided by an underlying grid (Figure 4.1). Nevertheless the algorithm is not limited to this specific domain decomposition. It works for any decomposition into convex regions. A domain decomposition into rectangular regions (Figure 4.2), for example, is a good compromise if the mesh contains regions with higher density. On the one hand such a domain decomposition gives more freedom at sizing the regions on the other hand the region, where a new point is located in, can efficiently identified.

After having divided the domain each processor is assigned one of these regions. All elements, which are sites, edges and triangles, in such a region are stored at the corresponding processor.

Sites are lying in exact one region and are stored at the processor which the region belongs to. But edges and triangles may overlap region borders! These elements are stored as follows: Triangles consists of three edges. Therefore, a triangle may be stored at any processor where at least one of its edges is stored. And an edge connects two sites. So, an edge may be stored where at least one site is located.

Thus, members of an element may be located remotely. This means that an edge of a triangle might not be local but stored at another processor. Or a site of an edge might be located remotely as well. Hence, pointers to these remote

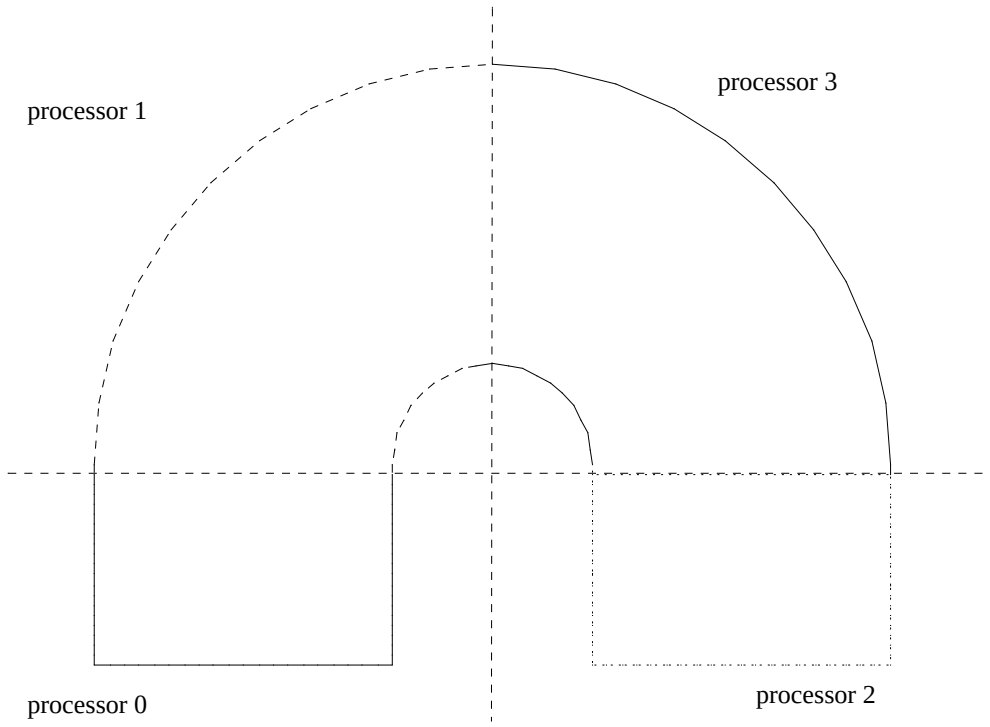

Figure 4.1: A domain is divided by an underlying grid. At the triangulation sites are stored at the processor where they are located. Overlapping elements are only stored at one processor.

elements of the mesh are global pointers.

Additionally to the data structure already described, there is a list on each processor connecting all local triangles to allow easy access. This list is used when the local triangles have to be processed. For example, in order to evaluate all triangles of the initial mesh this list is used to step through all local triangles.

4.2.2 The Parallel Algorithm

As described in Chapter 2 the Bowyer-Watson algorithm is applied as follows:

1. Find an infeasible triangle, i.e., a triangle that does not satisfy the given quality constraints (Sections 2 and 2.2.1).

2. Calculate the circumcenter of the infeasible triangle as new point.

3. Expand the cavity. While expanding check if the new point is outside or too close to a boundary edge. This is not known in advance. If the new point turns out to be outside or too close to a boundary edge, split that edge and start cavity expansion again using the edge center as new point.

4. Create new triangles.

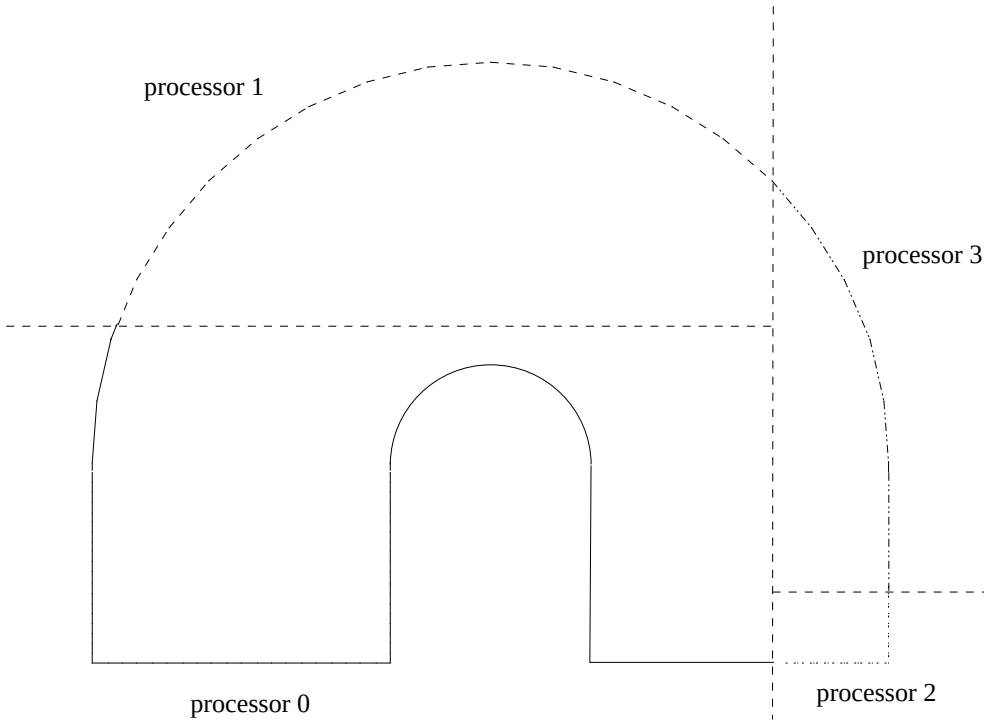

Figure 4.2: A domain is divided into rectangular regions. Assuming points are dense at the right bottom of the domain, the region of processor 2 is small.

This algorithm is applied until all triangles satisfy the quality constraints.

In the parallel version there is one major problem to be solved, the collision of cavities: Points are inserted in parallel. As the cavity expands it may reach remote elements. Then the available part is frozen and a remote request is sent in order to expand the cavity remotely. This frozen part of the cavity cannot be used for any further operations until a request from the remote processor says what to do with it. In the meantime another infeasible triangle is taken and computation goes on.

If an expanding cavity collides with the frozen cavity, all triangles of the expanding cavity are released. The option of keeping it and freeze it as well for complete expansion later could easily result in a deadlock, since, two cavities can block each other.

Taking that into account, Point 3 and 4 has to be changed (the situation concerning the new point being outside or too close to a boundary edge still exist but is not mentioned below, since, it is not necessary for understanding the parallel concept):

3a. Expand the cavity. If the cavity can be expanded completely, create new triangles as next step (4a) otherwise, if a remote edge or triangle has to be accessed, freeze the local part of the cavity and send a request to expand the

cavity remotely (3b). If, however, an already frozen cavity is hit, release the triangles of the current cavity and skip Point 4. In this case cavity expansion failed.

3b. Remote expansion of the cavity. If the cavity can be expanded completely, send a request to the process where the new point is located to create new triangles (4b), otherwise, if a remote edge or triangle has to be accessed, freeze the local part of the cavity and send a request to expand the cavity remotely (3b). If, however, an already frozen cavity is hit, release local triangles of the current cavity and send a request to the process where the new point is located to initiate the release of the rest of the triangles of the current cavity 4b.

4a. Create elements as in Step 4 of the sequential algorithm.

4b. Collect parts of the cavity. If all parts arrived and all are valid, this means no other cavity was hit, create new triangles (all new triangles are stored locally, to comply with the restrictions in Section 4.2.3, but remote updates of pointers have to be initiated), otherwise, if another cavity was hit, initiate the release of all triangles of that cavity (locally and remotely).

Detailed Description of the Algorithm

PROCEDURE *parallel_DT*

Read in sites and triangles of the initial triangulation and distribute them to the processors.

Do until (*the following condition is checked at loop end*) all infeasible triangles are deleted and no system work (i. e., serving remote requests) left. Note: Remote service requests may produce infeasible triangles.

Do while an infeasible triangle is available[1]

Mark the triangle as in use.

Try to insert its circumcenter, call *insert_point*.

Serve remote requests.

End do

Serve remote requests.

End do

PROCEDURE *insert_point*

Expand the cavity: Start at the infeasible triangle, which, by definition, is in conflict with the new point (circumcenter or center of a boundary edge to be split). In a recursive way check the neighbors of the triangle whether they are in conflict with the new point.

Switch on the result of the cavity expansion

Case [Cavity could be expanded completely]

Remove old triangles and create new ones. This can be done locally and does not need any interaction with other processors (exception: sites of involved triangles might be remote, in this case remote information is needed in order to evaluate the new triangles).

Case [Cavity has to be expanded to one or more remote processors]

Initiate a remote cavity expansion, call *remote_cavity_expansion* remotely.

Store the local part of the cavity which is already computed at the process where the new triangles will be created. Those are always created and stored at the process where the new point is located. Therefore, all parts of the cavity have to be collected at that process.

If that process is the local one

Then store the current part of the cavity locally, call *collect_cavity*.

Else send this part of the cavity to the remote process, call *collect_cavity* remotely.

End if

Case [The new point is too close to a boundary edge]

Split the boundary edge. Release all triangles and start again point insertion with the center of the boundary edge as the new point, call *insert_point*.

Case [Cavity Collision]

Release all triangles and store the initial infeasible triangle for a later try.

End switch

PROCEDURE *Remote_cavity_expansion*

Continue the cavity expansion which was started at a remote processor

Switch on its results

Case [Cavity could be completed]

Send the current part to the process where the new triangles are created, call *collect_cavity* locally or remotely depending on the location of the new triangles.

Case [Cavity has to be expanded to one or more remote processors (also see *insert_point*)]

Initiate a remote cavity expansion, call *remote_cavity_expansion* remotely.

Send the current part of the cavity to the process where the new point is located, call *collect_cavity* either locally or remotely.

Case [The new point is too close to a boundary edge]

Release all current triangles.

Initiate a new point insertion with the center of the edge to be split as new point at the process where old new point was located, call *collect_cavity* locally or remotely.

Case [A triangle in use was hit by expanding the cavity]

Release all current triangles.

Call *collect_cavity* where the new point is located to tell:

Release all triangles.

Store the initial infeasible triangle for a later try.

End switch

PROCEDURE *collect_cavity*

Store incoming part of the cavity.

If the collection of the cavity is completed

Then

Switch on its status

Case [Cavity is valid, this means, it could be expanded completely]

Delete all triangles (the local as well as the remote ones) of the cavity.

Create new triangles and store these locally.

Update all border edges of the cavity. Do this partly on remote processors.

Case [Cavity is invalid because the new point is too close to a boundary edge (which has to be split now)]

Release all triangles of the cavity (including the remote ones) and store the starting triangle for a later try.

Initiate point insertion with the center of the boundary edge to be split as new point at the process where this new point is located, call *insert_point* locally or remotely.

Case [Cavity is invalid because of a cavity collision]

Release all triangles of the cavity (including the remote ones).

End switch

End if

Processor 0	Processor 1	Processor 2	Processor 3
		cav exp start, requests rem cav exp at 0	
rem cav exp (from 2), requests rem cav exp at 2 and 3			
		rem cav exp (from 0), pro col data	rem cav exp (from 0), requests rem cav exp at 1
	rem cav exp (from 3), requests rem cav exp at 0 and 3		
rem cav exp (from 1), requests pro col data at 2			rem cav exp (from 1), hits other cavity, requests pro col data at 2
		pro col data (from 3)	
		pro col data (from 0), all parts received, releases locally and requests rem release at 0, 1 and 3	
rem release (from 2)	rem release (from 2)		rem release (from 2)

Table 4.1: An example of an extensive cavity expansion. ("cav exp"... cavity expansion, "rem cav exp"... remote cavity expansion, "pro col data"... process collected data, "rem release"... remote release).

Table 4.1 shows an example of a run where the cavity expands over four processors. The cavity expansion starts at processor 2, initiates a remote cavity expansion at processor 0, processor 0 initiates remote cavity expansions at processor 2 and processor 3. Then processor 3 initiates a remote cavity expansion at processor 1 which does the same at processor 0 and processor 3. But at processor 3 the cavity hits another cavity and, therefore, the current cavity cannot be expanded further. Thus, it has to be released.

After having collected all parts of the cavity at processor 2 all parts of the cavity are released, locally as well as remotely. This example shows that collisions

[1] Triangles may exist but may not be available if they are part of a frozen cavity.

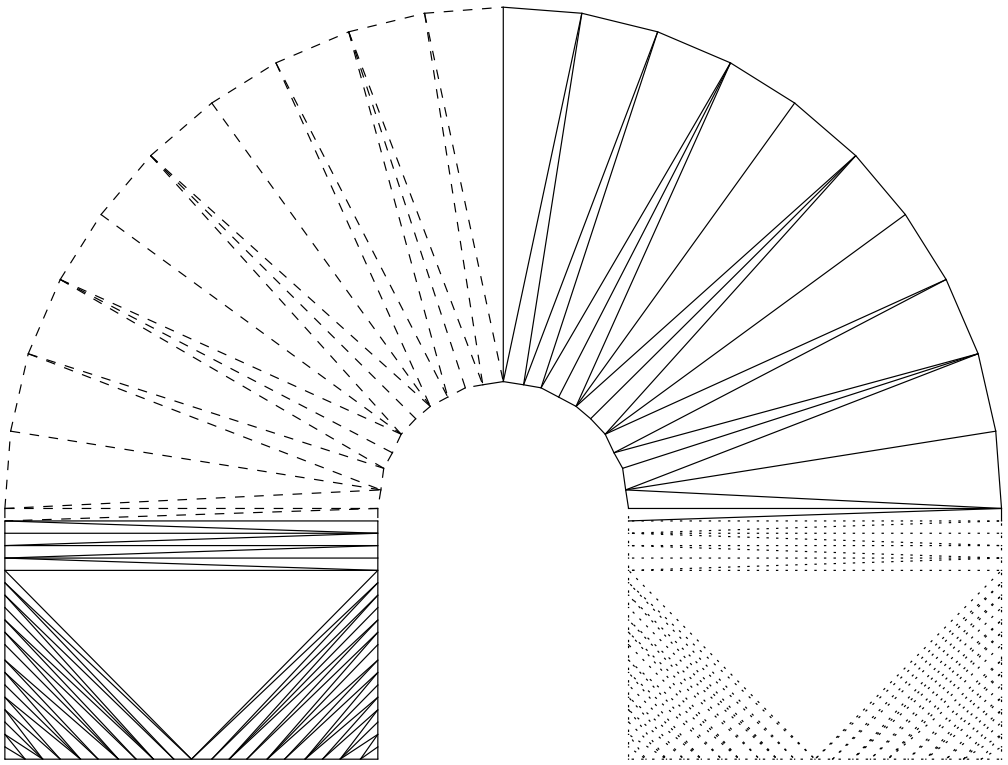

Figure 4.3: The initial triangulation distributed over four processors.

between two cavities can be expensive, especially if so many remote requests have to be served.

Figure 4.3 shows an example of an initial triangulation distributed over four processors. The parallel Bowyer-Watson algorithm was applied and produced the output shown in Figure 4.4.

4.2.3 Implementation Details

Allocation of Elements

The most important data structures are shown with more details in the following to understand some important tricks to make the code more efficient, shorter and easier to understand. (The source code is shown in Appendix A.2.)

point

- Two real numbers to represent the coordinates

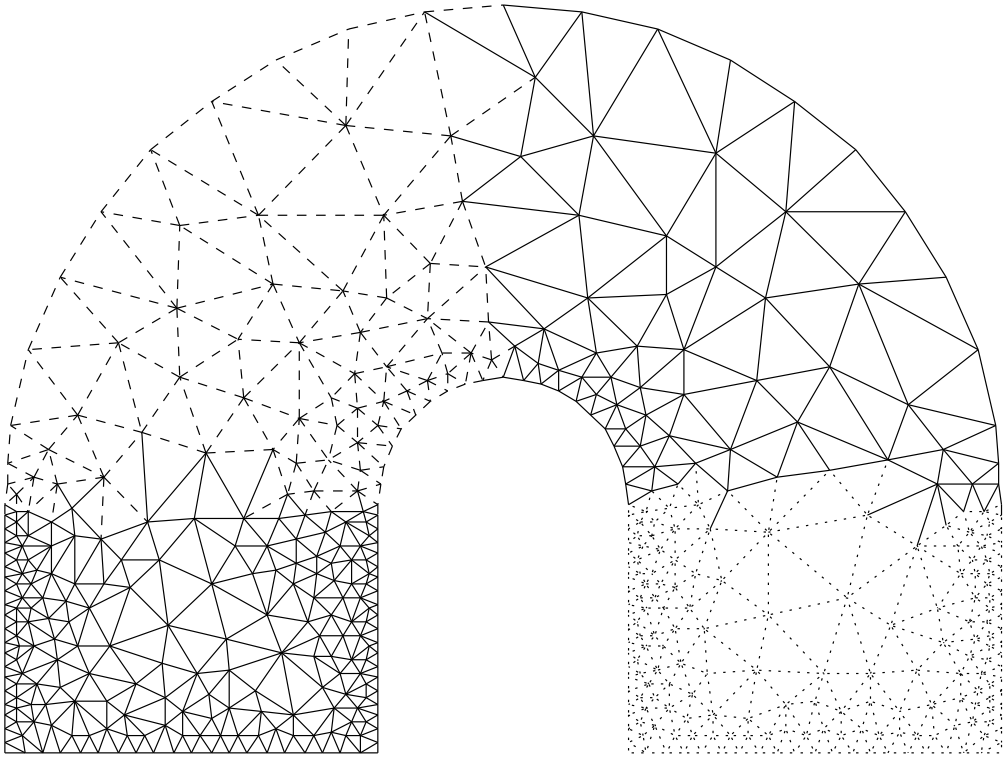

Figure 4.4: The parallel Bowyer-Watson algorithm used for the triangulation.

global pointer

- Processor ID
- Local pointer

site

- Point
- Pointer to the first element of the edge list

edge

- Two global pointers to triangles
- Local pointer to the local site of the edge
- Global Pointer to the other site of the edge

triangle

- Pointer to the next triangle in the local triangle list
- Pointer to the previous triangle in the local triangle list
- Circumcenter
- The square of the circumradius
- Two local pointers to edges
- Global pointer to the third edge
- Flag to indicate if this triangle is part of a cavity

In order to save memory, even more the cost of dereferencing global pointers, to reduce the number of special cases and, therefore, shorten the code in general the following restrictions are made:

- An edge is always allocated at the processor where at least one site is allocated, too. This might seem obvious anyway but if each site of a triangle is located at an other processor the three edges cannot be allocated at the same processor. But this might seem desirable at some point.
- A triangle is always allocated at the processor where at least one site and two edges are allocated.
- Elements of the edge list are always allocated at the same processor where the site is allocated even if adjacent edges stored at remote processors.

This makes it possible for an edge to have only one global pointer to a site as well as for a triangle to have only one global pointer to an edge.

It is always possible to satisfy these constraints: Let's start with the trivial case: the triangulation only consists of one triangle. At inserting a point, all new edges and triangles just have to be allocated at the processor where the new point is allocated. This will keep the above constraints satisfied.

Control Over the Remote Parts of the Cavity

As described in Section 4.2.2 the cavity can be expanded remotely. The cavity expansion may well go to more than one remote processor or expand on the same processor more than once. All these parts of the cavity have to be collected at the processor where the new point is located. Then the new triangles can be created at that processor.

Such a part of the cavity is always sent with a header which contains data like the size of the cavity, the number of triangles etc. The routine which collects the parts of the cavity needs to know when the cavity is complete. There is no global instance which initiates remote cavity expansion but it can be done by every process who is involved in the cavity expansion. To keep track of all these parts the following data structure is sent along with the header:

remote flow control

- ID (a unique number) of the part of the cavity where the remote request for expanding the cavity was coming from (father).
- ID (a unique number) of the current part of the cavity.
- Number of remote requests for expanding the cavity initiated by the current part of the cavity (number of sons).

Each call for a remote cavity expansion gets a unique number. When the cavity is expanded as far as locally possible its data are sent to the processor where the new point lies. The record shown above is located in the header of this message. The first field is the unique number of this part of the cavity expansion. The second field contains the unique number of the part of the cavity expansion where the current request came from. The third field denotes the number of remote cavity expansions which are initiated by the current part. This information is sufficient for the collection routine to determine whether all parts of the expanded cavity are received.

For example, a processor receives a part of the cavity. This processor knows that the cavity cannot be seen as completed before the father, described in the "remote flow control" record is received as well. And there also have to be received two parts with the current part of the cavity as father in order to complete the cavity.

4.3 Parallel Mesh Generation Using Constrained Delaunay Triangulation

4.3.1 Constrained Delaunay Triangulation

Assume a given set of points in a plane together with a set of non-crossing, straight-line edges the *constrained Delaunay triangulation* [9] is the triangulation of the points with the following properties:

- The pre-specified (constrained) edges are included in the triangulation.
- The circumcircle of each triangle does not contain any other point unless it is hidden behind a pre-specified edge. This means that if line segments are drawn from each point of the triangle to that other point, at least one of these line segments crosses a pre-specified edge (Figure 4.5).

Pre-specified or constrained edges cannot be removed. At mesh generation they are handled similar to boundary edges, points on one side of the edge do not have any influence on the triangulation on the other side of the edge.

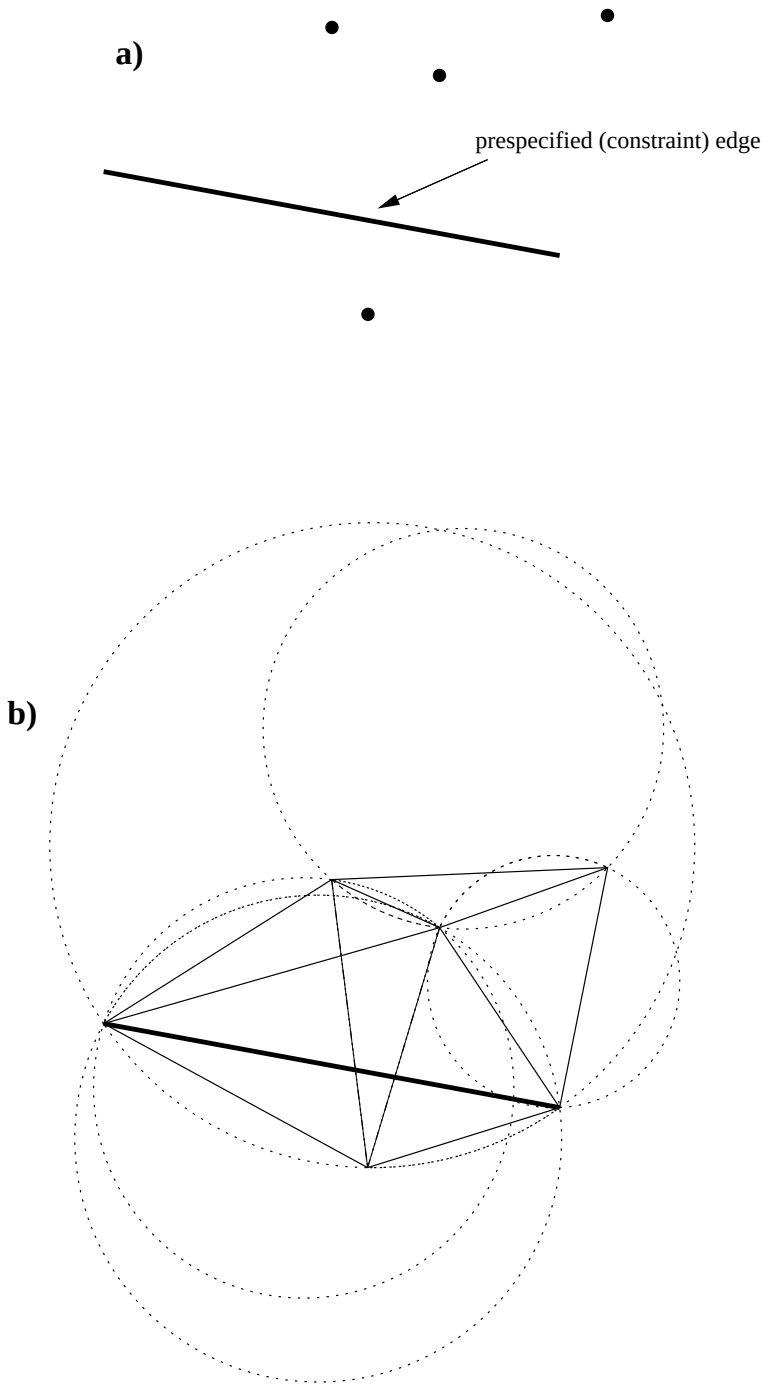

Figure 4.5: The constrained Delaunay triangulation: (a) A set of four points and one pre-specified (constrained) edge is given. (b) Circumcircles of triangles may contain another point only when that point is hidden behind the pre-specified edge: The dotted line segments from at least one point of these triangles have to cross the pre-specified edge.

An exception is the fact that such a constrained edge can be split. This is needed for reaching good mesh quality as described in Section 2.2.1. In this case a point insertion on one side of the edge may cause the split of the edge and then, indeed, a change of the triangulation on the other side of the edge.

4.3.2 Use of Parallel Constrained Delaunay Triangulation

As explained in Section 4.1 generating the entire Delaunay mesh in parallel as one piece is a quite complex task. This results in increasing communication and implicit synchronization as well as in a greater number of special cases to be considered at designing such a program.

Communication is done asynchronously and, thus, does not block a process but, nevertheless, it is expensive and decreases performance. The implicit synchronization, mentioned above, is caused by cavity collisions. In this case one cavity has to be released and recomputed at a later point of time.

Triangulating a mesh in the constrained way means much less communication, no synchronization and much less effort to implement the algorithm [11]. Nevertheless the *constrained parallel Delaunay* algorithm produces a high quality mesh which, depending on certain parameters (described in Section 4.3.5), can be Delaunay, too. Results of runs of both algorithms, the *entire mesh* algorithm and the *constraint mesh* algorithm, are discussed in section 4.4.

4.3.3 The Data Structure

In the "entire mesh" version the border between processors are along an underlying auxiliary grid (Section 4.2.1). In this way elements can overlap grid lines and be stored distributed.

In the constrained Delaunay triangulation the border between processors is a graph of edges. Such a border edge is duplicated and stored at two processors. Since triangles cannot overlap borders between processors, they are stored completely at their processor and they go along with local pointers for references in their data structures.

A triangle contains three local pointers to its edges. A regular edge contains two local pointers to sites and two local pointers to triangles while a border edge owns only one local pointer to a triangle and a global pointer to the second instance of the edge at the neighboring processor. A site contains a list of local pointers to edges. Sites at the border between processors are duplicated but they are not marked as border sites. They only store pointers to local edges in their edge lists and only can be determined as border site by checking if there is a pointer to a border edge.

