FRAMEWORK FOR MASSIVELY PARALLEL ADAPTIVE FINITE ELEMENT CFD ON TETRAHEDRAL MESHES

NICLAS JANSSON[†], JOHAN HOFFMAN[‡], AND JOHAN JANSSON[§]

Abstract. In this paper we describe a general adaptive finite element framework for unstructured tetrahedral meshes without hanging nodes suitable for large scale parallel computations. Our framework is designed to scale linearly to several thousands of processors, using fully distributed and efficient algorithms. The key components of our implementation, local mesh refinement and load balancing algorithms are described in detail. Finally, we present a theoretical and experimental performance study of our framework, used in a large scale Computational Fluid Dynamics (CFD) computation, and we compare scaling and complexity of different algorithms on different massively parallel architectures.

Key words. adaptive methods, load balancing, unstructured local mesh refinement

1. Introduction. Adaptive mesh refinement methods are effective techniques used for reducing computational cost of a finite element based solver. By using error indicators, an adaptive solver can add more elements in regions of interest, for instance where the local error is large. Hence, locally enhance the resolution of a solution, with minimal extra computational cost. However, for most computationally demanding large scale problems, these reductions are often not enough, and they are still too computationally expensive for ordinary computers.

By utilizing parallel computing one could gain access to the large amounts of memory and processors demanded by for example complex flow problems. Despite all the work in the field of adaptive finite element methods most of this work could not easily be applied in the parallel setting, mostly due to the distributed memory model required by most larger message passing parallel computers.

As mentioned, the main obstacle towards an adaptive parallel finite element solver is the distributed memory model, which adds additional constraints on both solver and mesh refinement methods. This paper addresses the problem of adaptive refinement of unstructured tetrahedral meshes without hanging nodes. Most state-of-the-art finite element packages with support for parallel processing, bypass this problem by distributing the entire mesh to all processors, assigning parts of the mesh to each processor [18, 2]. This lowers computational time, but not memory requirement.

In this paper, we present a fully distributed adaptive solver framework were all components, assembly, error estimation, refinement and load balancing scalable to thousands of processors in parallel with fully distributed data. We here describe an open source implementation DOLFIN [20]. To our knowledge this is one of the few unstructured finite element solvers which has this, regarding open source solvers we believe it is the first.

The remainder of this paper is organized as follows. First we give an overview of our solver (section 2), the basic components and the parallelization strategy. In

[1] Previously appeared as *Parallel Adaptive FEM CFD*, Tech. Rep. KTH-CTL-4008, Computational Technology Laboratory, 2010, `http://www.publ.kth.se/trita/ctl-4/008/`.

[†] Computational Technology Laboratory, Computer Science and Communication, KTH, SE-10044 Stockholm, Sweden (`njansson@csc.kth.se`.)

[‡] Computational Technology Laboratory, Computer Science and Communication, KTH, SE-10044 Stockholm, Sweden (`jhoffman@csc.kth.se`.)

[§] Computational Technology Laboratory, Computer Science and Communication, KTH, SE-10044 Stockholm, Sweden (`jjan@csc.kth.se`.)

section 3 we describe error estimation, in section 4 we give some background on local mesh refinement and the challenge with parallel implementations. We present two methods which we use in our solver and compare them to related work. In section 5 we describe the solver's load balancing framework. Finally, in section 6 we present a theoretical and experimental performance analysis of our solver.

2. Solver. The adaptive flow solver is based on a General Galerkin (G2) finite element method [15]. It consists of four major components, the main Navier-Stokes equations solver, the dual Navier-Stokes equations (on the same mesh) solver, error estimation and finally mesh refinement. This loop is repeated until some convergence criterion is satisfied.

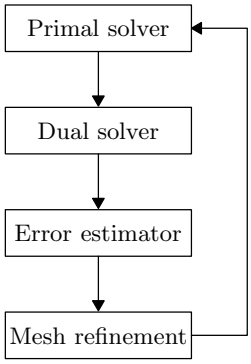


FIG. 2.1. *Overview of our adaptive framework.*

2.1. Error estimation. Adaptivity is here based on a posteriori error estimation, where we always refine a subset of the largest indicators. Since we work with a fully distributed solver we have to choose these indicators from a global perspective. Due to the large problem sizes for which we are aiming, gathering all error indicators on each processor is not an option. Instead we present two different methods, one parallel merge routine and one interval halving method.

2.2. Mesh adaption. The solver's mesh adaption routines can be based on different local mesh refinement methods. These methods enhance the mesh resolution locally from the given error indicators. In the parallel setting there is an additional problem not present in the serial case. As the mesh is refined, new vertices are added arbitrarily at any processor. Hence the work distribution changes over time, rendering an initially good load balance useless.

Therefore in order to sustain a good parallel efficiency the mesh must be repartitioned and redistributed after each refinement. In other words dynamic load balancing is needed. In the worst case, the load balancing routine must be invoked every time a mesh is adapted, it has to be rather efficient, and for our aim, scale well for a large number of processors.

2.3. Implementation. This work is implemented as a parallel version of the incompressible flow solver in Unicorn [13, 14], which is based on the finite element library DOLFIN [19, 20]. At the time of this work, DOLFIN was not fully parallelized. We created a parallel branch based on initial work described in [17].

This version is a fully parallel variant of DOLFIN, where everything in the preprocessing, assembly and post-processing chain is parallelized. For efficiency the par-

allelization utilizes a fully distributed mesh, where each processor only stores a small portion of the entire mesh, which reduces memory footprint and allows efficient usage of thousands of processors, each with a small amount of local memory. The overlap is stored as ghosted entities on each processor, and a unique global numbering glues the smaller meshes together into a consistent global mesh. Parallel matrix/vector representation and linear solvers are handled through PETSc [1]. This experimental version has also proven to scale well for a wide range of processors.

3. Error estimation. We here present two different strategies for selecting error indicators for refinement. The first method selects a percentage of the largest indicators, while the other method selects indicators such that the sum of the selected indicators reduce the estimated error by a given percentage.

Parallelization of the first method is more or less straightforward. Clearly it is trivial to gather all indicators onto all processors, select the local indicators which corresponds to the given percentage of the global indicators. However, as mentioned before, for larger problem sizes this is not an option due to memory constraints.

To reduce the memory footprint we use a parallel merge routine. Each processor stores the defined largest percentage from the local indicators, then communicates these with all other processors, and for each received set of indicators, merges these indicators with the local ones. In the end all processors will have a list of indicators that are valid from a global perspective. Let L be a sorted list of local error indicators e on the processor, N_c be the number of cells on the processors, p the given percentage and P the number of processors, then the algorithm can be expressed as in Algorithm 1.

Algorithm 1: Method 1

$S = \{e_i \in L \,|\, i \geq (1-p)N_c\}$
for $i = 1$ *to* $P - 1$ **do**
  $src \leftarrow (rank - i + P) \mod P$
  $dest \leftarrow (rank + i) \mod P$
  Send S to $dest$
  Receive cells from src
  $S_{\text{global}} \leftarrow \text{merge}(\text{received}, S_{\text{global}})$
end

For the second method, the problem lies in selecting a subset of indicators whose sum is a given percentage of the total sum. Since the summation has to be done globally, the parallelization involves a bit more work.

One solution to this problem is interval halving, first compute a local sum of a subset of the largest local indicators. Then compute the global sum and compare it against the target value. The interval halving enters as a threshold value, for which each processor computes the local sum of all indicators which are larger. If the global summation does not reach the given target value, the threshold value is changed, and the process is repeated until convergence. Let P, p be as before, e be all error indicators (globally), e_l, e_g local and global sums of indicators respectively, t be the global threshold value and c a global cutoff value, then the algorithm can be expressed as in Algorithm 2.

4. Local mesh refinement. Local mesh refinement has been studied by several authors over the past years. The general idea is to split an element into a set of

Algorithm 2: Method 2

$t \leftarrow p \sum e$
$max_e = \max(e)$
$min_e = \min(e)$
while $|e_g - t| > \epsilon$ **do**
  $c \leftarrow (max_e + min_e))/2$
  $e_l \leftarrow \sum_{e_i \geq c} e_i$
  $e_g \leftarrow \text{Allreduce}(e_l)$
  if $e_g > t$ **then**
    $min_e = c$
  else
    $max_e = c$
  end
end

new ones in order to improve the solution in that region. For most finite element formulations, mesh refinement has a constraint that the produced mesh must be valid. A mesh is considered valid if there are no "hanging nodes", that is no node should be on another element's facet. How elements should be split in order to ensure this differs between different methods. Often one uses some kind of edge bisection scheme.