Border edges are considered as constrained. Hence, they cannot be removed, only be split. This means they are handled like boundary edges with the only

difference that, in the case of splitting a border edge, a split of its duplicate on the neighboring processor has to be initiated, too.

4.3.4 Algorithm for Parallel Constrained Delaunay Triangulation

Overview

The main differences to the algorithm of the "entire mesh" version, as described in Section 4.2.2, is that the cavity expansion never has to be stopped because of a collision with another cavity and, thus, triangles do not have to be marked as in use for synchronization. The cavity even may hit a border edge to another process and still does not need to interact with that process. Border edges are constraint edges and, therefore, points of another process are "hidden" behind border edges. Only the case of a border edge split causes a request to the neighboring process to split its instance of the border edge as well.

Details of the Algorithm

The following algorithm generates a quality mesh using the *constrained Delaunay triangulation*:

PROCEDURE parallel_CDT

Read in sites and triangles of the initial triangulation and distribute them to the processors.

Do until (*the following condition is checked at loop end*) all infeasible triangles are deleted and no system work (i. e., serving remote requests) left. Note: Remote service requests may produce infeasible triangles.

Do while an infeasible triangle is available

Mark the triangle as in use.

Try to insert its circumcenter, call *insert_point*.

Serve remote requests.

End do

Serve remote requests.

End do

PROCEDURE *insert_point*

Expand the cavity: Start at the infeasible triangle, which, by definition, is in conflict with the new point (circumcenter or center of a boundary edge to be split). In a recursive way check the neighbors of the triangle whether they are in conflict with the new point.

Switch on the result of the cavity expansion

Case [Cavity could be expanded completely]

Remove old triangles and create new ones. This can be done locally and does not need any interaction with other processors.

Case [The new point is too close to a boundary or border edge]

Split this edge. Release all triangles and start again point insertion with the center of this edge as the new point, call *insert_point*.

If the edge, which the new point is too close to, is a border edge

Then initiate a remote split of this edge and a remote point insertion with the center of the split edge at the neighboring process, call *insert_point* remotely.

End if

End switch

Figure 4.6 shows a finished triangulation with the triangulation in Figure 4.3 as start point where the parallel constrained Delaunay triangulation was applied on. The entire mesh is still Delaunay due to certain parameters described in Section 4.3.5. In contrast to Figure 4.6 Figure 4.7 shows the four parts of the initial triangulation meshed on their own.

4.3.5 Implementation Details

Splitting a Border Edge

The parallel constrained Delaunay triangulation does not need synchronization between processes. The only message sent to a remote processor is to split a

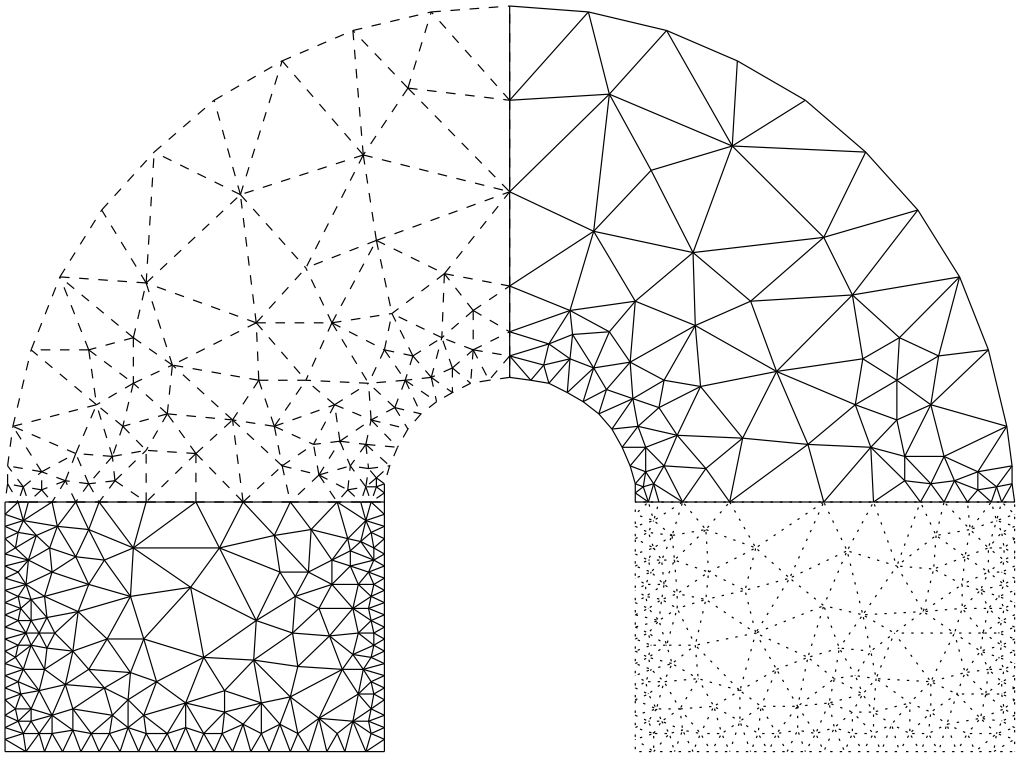

Figure 4.6: The parallel constrained Delaunay triangulation.

border edge. The data structure of a border edge contains the following data (the source code can be found in Section A.3):

- Two local pointers to two sites
- A local pointer to a triangle
- A global pointer to the instance of this edge at the neighboring process
- Two local pointers to the two halves of this edge in case it is already split
- A local pointer to the "father" edge

If a border edge has to be split two new edges are created. Their data structure is filled, except the pointers to their second instances at the remote process because they are not known at that point of time. After having split the local instance of the edge a remote split is requested. When the remote split has been done the pointer to the remote split edges is sent back to the local processor.

If a local split is necessary before these pointers are available the algorithm has to step down through this tree of split edges until an edge can be found which contains already a pointer to its remote instance. The path through this tree is sent with the split request to step up the tree at the remote processor.

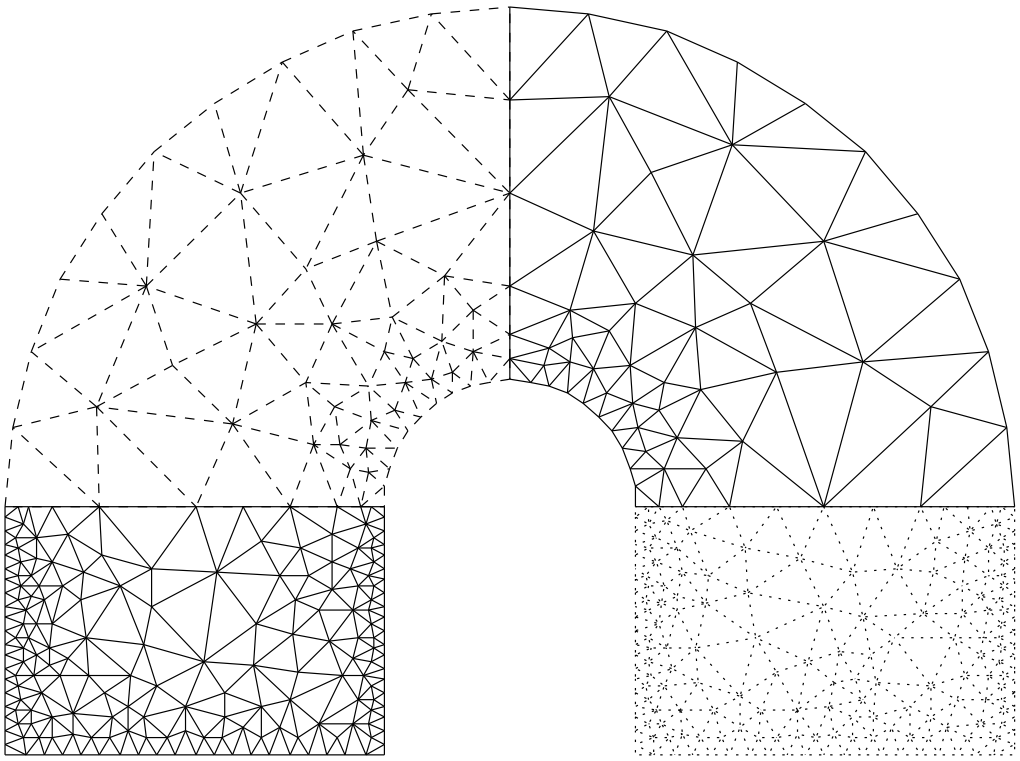

Figure 4.7: Triangulation of the parts of the initial initialization on their own.

PROCEDURE *split_edge*

Allocate space for two edges. Then fill the entries of the border edge records. The pointers to the second instances at the remote process of these edges are assigned NULL. As well the pointers to the new edges.

Enter the address of the new edges at the corresponding entries of the current edge.

Do while the current edge has not got a pointer to its remote instance, yet, do:

Go to the father's edge. Find out which one of the new edge pointers of the father edge points to the current edge. Keep this information stored (path).

Set the father's edge as current edge.

End do

A pointer to the remote instance of a predecessor of the new edges is now available. Initiate a split of the remote instance of the edge, call *remote_split_edge* remotely. As arguments send the available address of the remote instance of the found edge and the path to get up again on the top level to find the edge which actually has to be split.

PROCEDURE remote_split_edge

Get up in the border edge tree starting from the given edge following the path.

If the edge has been split locally, yet

Then enter the pointers to the remote instances (these pointer were sent along this request).

Else allocate space for two border edges. Fill the entries of their records.

Send the pointers of the new edges back to the original processor, call *remote_instance_assignment* remotely.

End if

PROCEDURE remote_instance_assignment

Assign the sent pointers to the corresponding entries of the specified edges.

Keeping the Mesh Delaunay

Point insertions do not expand their cavity beyond constrained edges. Therefore, if a constrained edge is changed to a regular edge, the triangulation might not be Delaunay anymore as shown in Figure 4.8.

For computing a triangulation in parallel it might be useful and necessary for load balancing to shift the border in order to enlarge the submesh of one processor and to reduce the submesh of another one. In such cases it is desirable to keep the mesh entirely Delaunay.

Keeping the entire mesh Delaunay can be controlled by the criteria of when a new point to be inserted is *too close* to a border edge. If *too close* means that α in Figure 2.6 is greater than $\pi/2$ (this is equal to the new point not lying in the smallest circle going through the end points of the edge, called border

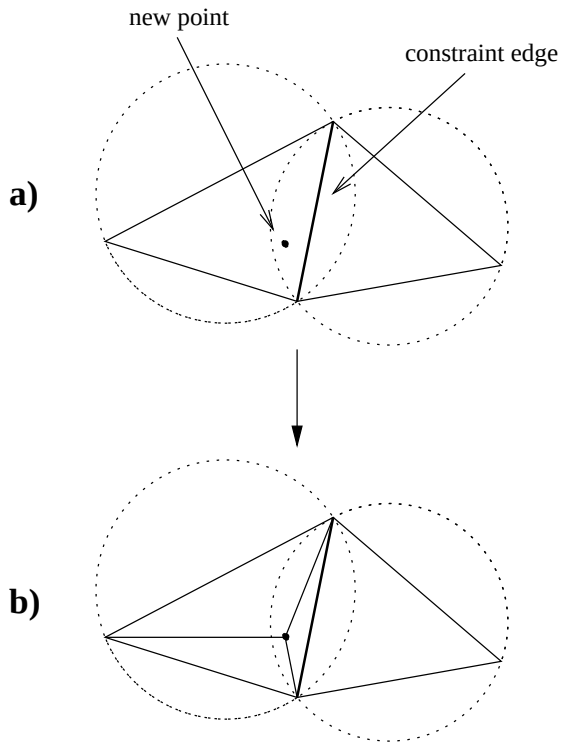

Figure 4.8: If a new point is inserted in a Delaunay triangulation with a constrained edge (a) and the edge is not split then the mesh might not be Delaunay anymore from the point of view seeing the mesh as entire one (b).

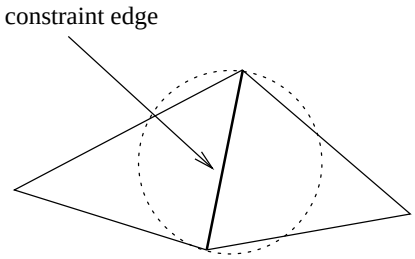

Figure 4.9: The entire mesh stays Delaunay as long as no other points lie within the circle which the constrained edge is diameter of.

edge circle), see Figure 4.9, the mesh will stay Delaunay. This results from the fact that the circumcircle of a border triangle will not overlap any remote area outside the border edge circle. And, since, the area within the border edge circle is free of any site the Delaunay constraint ("no site within the circumcircle of any triangle") in not violated.

But it has to be considered that the path of border edges between two processors must not have angles smaller than $\pi/3$. If there are angles smaller than $\pi/3$,

this path has to be improved which can be done by using the "angle improvement algorithms" shown in Section 4.3.6.

4.3.6 The "Angle Improvement" Algorithm

Overview

The constrained Delaunay triangulation provides a possibility for efficiently computing a quality mesh in parallel. But before computing the mesh the input mesh has to be distributed to the processors.

Several mesh partioning software packages exist: Chaco [34], METIS [37], PARTY [45], JOSTLE [52] and SCOTCH [43]. But they all do not care about quality of the mesh partioning path (see paragraph below). The algorithm for generating meshes does not allow edges to embed angles less than the minimum angle for any quality triangle inside the mesh (see Section 2.2.1). Border edges between adjacent parts of the distributed mesh are constrained and, hence, handled like boundary edges. Therefore, border edges also have to satisfy the same constraints for angles as the boundary edges.

In this section two algorithms are presented which improve the mesh partioning path such that the angles at that graph are not smaller than a given angle (in general the given minimum angle of the triangles in the mesh).

The Mesh Partitioning Path

The results of Delaunay mesh partioning are paths which divide one submesh from another. Such a path consists of edges. The mesh is divided by these edges. They belong to both adjacent submeshes, each of them holds an instance of each edge.

Each edge embeds angles with its two neighbors (the edges at the ends of the path have only one neighbor each), each angle must not be smaller than a given limit, a user specified restriction on mesh generation.

Since mesh partioning algorithms do not take care of that restriction in general, a treatment afterwards may be necessary to improve the mesh partitioning path. The two following algorithms, developed by the author, work for minimum angles specified up to $\pi/3$.

The "Alternative Path" Algorithm

The Algorithm. If two edges are found which embed an angle β smaller then the given limit, an alternative path has to be found. Let's denote the two edges as a and c with their common endpoint B and A and C as the other end points, respectively.

a and c embed one or more triangles containing B as one corner. Each of these triangles consists of one edge which does not contain B. The new path goes

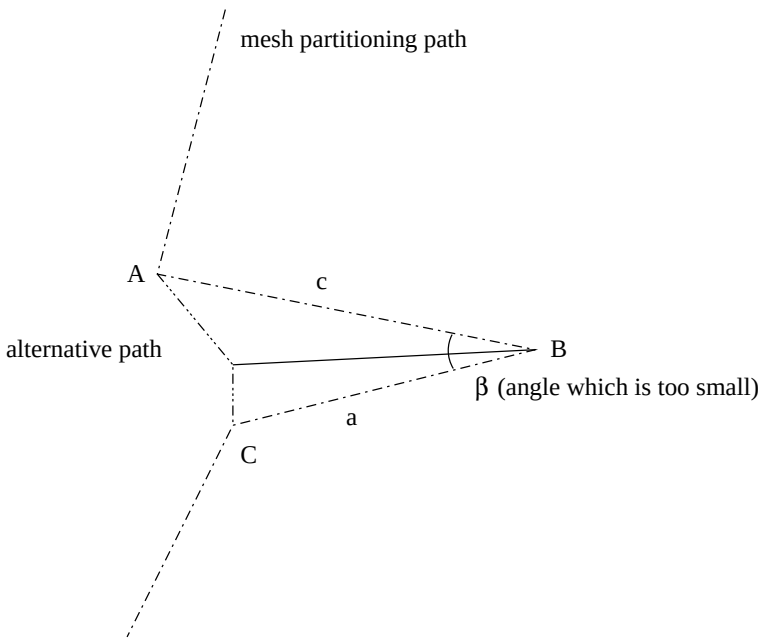

Figure 4.10: Applying the "Alternative Path" algorithm for improving the mesh partitioning path removes infeasible angles by finding an alternative path. In this figure β is too small. Thus edges a and c are replaced by two other edges which embed a greater angle.

along these edges (see Figure 4.10). This procedure has to be applied as long as the partitioning path contains infeasible angles.

If the partitioning path is a winding one the replacement path for two edges with an infeasible angle might touch other edges of the partioning path that could results in a dead end (see Figure 4.11). This has to be cut off of the partitioning path. This special case does not effect the proof in the next section since no other edges are used than the one already in use.

Proof of the Algorithm. In order to verify this algorithm two things have to be proofed:

- There is an alternative path containing no infeasible angles.
- The algorithm will find such a path.

Theorem: *If the angle, embedded by two adjacent edges of the path, is smaller than $\pi/3$ the new path between the two non common endpoints contains only edges shorter than the longer one of the original edges.*
Proof: Figure 4.12 shows two edges, a and c, of the partitioning path. c is assumed to be the longer edge. The dashed circle is the circle through A, B and C, assuming a has the same length as c. The new path is going from A to C.

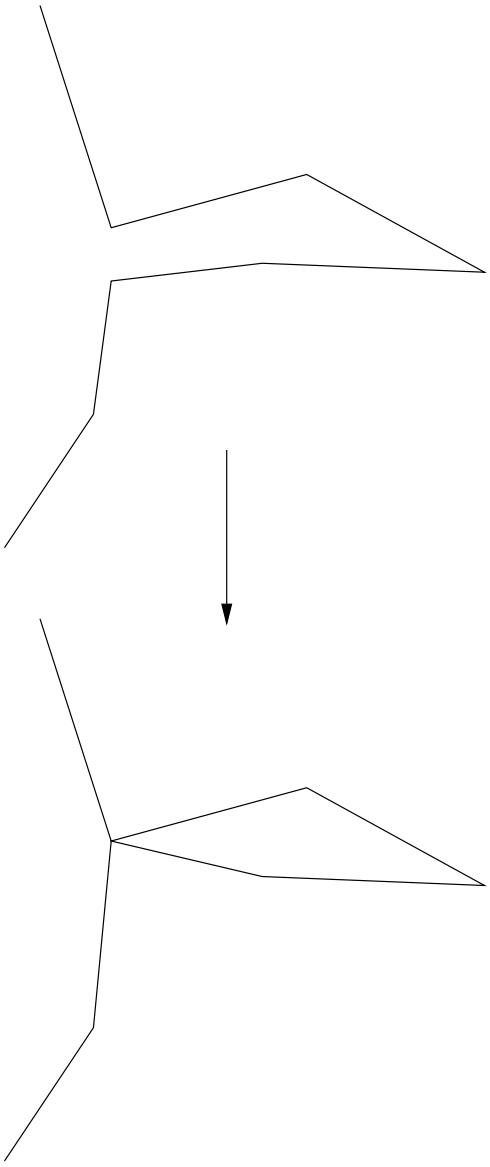

Figure 4.11: The partioning path improvement algorithm might produce dead ends which have to be cut off.

All vertices of that path have to be within the dashed circle otherwise A or C were in the circumcircle of the embedded triangle which contains the vertex outside the dashed circle. This would be a contradiction to the Delaunay constraint. Thus, also all new edges have to be within the dashed circle.

An edge in the new path which is longer than c has to have one vertex v_0 outside of the solid circle which has its center in B and the length of c as radius. The other vertex v_1 has to be outside the dotted circle. This circle has c as diameter. This circle results from the following consideration: The new edge

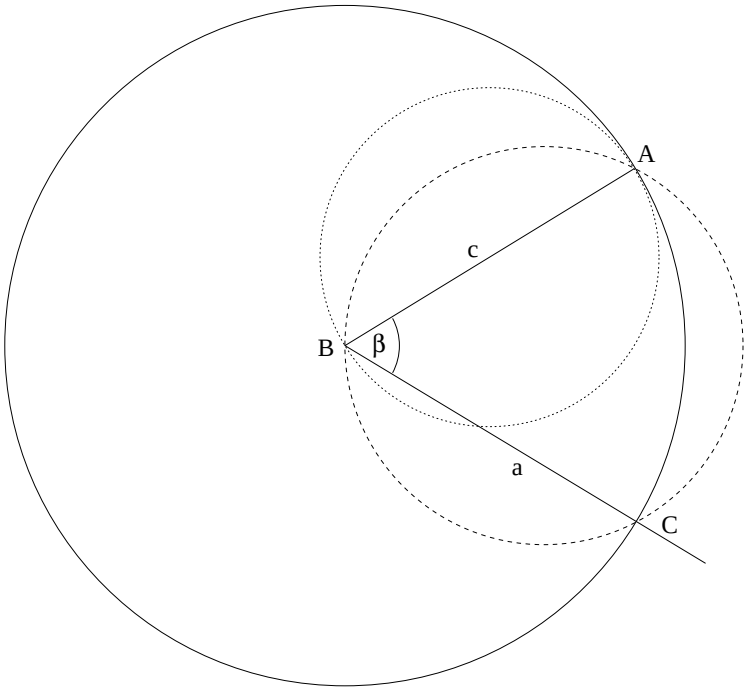

Figure 4.12: Two edges of the partitioning path with angle $\beta = \frac{\pi}{3}$.

(which is longer than a) is part of a triangle defined by this edge and B. The circumcircle of that triangle must not contain A, this constraint would be violated by putting v_1 within the dotted circle. In order to get a lower bound the dotted circle only represents the minimum area to be excluded.

It's easy to recognize in Figure 4.12 that, in order to maximize the length of the new edge, v_0 has to be as close as possible to A while v_1 has to be in one of the other two corners of the allowed area. In case v_1 is assumed to be in the corner of C and β is $\pi/3$, such that A, B and C would result in an equilateral triangle, the new edge would be just as long as c. This can be considered to be the exclusive upper bound. In case v_1 is at the corner where the dotted circle intersects a the new edge would be the height of that triangle and shorter than c, too.

If A and v_0 are identical the dotted circle rule does not apply. Then v_1 can be close to C and the exclusive upper bound for the length of the new edge is the length of c, again.

For a assumed to be shorter than c only the upper bound is lowered for the case where v_1 is close to C. Proof done.

In that way two adjacent edges in the partitioning path which include an angle smaller than $\pi/3$ can be replaced by a path of edges each of them shorter than the longer original one. Assuming the mesh consists only of a finite number of elements the algorithm has to halt latest by reaching the smallest edge size.

Cutting off a dead end does not increase the maximum edge length of the path.

If the algorithm halts a path is found only with angles greater than $\pi/3$. Otherwise the theorem could be applied again.

Side Effects The partioning path may wander by applying that algorithm. It might touch the mesh boundary or even other partioning paths. But it will never intersect them in the sense that the subdomains will overlap. This results from the consideration that the constraints are the same for all paths. But it might divide a submesh into non contiguous parts. However, this seems to be rather improbably since the algorithm straightens the path in general. Nevertheless it has to be considered at implementation.

The "Improvement by Mesh Refinement" Algorithm

This approach basically runs the mesh refinement on a single processor as long the partitioning path contains infeasible angles. In order to improve the partioning path two edges with an infeasible angle are detected. An embedded triangle is chosen. It has to contain one of these edges. This triangle is refined by inserting its circumcenter or by splitting the mesh boundary if close enough.

At this refinement a cavity is created. The section of the partitioning path, which begins at the first touch with the cavity and ends where it leaves the cavity, is replaced by two edges: One from the point where the partitioning path enters the cavity to the inserted vertex and one from that vertex to the point where the partitioning path leaves the cavity. Figure 4.13 shows applying this algorithm. This procedure has to be repeated as long as there are infeasible triangles in the partitioning path.

This algorithm works because Paul Chews algorithm halts and produces a mesh which satisfy the users constraints regardless of the sequence of triangles refined. Hence, after some point of time all triangles along the partitioning path satisfy the constraint of the minimum angle and, therefore, the partitioning path does so, too.

4.4 Parallel Entire Mesh vs. Parallel Constrained Delaunay Triangulation

Four series of runs were performed to compare these two algorithms. Two series were done on an SP-2, shown in Table 4.2. The other two series, shown in Table 4.3, were performed on a cluster of PCs (see Section 7.1).

These runs show that the use of the *entire mesh* algorithm takes more time than the use of the *constrained mesh* algorithm. This is true for comparing the two series of runs on the SP-2 and the other two series on the cluster of PCs.

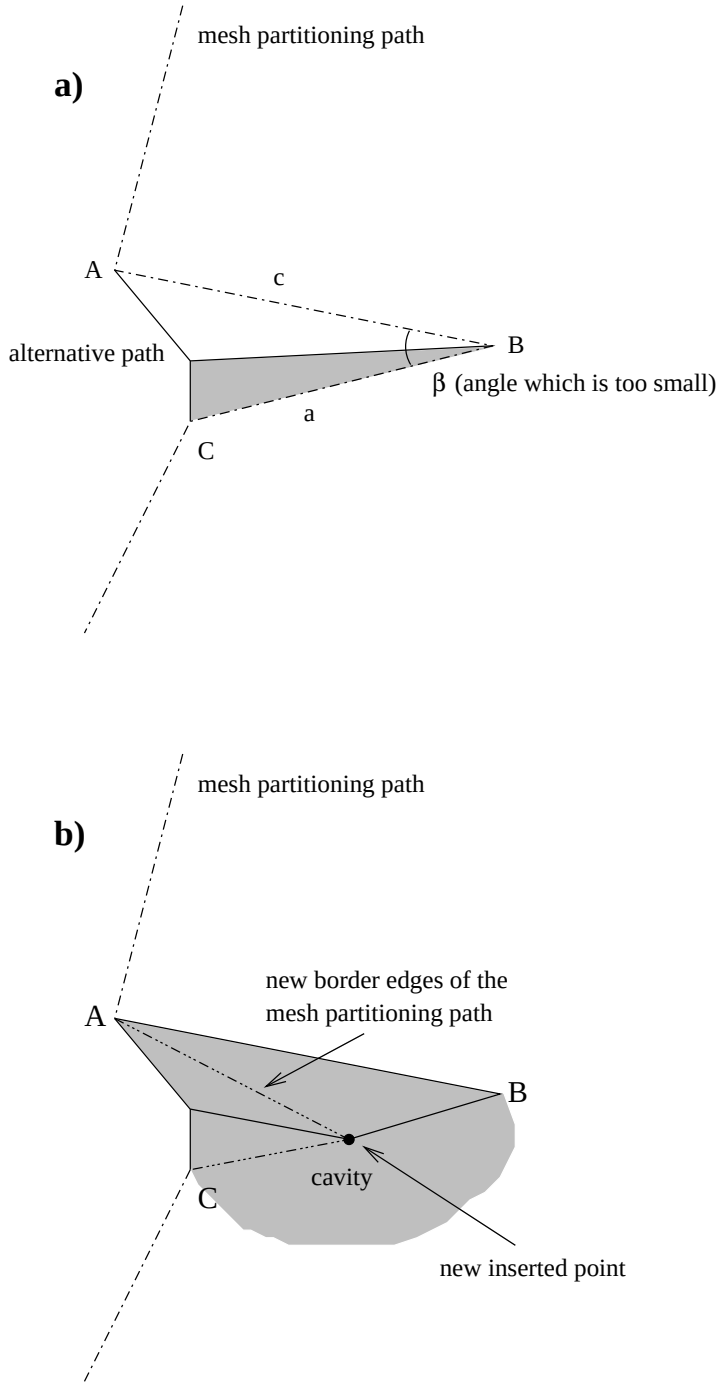

Figure 4.13: Applying the "Improvement by Mesh Refinement" algorithm for improving the mesh partitioning path removes infeasible angles by refining the mesh and putting the path through the new point. At (a) β is too small. An embedded triangle is chosen. At (b) edges a and c are replaced by two new edges which are formed by inserting the circumcenter of the chosen triangle.

In both comparisons the execution time is significantly higher at the runs which compute an entire mesh than the runs computing the constrained Delaunay triangulation. The higher run times correspond to the higher number of remote service requests to be sent when computing an entire mesh.

Runs on the SP-2 and runs on the cluster of PCs cannot be compared due to following reasons:

- Communication between the nodes of the SP-2 is much faster than between the PCs of the cluster.
- The layer is implemented on top of Active Messages at the SP-2, PVM is used on the cluster.
- The main memory of the PCs is much more capable than the one of the nodes of the SP-2. This causes the problem size to be chosen higher on the cluster of PCs than on the SP-2.

Nevertheless, the objective of comparing both algorithms was reached by these runs.

Another circumstance seems to be remarkable at the runs on the PC cluster: Each PC is equipped with two processors. Parallel processes can be distributed in two ways. (i) each process runs alone on a PC, (ii) two processes are assigned to a PC running on its two processors. For example: Four parallel processes can be distributed to four or two PCs using only one or both processors in each PC, respectively.

Running two processes on two processors in a single PC instead of using two PCs reduces communication overhead, since, messages can be sent via shared memory. Looking at Table 4.3 the opposite seems to be true, i.e., the run where four processes are distributed to four PCs are significantly faster than the the run where four processes are using only two PCs.

This can easily be explained by the fact that the triangulation algorithms frequently access main memory. This means that the memory bus is the bottleneck in such a constellation. Indeed, processes which mainly access the cache instead of the common main memory will show speed up compared to a distribution where each process runs on an extra PC.

Number of Nodes	2	4	8
Execution Time (in sec)	4.38	2.89	1.91
Number of RSRs	516	807	866

The triangulation as *entire mesh*.

Number of Nodes	2	4	8
Execution Time (in sec)	3.59	2.29	1.48
Number of RSRs	63	126	126

The triangulation using the *constrained Delaunay meshing* algorithm.

Table 4.2: Computing the triangulation as *entire mesh* and using the *constrained Delaunay meshing* algorithm *on an SP-2* using up to eight nodes. The total execution time and number of remote service requests sent by *each* processor are shown in this table. The size of the mesh reached 22K sites and 43K triangles.

Number of Processors	1	2	4	4	8
Number of PCs	1	2	2	4	4
Execution Time (in sec)	117	83	71	58	58
Number of RSRs	0	971	1439	1441	1812

The triangulation as *entire mesh*.

Number of Processors	1	2	4	4	8
Number of PCs	1	2	2	4	4
Execution Time (in sec)	117	61	48	41	40
Number of RSRs	0	127	191	191	222

The triangulation using the *constrained Delaunay meshing* algorithm.

Table 4.3: Computing the triangulation as *entire mesh* and using the *constrained Delaunay meshing* algorithm *on a cluster of PCs* (see Section 7.1). There are at most four PCs involved with either one or two processors in use on each PC. The total execution time in seconds and the average number of remote service requests sent by *each* processor are shown in this table. The size of the mesh reached 122K sites and 244K triangles.

Chapter 5

Asynchronous Programming Tools

In this chapter an overview of the current available asynchronous programming tools is given: Active Messages, LAPI, PORTS and DMCS. All these tools provide *remote service requests*, *put* and *get* operations. Operations of this kind are typical for asynchronous programming as described in Chapter 3. Commands like *send* and *receive* are not provided. Those belong to the "synchronous world".

Active Messages provides a handy set of functions which are easy to use. LAPI is not published, yet, and should replace Active Messages on IBM parallel computers, since, support for further updates of Active Messages seems to be denied. PORTS, which can be seen more as a concept as a completely defined tool, and DMCS, as an implementation of PORTS, provide a slightly larger set of commands. All these tools are on a very low level and, out of that reason, only of limited use at big programs. This issue will be covered in the next chapters.

5.1 Active Messages

The *Active Messages* layer [7, 20] supports asynchronous communication between instances of parallel programs. It contains the following five basic operations to be called with their arguments:

request:

- destination processor
- handler routine
- 0 to 4 integer arguments

The invocation of *handler routine* is requested at *destination processor*. *handler routine* is called with up to four integer arguments additionally to the standard argument, the requesting processor.

reply:

- requesting processor
- handler routine
- 0 to 4 integer arguments

The *reply* operation works in the same way as *request* except that it only may be called from a *request handler routine*.

store:

- destination processor
- source (local)
- destination (remote)
- size of block
- handler routine
- integer argument
- (end function)
- (two end function arguments)

The *store* operation transfers a block from the local *source* to the remote *destination*. After finishing the transfer *handler routine* is called at the remote processor with one user defined integer argument additionally to the standard arguments which are the requesting processor, the *destination* address and *size of block*.

There are two versions of *store* operations. One blocks until data are sent and *source* is usable again, the other one exits immediately and calls an *end function* after completion of sending. The end function is called with two user defined arguments.

get:

- source processor
- destination (local)
- source (remote)
- size of block
- handler routine
- integer argument

The *get* operation works like a request for a *store* operation from *source processor* to the local one.

poll: *Polling* has to be done frequently in order to keep the network clear. This operation gets all pending messages and calls their *handler routines*.

5.2 LAPI

The Low-level Applications Programming Interface (LAPI) [48] is IBM's new communications applications programming interface. Like *Active Messages* LAPI is a thin layer above the hardware abstractions layer and includes reliability and flow control. LAPI uses a basic "*Active Message* Style" mechanism which provides a one-sided communications model. The library provides a relatively small set of functions close to *Active Messages*.

5.3 PORTS

The PORTS [19] working group took the first step toward a common run-time specification for compilers by developing the PORTS0 thread interface. The PORTS1 interface for communication, etc. is being developed. Guidelines have been worked out.

DMCS is a kind of implementation of these guidelines and, therefore, rather similar to what PORTS1 will be when finished. Hence, DMCS is described instead of PORTS done in Section 5.4.

5.4 DMCS

DMCS ([12]) is an implementation of the guidelines worked out by the PORTS working group and some extension. In this section the relevant parts for this work will be described.

Global Pointers Module. DMCS provides the notion of a global pointer through which remote data can be accessed. When a program using DMCS runs on N processors, each processor is assigned an unique integer id (known as a *dmcs_context*, and returned by the routine *dmcs_mycontext()* in the range $0 \ldots N-1$. Any processor can access local data through a regular pointer. However, any remote data must be accessed through a *global pointer*. A global pointer consists of a *dmcs_context* and a local pointer. Routines are provided to make a global pointer out of a local pointer and a *dmcs_context*, and also to extract these fields from a global pointer. This module is the

only module that depends on the homogeneity of the underlying hardware (in the determination of unique context numbers).

Acknowledgment Variables. Since DMCS is asynchronous by nature, a mechanism is provided to enable programs to find out about data transfer completion and to synchronize processes. This mechanism is an acknowledgment variable. An acknowledgment variable is represented as a 16-bit in-

teger for efficiency reasons. An acknowledgment variable can have three states: *cleared*, *set* and *uninitialized*. To use an acknowledgment variable, a program must first request one using the routine *dmcs_newack()*, which returns an unused acknowledgment variable. This return value can be used as a handler to perform various operations on the acknowledgment variable. For example, *dmcs_testack()* checks if the acknowledgment variable has been set or not, and returns immediately. On the other hand, *dmcs_waitack()* waits until the variable in question has been set. It is also possible to clear an acknowledgment variable using *dmcs_clearack()*. Finally, it is possible to "anticipate" the use of an acknowledgment variable in future data transfer using the function *dmcs_anticipateack()*. The last routine has an important role in one-sided data transfer operations, which will be described later. Finally, it is possible to use the same acknowledgment variable in conjunction with multiple data transfer operations. Use of an acknowledgment variable is always optional and a value of NULL for an argument of this type indicates that the particular acknowledgment variable in question is not being used.

Get and Put Operations. DMCS provides routines for one-sided communication. In other words, communication does not happen through a pair of matched sends and receives. Instead, get (*dmcs_get*) and put (*dmcs_put*) routines are provided for fetching remote data and storing remote data respectively. Both these operations are inherently asynchronous. Acknowledgment variables can be used to determine the state of one of these transfers: A Get operation transfers data from a source specified by a global pointer to a destination specified by a local pointer. A single acknowledgment variable passed as an argument to this routine is set when the data transfer operation is complete. A Put

operation transfers data from a local buffer specified by a local pointer to a remote data buffer specified by a global pointer. A put operation has associated with it three acknowledgment variables: a *local_ack*, which is set when the local data buffer can be reused by the application program, a *remote_ack*, which is set on the processor that initiated the put operation to indicate that the put operation on the remote processor is complete, and finally a *remote_remote_ack* which is set on the remote processor to indicate that the put operation there has been completed. For the remote-processor intimation to work correctly, it must first anticipate this put operation by calling *dmcs_anticipateack()* on the acknowledgment variable specified as the *remote_remote_ack* in the put operation. As mentioned previously, all these acknowledgment variables are optional and a value of NULL can be

passed as the corresponding argument to indicate disinterest in that particular acknowledgment variable.

Remote Service Requests. DMCS provides several kinds of remote service requests. A remote service request (*dmcs_rsr*) consists of a remote context, a function to be executed at the remote context and the arguments to the function. In

addition, a type argument is also passed, indicating the type of the remote service request. This type argument can take three possible values, and determines how the function at the remote end is executed. The function can be executed immediately on arrival, or it can be threaded. If the handler is threaded, then it can be executed as either a low-priority thread or a high-priority thread. All high-priority threads are served before any of the low-priority ones.

Finally, it is common in programs to transfer some data to a remote processor, followed by the invocation of a remote service request acting on the data just transferred. For such operations, it is more efficient to do the data transfer and the remote service invocation in one step, rather than transferring the data first, and then invoking the remote service request. This is because, the second approach usually involves some additional synchronization delays that can be avoided by the first approach. For this purpose, DMCS provides a *dmcs_put_rsr* routine, which combine the data transfer and the remote service action at the end of the data transfer. This is similar to the functionality specified in the generic active message specification.

Chapter 6

A Layer for Asynchronous Communication (LAC)

6.1 The Necessity for this Layer

Existing tools for asynchronous parallel programming as described in Section 5 only provide communication primitives on a rather low level. No powerful primitives are available for easy straightforward parallel programming. The newly developed layer takes the pain away and makes asynchronous parallel programming more comfortable. These improvements will be discussed in the next sections.

6.1.1 Elimination of the Three Way Communication

Three-way communication may be necessary if the communication tools described in Section 5 are used as the following example shows: A remote service request has to be sent to a remote processor. In order to execute this remote service request data (let's say: a single block) have to be transferred. The following steps are necessary (see Figure 6.1):

1. Send a remote service request to the remote processor to allocate memory space for the data block.

2. The remote processor sends back a remote service request that the data block can be transfered to the allocated space.

3. Transfer the data block to the remote processor. Execute the remote service request.

The data block only could be transferred at the beginning if space had been allocated in advance. This fact makes programming inconvenient and means much more work for the programmer.

Using the layer the user does not have to care about this problem. Calling a remote service request the layer even provides two options of having the block transferred to the remote processor: At the first option the layer remotely allocates space and transfers the data. Then these data can be accessed by a pointer which is given as parameter in the remote service request routine additional to the user specified ones. Any further handling of this memory space is in user's responsibility (for example: freeing it). At the other option the layer does not

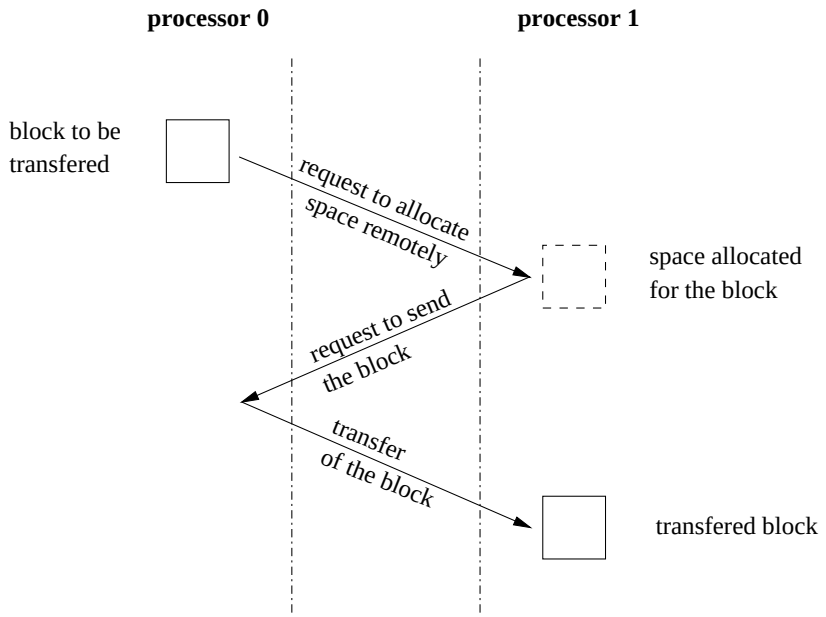

Figure 6.1: Three requests are necessary to transfer a block to a remote processor if the destination space is not allocated in advance.

allocate any extra space but just transfers the data block to its system buffer on the remote processor. Then the user has to read the data immediately because after finishing the remote service request, the buffer is reused by the system. This option means higher efficiency in most of the times because system buffers are preallocated and reused in general and the user has not to take care of the further use of this memory space.

The layer does not only mean more comfort for the user by making block transfers more convenient. It also might reduce communication. The layer works with preallocated system buffers. Therefore, data from one processor can immediately be transferred to the remote processor skipping the first two ways of the three-way communication. However, in case of very big sized data blocks the three-way communication still might be the more efficient way, depending on some system parameters discussed in Section 6.3.

6.1.2 Easy Transfer of the Local Environment

The layer allows calling a remote service request in a more flexible way than current communication tools. The number of arguments are to be defined by the user. This is a significant difference to the current tools where transfer of data only was provided for a very few integer arguments or a data block. Data had to be copied to one contiguous block in order to be able to call a remote service request. At the remote processor this block had to be unpacked.

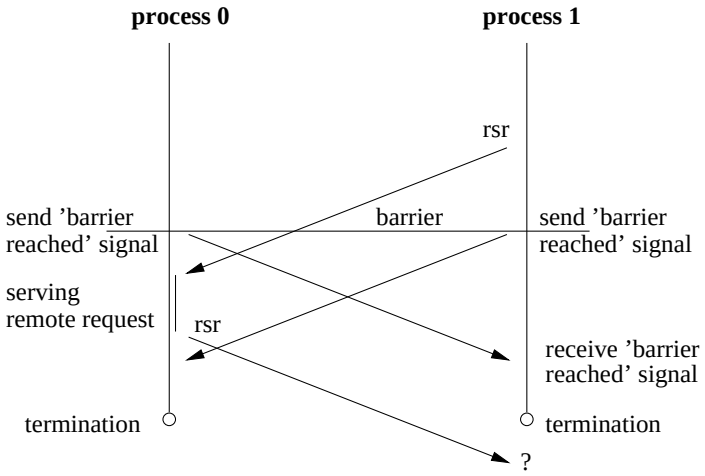

Figure 6.2: A simple barrier (just *send* and *receive*) does not work for asynchronous processes: Process 0 reaches the barrier and sends the "barrier reached" signal. Process 1 sends a *remote service request* and then reaches the barrier with sending the signal, too. After receiving the "barrier reached" signal from process 0, process 1 terminates. Process 0 receives and serves the request from process 1 which causes another *remote service request* to be sent to process 1. But this request goes nowhere, since, process 1 has already terminated.

This layer takes away the pain from the programmer. It provides a remote service request call (exactly specified in Appendix B) which allows the user to specify any number of simple arguments or data blocks to be transfered to the remote processor. In this way an easy transfer of the local environment along with the remote service request is possible.

6.1.3 Final Synchronization

Barriers can be used to synchronize processes. After stepping beyond a barrier every process should know the state of each other process. In the *synchronous parallel programming* model such a barrier can easily be implemented either as command provided by the communication tool or with a combination of *send* and *receive* commands: Each process which reaches the barrier sends a broadcast message to all other processes. After having received messages from all processes the process is allowed to step beyond the barrier.

Asynchronous processes run uncoupled and do not need barriers most of the time. But at the end they need to be synchronized such that each process knows when its job is finished. For example, a mesh is generated in parallel. At the end each process has to print its part of the mesh. When is it allowed to assume its job to be finished and start printing its part of the mesh? Does that concept of a barrier work here as well?

A barrier in the *asynchronous parallel programming* model is not as simple as in the *synchronous parallel programming* model: When a process reaches the

barrier it temporarily finished its work. In the same time a second process sends a *remote service request* and reaches the barrier as well. Receiving the signal from the first process it would expect that the the first process finished its work. But in reality the second process is serving the remote request. Let's assume this *remote service request* causes another *remote service request* to be sent to the second processor. But then this processor has already stepped beyond the barrier and may have already be terminated. Figure 6.2 shows this situation. Conclusion: A more sophisticated barrier has to be found in order to synchronize asynchronous processes.

The user should not take care about that. Thus, the layer provides a kind of barrier, a function whose return value corresponds to one of the two states:

Either all processes have finished their work and all messages which are sent are also received and handled or more work has to done by the local processor. To call this function means a request to pass the barrier. Depending on the return value it is rejected or accepted, respectively.

6.1.4 Debugging

Debuggers are not available for the current asynchronously parallel communication tools described in Section 5. But this layer provides the user with a minimum of debug information as tracing of execution of remote service requests.

First all functions which are used for remote service requests have to be registered. This is done by calling a function at the begin of the program with the function name and a user specified message as arguments. When tracing is activated the message, which was specified at registration, is printed when the remote service request is called. Then this message is printed again on the remote processor saying that the execution of the remote service request just has started, and a third time when the remote service request is finished.

Since output of more than one processors does not come in chronological order it often is hard or even impossible to find out when and where a remote service request was initiated. Therefore, each remote service request gets an unique id which is printed additionally to the user specified messages. In this way the run of the program can be exactly identified. Furthermore, the output could be used as input for a graphical post mortem debugger.

6.1.5 Flushing of Remote Service Requests

Using low level asynchronous communication tools intuitive straight forward programming may be a contradiction to efficiency: Operations often can be defined in a way that it makes much easier to send two or more remote service requests to a remote processor instead of gathering these requests and send them together in a single request. While the first possibility, sending several requests, might be

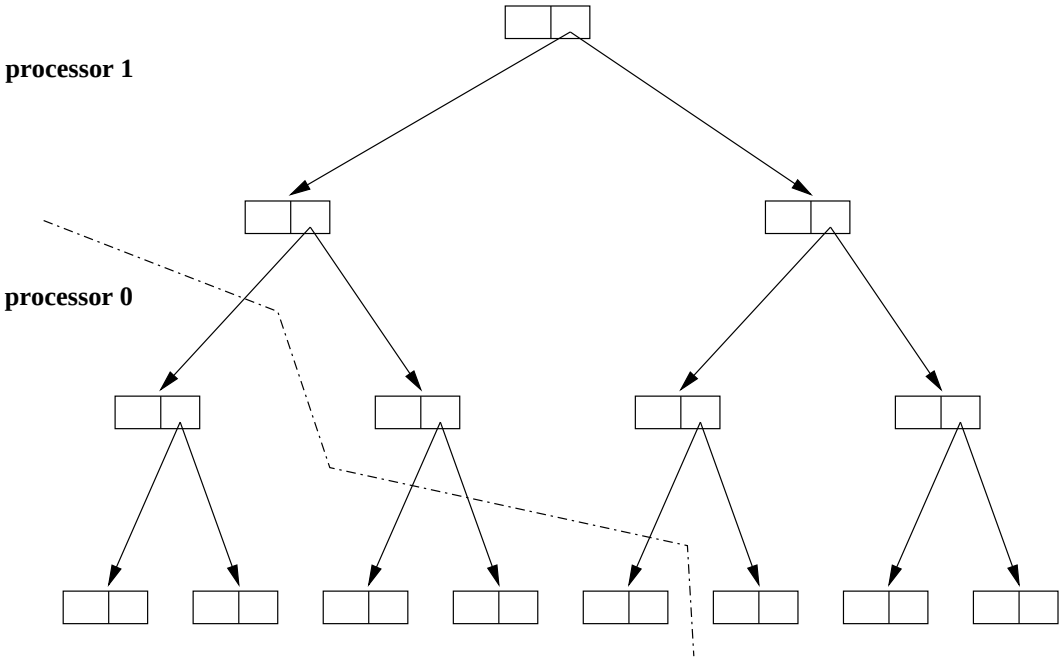

Figure 6.3: 15 numbers are stored in this tree. Each box contains a number and two pointers to its children. Four remote service requests are necessary to add these numbers. Or the information is gathered and sent in only one remote service request. This might increase the efficiency of the program but probably decrease the efficiency of the programmer.

the "easy to implement" way the second possibility, sending everything in just one request, is probably more efficient.

An example: Numbers have to be stored which is done in a tree. This tree is distributed to more than one processors. Let's assume, all numbers have to be added to get the sum as result. Starting at the root all numbers are added recursively. If a subnode of a node in this tree is stored remotely a remote service request is sent to this processor in order to continue adding the numbers of that branch (see Figure 6.3).

In order to be efficient it would be better not to send a remote service request for each remote subnode but to store all requests going to the same processor in a buffer. After finishing all local work all requests for each processor are packed into a single remote service request. This strategy helps to minimize the number of remote service requests and, therefore, makes the program more efficient. Unfortunately this strategy means much more work for the programmer.

The layer helps in that situation. Remote service requests can be sent delayed. This means that a "remote service request"-queue can be opened for each processor. In case of a delayed remote service request call, the remote service request is not immediately sent but stored in this queue. Then the user can flush this queue and all remote service requests are sent at once (internally as one remote service

request). Or the user decides to cancel the remote service requests waiting in the queue. This way guarantees an efficient implementation as well as an easy and straight forward programming.

6.2 The Concept of the Layer

In this section the concept of the layer is presented. The exact specification of it can be found in B. As the other asynchronous parallel programming tools this layer also provides a *remote service request*. This is the basic call to communicate on this layer. *Get* and *put* operations are not available because they are only special cases of the *remote service request*. In difference to the other communication tools an arbitrary number of variables and blocks can be sent along with every *remote service request*. The destination address does not have to be known to transfer a block. *Remote service requests* can be gathered in queues to be sent at once in order to increase efficiency.

As mentioned in 6.1.3 the layer provides a function which can be used as barrier for asynchronous processes. As far as synchronization variables are needed for asynchronous computations the layer provides an object where synchronization variables and functions on them are defined in. Also a minimum of debugging and tracing asynchronous processes is supported. In the following each topic is explained more exact:

***Remote Service Request*:** A *remote service request* is to be called with the id of the destination processor and the remote function as arguments. Further the kind of this request has to be specified. Like in Active Messages [20] a request has immediately, urgent or non-urgent to be executed. Immediately means while polling for *remote service requests*, the urgent *remote service requests* are handled before the non-urgent ones.

Afterwards an arbitrary number of variables can be specified. There are three types of variables: Simple variables as integer, blocks for regular and for direct transfer. Regularly transferred blocks are copied immediately from the user's buffer to a system buffer. This means that the user's buffer can be reused immediately after the call of the *remote service request* returns.

Directly transferred blocks are sent directly out of the user's space to the remote processor. When the block was sent a function, which has to be specified by the user at the call of the *remote service request*, is invoked. For example: An array has to be sent along with a *remote service request* and freed after use. When the *remote service request* is initiated the function returns immediately but if this array is specified as directly transferred block, it can't be freed immediately, since, it is still used from the system. After usage of the array the system calls the user specified function to

indicate that the space can be reused. Then this function should free the space.

In order to transfer a block along with a *remote service request* a destination address has to be specified. This can be any valid address at the destination processor. If that address is unknown, i.e., a space for the block is not allocated, yet, the user may ask the system either to allocate space for that block before transferring it or to put keep it temporarily in the system buffer. At the last option data are lost after the *remote service request* is finished.

Remote service requests do not have to be sent immediately after they are called but they can be stored in a queue. There can be opened a queue for each processor. The user can flush these queues at any point or cancel queues if its *remote service requests* are not needed anymore. While sending several *remote service requests* together may increase performance the later option to cancel *remote service requests* in queues may be very comfortable if sending of *remote service requests* depends on computations at a later point of time.

Synchronization: The layer provides two primitives for synchronizing processes, a function which can be used as barrier and an object which provides synchronization variables and functions on them. The first one is a function which has to be called by all processes in order to synchronize them. It returns 1 if all processes called this function and if there is no messages on the way. Returning 1 at one processor means returning 1 at all other processors as well.

The function blocks until these requirements are satisfied or until a *remote service request* arrives and returns 0. After having do this additional work the process can call the function again for another try to pass the barrier.

The second part provided by the layer is a routine for synchronization variables. These are bit arrays, a bit for each processor. Several operations are provided to handle these synchronization variables. For example, there are functions to set, reset and test these variables.

Polling and Serving Remote Requests: Each process should poll for arriving *remote service requests* frequently. Depending on their type these requests are executed immediately or stored in queues. The point of time when queued *remote service requests* are served is determined by the user.

Debugging and Tracing: The layer provides help at debugging and tracing asynchronous parallel programs. Information about the call and the execution of *remote service requests* are printed if debugging is turned on. Before that each function which can be executed as *remote service request* has to be registered by specifying a message which is one of the things printed at the

call and the execution of *remote service requests*. The following information is printed at calling a *remote service request*:

- The processor id where the remote service request is called.
- The message which was stored at registering the function which will be invoked remotely. In most cases this message will be the name of the function.
- The processor id where the remote service is requested.
- A unique id of this remote service request.

When the remote service request is done at the remote processor, the following information is printed twice, at entering the function and leaving it:

- The processor id where the remote service request is served.
- The message which was stored at registration of the function.
- The processor id where this remote service request was called from.
- The unique id of this remote service request.

Especially the unique remote service request id is very important to trace the run of a program. It might happen, for example, that the printout of entering and leaving a *remote service request* appears earlier than one of the call at the initiating process. Without this unique id it is easy to be left in confusion.

6.3 Implementation Details

Originally the implementation of this layer was done on top of Active Messages, [7, 20], but in order to have compatibility to platforms other than the ones where Active Messages can run on, the layer is now implemented on top of Ports. And Ports is implemented on Active Messages.

6.3.1 Remote Service Requests

This section gives an overview of how the layer works if a remote service request is called.

lac_rsr:

input: see Section 6.2.

1. A buffer, where all data, which are specified by the user as input, are gathered, has to be allocated. Control data are also added to this buffer: Remote processors manage memory at the local processor for having space available to transfer data to this processor. After usage they are free and can be reused. Therefore, this information has to be sent to that remote processor. And for efficiency reason this is made together with this remote service request. Information about freed blocks is obtained by calling *free_local_rec_buf_store*.
2. Having finished assembling the buffer: *transmission_manager* is called to send the remote service request.

free_local_rec_buf_store: The layer uses its own memory management in order to have pre allocated memory available for transfering data to a remote processor. This is used to avoid the 3-way communication, mentioned in Section 6.1.1. Each processor manages memory on each processor. This means that a processor has to allocate memory remotely and keeps control over it in order to transfer data immediately to that processor. This function helps controlling these memory blocks.

input: processor_id, buffer *, block_number, operation

output: return value, field of freed blocks

Depending on `operation` the following actions are taken:

`STORE`: Memory specified by `block_number` and `buffer` of processor `processor_id` are not needed anymore and, thus, have to be stored. If there is already too much freed memory for processor `processor_id` information about this freed memory has to be sent to that processor, even if it cannot be combined with a regular remote service request.