A common edge bisection algorithm bisects all edges in the element, introducing a new vertex on each edge, and connecting them together to form the new elements (see for example [4]). Other methods focus only on bisecting one of the edges, which edge depends on the method. For example one could select the edge opposite to its newest vertex, this method is often referred to as the newest vertex approach, described in [3]. Another popular edge bisection method is the longest edge [22], where one always selects the longest edge for refinement. In order to ensure that the mesh is free of "hanging nodes", the algorithm recursively bisects elements until there are no "hanging nodes" left.

4.1. The challenge of parallel mesh refinement. Performing the refinement in parallel adds additional constraints on the refinement method. Not only should the method prevent "hanging nodes", it must also be guaranteed to terminate in a finite number of steps.

In the parallel setting, each processor has a small part of the distributed mesh in the local memory. If a new vertex is introduced on the shared boundary between processors, the algorithm must ensure that the information propagates onto all the neighboring processors.

For an algorithm that bisects all edges in an element, the problem reduces after a propagation step to a global decision problem, deciding which of the processor's information should be used on all the other processors. For an algorithm that propagates the refinement (several times) like the longest edge, the problem becomes a set of synchronization problems: i) to detect and handle refinement propagation between different processors and ii) to detect global termination.

The first synchronization problem could be solved by dividing the refinement into two different phases, one local refinement phase and one global propagation phase. In the first phase, elements on the processor are refined with an ordinary serial local refinement method. This could create non conforming elements on the boundary

between processors. These are fixed by propagating the refinement to the neighboring processors in the second propagation phase. The next local refinement phase could create a non conforming element, and another propagation phase is needed with the possibility of another and so forth. However, if the longest edge algorithm is used, termination is guaranteed [6]. But the problem is to detect when all these local meshes are conforming, and also when they are conforming at the same time. That means, one has to detect global termination, which is a rather difficult problem to solve efficiently, especially for massively parallel systems for which we are aiming.

There has been some other work related to parallel refinement with edge bisection. For example a parallel newest vertex algorithm was presented by Zhang [5]. Since the algorithm does not need to solve the termination detection problem, scaling is simply a question of interconnect latency. Another work is the parallel longest edge algorithm done by Castaños and Savage [6]. They solve the termination detection problem with Dijkstra's general distributed termination algorithm, which simply detects termination by counting messages sent and received from some controlling processor. However, in both of these methods only a fairly small number of processors is used, less than one hundred, so it is difficult to estimate how efficient and scalable these algorithms are. For more processors, communication cost and concurrency of communication patterns start to become important factors. Our aim is to design an algorithm that scales well for thousands of processors.

4.2. A modified longest edge bisection algorithm. Instead of trying to solve the termination detection problem, one could try to modify the refinement algorithm in such a way that it would only require one synchronization step, thus less communication. With less communication overhead it should also scale better for a large number of processors.

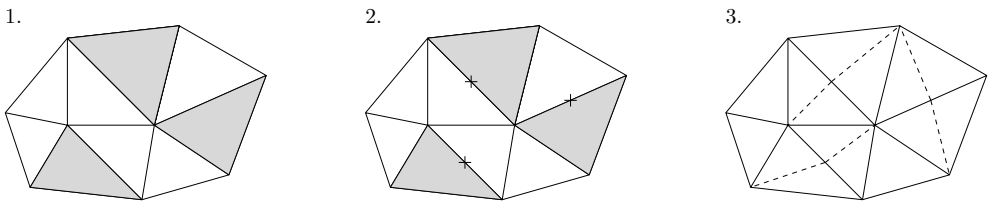


FIG. 4.1. *An example of the refinement algorithm used. First a set of elements are marked for refinement (1). The longest edges are found (2), and all elements connected to these edges are finally bisected, the dashed lines in (3).*

One simplification that can be made to the longest edge algorithm is that instead of recursively fixing "hanging nodes", elements could instead be bisected in pairs (or groups) (see Figure 4.1). With this modification, the algorithm would always terminate the refinement by bisecting all elements connected to the refined edge, it will never leave any non conforming elements, hence if the longest edge is shared by different processors, the algorithm must only propagate the refinement onto all elements (processor) connected to that edge, but then no further propagation is possible (see Figure 4.2). This makes the modified algorithm a perfect candidate for an efficient parallel algorithm, we here refer to this algorithm as *simple bisection*.