`FREE_STATUS`: The amount of freed memory which is in control of processor `processor_id` is returned.

`TAKE_OUT`: The indices of the first `block_number` memory blocks of processor `processor_id` are written to where `buffer` points to.

transmission_manager:

input: destination_processor, buffer *, buffer_size.

A remote service request has to be sent to the `destination_processor`. `buffer` points to a data block of size `buffer_size` which contains all the data specified by the user and some control data. Can space be found at the `destination_processor`?

yes: The data block is transfered and *rsr_handler* is called remotely.

no: A remote allocation of memory space has to be initiated. The data block where `buffer` points to, is stored in the transmission queue to be sent when space is available at the `destination_processor`.

rsr_handler: Routine which is called for executing the remote service request. Depending on the kind of the remote service request, it is executed immediately or it is put in a task queue. This queue is emptied by calling `lac_do_rsr` as described in Section B.4.

After exiting the remote service request the used memory blocks are given back by calling *free_local_buf_store*.

6.3.2 Synchronization

Asynchronous processes, which communicate with each other, also need synchronization. At least at the end, otherwise one process may terminate while another process still works and cannot get information from that process. Synchronizing asynchronous processes is not a trivial task. As described in Section 6.2 and B.3 the layer provides a function `lac_done`. All processes have to call it and once it returned 1 it is guaranteed that all processes went through this routine, that there is no *remote service request* queued (denoted as task queues in this algorithm) or still on the way.

There are three parts in this algorithm. Part three is the one which compares the number of *remote service requests* sent by each processor to the received ones in order to decide if there is left any more work to do within this layer.

But even to get to part three needs synchronization. This is done by part one and two alternatively. Two parts are necessary because in case of an exit of either part, a message is sent to all other processors. If work is done quickly enough the local processor might already enter this synchronization algorithm before there is a response from any other processor. Therefore, the synchronization routine is entered the other part, i. e., part one if the algorithm was entered from part two before and vice versa. The description of the algorithm sends and receives $S_{0...6}$ which are simple "synchronization messages", comments are in "Sans Serif" style. Figure 6.4 shows the flow control of this algorithm.

PROCEDURE *lac_done*

If your task queues are empty

Then

If you received S_0 from any process
(if other processes or you came from PART 2 or PART 3)

Then call PART 1.

Else call PART 2.

End if

Else go back to work by returning 0.

End if

PART *1*

Do while you have not received S_0 from all processors and your task queues are still empty.

Poll for incoming remote service requests.

End do

If your task queues are not empty anymore

Then go back to work, this means return 0.

Else (you got S_0 from all processors)
reset $S_{0...3}$ (might have been set at PART 2 and PART 3).

Send S_4 to all processors.

Do while you have not received S_4 from all processors and not S_5 from any processor and all your task queues are still empty

Poll for incoming remote service requests.

End do

If you got S_5 from any processor or your task queues are not empty anymore

Then send S_5 to all processors, this means you are leaving for work.

Go back to work, this means return 0.

Else (all processors touched the S_4 barrier)
send S_6 to all processors.

Do while you have not received S_6 from all processors and not S_5 from any processor

Poll for incoming remote service requests.

End do

If you got S_5 from any processor

Then send S_5 to all processors, this means you are leaving for work.

Go back to work, this means return 0.

Else reset S_4 and S_6.

End if

End if

End if

PART *2*

If you received S_5 from any processor
(this means any processor or you left PART 1 for work)

Then

Do while you have not received S_5 from all processors and all your task queues are still empty

Poll for incoming remote service requests.

End do

If your task queues are not empty anymore

Then go back to work, this means return 0.

Else reset $S_{4,5,6}$.

End if

End if

Send S_1 to all processors.

Do while you have not received S_1 from all processors and not S_0 from any processor and all your task queues are still empty

Poll for incoming remote service requests.

End do

If you received S_0 from any processor or your task queues are not empty anymore

Then send S_0 to all processors, saying that you left for work.

Go back to work, this means return 0.

Else send S_2 to all processors.

Do while you have not received S_2 from all processors and not S_0 from any processor

Poll for incoming remote service requests.

End do

If you received S_0 from any processor

Then send S_0 to all processors, saying that you left for work.

Go back to work, this means return 0.

Else reset $S_{1,2}$.

End if

End if

PART *3*

Send the number of remote service requests done for each processor to them.

Wait until having received the number of my remote service requests done by remote processors.

If the sum of these remote service requests differ from the number of remote service requests sent by you

Then send S_0 to all processors, saying that you leave for work again.

Leave for work by returning 0.

Else send S_3 to all processors.

Do while you have not received S_3 from all processors and not S_0 from any processor

Poll for incoming remote service requests.

End do

If you received S_0 from any processor

Then send S_0 to all processors, saying that you leave for work again.

Leave for work by returning 0.

Else reset S_3 for next time calling *lac_done*.

Return 1.

End if

End if

6.3.3 Transfering Blocks Directly

The layer is implemented on PORTS. PORTS does not allow to send more than one data block to be transfered along a remote service request (Right now not even that but standards seem to be going to include that). The layer allows to specify simple and block arguments in an arbitrary number. All these data are gathered at a single block to be sent at once.

As described in Section 6.2 and B.1 the user may specify data blocks to be transferred directly. That can mean an exception on this issue. Big data blocks might be transfered more efficiently if they are transfered separately instead of being copied to a system buffer and at the remote processor being copied a second time from the system buffer to the user's space.

The layer has to decide when to transfer these blocks directly or when to do it the regular way, copying them to the system buffer, transfer, copying them again, if requested. Following, the difference in costs of transfering such a block in different ways will be shown. Depending on the value of the destination address of the blocks which should be transferred directly, the argument `direct_block_dest` in `lac_rsr`, three cases have to be distinguished:

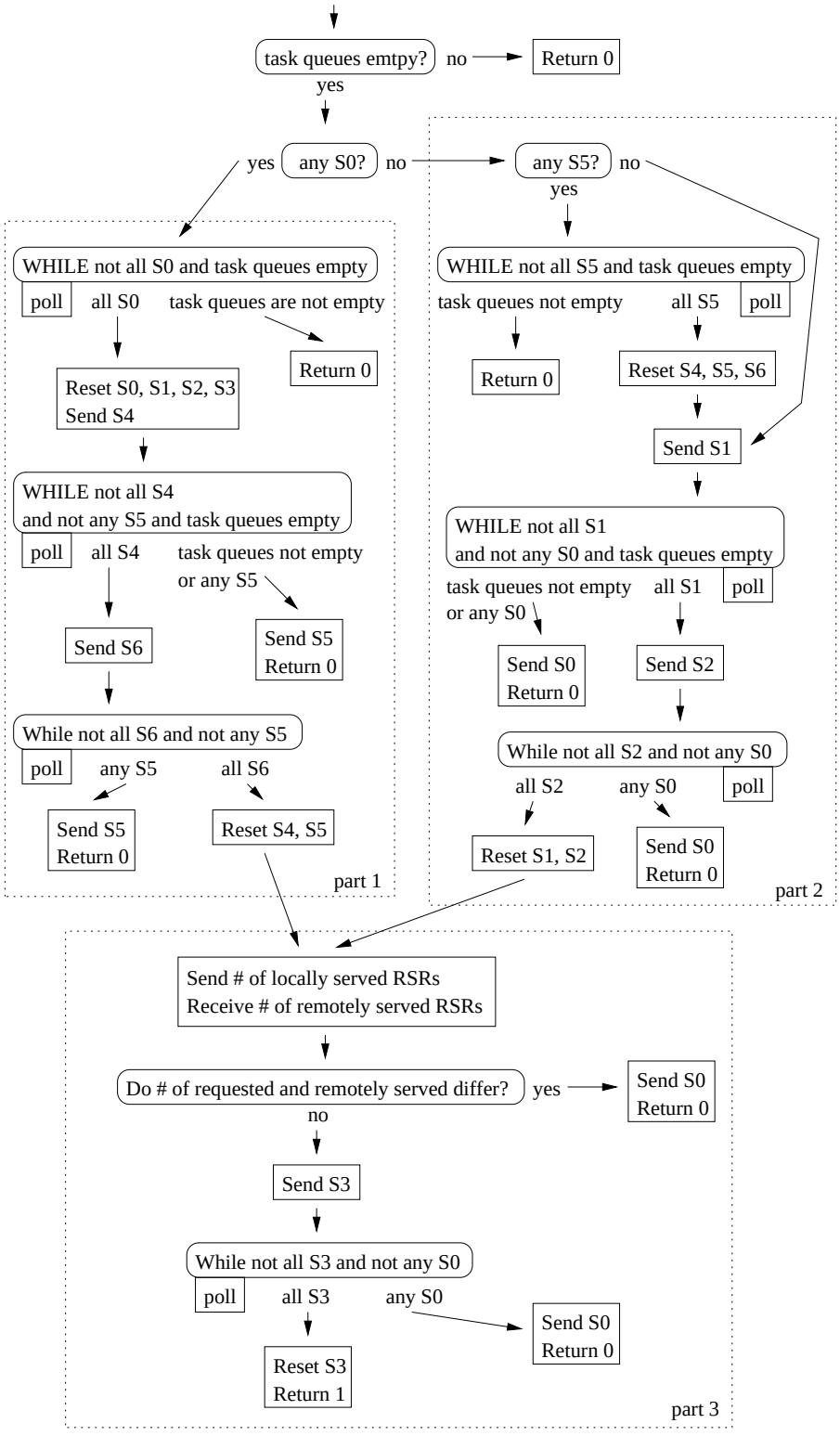

Figure 6.4: The algorithm for synchronizing asynchronous processes

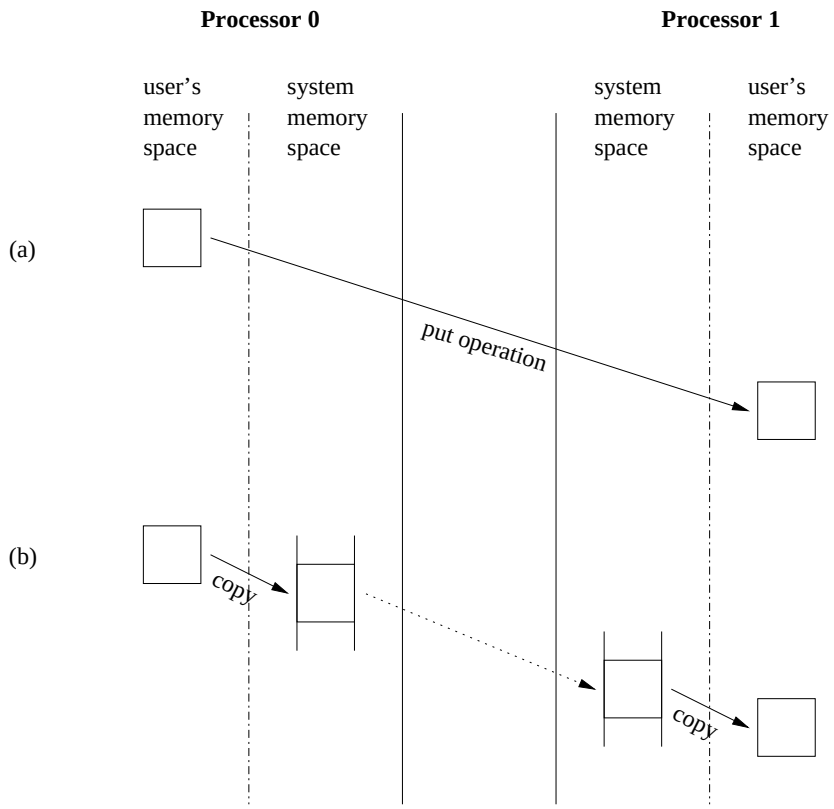

Figure 6.5: There are two ways to send a data block from the user's space if a valid destination address is specified by the user. Case (a) shows a direct transfer, case (b) uses the regular way by copying the block to the system buffer first and then after having been transfered to the remote system buffer copying again to the destination address.

A valid address: The destination address is given by the user (see Figure 6.5).

direct: Transfering the block directly to the remote address will cause an additional *put* operation.

regular way: Performing the transfer by copying the block to the system buffer first and then after having been transfered to the remote system buffer copying again to the destination address needs two copies of the block.

`ANYWHERE`**:** The user wants the block to be transfered to the user's space but does not specify any space for this block (see Figure 6.6):

three way direct: The block is copied directly to the user's space. Thus, space has to be allocated over there, first. In order to do so, a request is sent to the remote processor to allocate space for that block. A second request is sent back with the address to start the *put* operation for transfering the block: Costs: two requests and a *put* operation.

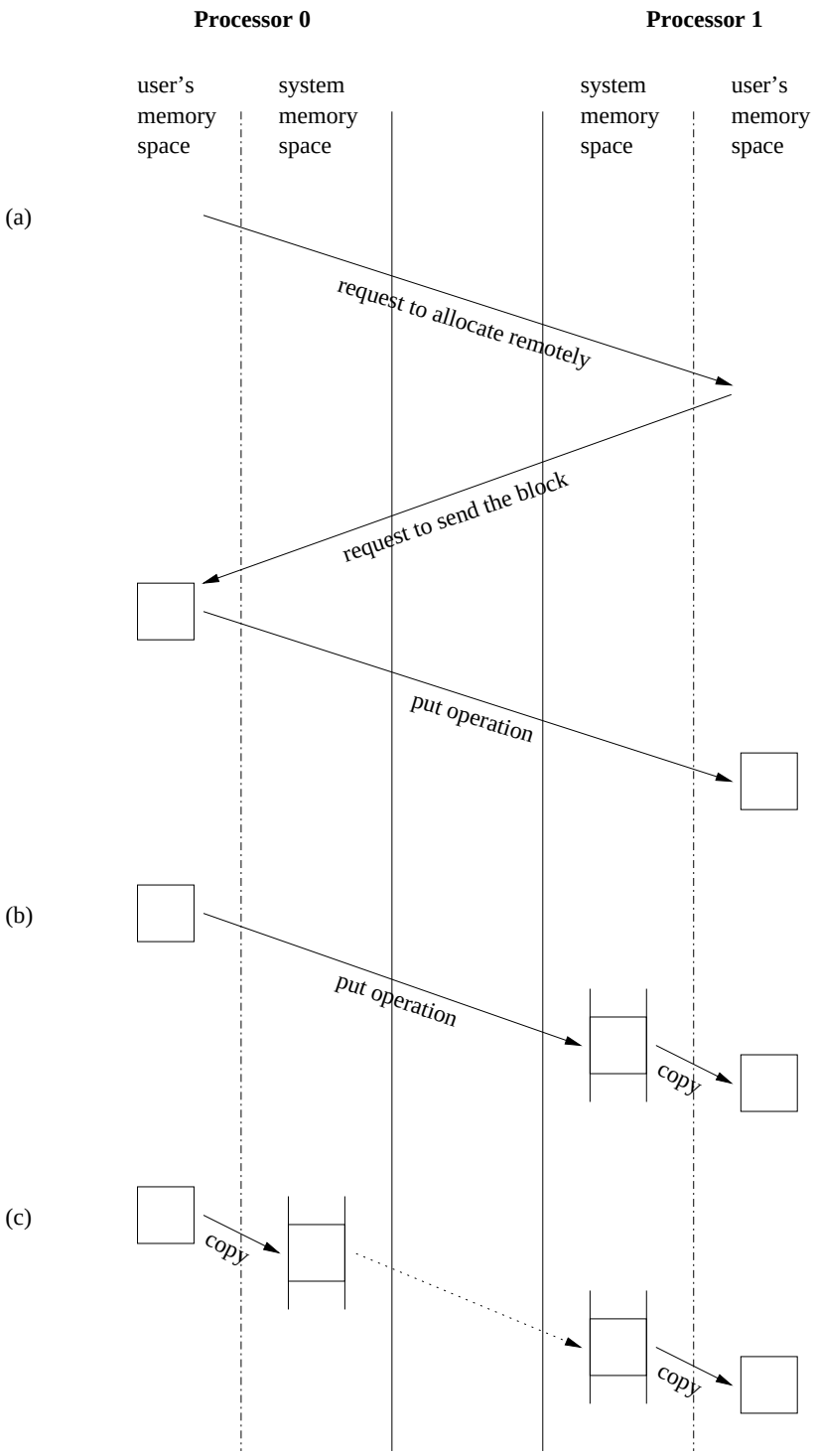

Figure 6.6: Ways to send a data block from the user's space if a valid destination address is not specified by the user. Case (a) uses two requests to get a space allocated and then sends the block directly, case (b) just sends the data block directly to the remote system space and then copies it to the remote user's space and case (c) does it the regular way.

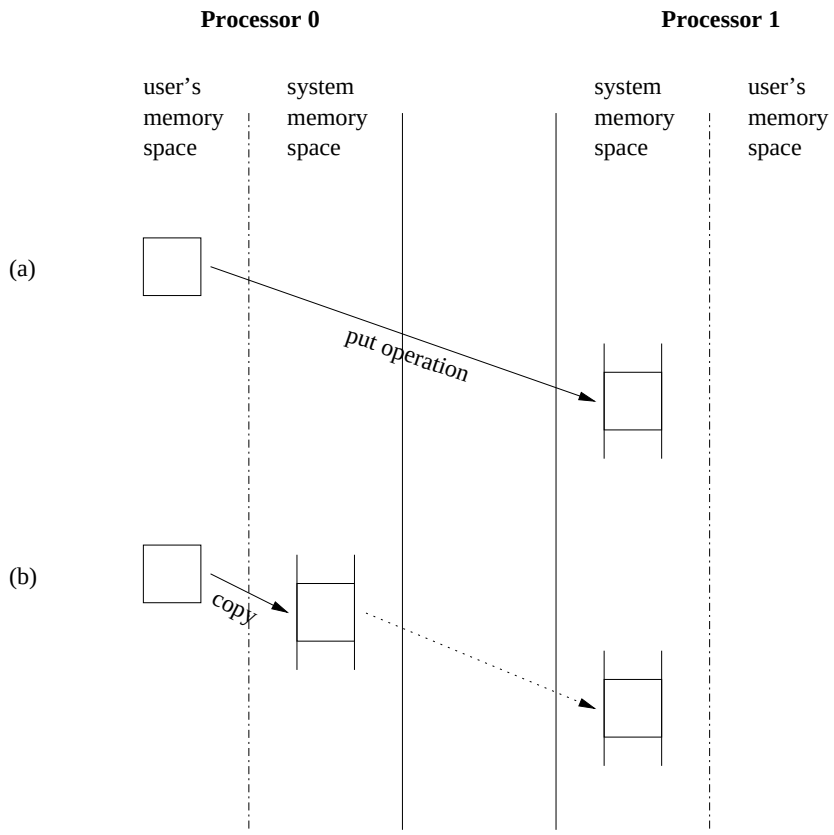

Figure 6.7: Ways to transfer a data block from the user's space to the remote layer's system memory. Case (a) shows a direct transfer to the remote system space, case (b) does it the regular way.

regular direct: The block is transfered directly to the system buffer of the remote processor and then copied to the user's space: a *put* operation and a copy.

regular way: In this case the costs are again two copies.

`NULL`: The data block just need to be transfered to the system buffer on the remote processor (see Figure 6.7):

direct: The costs are one additional *put* operation.

regular way: This time the regular way of getting the block to the remote processor takes one copy.

The cost of requests, copy and *put* operations depends on the underlying hardware platform and are to be found out through benchmarks. Depending on these data the layer will choose one of the described options. The regular data blocks can also be transfered in a direct way. The advantage of for the user to use the regular

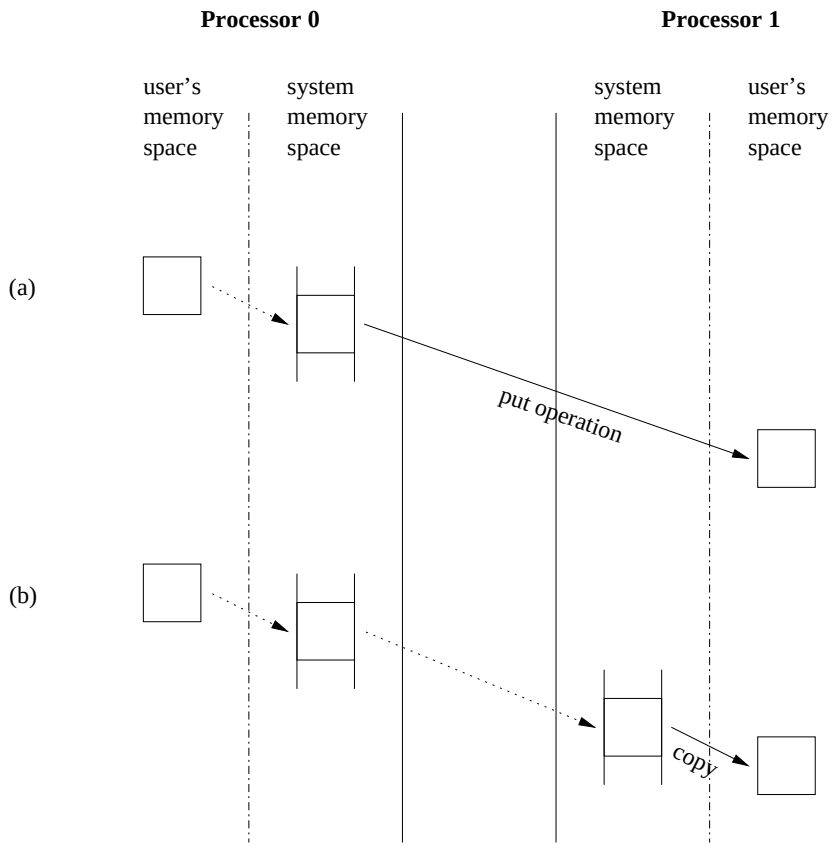

Figure 6.8: Ways to send a block from the layer's system memory if the user specified a valid destination address. Case (a) shows how to send the block directly from the local system space to the remote user's space, case (b) transfers the block along the regular way together with all other data.

data blocks is to be allowed to reuse the source memory space immediately after `lac_rsr` returns. Hence, the block has to be copied immediately to the layer's system buffer. But for the rest of the transfer there are different options depending on the destination address, the argument `reg_block_dest`:

a valid address: The destination address is specified by the user. There are two options for transfering the block (see Figure 6.8):

direct: The block is transfered directly to the user's space at the remote processor with a *put* operation.

regular way: The block is transfered with the rest of the data and written to the user's space by a copy.

`ANYWHERE`**:** This time the address at the remote user's space is not known (see Figure 6.9):

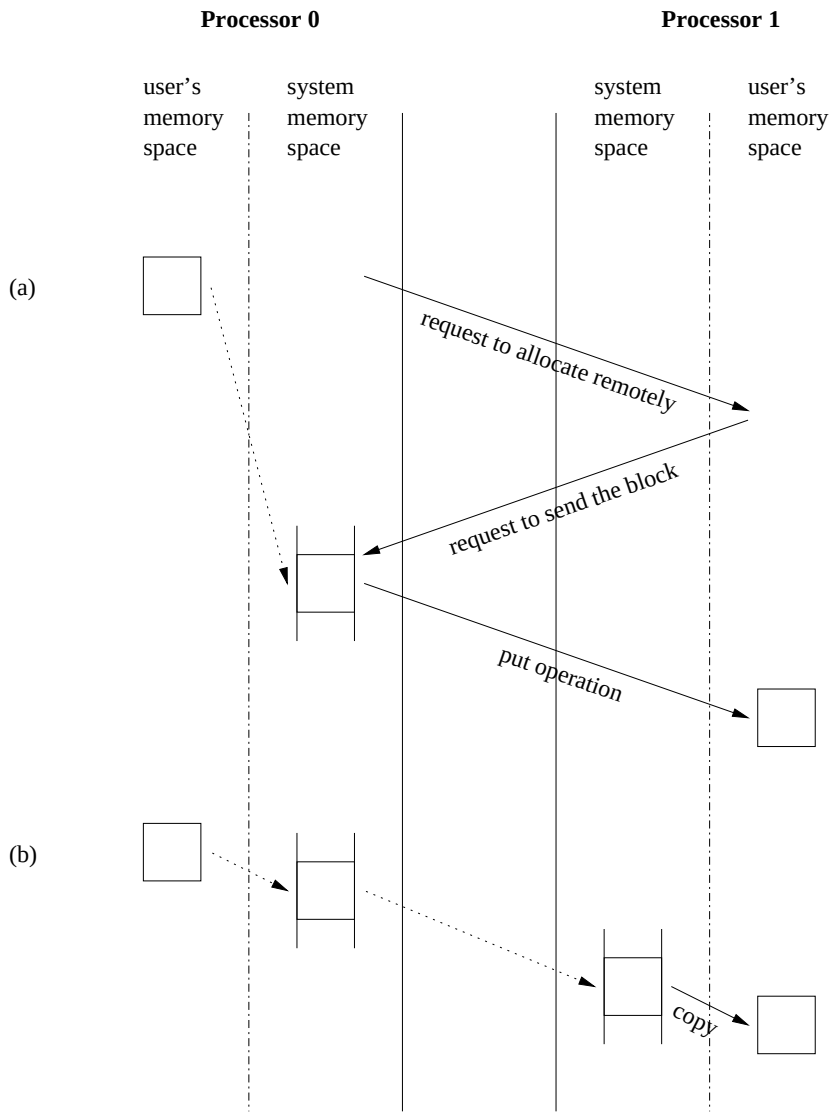

Figure 6.9: Ways to send a block from the layer's system memory if the user did not specify a valid destination address. Case (a) requests a remote allocation of user's memory and then requests sending the block directly to that space, case (b) does it the regular way

three way: Two requests and a *put* operation are necessary to transfer the block directly to the remote user's space.

regular way: The data block is transfered to the remote system space together with the other data and then a copy is done to put the data block to the user's space.

`NULL`: Here there is no choice, the block will always be transfered together with the other data to the system buffer of the remote processor.

Chapter 7

Evaluating the Layer for Asynchronous Communication

First the layer was implemented on top of Active Messages (Section 5) which is a very efficient tool for asynchronous low level communication but lacks of portability.

To avoid the problem of non-portability the layer was built on top of PVM, too. In this way the layer can be used on every machine where PVM runs on.

As soon as PVM is running the question arises why to use LAC. How much is it worth to use the layer (built on PVM) instead of using just PVM? What are the differences? How do they compare? This section answers these questions by running a synchronous PVM version and an asynchronous LAC one of a given problem described in Section 7.1. In the other sections the differences in run times (7.2 and 7.3), structure (7.4), ease of programming (7.5) and debugging (7.6) is pointed out.

7.1 The Test Case

The task to be done by the two program versions is to refine a given Delaunay mesh using the *Parallel Constrained Delaunay Triangulation* as described in Section 4.3. The mesh is divided along edges of the initial triangulation. These edges are considered as constrained and can only be split, not removed.

The runs of the triangulation programs are compared on a cluster of one master and five slaves, described in Table 7.1. Linux 2.2.1 SMP Kernel, based on the S.u.S.E. distribution, and PVM 3.4 is running on that parallel machine.

Since the cluster consists of five slaves the runs are performed with one to five processes, such that only one process is running on one slave. Figure 7.1 shows the initial triangulation to be refined. The distribution of this initial triangulation for two, three, four and five processes is defined in Figure 7.2, Figure 7.3, Figure 7.4 and Figure 7.5, respectively.

7.2 Efficiency Benchmarks

Triangles in a quality triangulation have to be well sized and well shaped (Sections 2 and 2.2.1). For benchmarking the amount of triangles calculated can

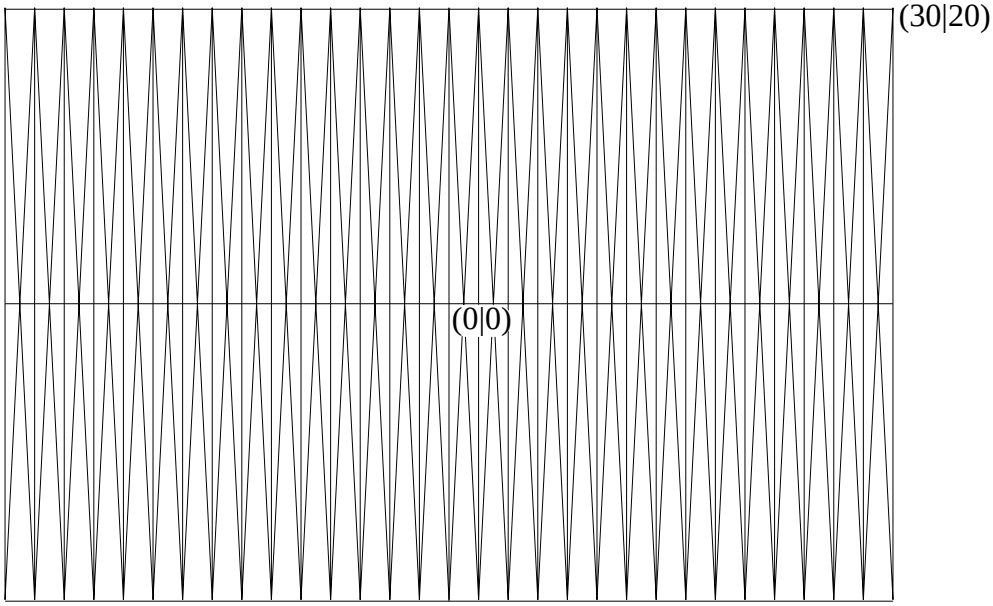

Figure 7.1: The initial triangulation to be refined.

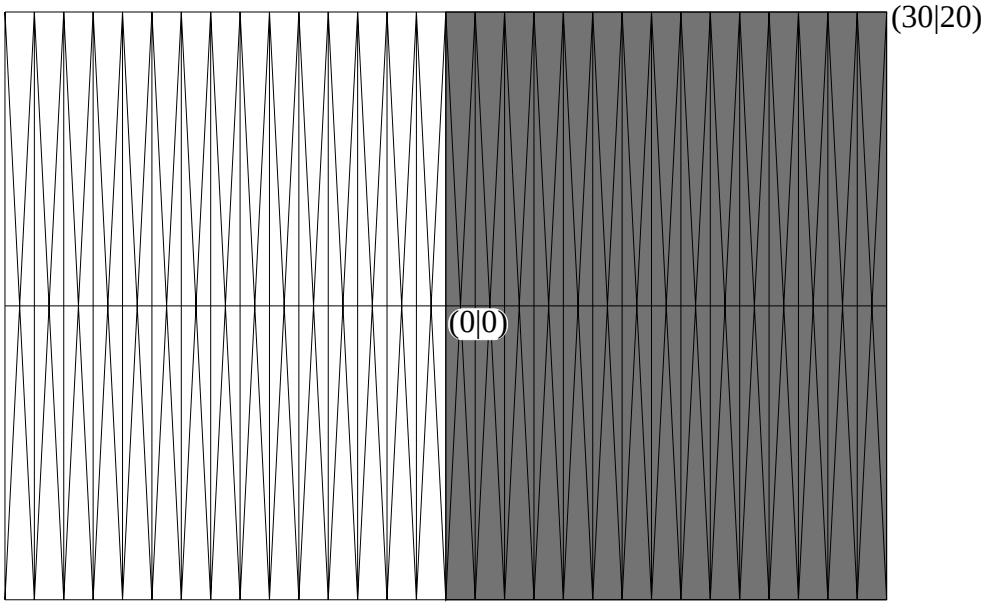

Figure 7.2: The distribution of the initial triangulation for two processors.

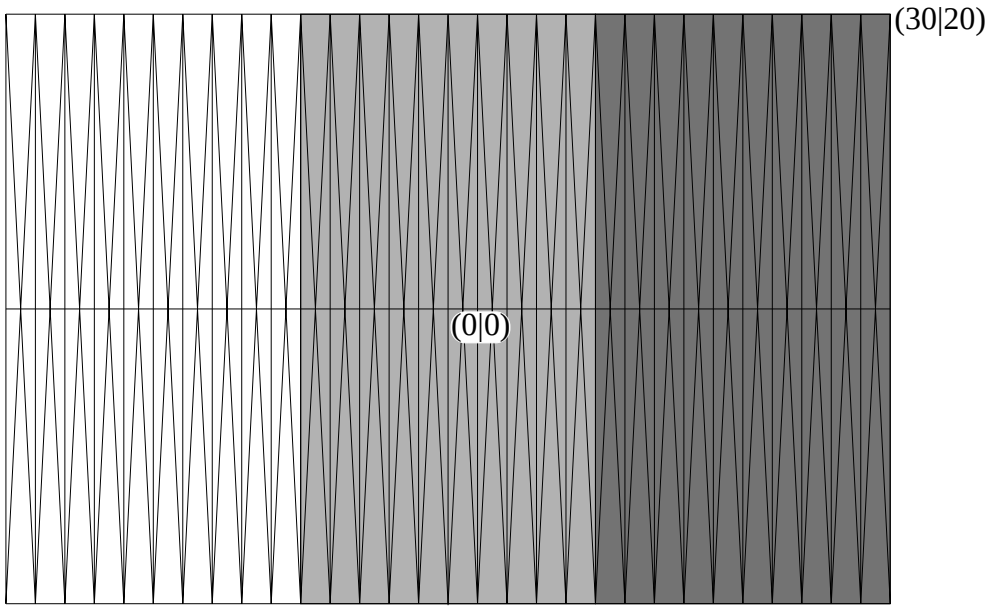

Figure 7.3: The distribution of the initial triangulation for three processors.

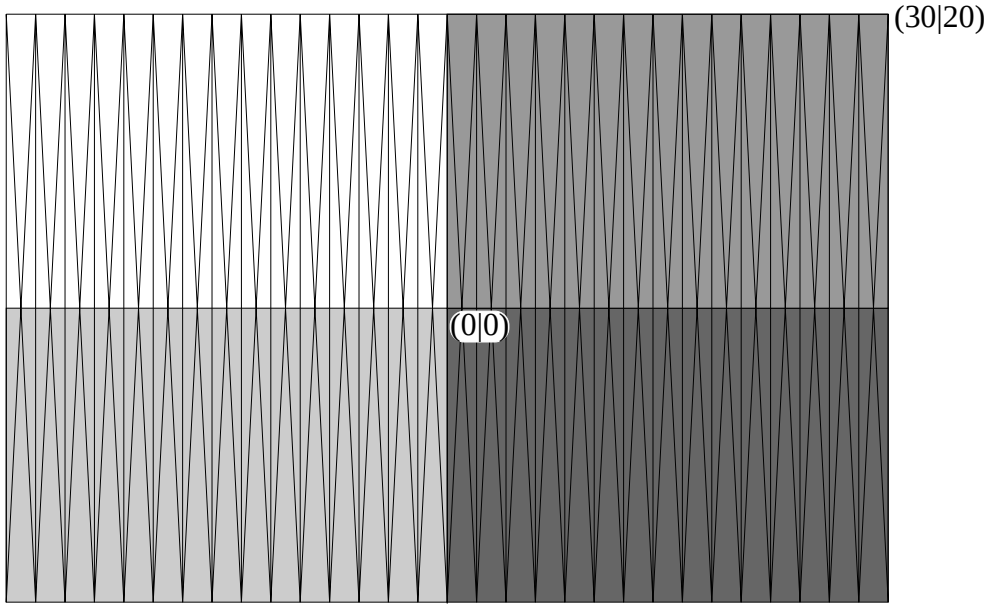

Figure 7.4: The distribution of the initial triangulation for four processors.

Master	
Mainboard	ASUS P2BS-ATX, 100 MHz
CPU	Pentium II 350 MHz
Memory	256 MB
Network	2 × 3Com Fast-Etherlink 3C905-TX

Slaves	
Mainboard	ASUS P2BS-ATX, 100 MHz
CPU	2 × Pentium II 350 MHz
Memory	256 MB
Network	3Com Fast-Etherlink 3C905-TX

Table 7.1: The cluster where to run the PVM and LAC version for comparison. There are one master and five slaves.

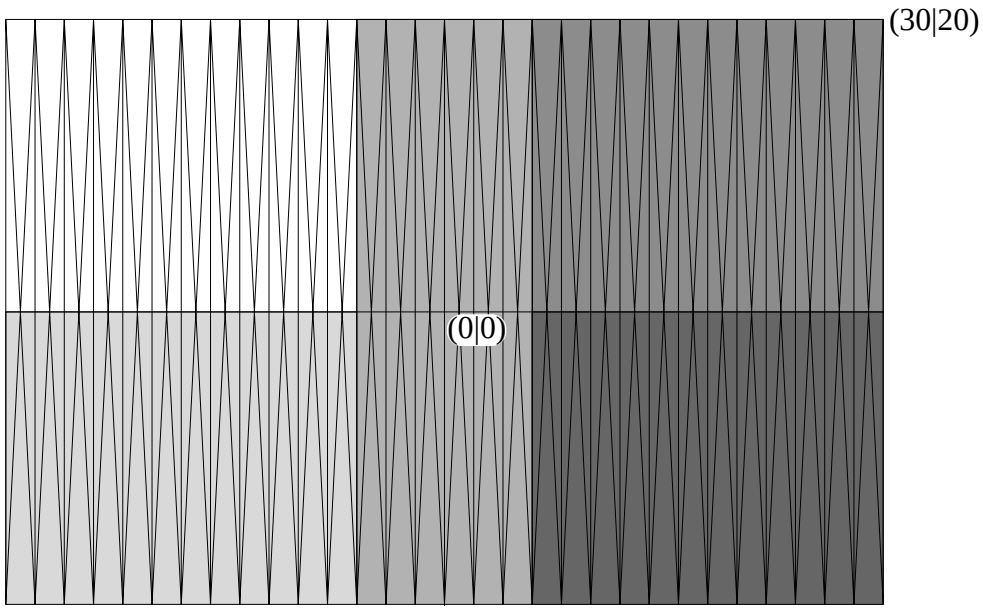

Figure 7.5: The distribution of the initial triangulation for five processors.

be controlled by the maximum size of the triangles, that means the number of triangles produced depends on the "well sized" constraint.

For benchmarking the maximum size of triangles is chosen such that the machine does not page. Although the bigger amount of available memory on a parallel machine is one of the pros of parallel computing benchmarking is performed without this effect otherwise numbers for speedup would be without any meaning.

The goal of this benchmarking was to compare the run times of the synchronous and asynchronous version of the triangulation program. Therefore, on the one hand the speedup is of interest on the other hand both versions should

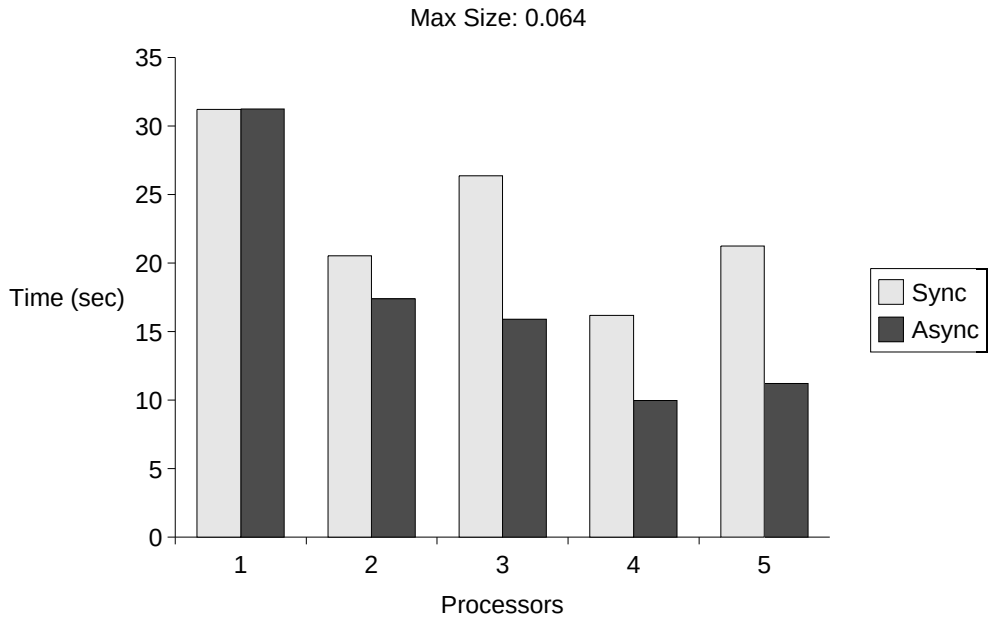

Figure 7.6: The results of the runs with maximum triangle size of 0.064 units which corresponds to 2.9 M triangles

be compared in the situation of maximum load.

Out of these reasons the first series was started with a maximum size just that the produced triangles fit into a single machine without paging. Then runs are done with two, three, four and five processors. The run times can be seen in Figure 7.6. In the next step the maximum size for two processors are chosen and the runs are performed with two, three, four and five processor (run times in Figure 7.7). The other series of runs are analogously started with three, four and five processors, where the results are shown in Figure 7.8, 7.9 and 7.10, respectively. Every run of a series is done at least five times and the average run time finally is presented. Runs where the run time shows a great deviation does not count. Deviations of the used run times are small. Table 7.2 summarizes the run times of all runs.

7.3 Differences in Run Times

The benchmarks, presented in the previous section, were performed for comparing the run times of the synchronously and the asynchronously parallel version of the Constrained Delaunay Triangulation. The main conclusion is that there is a significant speedup of the asynchronously parallel version compared to the synchronously parallel version. Table 7.4 shows this speedup while Table 7.3 presents the ratios of the run times of the asynchronously parallel version to the run times of the synchronously parallel version.

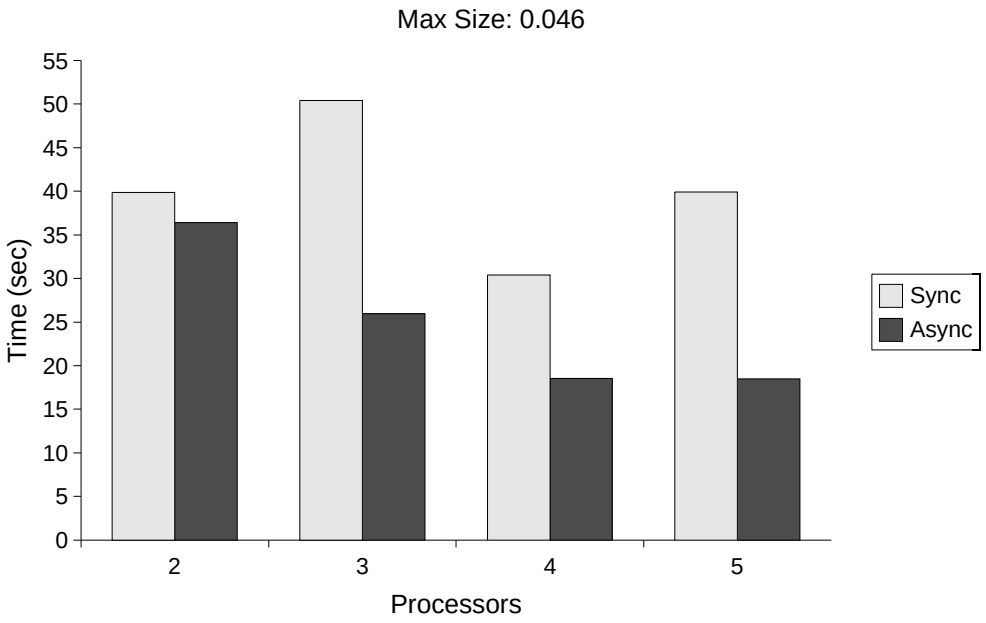

Figure 7.7: The results of the runs with maximum triangle size of 0.046 units which corresponds to 5.7 M triangles

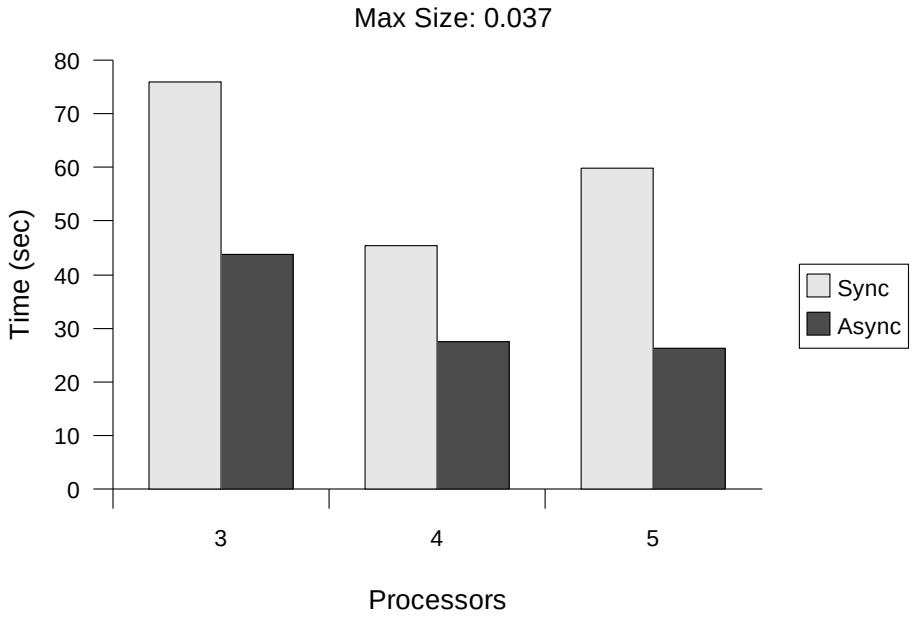

Figure 7.8: The results of the runs with maximum triangle size of 0.037 units which corresponds to 8.6 M triangles

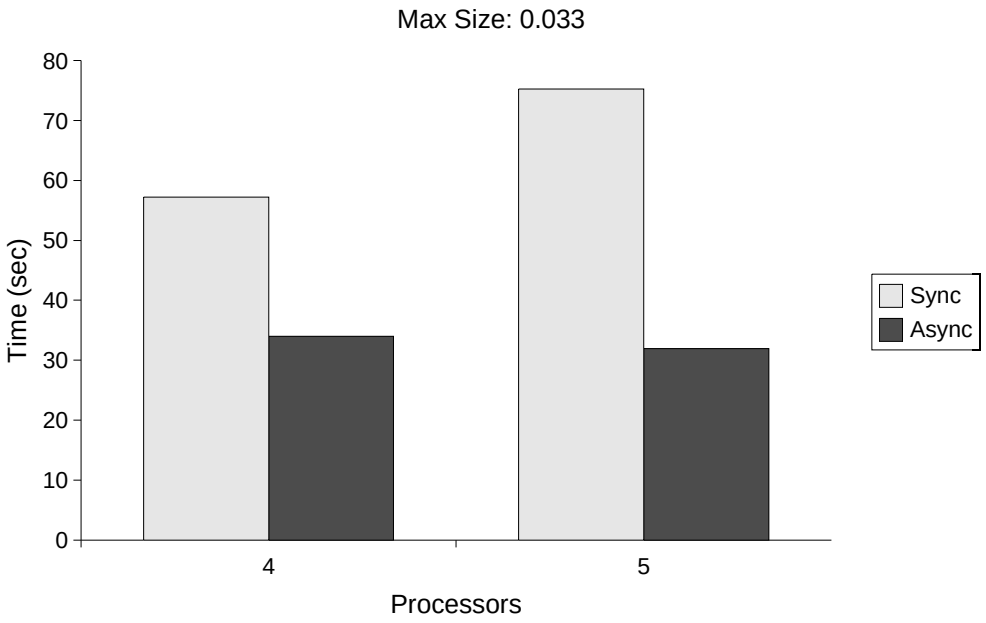

Figure 7.9: The results of the runs with maximum triangle size of 0.033 units which corresponds to 11 M triangles

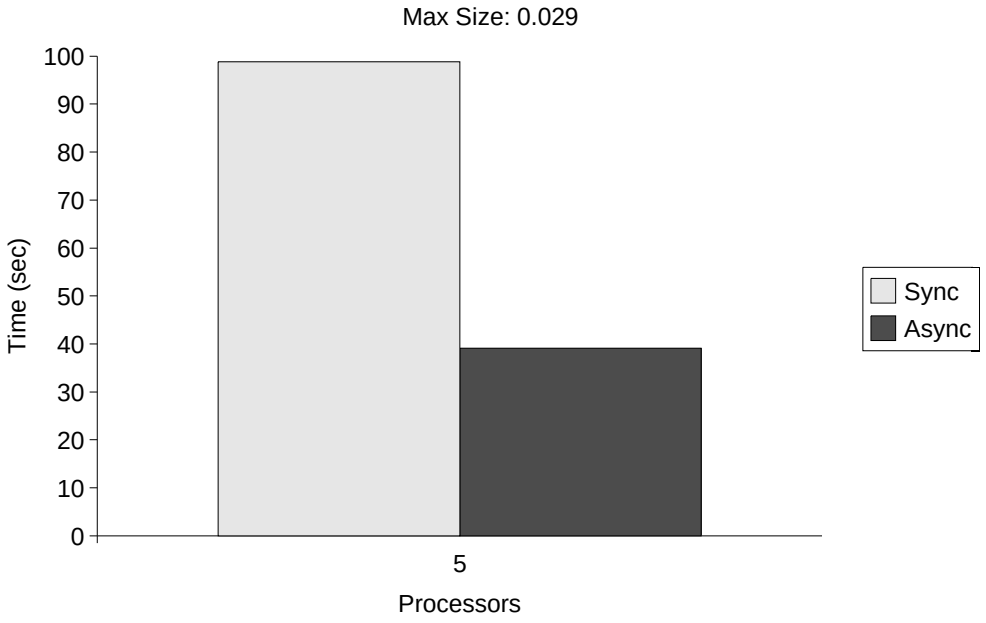

Figure 7.10: The results of the runs with maximum triangle size of 0.029 units which corresponds to 14.5 M triangles

Max triangle size	Version	Processors				
		1	2	3	4	5
0.064	Sync	31.2	20.5	26.4	16.2	21.2
	Async	31.2	17.4	15.9	9.9	11.2
0.046	Sync		39.9	50.4	30.4	39.9
	Async		36.4	25.9	18.5	18.5
0.037	Sync			75.9	45.4	59.9
	Async			43.8	27.6	26.4
0.033	Sync				57.2	75.3
	Async				34	31.9
0.029	Sync					98.8
	Async					39.1

Table 7.2: The summary of the run times (sec) on the cluster

Max triangle size	Processors				
	1	2	3	4	5
0.064	1	0.85	0.6	0.62	0.53
0.046		0.91	0.51	0.61	0.46
0.037			0.58	0.61	0.44
0.033				0.59	0.42
0.029					0.4

Table 7.3: The ratio of the run times of asynchronous version to the run times of the synchronous version. The absolute run times are shown in Table 7.2

There are two reasons for the higher efficiency of the asynchronous version, as described in Section 7.4.4 on page 94:

- The communication can be hidden in the asynchronous version. This means that a request is given to the network to be served remotely but there is no need to wait until it is really sent. The synchronous version always has to wait for the answer and, therefore, also for setting up the message to be sent.

Max triangle size	Processors				
	1	2	3	4	5
0.064	1	1.18	1.66	1.63	1.9
0.046		1.1	1.94	1.64	2.16
0.037			1.73	1.65	2.27
0.033				1.68	2.36
0.029					2.53

Table 7.4: The ratio of the run times of synchronous version to the run times of the asynchronous version. The absolute run times are shown in Table 7.2

- No waiting for messages is necessary at the asynchronous version. At the synchronous version each process has to synchronize after having locally refined its mesh as much as possible. Then the processes have to get synchronized to refine the rest of the mesh where access to remote triangles is necessary.

The results show a much better performance on the runs with two and four processors compared to the runs with three and five processors. This observation can be explained by the fact that the necessary communication depends largely on the border lengths, actually the maximum border lengths. When the initial mesh is distributed into regions such that each processor gets one region, each of these regions must have borders to at least one adjacent region. The maximum border length is defined as the maximum over all border lengths of these regions. Distributing the initial mesh to two processors implies a maximum border length of 40, three processors means a length of 80, four processors 50 and five processors 80. Therefore, the distribution to three and five processors both means long borders of length 80 causing poor performance.

However, this effect becomes smaller when the amount of data to be computed is increased. For example, refining the mesh to a maximum triangle size of 0.064 means 9.9 seconds for the four processor distribution of the asynchronous version. Five processors need 11.2 seconds, this means a negative speedup. But refining the mesh to a maximum triangle size of 0.029 results in a positive speedup, 34 and 31.9 seconds, respectively. This comes from the fact, that the number of edge splits only depends on the square root of the number of triangles.

Table 7.4 shows that the run time ratio of the synchronous version to the asynchronous version becomes higher if communication costs are higher, especially at the three and five processor distribution. That can be explained by the two reasons mentioned above: First, communication can be hidden at the asynchronous version. Second, at the synchronous version longer borders mean a smaller number of locally computed triangles which implies a higher amount of time to wait at the point of time where access to remote triangles is necessary considering the structure of both algorithms explained in section 7.4.

In general the speedup of the asynchronous version compared to the synchronous one increases with the number of processors used for mesh refinement. This results from the difference in scalability of the two versions as explained in Section 7.4.4 on page 95. The scalability of the asynchronous version is acceptable, especially if the amount of triangles to be computed is increased. But there is bad scalability at the synchronous version due to the increasing waiting at the final step when the part of the mesh is refined where access to remote triangles is needed.

The speedup of the asynchronous version also increases with the increasing amount of data to be computed. This comes from the final synchronization at the very end of the asynchronous algorithms. This synchronization step counts for

timing and is a quite complex task which is performed by the layer as described in Section 6.1.3, 6.3.2 and B.3. This has to be done no matter how many data are computed. Increasing the portion of time needed for computation of the mesh by increasing the number of triangles to be calculated means a better performance of the asynchronous algorithm.

7.4 Differences in Structures

In this section the synchronous and asynchronous algorithms are presented whose description emphasizes the important parallel parts. On details of the sequential algorithm please refer to Section 2.2.3.

7.4.1 Common Data Structure

In both versions a border edge, which is an edge between two processors, has two instances, one on each processor. Such an instance of a border edge contains a pointer to the remote instance additionally to some local data. If a new border edge is created each instance needs the address of the other (remote) instance. That is one issue which has be considered at the algorithms in the following sections.

7.4.2 Structure of the Synchronous Algorithm

The synchronous parallel algorithm consists of two major parts: After an initialization, consisting of reading and distributing the initial mesh,

1. the triangulation is refined as much as locally possible, where all infeasible triangles are removed as long as not communication to another processor is required, and

2. messages are send and received in order to refine the rest of the mesh.

   One processes at a time gets the permission to send messages to all processes it needs to refine the mesh which is too close to the border to be triangulated just locally. After the process has finished, the permission is passed on the next process.

A detailed description of this algorithm follows. First, the mesh is refined as long as there is no need to split a border edge. If this happens, the current infeasible triangle is stored for the second step. When no infeasible triangles are left to refine in this step, the second step is performed, the splitting of the border edges which causes communication. A token is used to tell which process is currently allowed to split its border edges and to send messages to split these border edges at the adjacent processor as well. Figure 7.11 shows the design of this algorithm.

PROCEDURE *generate_mesh_synchronous_parallel*

Get and distribute the initial triangulation.

Timing starts.

Do while an infeasible triangle is available

Try to insert its circumcenter

If a border edge between two processors has to be split

Then keep the infeasible triangle for later.

End if

End do

Make the kept triangles available.

Do until (*the following condition is checked at loop end*) all processors are ready.

If it is my turn (the sequence of slaves has to be determined)

Then

Do while an infeasible triangle is available

Try to insert its circumcenter.

If a border edge between two processors has to be split

Then split this edge locally.

Send a message to start a remote split of the edge. The local addresses of the new border edges come with this message.

Receive the addresses of the new remote border edges.

End if

End do

Else

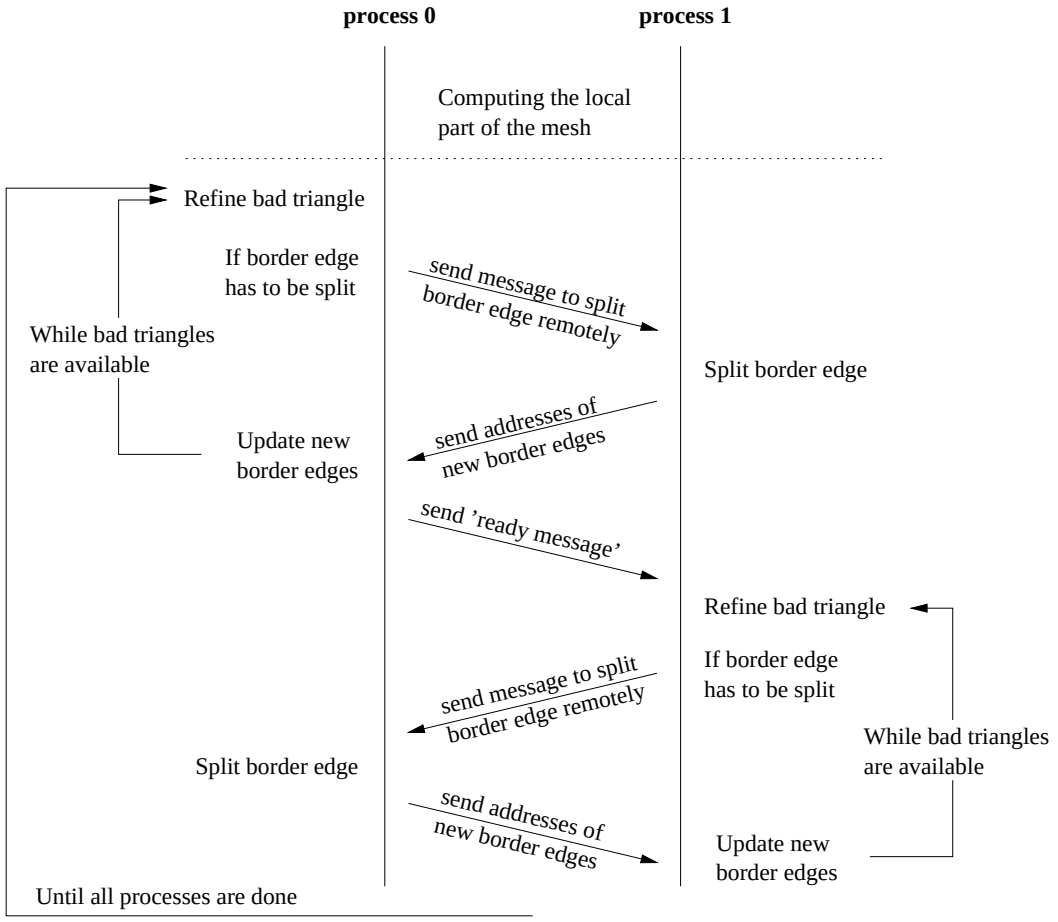

Figure 7.11: The algorithm for performing the *Parallel Constrained Delaunay Triangulation* synchronously.

Do while I get messages from other processors

Receive edge to split.

Split edge locally.

Send back the addresses of the new border edges.

End do

End if

End do

Timing ends.

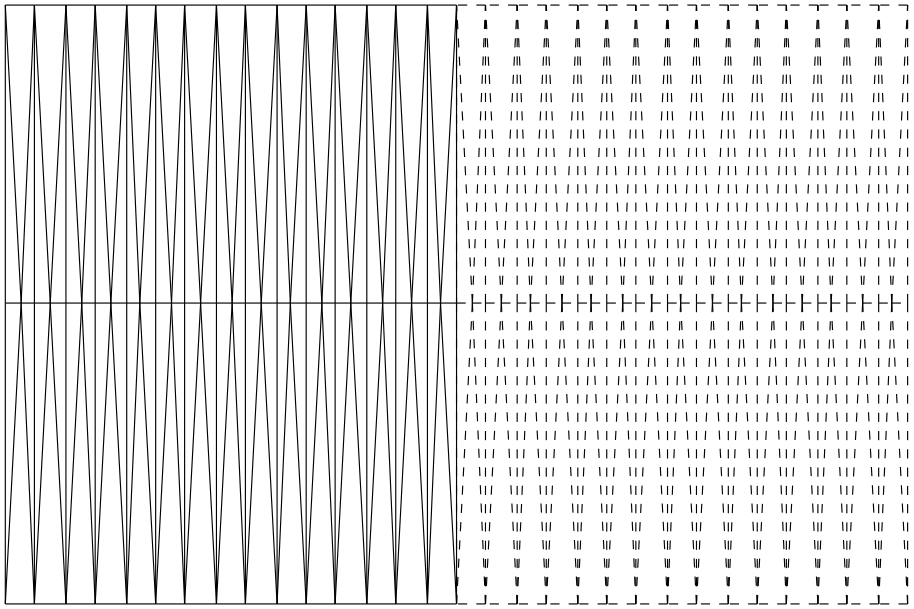

Figure 7.12: The initial mesh distributed to two processors.

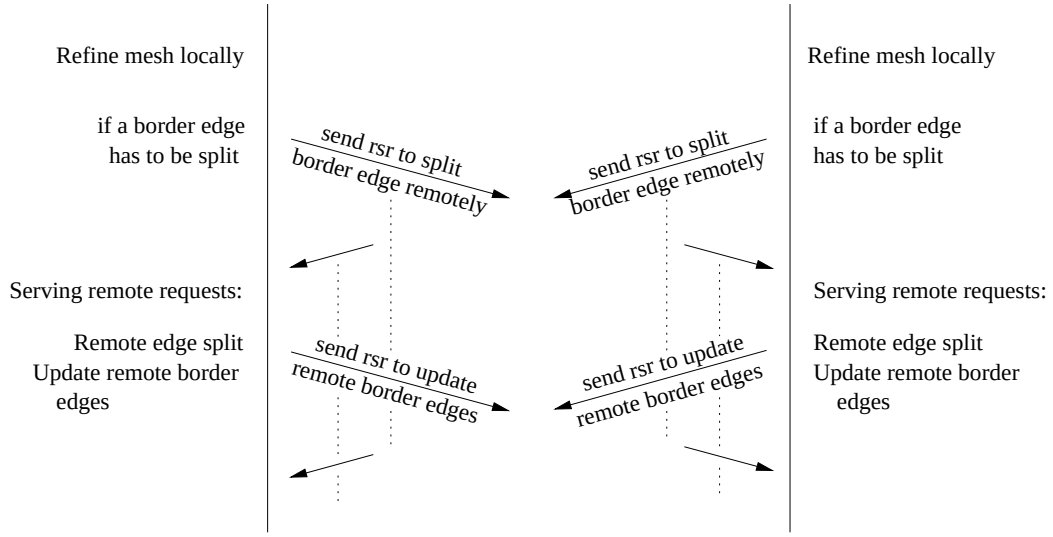

Figure 7.13: The asynchronous algorithm for performing the *Parallel Constrained Delaunay Triangulation*.

Figure 7.12 shows the initial triangulation to start with. Then the synchronous version refines the mesh as much as locally possible. Figure 7.14 shows the result after this part. After triangulating the rest, where communication is possible, the final result can be seen in Figure 7.15.

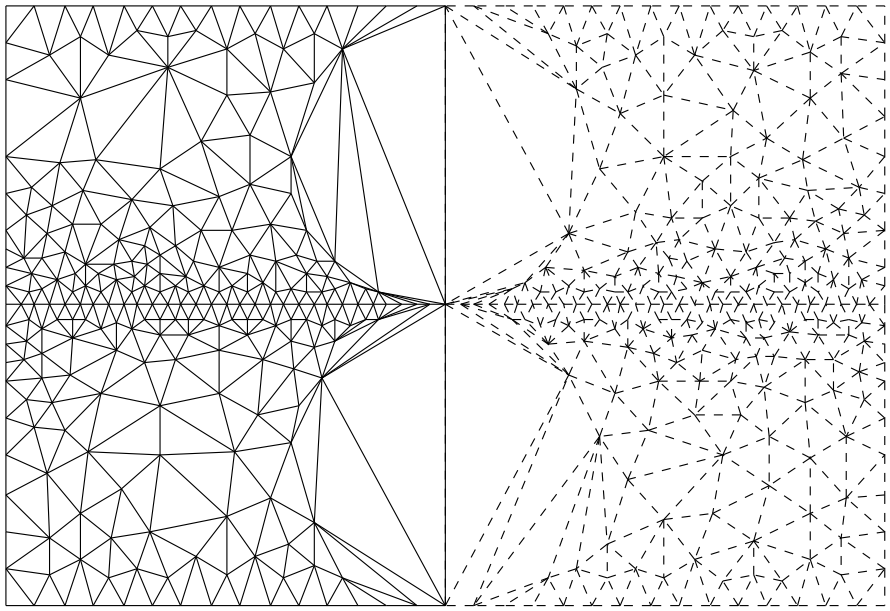

Figure 7.14: The mesh after the step of refining the mesh locally and before the step of splitting border edges.

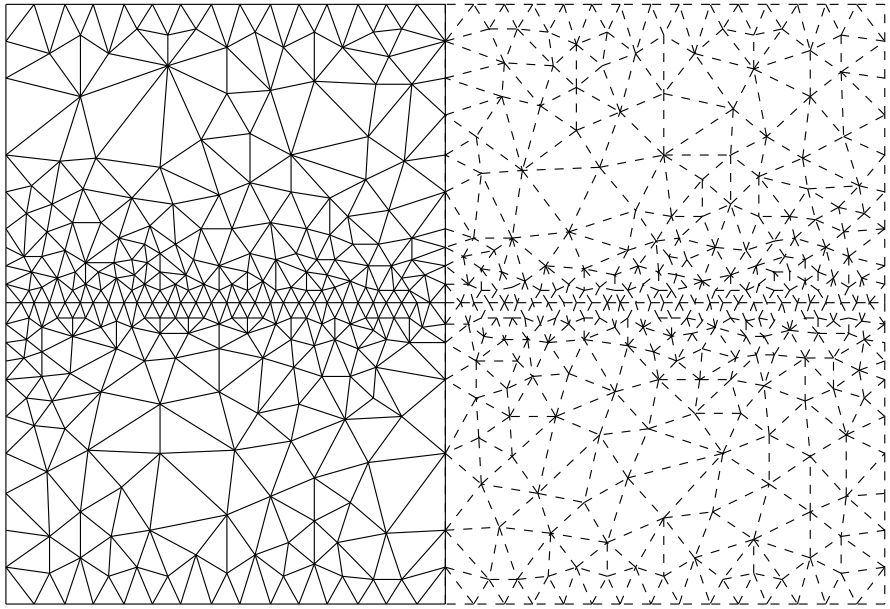

Figure 7.15: The final mesh produced by two processors.

7.4.3 Structure of the Asynchronous Algorithm

The asynchronous algorithm only consists of one part. Like the synchronous version the mesh by evaluating the local triangles but in case when a border edge has to be split the edge is not put away but split locally and just a request for a remote split is sent. Figure 7.13 illustrates the following algorithm.

PROCEDURE *generate_mesh_asynchronous_parallel*

Get and distribute the initial triangulation.

Timing starts.

Do until (*the following condition is checked at loop end*) all processes are ready.

Do while a bad triangle is available

Try to insert its circumcenter.

If a border edge between two processors has to be split

Then split the border edge locally.

If the remote instance of the border edge of the border edge is known (the update could still be on the way)

Then send remote service request to split border edge remotely, too (call *split_remote_border_edge* remotely).

Else mark border edge as not yet remotely split.

End if

End if

Serve remote requests.

End do

Serve remote requests.

End do

Timing ends.

PROCEDURE *split_remote_border_edge*

If border edge is already split

Then update pointers of new border edges.

If one of the new border edges is already split

Then send remote service request to split border edge remotely, too (call *split_remote_border_edge* remotely).

Else split border edge locally.

Send remote service request to update pointers at the remote instances of the new border edges (*update_remote_border_edges*).

End if

End if

PROCEDURE *update_remote_border_edges*

Update pointers of new border edges.

If one of the new border edges is already split

Then send remote service request to split border edge remotely, too (call *split_remote_border_edge* remotely).

End if

7.4.4 Comparison of Structures

One major difference between the structures of the different versions is the point of time when communication happens. At the synchronous version this is done after the independently local mesh generation is done. Then each processor may either send or receive messages. The asynchronous version allows to send remote service requests anytime. This yields two advantages.

Communication Hiding. At the synchronous version all communication is done at the end. Then hiding of communication is poorly possible. Communication is hidden when the process does not wait till sending a message is completed but continues with local work in the meantime. But at the

end of the triangulation the processes may run out of local work. Another effect may be a higher number of collisions caused by heavy network traffic when sending everything at the end.

The asynchronous version sends its requests distributed over the run time. After sending a request work is available to continue while the communication unit (ports, etc) is still busy with handling the message.

Waiting for Messages. None of the processes has to wait for any incoming remote service request. Processes have to poll the network and only if a request is pending the process interrupts its work (only at user defined places, therefore, it should be safe) serves the remote request and is continuing its work.

At the synchronous algorithm there is one active process and the others are passive at a time in the second part where communication is performed. The active process sends messages when it needs to have a border edge split remotely. The passive processes are waiting to receive such a message. When the active process has finished, one of the passive processes becomes active. This solution forces the passive processes to wait incoming messages and to waste time.

These two advantages are supposed to be the reason for the higher performance of the asynchronous version but there is also a disadvantage.

Polling for RSRs. In order not to jam the network polling has to be done frequently. Depending on the implementation this might consume a considerable amount of time. PVM as underlying communication tool needs quite some time for such a poll but is not sensitive in terms of having the receive buffers overflown. A strategy to improve run times in this case is to reduce the number of polls.

The Scalability of the Parallel Algorithms

The asynchronous version does not have any problem with scalability. Since, no synchronization is performed more processors would not mean more waiting for any messages. But this would cause the passive processes in the synchronous version to wait even longer. To improve scalability more sophisticated algorithms needs to be developed to allow more than one active process at the part where communication is performed.

The results of the runs allows to make the conclusion that communication costs depend on the length of the border. The speedup at the runs with two and four processes is much better than the speedup of using three or five processes. The lengths of the borders of processes at the two, three, four and five processes, i. e., the maximum length of border edges over all processes of each distribution, are 40, 80, 50 and 80 units, respectively.

7.5 Differences in Programming

7.5.1 Quantitative Evaluation

The objective of this section is to gain some measure about length and complexity of the code of both versions. Software metrics [22, 30, 51] are used on this behalf. The numbers obtained are compared to say if and where one version differs to the other.

CCCC [38] was used to measure the synchronous and the asynchronous version. CCCC is a tool for the analysis of source code in various languages, which generates a report in HTML format on various measurements of the code processed. Although the tool was originally implemented to process C++ and ANSI C, facilities have recently been added to allow Java and Ada 95 source files to be recognized and processed as well. The name CCCC stands for *C and C++ Code Counter*.

The following metrics are interesting as result in the comparison between the two different versions:

- LOC = Lines of Code: Number of non-blank, non-comment lines of source code counted by the analyzer.
- MVG = McCabe's Cyclomatic Complexity: A measure of the decision complexity of the functions which make up the program. The strict definition of this measure is that it is the number of linearly independent routes through a directed acyclic graph which maps the flow of control of a subprogram. The analyzer counts this by recording the number of distinct decision outcomes contained within each function, which yields a good approximation to the formally defined version of the measure. More details on McCabe's Cyclomatic Complexity can be found in [22, 51].
- FUN = Number of Functions

The results of the run of CCCC, shown in Table 7.5, indicates that the qualitative differences in programming both versions are insignificant. The lines of code are a little bit fewer at the asynchronous version than in the synchronous one because of more straight forward programming while the number of functions are greater caused by packing certain tasks into separate functions to be called as remote service requests. The issue of more straight forward programming, the need of extra functions and other details on the much more meaningful, qualitative differences are explained in the upcoming section.

7.5.2 Qualitative Evaluation

In this section the differences in Programming are explained which cannot be described by numbers. As mentioned in the section above, the asynchronous

Metric	LOC	MVG	FUN
Sync	781	143	10
Async	697	122	12

Table 7.5: The results of the run of CCCC on the synchronous and asynchronous version.

programming style is a more straight forward one. If remote action is needed a remote service request is sent and work is continued. There is no need to wait for the correct point of time to send the message, sending is always possible. Synchronous programming only allows to communicate at certain points in the program flow. However, an asynchronous parallel program frequently needs to poll for incoming requests.

For example, in the case of the asynchronously *Parallel Constrained Delaunay Triangulation* a point is always inserted (if not the circumcenter of a triangle then the center of a boundary or border edge). If any remote action has to be performed, a remote service request is sent to the adjacent process also to split the edge.

In the synchronous version in the case of a split border edge where remote action is necessary, point insertion is rejected because the adjacent process is not expecting any message coming in at this point of time. Since, the point of time and the number of messages are not predictable in this algorithm immediate sending and receiving would hardly work.

But there is a price to pay for this more straight forward programming. When a remote service request is sent, an additional state has to be defined and kept. For example, when a border edge has to be split the address of the remote instance needs to be known. This split of a border edge produces two new border edges where the addresses of the remote instance cannot be entered immediately, since, those might not be created, yet. Therefore, a remote split of the border edge is requested. And the two new edges have to be left in a non-regular state, namely, a state where the remote instances are not known, yet. If a border edge in such a state has to be split, another new state has to be defined. This happens if a border edge has to be split before the address of the remote instance is known. Figure 7.16 shows the different states of a border edge.

These states causing a much higher complexity than in synchronously parallel programs. In case of the synchronously parallel version the new border edge is created, a message is sent to the adjacent process and the local process is waiting for the addresses of the two new remote instances. In this way there is just one state, the regular state, which makes the program much easier.

Designing and implementing an algorithm is one thing, testing and getting the code running another. The higher complexity of asynchronous programs makes tracking down bugs more difficult as the next section will show.

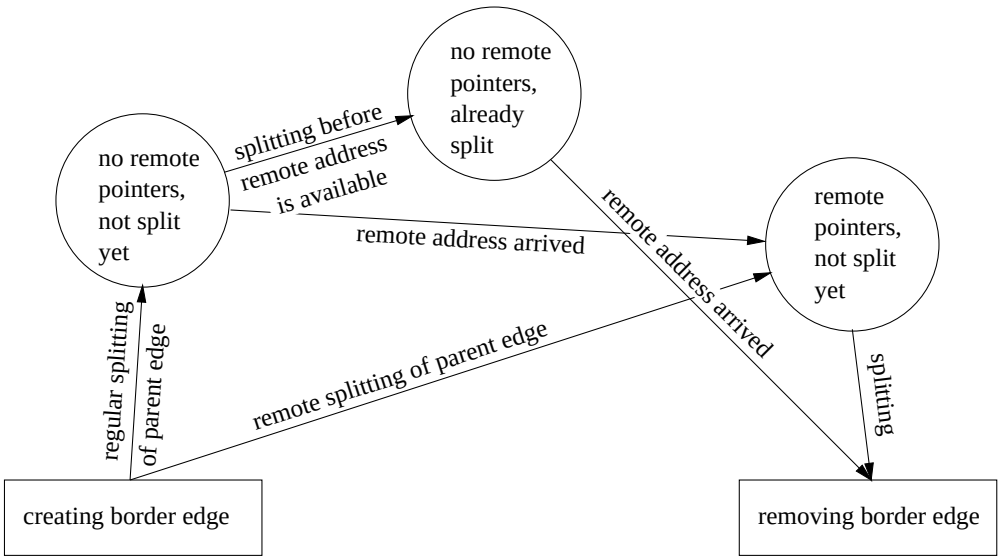

Figure 7.16: The three possible states of a border edge at the asynchronously parallel version. Beginning with the creation of the edge and ending with its replacement by two new border edges.

7.6 Differences in Debugging

PVM provides the possibility to start each process under a debugger. This provides all the usual comfort. Synchronously parallel programs can be debugged in this way. Unfortunately it is not that easy with asynchronously parallel programs. The reason is their indeterminism.

Generally spoken, synchronously parallel programs are rather deterministic, asynchronously parallel programs are not. For example, the triangulation may differ between two runs of the asynchronous version while it always stays the same at runs of the synchronous version. Figure 7.17 shows an example how the triangulation may develop differently running the asynchronously parallel version.

It is often hard to find out how a state in an asynchronous programm is reached. The layer provides tracing, it tells when a remote service request is called, an execution starts and finishes. In this way the program flow can be exactly recovered even if it gets as complex as in Table 4.1 on page 33.

But even with this feature debugging an asynchronously parallel program remains difficult. Sometimes it is necessary to set a breakpoint in a handler often without knowing when exactly it is called and what state are certain data in. And there is often a great choice, since, as described in Section 7.5.2, objects may be in many more states than in corresponding synchronously parallel programs.

Furthermore, this indeterminism makes some errors transient. Not only that errors may get rarely visible, they even might not appear at all when computation is slowed down, for example, in case of debugging.

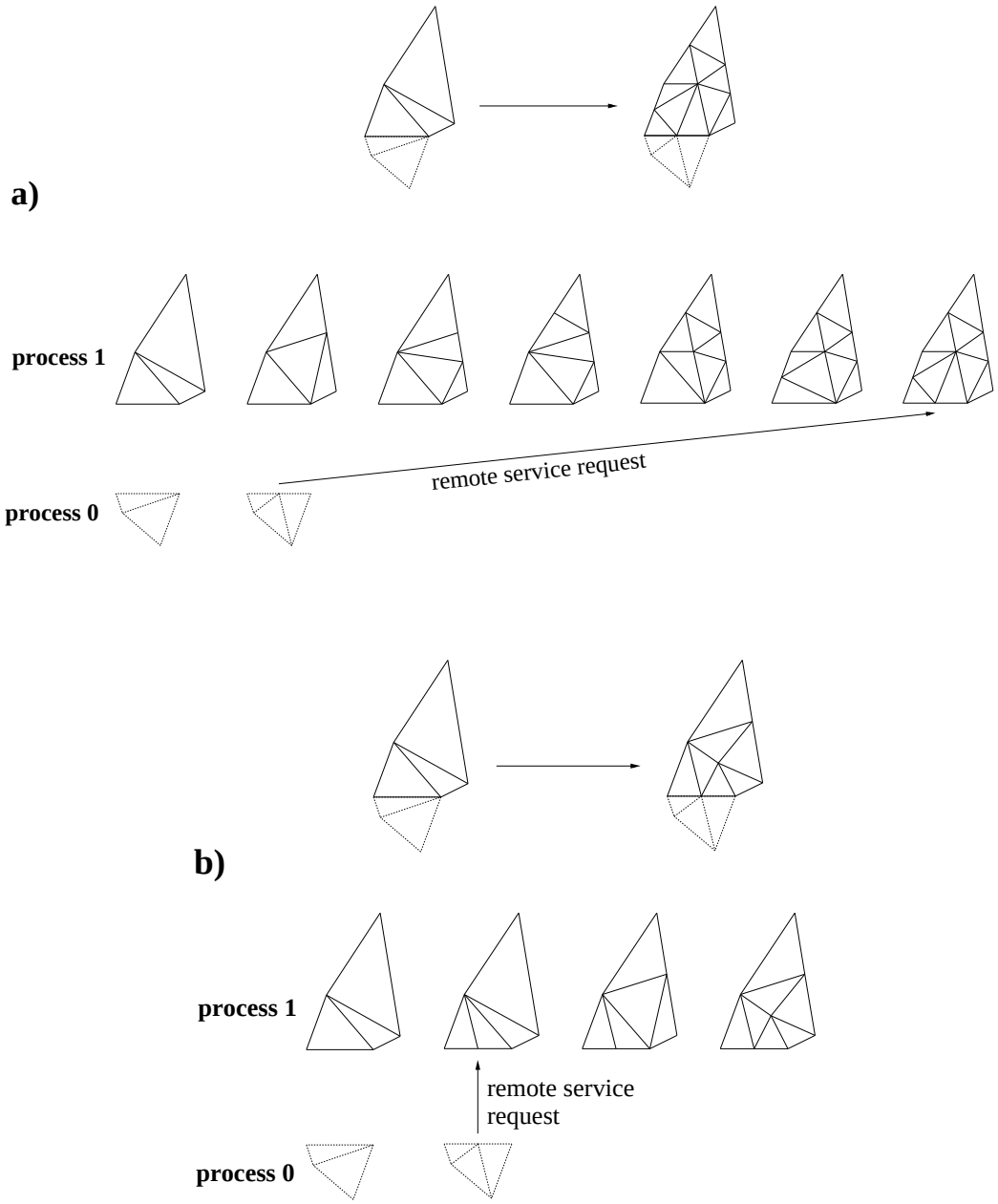

Figure 7.17: The indeterminism in the asynchronous version: The final mesh differs depending on the point of time when the remote service request from process 0 reaches process 1. At (a) the remote service request reaches process 1 after five local point insertions while at (b) the remote request is served before any local point insertions are done.

Conclusions

In this work research was done in two major fields: Parallel computing and Delaunay mesh generation. Parallel computing is supposed to increase the performance of computational systems but its usefulness highly depends on the application to be parallelized. The application which was dealt with in this work comes from the second field of research, Delaunay mesh generation.

First, the synchronous and asynchronous parallel programming model were compared. The result shows that on the one hand synchronous parallel programming is mainly used due to its simpler handling. On the other hand, algorithms with unpredictable behavior are hard to be implemented using synchronous parallelism, they better can be implemented in the asynchronous parallel model.

The nature of the Bowyer-Watson algorithm, which is used for Delaunay mesh generation, turned out to fit well into the asynchronous parallel model. Therefore, a parallel Delaunay triangulation algorithm was developed using asynchronous communication.

In order to decrease communication costs the constrained parallel Delaunay triangulation was introduced. This algorithm produces a mesh of similar quality than the parallel "entire mesh" Delaunay triangulation but achieves a significant performance improvement.

Then currently available asynchronous communication tools were investigated. None of them satisfies the needs of parallel programmers to make programming more comfortable and efficient. Thus, a new layer was developed to provide better support for asynchronously parallel programming.

Finally asynchronous programming using this newly introduced layer was evaluated. The parallel constrained Delaunay triangulation was implemented two times, first, using synchronous parallel programming based on PVM and, second, using asynchronous parallel programming with the newly developed layer. Numerical experiments demonstrate a much better performance of the implementation using the asynchronous communication layer.

The results of this work show that parallel computing using asynchronous communication includes the potential to efficiently solve problems which have not been solved efficiently up to now.

Future activities dealing with asynchronous communication should consider the importance of out-of-core computations. At first glance parallel computation and out-of-core computation do not seem to have anything in common. However, in both cases computations are carried out which have to deal with data not available in local main memory. In case of parallel computing the non-local data are at remote processors, in case of out-of-core computations those data can be found in a mass storage device. The main difference between parallel and out-of-core computation is the location of computation. Both issues could be combined

in a new model. The programmer should not have to deal with all these things. Therefore, the new model should provide a global address space and efficient memory management tools. In Appendix C such a model, the *delayed procedure execution* model, is presented.

If data are not local a delayed procedure is not executed immediately but delayed, either locally as long as data are not in the main memory or remotely if data are on remote processors. The key for this model is its memory management which organizes the global address space in buckets which can be moved from one physical storage unit to another one.

Especially problems which are not only computational intensive but also need much memory space are the target of this model. For example, parallel Delaunay triangulation is an application which needs large amounts of memory. But, of course, this model is not limited to mesh generation but is open to all tasks dealing with parallelism or out-of-core computation.

Appendix A

The Data Structure

A.1 Sequential Delaunay Triangulation

In this section the C source code of the data structure described in Section 2.2.4 is shown.

```
typedef struct point_s        /* contains the coordinates of a
                                 point */
{
    double x, y;
}
point;

typedef struct list_el_s      /* element of edge lists */
{
    struct list_el_s *next;
    struct edge_s *item;
}
list_el;

typedef struct site_s         /* data structure for sites */
{
    point sit_co;             /* location */
    list_el *edge_list;       /* points to the list of edges
                                 ending at this site */
}
site;

typedef struct edge_s
{
    struct triangle_s *tris[3]; /* pointers to the two adjacent
                                   triangles */
    site *sits[2];            /* pointers to the two sites of
                                 the edge */
}
edge;

typedef struct triangle_s     /* delaunay triangle */
{
    struct triangle_s *next_tri; /* there is a list of all
```

```
                                triangles, 'next_tri' points to
                                the next triangle in this list,
                                NULL if it is the last one */
    struct triangle_s *previous_tri; /* points to the previous
                                triangle in the list, NULL if
                                the first one */
    point cc;                   /* circumcenter */
    double rad_sqr;             /* square of the circumradius */
    edge *eds[3];               /* pointers to the three edges */
}
triangle;
```

A.2 "Entire Mesh" Parallel Delaunay Triangulation

In this section the C source code of the data structure described in Section 4.2.3 is shown.

```
typedef struct point_s          /* contains the coordinates of a
                                   point */
{
    double x, y;
}
point;

typedef global_ptr_s
{
    unsigned int proc_id;
    void *local_ptr;
}
global_ptr;

typedef struct list_el_s        /* element of edge lists */
{
    struct list_el_s *next;
    global_ptr item;
}
list_el;

typedef struct site_s           /* data structure for sites */
{
    point sit_co;               /* location */
    list_el *edge_list;         /* points to the list of edges
                                   ending at this site */
}
```

```
site;

typedef struct edge_s
{
    global_ptr tris[2];       /* pointer to the two adjacent
                                 triangles, note: both might be
                                 remote */
    site *sit0;               /* pointer to the local site of
                                 the edge */
    global_ptr sits1;         /* pointer to the global site of
                                 the edge */
}
edge;

typedef struct triangle_s     /* delaunay triangle */
{
    struct triangle_s *next_tri; /* there is a list of all
                                 triangles, 'next_tri' points to
                                 the next triangle in this list,
                                 NULL if it is the last one */
    struct triangle_s *previous_tri; /* points to the previous
                                 triangle in the list, NULL if
                                 the first one */
    point cc;                 /* circumcenter */
    double rad_sqr;           /* square of the circumradius */
    edge *eds[2];             /* pointer to the local edges */
    global_ptr eds2;          /* pointer to the global edges */
    char in_use;              /* nonzero if involved in cavity*/
}
triangle;

typedef struct remote_flow_control_s
{
    unsigned int father;
    unsigned int me;
    int numb_of_sons;
}
remote_flow_control;
```

A.3 Parallel Constrained Delaunay Triangulation

The following C source code describes the data structure of a border edge (see Section 4.3.5).

```
typedef struct border_edge_s
{
    site *sits[2];            /* pointers to the two sites */
    triangle *tri;            /* pointer to the local triangle*/
    global_ptr dup_border_ed; /* pointer to the second instance
                                 of this border edge at the
                                 remote processor, NULL pointer
                                 if not available */
    struct border_edge_s *new_eds[2]; /* if the edge gets split
                                 these are the pointers to the
                                 split edges */
    edge *father_ed;          /* pointer to the father edge */
}
border_edge;
```

Appendix B

Specification of the LAC

B.1 Remote Service Requests

```
void lac_rsr(proc_id, rem_fun, kind_of_rsr,
             int_nr, direct_block_nr, reg_block_nr,
             {int} int_nr times,
             {direct_block_src *, size, direct_block_dest *}
                 direct_block_nr times,
             {reg_block_src *, size, reg_block_dest *}
                 reg_block_nr times,
             {direct_end_fun, direct_end_arg0, direct_end_arg1}
                 if direct_block_nr)
```

This is the basic call to communicate on that layer. It initiates a remote service request on a remote processor `proc_id`. At that processor `rem_fun` will be executed, either immediately or delayed depending on `kind_of_rsr`. If `kind_of_rsr` is `IMMEDIATE` then `rem_fun` is executed as soon as the remote processor receives the remote service request. This means `rem_fun` is executed inside of `poll_rsr` (see Section B.4) and must not contain heavy computations or remote service request calls. If the value of `kind_of_rsr` is `REGULAR` `rem_fun` is executed at the next call of `do_rsr` (see Section B.4) once it arrives there.

The next three arguments at the remote service request call specify the number and type of arguments and data blocks to be transfered along the request. There are three types of data: the regular argument which occurs `int_nr` times, consisting of four bytes like an integer, the block to be transfered directly (`direct_block_nr` times) and the regular block (`reg_block_nr` times) which has to be copied immediately to the system buffer.

To transfer a regular block, the third kind of arguments, its source address `reg_block_src` and size `size` has to be specified. Then the destination address `reg_block_dest` can be specified. The call can handle three kinds of destination addresses: a regular address, where the block is transfered to, `NULL` if the block should just be kept in the system buffer at the remote processor or `ANYWHERE`. In this case memory is allocated for that block at the remote processor and it is transfered to that space. Then the user has the responsibility to take care of this part of the memory, i. e., freeing it.

If such a block is specified at a remote service request call, then the block is copied immediately to a system buffer where data are gathered to be sent

to the remote processor. This is different at the blocks which are specified as the second kind of arguments. These blocks may be transferred directly to the remote processor. Therefore, the memory space where these blocks are, can't be reused immediately because sending can take a while. Hence, a function, `direct_end_fun`, has to be specified which is called as soon as the memory, where these blocks are allocated at, can be reused. This function takes two four byte arguments, `direct_end_arg0` and `direct_end_arg1`.

Such an end function is one possibility of giving feedback to the user when he can reuse the memory space of the block being transfered directly. Another possibility would be to use acknowledgment variables. Those are set if the memory can be reused again. Ports, Section 5.3, and DMCS, Section 5.4, use acknowledgment variables, while Active Messages, Section 5.1, uses end functions.

It turned out that using end functions supports better asynchronous programming while acknowledgment variables are useful for a more synchronized programming. Acknowledgment variables have to be checked until they are set. Then action can be set, like freeing or rewriting the memory space. But this is a contradiction to the asynchronous programmer's philosophy, that is: "send and forget". Using end functions these actions will take place automatically without any checking of results of past calls.

`rem_fun` should be defined as following:

```
void rem_fun(source_id,
             {int} int_nr times,
             {direct_block_dest *, size} direct_block_nr times,
             {reg_block_dest *, size} reg_block_nr times)
```

`direct_end_fun` should be defined in the following way:

```
void direct_end_fun(direct_end_arg0, direct_end_arg1)
```

Before being sent to remote processors remote service requests can be queued in order to be sent at once. Therefore, a queue for each processor can be specified where remote service requests are stored until the queue is flushed either manually by the user or automatically because it got too long. This service is only available for remote service request of kind `REGULAR`, `IMMEDIATE` remote service requests are always sent immediately and, therefore, one by one.

```
void lac_queue_len(proc_id, length)
```

has to be used to specify the length of the queue for processor `proc_id`, where `length` is an integer which denotes the maximum queue length. If the number of queued remote service requests exceeds `length` then the queue is flushed. The default value of `length` is 0, this means there is no queuing of remote service requests at initialization. If `proc_id` is negative the maximum length of all queues are set to `length`.

```
void lac_flush(proc_id)
```

flushes the queue for processor `proc_id` and all remote service requests in this queue are sent. `proc_id` as a negative value means to flush the queues for all processors.

```
void lac_cancel(proc_id)
```

cancels all remote service requests in the queue for processor `proc_id`. `proc_id` as negative value means to cancel all remote service requests queued for any processor. All `direct_end_fun`s, if specified and not yet called, of the canceled remote service requests are invoked.

B.2 Registration of Remote Functions

All functions used as remote service request, i. e., `rem_fun` in `lac_rsr`, have to be registered by calling:

```
void lac_fun_register(fun, debug_msg)
```

`fun` is the function to be registered and `debug_msg` a string that is used for debugging or tracing as explained in Section B.5. On every processor the same functions have to be registered in the same sequence such that the state of registering these function is the same at the local processor initiating the request and the remote processor serving the request.

B.3 Synchronization

As described in Section 6.1.3 the layer provides a function for explicit synchronization:

```
int lac_done()
```

This function blocks as long as one of the following two states can be reported by returning 0 or 1: If 0 is returned, remote service requests still have to be done. Otherwise, if 1 is returned, all remote service requests are served, all processors entered `lac_done` and 1 is returned at all processors.

Handling synchronization variables is supported by the function `lac_syncro`. A synchronization variable is implemented as field of as many bits as processors in use. Hence, each bit belongs to a certain processor. Each synchronization variable has an instance at every processor. This synchronization function deals with the local instance of the synchronization variables as following:

```
int lac_syncro(proc_id, sync_var, op)
```

where `op` may be one of the following:

`RESET_ALL`: All bits of all synchronization variables are reset, `proc_id` and `sync_var` are ignored.

`TEST_BIT_OF_PROC_OF_VAR`: If bit `proc_id` of variable `sync_var` is set, a nonzero value is returned, otherwise zero. If `proc_id` is negative a nonzero value is returned if any bit of the variable is set.

`TEST_ALL_BITS_OF_VAR`: If all bits of `sync_var` are set, a nonzero value is returned, otherwise zero. `proc_id` is ignored.

`STORE_VAR`: The name of `sync_var` is stored for further usage with `op` equal to `TEST_ALL_BITS_OF_BOTH_VARS`. `proc_id` is ignored.

`TEST_ALL_BITS_OF_BOTH_VARS`: `sync_var` and the last stored variable are ored. If all bits of the result are set, a nonzero value is returned otherwise zero. `proc_id` is ignored.

`SET_BIT_OF_PROC_OF_VAR`: Bit `proc_id` of `sync_var` is set. If `proc_id` is negative all bits of `sync_var` are set.

`RESET_VAR_EXCEPT_BIT_OF_PROC`: All bits of `sync_var` are reset except the bit `proc_id` which is set. If `proc_id` is negative, `proc_id` is handled as being the id of the local processor.

`SEND_VAR_TO_PROC`: A request is sent to processor `proc_id` to set the bit of the local processor at its instance of variable `sync_var`. If `proc_id` is negative, a request is sent to all processors to set the bit of the local processor at their instances of variable `sync_var`. This also means that the bit of the local processor is set at the local instance of `sync_var`.

B.4 Polling and Serving Remote Service Requests

In order to receive remote service requests polling has to be done:

```
void lac_poll()
```

This function should be called frequently in order to keep the network clear. Remote service requests of type `IMMEDIATE` (see B.1) will be executed immediately, that means inside the call of `lac_poll`. No communication to other processors may be performed inside of these immediate remote service requests, i. e., no `lac_poll` or `lac_rsr`.

All other remote service requests are queued and executed when

```
void lac_do_rsr()
```

is called.

B.5 Debugging and Tracing

As mentioned in Section 6.1.4 minimum debugging and tracing information can be obtained by using the provided commands of *LAC*. As described in Section B.2 all functions used as remote service requests have to be registered in advance. At registration these functions a string (`debug_msg`) has to be specified that is printed once at the initiating processor when the remote service request is called to remotely execute `fun` and twice at the processor where `fun` is executed, first when `fun` is called and finally when it is terminated.

Debugging can be switched on or off by calling:

```
int lac_debug(op)
```

where `op`, equal to `LAC_DEBUG_ON`, switches debugging on and, equal to `LAC_DEBUG_OFF`, off. If `op` is equal to `LAC_DEBUG_STATUS` the function returns a nonzero value if debugging is on otherwise zero. The default value is `LAC_DEBUG_OFF`.

At calling a remote service request the following information is printed:

- The processor id where the remote service request is called.
- The message which was stored at registering the function which will be invoked remotely. In most cases this message will be the name of the function.
- The processor id where the remote service is requested.
- An unique id of this remote service request.

When the remote service request is done at the remote processor, the following information is printed twice, at entering the function and leaving it:

- The processor id where the remote service request is served.
- The message which was stored at registration of the function.
- The processor id where this remote service request was called from.
- The unique id of this remote service request.

Especially the unique remote service request id is very important to trace the run of a program. It might happen, for example, that the printout of entering and leaving a remote service request comes earlier than the call at the origin processor. Without this unique id it is easy to be left in confusion.

B.6 Miscellaneous

The layer has to be initialized by calling

```
void lac_initialize(argc_p, argv_p)
```

where `argc_p` and `argv_p` are pointers to the known `argc` and `argv`.

The id of the local processor and the number of processors are returned by:

```
int lac_proc_id()
int lac_proc_num()
```

respectively. Processor ids are integers and in the range of 0 to `lac_proc_id` − 1.

```
void lac_end()
```

should be called at the end of each process.

Appendix C

Delayed Procedure Execution (DPE)

Nowadays processor performance is rapidly increasing as long as data are locally available. But if applications become too large to fit on a single processor and its local memory in terms of execution time or memory requirements, computation has to be distributed to more than one processing unit (*parallel* computation) or data have to be stored on a second level storage (*out of core* computation).

In both cases non local data are involved in computations which may cause a significant decrease on performance. Solutions for either problem can be found in [4, 35, 50] but they only show reasonable performance at computations on regular data.

These solutions are using the *synchronously parallel* programming model or *prefetching* at *out of core* computations. There knowledge has to be available in advance when a message will arrive to be received (section 3) or which data will be needed next to be prefetched, respectively. This information is missing at unpredictable computations on irregular data which, in this case, will result in poor performance.

Available tools for asynchronous parallel communication, as described in sections 5 and 6, only cover part of this issue, the parallel communication on a low level.

The *delayed procedure execution* model addresses problems appearing at implementing the following kind of computations:

- The *asynchronously parallel* programming model allows efficient implementations. The *delayed procedure execution* model supports *asynchronously parallel* programming.

- *Out of core* computations need data in primary memory to perform operations. If needed data are not in primary memory they have to be fetched from secondary memory. In order not to wait for that I/O operation data can either be prefetched, possible for predictable computations, or the execution of the operations has to be delayed until data arrive. The DPE model supports the second way by putting such operations in a queue to be executed later while computation, which is independent of these operations, goes on. The *delayed procedure execution* model supports this way.

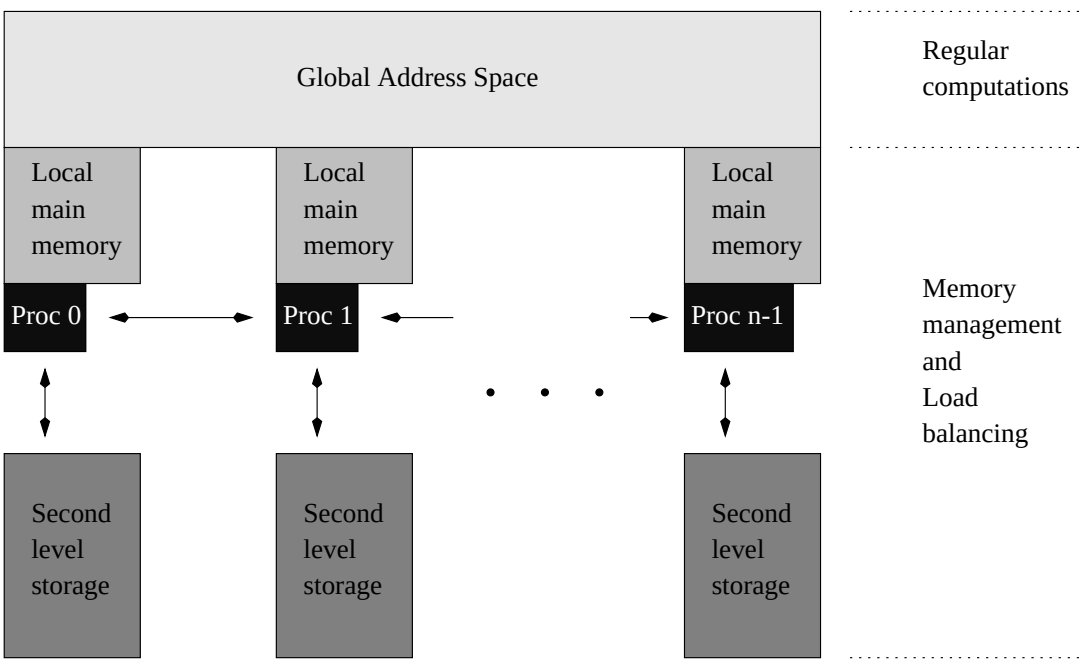

Figure C.1: The *delayed procedure execution* model outreaches two levels of abstraction. Regular Computations are performed in the global address space. On a lower level the user is allowed to map that global address space to local main memory or second level storage of processors. At this level memory management and load balancing is performed.

C.1 Overview

The *delayed procedure execution (DPE)* model distinguishes two levels of abstraction in parallelism and *out-of-core* computations. On the higher level a global address space is provided where computations are performed. The lower level is the home of the memory management and the load balancing where only the user is able to and has to deal with processors since he is the one who knows about locality of data. Figure C.1 shows the different levels of abstraction in the *delayed procedure execution* model.

Distinguishing these two levels of abstraction is the major key of the *DPE* model. Algorithms and programs can be designed regardless of how the global address space is mapped physically. This eases the design process with all its consequences as less source code, fewer errors, etc.

The global memory is organized in buckets. Buckets are blocks of memory whose management is controlled by the user. Routines to do so are available in the DPE model. Buckets can be created at one processor and, if necessary, migrate to another processor or from the main to a second level memory. They can also be split if they get too big, etc. Figure C.2 shows the location of buckets in main memories and in second level storages.

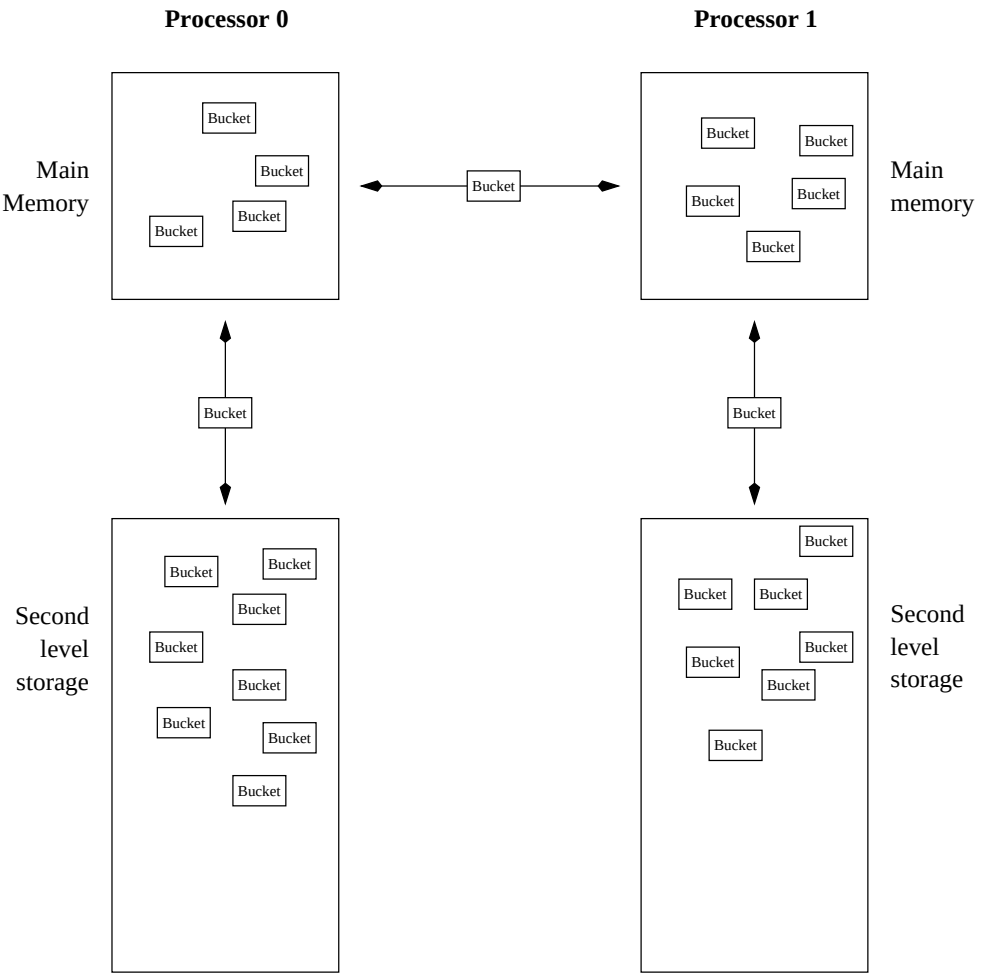

Figure C.2: The main and second level memories of two processors. The global address space is managed through buckets. These are memory blocks which can be created, shifted between processors or between main and second level memory, split, etc.

Global pointers are used to address global data. These global pointers point into buckets. Since, *delayed procedures* are performed on global data all addressed buckets have to be checked if local or non-local. If all buckets of the needed global data are local, the procedure is executed immediately as a regular procedure. Otherwise its execution is delayed. Then the procedure is stored either in a local task queue to wait for the arrival of the data or it is transferred to a remote processor. The decision is based on the location of the involved data, on load distribution and on user preferences.

The DPE model is designed to support all kind of asynchronous programming on all kind of computer architecture. The regular computation of a problem, which is performed at the higher level of the DPE model, can remain unchanged regardless of which platform the program is running on. Only the memory man-

agement and the load balancing is subject to adaptions if a different architecture is used. Therefore, the user should not mess up with these two levels of the DPE model. In the next section more details of the model are explained.

C.2 The Concept of the DPE Model

C.2.1 Computations in the Global Address Space

This section copes with the higher level part of the *delayed procedure execution* model. Here computations are performed in a global address space. Local memories and processors are hidden on this level where the following issues are supported:

Global pointers are used to address global data where *delayed procedures* can be performed on. They consist of two components: A physical address and the id of the bucket where the global data are located. Since, it can happen that buckets are removed and data are moved to another bucket, a global pointer could have to be updated. This is done on demand, for example, automatically at a *delayed procedure* call. Furthermore, *global pointers* can be copied and incremented.

The *delayed procedure* call: Operations on global data only can be performed by using a *delayed procedure* call. If all specified global data are local the operations are performed immediately. Otherwise their execution is delayed. Two routines, a basic call and an extended version, are available for calling *delayed procedures*.

The basic call allows an arbitrary number of local variables, simple integers or blocks, and global data. Global data have to be specified in one of three possible ways: They can be read only data. This means that data specified in this way will only be read from their location. Then data can be specified as write only. This means that data produced at the current *delayed procedure* and written to their location after the procedure has terminated. As third options global data can be specified as both, read and write. This is a combination of the first two options. For example, let's assume a, b and c are global data. In order to perform the operation $c = a + b$, a and b has to be specified as read data, while c is specified as write data. If operation $c = c + a$ need to be done then a has to be specified as read data again but c has to be specified as both, read and write data.

Whenever, immediately or delayed, or wherever, local or remote, the procedure is executed, all global data are local then. This might be the data themself if the global data are local or, otherwise, just a local copy. Then these data can be accessed by local pointers. This is important to achieve sufficient efficiency.

Then it can be determined where the procedure should be performed by specifying a global pointer. Wherever the bucket, where this pointer points to, lies, the procedure is executed. If no pointer is specified the system schedules the execution of the procedure. Then load balancing is a criteria.

In order to make save reading and writing of global data at shared memory architectures or at using preemptive threads, buckets can be locked for exclusive use for the duration of reading and writing. If the bucket is local then the state of locking will last the whole execution time of the procedure.

The extended version of the *delayed procedure* call the specification of where the procedure has to be executed is combined with a function. Such a function may come from the low level part of the DPE model and, for example, may control the load balancing.

Another feature of the extended version is the possibility of specifying "end functions" for local blocks. Out of efficiency reasons local block may be specified for direct use by the system. Two cases have to be distinguished: First, the procedure is executed at a remote processor. Then the block has to be transferred to that remote processor. "Direct use by the system" means that the block is sent directly from the original location. Since, the *delayed procedure* call generally returns quicker than sending is done, the block is not allowed to be reused immediately afterwards. When the use by the system is finished the "end function" is invoked. In the second case the *delayed procedure* is executed locally. Then the block is needed until the procedure finishes. After finishing the procedure, the "end function" is called.

"End functions" may do, for example, cleaning actions as freeing the block. If the block has to be reused by the user immediately after the return of the *delayed procedure* then the the local blocks can be declared as at the basic call: The block is immediately copied to a system buffer and the "end function" is not needed.

***Delayed procedure* call queues** collect *delayed procedure* calls (in the case that they cannot be executed immediately) to be initiated at once. First, this may improve the efficiency of the program, i. e., if these calls go to a remote processor, and, second, provides convenience for the programmer, since, he can cancel the calls in the queue if necessary. Otherwise the queue can be flushed automatically or manually.

Allocating global memory space is done by specifying size of the object or, if the type of this object is already registered, only the type needs to be specified. Registering types of objects is used to increase efficiency of allocation and freeing memory, bucket splitting and garbage collection. More about registering memory items can be found in section C.2.2.

Synchronization similar to the one described in section 6.2, is also available in the *delayed procedure execution* model. where certain algorithms are developed to synchronize asynchronous processes.

Debugging: A minimum of debugging features are provided by system. This basically is the possibility to trace *delayed procedure execution* and similar to the one presented in section 6.2.

C.2.2 Memory Management and Load Balancing

In this section the routines for managing the memory of the global address space are explained. This has to be done on a lower level of abstraction than in section C.2.1. The user has to deal with processors, how to distribute the data and how to balance the load. Generally there are two possibilities to balance the load. First, by data distribution, since, operations on global data are mostly performed where these data are located (*data centric computations*). Second, by specifying a certain bucket at calling a *delayed procedure*. Especially the extended call allows a function to specify this bucket. This function can operate on this lower level. In such a way the higher and lower level computations are not messed up. Alternatively the user may not specify any bucket as location for the execution in order to let the system balance the load. But this option will not be as efficient, since, it only is the user who knows about locality of data. At this level primitives are provided for the following features:

Manipulating Buckets. A bucket is an object where memory is managed. All global data are allocated in buckets. The following routines are available to manipulate buckets.

Creation: Buckets can be created at specified processors.

Overflow Function: A function and the amount of data can be specified and if the amount of data contained by the bucket exceeds the specified amount of data then this function has to be called. For example, a bucket is created. More and more data are allocated. If the bucket overflows the specified "overflow function" is called. This function may split the bucket.

Splitting: Buckets may be split. For example, this is useful if the bucket becomes too big or if data of a bucket should be distributed to other processors. Buckets only can be split efficiently if memory items are registered. Then the user has to provide a function which is used by the system to decide which item is put into which of the new buckets. This function is called for each item at splitting. Unregistered items are only distributed equally to the new buckets which might not result in a good locality. Buckets also can be merged. The system also

provides a function to get the current size of the bucket. Exceeding a certain size may be a reason for splitting.

Garbage Collection: If the fragmentation of the memory becomes a problem garbage collection can be requested.

Migration of buckets is also possible. Buckets can be migrated from one to another processor or from main memory (*in-core*) to the second level memory (*out-of-core*) or vice versa.

Basic Low Level Calls. The system provides function calls to find out how many processors are working in parallel and the ID of the local processor.

C.3 DPE Model Implementation

In this section a few ideas how to implement the DPE model are described.

C.3.1 Global Pointers and Bucket Tables

A global pointer consists of two components: the bucket and the physical address. A bucket table is used to check the validity of global pointers. In each local memory is a copy of the bucket table. If a global pointer is used its bucket is looked up in the local copy of the bucket table. If the bucket still exists the global pointer is considered as up to date. If the bucket is located in a remote memory the request is sent to the remote processor and there the remote copy of the bucket table is checked again.

Buckets can be removed by bucket manipulation commands (see section C.4.2). Then an entry in the bucket table will tell how to translate the global pointer to an up to date one. The deletion of a bucket may be caused by three commands:

`dp_buck_migrate` (see section C.4.2) only moves the bucket from one to another memory. Each global pointer pointing to an item in this bucket just gets a new bucket id and an offset is added to the physical pointer.

`dp_buck_garb_coll` (see section C.4.2) moves items within the bucket. It also gets a new bucket id but each physical pointer has to be translated with a list. This list consists of sorted old addresses and its offsets. For each address n to $n + 1$ in the table offset n has to be added. It's easy to see that the efficiency depends on the number of items having been moved not contiguous with other items. But, since, garbage collection should only be necessary if the application uses rather large items (smaller items should be able to fill into the gaps) the efficiency should be sufficient.

`dp_buck_split` (see section C.4.2) causes a situation similar to the garbage collection. The difference is an additional column in the list telling about the new bucket id. The efficiency also depends of how the bucket is split, how many single small items need an extra entry.

If buckets are manipulated changes are entered in the local copy of the bucket table. Then these changes are broadcasted to all remote memories. But, since, a change in the local copy of the bucket table may not reach the remote copy of the bucket table before a request from the remote memory to the local one has left, the request has to check the local copy of the bucket table, too.

C.4 The DPE Specification

In this section the concept and the calls of *delayed procedures* are specified.

C.4.1 High Level DPE Primitives

Global Pointers

Global data are addressed by global pointers. Global pointers contain a pointer to the bucket where the item is in and a physical address to find the item in the bucket.

Global pointers do not become invalid by moving or splitting buckets. Routines as `dp_xxx_call` translate old global pointers, given as input arguments, to up to date global pointers and update them. In this way global pointers are updated on demand.

Global pointers can be copied or incremented by calling:

```
dp_cp_global_ptr(global_ptr_dest, global_ptr_src)
dp_gptr_incr(global_ptr, step)
```

The later command changes `global_ptr` to point to a `step` bytes higher address. A negative `step` decrements `global_ptr`.

The Delayed Procedure Call

A procedure can be called to be executed when the specified data are available. The return value shows if the procedure was executed immediately or delayed. The buckets where the specified global data are in may be locked for exclusive use of that procedure. This can be useful at preemptive threads or shared memory parallel computers. *Delayed procedure* calls within a *delayed procedure* are allowed (recursive).

The DPE model recognizes deadlocks caused by the exclusive use of buckets. These deadlocks are resolved automatically by the system.

The Basic Call

```
int dp_basic_call(fun, exec_pointer *, exclusive,
                  l_nr, lb_nr,
                  {int} l_nr times, {local_ptr, size} lb_nr times,
                  gr_nr, gw_nr, grw_nr,
                  {global_ptr *, size} gr_nr, gw_nr, grw_nr times}
```

`dp_basic_call` invokes function `fun` when the specified global data are available. The return value is 1 if all global data are local and `fun` is immediately executed (as a regular function) otherwise 0 if the execution is delayed due to the non locality of the data.

`exec_pointer` is a pointer to a global pointer specifying the bucket in whose context `fun` should be executed. `exec_pointer` may be a `NULL` pointer to indicate that the user does not care. `exec_pointer` is updated if not `NULL` (see Section C.4.1). Note: If commands like `dp_buck_alloc` are used in `fun` `exec_pointer` is expected NOT to be `NULL`.

If exclusive access to global data is required (`exclusive` is nonzero), buckets, where the global data are in, are locked while reading/writing. This means that if the bucket is local, it is locked for other processes during the whole execution otherwise, if the bucket is remote, only an interruption of the implicit reading to/writing from the local copy at the begin/end is forbidden.

The reason of locking buckets is not to synchronize user processes but only to provide the possibility for the exclusive use of atomic I/O to the global address space.

If a bucket is exclusively used by a procedure and this procedure calls another procedure (child) which is executed immediately then this child is allowed to use the buckets locked by its father. This right is inherit to all successors as long as they are executed immediately.

`fun` can be called with local and global data specified. `l_nr` and `lb_nr` is the number of four byte (int) and block arguments, respectively. Afterwards `l_nr` four byte arguments (int) and `lb_nr` block arguments, consisting of a pointer to and the size of block, are following.

Then the blocks of global data are next. Three parameters specifying the number of blocks which are to read only, to write only and both, to read and to write, respectively. Each block is specified by a global pointer and its size. The global pointer is updated if necessary (see Section C.4.1).

`fun` should be declared as following:

```
void fun(execute,  {int} l_nr times,
         {local_ptr} lb_nr, gr_nr, gw_nr and grw_nr times)
```

`execute` is 1 if `fun` is executed immediately otherwise 0. The other parameters correspond to the call of `dp_basic_call`. Note: Only local pointers point to the

specified global data. This might be a local copy or the real location of the global data if the bucket is local. Local copies of remote blocks are written back to their origin destination if specified as `gw` or `grw` block.

The Extended Call

```
int dp_ext_call(fun, exec_buf, exec_buf_len, exec_fun,
                l_nr, lb_nr,
                {int} l_nr times, {local_ptr, size} lb_nr times,
                gr_nr, gw_nr, grw_nr,
                {global_ptr *, size, exclusive} gr_nr, gw_nr
                    and grw_nr times,
                {end_fun, end_arg0, end_arg1} if lb_nr)
```

`dp_ext_call` is the extended version of `dp_basic_call`. The differences are:

- As in `dp_basic_call` the bucket, where `fun` is executed, is specified by a global pointer pointing into this bucket. This global pointer `exec_pointer` has to be the output of the routine `exec_fun` which has to be declared as following:

  ```
  exec_fun(exec_pointer, exec_buf)
  ```

  where `exec_buf` is a block of length `exec_len` and `exec_pointer` points to a space provided to enter the global pointer.

  `exec_fun` is called to determine which bucket has to be executed at. If this bucket is remote the procedure call is traveling to the remote processor. Then `exec_fun` is executed a second time to check if `fun` still has to be executed at the same bucket. The bucket may have been split or migrated to another processor making the procedure call traveling again.

  `exec_len` equal 0 means that the user doesn't care where the procedure is executed. If `exec_len` is not equal 0 and `exec_fun` is equal `NULL`, `exec_buf` is assumed to be a pointer to a global pointer like `exec_pointer` in `dp_basic_call`.

  For example, `dp_basic_call(..., exec_pointer, ...)` is equal to `dp_ext_call(..., exec_pointer, -1, NULL, ...)` or equal to `dp_ext_call( ..., exec_pointer, sizeof(global_ptr), e_fun, ...)` where `e_fun` is defined as

  ```
  e_fun(exec_pointer, exec_pointer0)
  {
      dp_cp_global_ptr(exec_pointer, exec_pointer0);
  }
  ```

There is a small difference in the later option: `exec_pointer` would not be updated.

- The `exclusive` argument has to be specified for each block of global data. If `exclusive` is nonzero the bucket, where this block of global data lies in, is locked. More than one blocks of the specified global data can be located in the same bucket. If at least one of their `exclusive` arguments is nonzero the bucket is locked.

- Local read only blocks may be not usable immediately after the return of `dp_ext_call` if the execution of the procedure is delayed. Then, for example, it may be necessary to transfer the block to a remote processor or just to wait till the specified non local data arrived. In both cases the block, although read only, can't reused immediately. In the first case `end_fun` is called as soon as the block is transferred to the remote processor, in the second case the call takes place after `fun` is executed. `end_fun` is not executed if `fun` is executed immediately or if no local read only blocks are specified (`lb_nr` equal 0). Then `end_fun`, `end_arg0` and `end_arg1` is ignored.

  `end_fun` will always called at the location where the *delayed procedure* is called and should be declared as following:

  ```
  end_fun(end_arg0, end_arg1)
  ```

  `end_arg0` and `end_arg1` are the arguments specified in `dp_ext_call`. `end_fun` is always executed locally, this means at the processor where `dp_ext_call` was made.

  If `end_fun` is specified as `NULL` and `end_arg0` is nonzero no end function is executed when the read blocks are usable again. If `end_fun` is specified as `NULL` and `end_arg0` is zero the read only blocks are buffered and usable immediately after `dp_ext_call` returns.

`fun` should be declared as done in Section C.4.1.

Queuing Delayed Procedure Calls

Depending on the architecture and application it may be more efficient and more convenient for the programmer not to send delayed requests to the remote processor or to the second level storage immediately one by one but to queue them Then the requests can be collected and sent at once. This optimization can reduce the number of messages between the processors significantly.

```
void dp_call_queue(queue_len)
```

specifies the length of the queue. `queue_len` specifies the maximum number of calls stored in the queue. If the number of stored calls exceeds `queue_len` all calls in the queues are flushed. Initially `queue_len` is set to 0. It can be changed anytime, even to different values on different processors on a distributed memory architecture.

```
dp_call_flush(void)
dp_call_cancel(void)
```

flushes and cancels, respectively, all calls in the queue if any.

The following is useful only on a distributed memory architecture: *Delayed procedures* in a queue which are flushed together and executed on the same remote processor are seen as a group and they get their own queue for *delayed procedure* calls on that processor. But its `queue_len` is used. All *delayed procedures* called in these *delayed procedures* are stored in this queue. After the last call of that group, the queue is flushed automatically and removed. Of course, `dp_call_flush` and `dp_call_cancel` can be used, too. Note: `dp_call_flush` and `dp_call_cancel` only has effect on the queue of the group while `dp_call_queue` effects the settings of the memory of the processor.

Memory Allocation

Memory in a bucket can be allocated using

```
dp_alloc(global_ptr, item, size)
```

where `global_ptr` is the global address of the allocated memory. `item` specifies the kind of item, may be -1 if not registered, and `size` specifies the size. This argument is ignored if the size was already specified at registration. More details about registration of memory items are explained in Section C.4.2

```
dp_free(global_ptr, item)
```

frees the memory space where `global_ptr` is pointing to.

Synchronization

Synchronization is supported: A synchronization object and a routine to check if all delayed procedures have been executed, yet. This issue will be handled similar as done in Section 6.2, 6.3.2 and B.3.

C.4.2 Low Level DPE Primitives

Bucket Manipulation

The DPE model provides a global address space which is organized in buckets. Buckets are transferable between memories of processors as well as between main memory and second level memory.

The memory management is one of the major keys of the DPE model. It has to be controlled by the user. The DPE model just provides routines for easy managing the global memory. This is the better way because the locality of irregular data is highly application dependent and can only be exploited by the user who has the knowledge about it.

The price to pay is that the user has to step down to a lower level for managing the memory (i. e., to work with processors).

Creation

```
dp_buck_create(proc_id, id_pointer)
```

creates a bucket at processor `proc_id`. `id_pointer` is a global pointer which points into the bucket. It can be used to initiate procedure calls at this bucket but not to address items.

It is often necessary to react when the amount of data in a bucket exceeds a certain limit. A function may be specified to be executed locally when the bucket "overflows". A bucket "overflows" if the entire space (the used one and the freed but unused gaps (see Section C.4.2) exceeds `overflow_size`.

```
dp_buck_overflow_fun(id_pointer, overflow_size,
                     overflow_fun, arg_block, arg_block_size)
```

specifies this function to be declared as following:

```
overflow_fun(id_pointer, arg_block)
```

The argument block is `arg_block_size` large. `dp_buck_overflow_fun` can be called several times to overwrite the previous value. Specifying `dp_overflow_fun` with `NULL` will disable the overflow feature. This feature will also be disabled after the first call of `overflow_fun`.

To be exact: `overflow_fun` is called after the current procedure, which is executed on this bucket, is finished. Then the bucket is free and bucket manipulation can be performed immediately.

Splitting

If a bucket contains too many data, the user may want to split it which is provided by a routine. Unregistered items will just be distributed equally between the new

buckets. But then data may have a bad locality. Therefore, memory items can be registered. At compile time the number of different memory items has to be specified. Then the size of such an item can be specified by calling

```
dp_buck_item_size(id_pointer, item, item_size)
```

`id_pointer` denotes the bucket where `item_size` is assigned to item `item`. Allocating a registered memory item in a bucket without registered size is also possible. Calling `dp_buck_item_size` afterwards or a second time for the same item in the same bucket results in ignoring this call.

Specifying the size of memory items also allows a more efficient memory management. But the main reason to specify such items, with or without size, is to decide about the distribution of these memory items to the split buckets. For that reason a split function has to be specified for each kind of items:

```
dp_buck_split_fun(id_pointer, item,
                  split_fun, arg_block, arg_block_size)
```

where `split_fun` has to be declared as

```
int split_fun(item *, arg_block)
```

It has to return the index of the new bucket where `item` has to be put in. The arguments are pointers to the memory item and to the argument block. `dp_buck_split_fun` is allowed to be called more than once, each time overwriting the previous value. To know more about the data in the bucket

```
dp_buck_for_each(id_pointer, item,
                 for_each_fun, arg_block, arg_block_size)
```

can be called. For each item of kind `item` in bucket `id_pointer` `for_each_fun` is called which has to be declared as:

```
for_each_fun(item *, arg_block)
```

A bucket can be split by calling:

```
dp_buck_split(id_pointer, new_id_pointers, new_buck_num)
```

where `id_pointer` is a global pointer to specify the bucket to be split (it is removed after splitting), `new_id_pointers` has to pointer to an array to contain `new_buck_num` global pointers to buckets after return. This function is only allowed to be called if the bucket is free. The new buckets will be free of garbage.

Buckets also can be merged by calling

```
dp_buck_merge(id_pointers, buck_num)
```

where basically the items of buckets `id_pointers[1 ...buck_num]` are put into bucket `id_pointers[0]` which remains existing while buckets `id_pointers[1 ...buck_num]` are removed. Merging buckets is only possible if the registration of items was the same at all involved buckets. The rest of stored values as, for example, the overflow function will remain as it was in at bucket `id_pointers[0]`.

Garbage Collection

The user may want to check about the fragmentation of the memory space of a bucket and then decide to perform garbage collection. A point of time to do this could be the execution of `overflow_fun` (see Section C.4.2).

```
dp_buck_get_frag(id_pointer)
dp_buck_garb_coll(id_pointer_new, id_pointer_old)
```

`dp_buck_get_frag` returns the percentage of unused memory in the bucket, specified by `id_pointer`, due to fragmentation of the space. `dp_buck_garb_coll` performs garbage collection and moves the data into a new bucket, `id_pointer_new`. The old one, `id_pointer_old`, does not longer exist anymore. This function is only allowed to be executed when the bucket is free.

Migration

The user may want a bucket to be transferred to another processor or *in core* or *out of core*.

```
dp_buck_migrate(proc_id, id_pointer_new, id_pointer_old)
```

transfers bucket `id_pointer_old` to bucket `id_pointer_new` (output parameter) on processor `proc_id`. If `proc_id` is specified as -1 the bucket migrates from *in core* or *out of core* to *out of core* or *in core*, respectively. Migrating to out of core the bucket does not get a new bucket id and `id_pointer_new` is unused. But there will be a new id if the bucket migrates from *out of core* to *in core*. If the bucket gets a new id the old bucket is removed.

`dp_buck_migrate` returns zero at failure. This might be the case if no new id pointer is available for the time being. Try later again. This operation is only allowed when the bucket is free.

Miscellaneous Managing Functions

For managing buckets there are the following commands available, additionally:

```
dp_buck_get_size(id_pointer)
dp_buck_exec_at_free(id_pointer, at_free_fun, arg)
```

The first one returns the size of the bucket in bytes, to be exact: the used space for this bucket. This also includes the freed but enclosed space (see Section C.4.2). The second command executes `at_free_fun` locally as soon as bucket `id_pointer` is free, this means, no procedure is executed on it. `at_free_fun` should be declared as following:

```
at_free_fun(id_pointer, arg)
```

Basic Low Level Calls

```
int dp_proc_num()
int dp_my_proc_id()
```

returns the number of processors in the system and its own processor id, respectively.

```
dp_proc_to_pointer(proc_id, proc_pointer)
```

converts `proc_id` to a global pointer (`proc_pointer`) which tells `dp_xxx_call` that the procedure has to be executed at the local space of processor `proc_id`.

Load Balancing

Load balancing is performed automatically if a delayed procedure is called and no bucket is specified where to be executed. Data location is also considered.

Load balancing can also be done by the user in two ways: First, buckets can be migrated and, second, a processor can be specified where to execute the delayed procedure (see Section C.4.2).

C.4.3 Debugging and Diagnostics

Parallel and especially asynchronous systems are poor on debuggers. Therefore, often the old fashion print statements are used to debug parallel code. The DPE model helps. If compiled with the debug option, each *delayed procedure* call is equipped with a unique identifier which is printed at the procedure call, begin and end of the execution. A debug string can be registered for each procedure, for example the procedure name, which is printed with each statement. The debug output can be used to build a graphical post mortem debugger.

If compiled with the diagnostic option the use of the DPE model is checked. For example range checks for the input is performed.

The DPE model copes with this issue as it is done in Section 6.2 and B.5.

Bibliography

[1] J. D. Boissonnat. Geometric structures for three dimensional shape representation. *ACM Trans. Graphics*, 3:266–286, 1984.

[2] J. D. Boissonnat, O. Devillers, R. Schott, M. Teillaud, and M. Yvinec. Applications fo random sampling to online algorithms in computational geometry. Technical Report 1285, INRIA, Sophia Antipolis, 1990.

[3] A. Bowyer. Computing Dirichlet tessellations. *The Computer Journal*, 24(2):162–166, 1981.

[4] P. Brezany and A. Choudhary. Techniques and optimizations for developing irregular out-of-core applications on distributed-memory systems. Technical Report 96-4, Institute for Software Technology and Parallel Systems, University of Vienna, 1996.

[5] J. C. Cavendish. Automatic triangulation of arbitrary planar domains for finite element method. *Int. J. Numer. Methods Eng.*, 8:679–696, 1974.

[6] Z. J. Cendes, D. N. Shenton, and H. Shahnasser. Magnetic field computations using Delaunay triangulations and complementary finite element methods. *IEEE Trans. Magnetics*, 21:898–903, 1985.

[7] C. Chang, G. Czajkowski, and T. von Eicken. Design and performance of Active Messages on the SP-2. Technical Report 96-1572, Cornell University, Computer Science Department, 1996.

[8] P. L. Chew. Guaranteed-quality mesh generation for curved surfaces. In *Proceedings of the 9th Annual Symposium on Computational Geometry (SCG '93)*, pages 274–280. ACM Press, San Diego, CA, 1993.

[9] P. L. Chew. Constrained Delaunay triangulations. *Algorithmica*, 4:97–108, 1989.

[10] P. L. Chew. Guaranteed-quality triangular meshes. Technical Report TR89-983, Cornell University, Computer Science Department, 1989.

[11] P. L. Chew, N. P. Chrisochoides, and F. Sukup. Parallel constrained delaunay triangulation. In *Special Symposium on Trends in Unstructured Mesh Generation*. Northwestern University Evanston, IL, 1997.

[12] N. P. Chrisochoides, I. Kodukula, and K. Pingali. Data movement and control substrate for parallel scientific computing. In *Proceedings of the*

Workshop on Communication and Architectural Support for Network-based Parallel Computing, San Antonio, Texas, 1997.

[13] N. P. Chrisochoides and F. Sukup. Task parallel implementation of the bowyer-watson algorithm. In *5th Conference on Numerical Grid Generation in Computational Fluid Dynamics and Related Fields*. Mississippi State University, Starkville, MS, 1996.

[14] P. Cignoni, D. Laforenza, C. Montani, and R. Perego. Evaluation of parallelization strategies for an incremental Delaunay Triangulator in E^3. *Concurrency: Practice and Experience*, 7(1):61–80, 1995.

[15] P. Cignoni, D. Laforenza, C. Montani, R. Perego, and R. Scopigno. Evaluation of parallelization strategies for an incremental Delaunay triangulator in E^3. *Concurrency: Practice and Experience*, 7(1):61–80, 1995.

[16] K. L. Clarkson. Randomized geometric algorithms. *Computers and Euclidean Geometry*, 1:117–162, 1992.

[17] K. L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. *Comp. Geom.: Theory and Applications*, pages 185–121, 1993.

[18] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete and Computational Geometry*, 4(1):387–421, 1989.

[19] Ports Consortium. PORTS: POrtable RunTime System. `http://www.cs.uoregon.edu/research/paracomp/ports`.

[20] D. Culler, K. Keeton, C. Krumbein, L. T. Liu, A. Mainwaring, R. Martin, S. Rodrigues, K. Wright, and C. Yoshikawa. *Generic Active Message Specification, Version 1.1*. University of California, Berkeley, 1994.

[21] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin Heidelberg New York, 1997.

[22] C. Ebert and R. Dumke. *Software-Metriken in der Praxis*. Springer-Verlag, Berlin Heidelberg New York, 1996.

[23] H. Edelsbrunner. *Algorithms in Combinatorical Geometry*. Springer-Verlag, Berlin Heidelberg New York, 1987.

[24] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. In *Proc. 8th ACM Symp. Comp. Geometry*, pages 43–52, 1992.

[25] J. E. Flaherty, P. J. Paslow, M. S. Shephard, and J. D. Vasilakis. Adaptive methods for partial differential equations. *SIAM Proceedings Series*, Philadelphia, 1989.

[26] S. Fortune. Sweepline algorithms for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[27] S. Fortune. Voronoi diagrams and Delaunay triangulations. In F. K. Hwang and D.-Z. Du, editors, *Computing in Euclidean Geometry*, pages 193–233. World Scientific, Singapore, 1992.

[28] S. Fortune. A note on Delaunay diagonal flips. *Pattern Recognition Letters*, 14:723–726, 1993.

[29] N. T. Frink, P. Parikh, and S. Pirzadeh. A fast upwind solver for the Euler equations on 3D unstructured meshes. *AIAA*, 29:515–522, 1991.

[30] T. Gilb. *Software Metrics*. Winthrop Publishers Inc. (USA edition), Cambridge, MA, 1988.

[31] P.J. Green and R. Sibson. Computing Dirichlet tessellations in the plane. *Computer Journal*, 21(22):168 – 173, 1977.

[32] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delauanay and Voronoi diagrams. In *ICALP 90*, pages 414 – 431. Springer-Verlag, Berlin Heidelberg New York, 1990.

[33] L. J. Guibas and J. Stolfi. Ruler, compass and computer: the design and analysis of geometric algorithms. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, pages 111 – 165. Springer-Verlag, Berlin Heidelberg New York, 1988.

[34] B. Hendrickson and R. Leland. The Chaco user's guide: Version 2.0. Technical Report SAND94–2692, Sandia National Laboratories, Albuquerque, 1994.

[35] Y.-S. Hwang, B. Moon, S. Sharma, R. Ponnusamy, R. Das, and J. Saltz. Runtime and language support for compiling adaptive irregular programs on distributed memory machines. *Software Practice & Experience*, 25(6):597–621, 1995.

[36] A. Jameson. Computational algorithms for aerodynamic analysis and design. *Applied Numerical Mathematics: Transactions of IMACS*, 13(5):383–422, 1993.

[37] G. Karypis and V. Kumar. Using METIS and ParMETIS. Department of Computer Science and Engineering, University of Minnesota `http://www-users.cs.umn.edu/ karypis/metis/metis.html`.

[38] T. Littlefair. *User's Guide for CCCC*. Edith Cowan University, Australia. `http://www.fste.ac.cowan.edu.au/~tlittlef/cccc_ug.htm`, `http://www.fste.ac.cowan.edu.au/~tlittlef`.

[39] R. Löhner, J. Camberos, and M. Merriam. Parallel unstructured grid generation. *Comp.Meth.Appl. Mech.Eng*, 95:343–357, 1992.

[40] T. Okusanya and J. Peraire. Parallel unstructured mesh generation. In *5th Conference on Numerical Grid Generation in Computational Fluid Dynamics and Related Fields*. Mississippi State University, Starkville, MS, 1996.

[41] T. Okusanya and J. Peraire. 3D parallel unstructured mesh generation. In *Joint ASME/ASCE/SES Summer Meeting*. Norris Center, Northwestern University, 1997.

[42] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, Cambridge, 1994.

[43] F. Pellegrini and J. Roman. SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. *Lecture Notes in Computer Science*, 1067:493–498, 1996.

[44] J. Peraire, J. Peiro, and K. Morgan. *The FELISA System*. Computational Aerospace Science Laboratory, MIT, Cambridge, 1994. `http://abweb.larc.nasa.gov:8080/ kbibb/felisa.html`.

[45] R. Preis and R. Diekmann. The PARTY partitioning-library, user's guide – version 1.0. Technical Report tr-rsfb-96-024, University of Paderborn, 1996.

[46] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, Berlin Heidelberg New York, 1985.

[47] J. Ruppert. A new and simple algorithm for quality 2-dimensional mesh generation. In Mihalis Ramachandran, Vijaya; Bentley, jon; cole, Richard; Cunningham, William H.; Guibas, Leo; King, Valerie; Lawler, Eugene; Lenstra, Arjen; Mulmuley, Ketan; Sleator, Daniel D.; Yannakakis, editor, *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '93)*, pages 83–92. SIAM Press, Austin, TX, 1993.

[48] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. Govindaraju, K. Gildea, P. DiNicola, and C. Bender. Performance and experience with LAPI: A new high-performance communication library for the IBM RS/6000 SP. In *Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP-98)*, pages 260–266. IEEE Computer Society, Los Alamitos, CA, 1998.

[49] M. I. Shamos and D. Hoey. Closest-point problems. In *Proc. 16th Annual Symp. Foundations of Computer Science*, pages 151–162. IEEE Press, 1975.

[50] R. Thakur. *Runtime Support for In-Core and Out-of-Core Data-Parallel Programs.* PhD thesis, Dept. of Electrical and Computer Eng., Syracuse University, Syracuse, 1995.

[51] G. E. Thaller. *Softwaremetriken einsetzen, bewerten, messen.* Verlag Heinz Heise, Hannover, 1994.

[52] C. Walshaw, M. Cross, S. Johnson, and M. Everett. JOSTLE: Partitioning of unstructured meshes for massively parallel machines. In N. Satofuka, J. Periaux, and A. Ecer, editors, *Parallel Computational Fluid Dynamics: New Algorithms and Applications*, pages 273–280. Elsevier, Amsterdam, 1995.

[53] D. Watson. Computing the n-dimensional Delaunay tessellation with applications to Voronoi polytopes. *The Computer Journal*, 24(2):167–172, 1981.

[54] R. Williams and E. Felten. Distributed processing of an irregular tetrahedral mesh. Technical Report C3P793, Caltech, 1989.