However, notifying an adjacent processor of propagation does not solve the problem entirely. As mentioned in section 2, all mesh entities shared by several processors must have the same global number in order to correctly represent the distributed mesh.

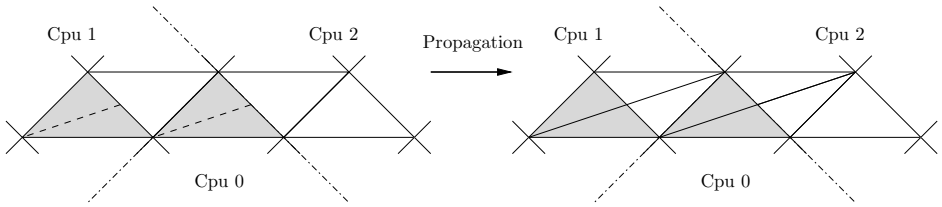


FIG. 4.2. *An example of the two simple cases of a propagation. Shading refers to elements marked for refinement, dashed lines show how a processor want to bisect an element and dash dotted lines refers to the mesh partitioning.*

The refinement process must therefore guarantee that all newly created vertices are assigned the same unique number on all the neighboring processors. Another problematic case arises when processors refine the same edge and the propagation "collides" (see Figure 4.2). In this case the propagation is done implicitly but the processors must decide which of the new numbers to use.

A more complicated case is when an element receives multiple propagations (possibly from different processors) on different edges (see Figure 4.3). Since the modified longest edge algorithm only allows one edge to be bisected per element, one of the propagations must be selected and the other one rejected. This however adds a difficulty to the simple algorithm. First of all, how should the processors decide upon which edge to refine? Clearly this can not be done arbitrarily since when a propagation is forbidden, all refinement done around that edge must be removed. Thus, in the worst case it could destroy the entire refinement.

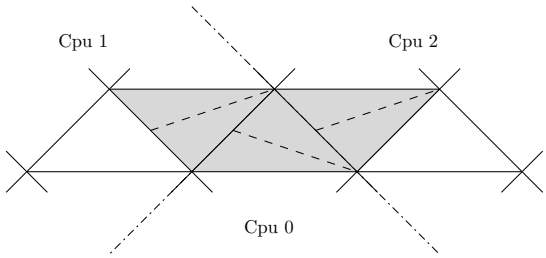


FIG. 4.3. *An example of the problematic case with multiple propagations. Shading refers to elements marked for refinement, dashed lines show how a processor want to bisect an element and dash dotted lines refers to the mesh partitioning.*

To solve the edge selection problem perfectly one needs an algorithm with a global view of the mesh. In two dimensions with a triangular mesh, the problem could be solved rather simply since each propagation could only come from two different edges (one edge is always facing the interior). By exchanging the desired propagation edges, processors could match theirs selected edges with the propagated ones, in an attempt to minimize the number of forbidden propagations. However, in three dimensions the problem starts to be so complicated that multiple exchange steps are needed in order to solve the problem. Hence, it becomes too expensive to solve.

Instead we propose an algorithm which solves the problem using an edge voting scheme. Each processor refines the boundary elements, finds their longest edge and cast a vote for it. These votes are then exchanged between processors, which add the received votes to their own set of votes. Now the refinement process restarts,

but instead of using the longest edge criteria, edges are selected depending on the maximum numbers of votes. In the case of a tie, the edge is selected depending on a random number assigned to all votes.

Once a set of edges has been selected from the voting phase the actual propagation starts by exchanging these with the other processors. However, the voting could fail to "pair" refinements together. For example, an element could lie between two processors which otherwise does not share any common face. Each of these processors wants to propagate into the neighboring element but on different edges (see Figure 4.4). Since the processors on the left and right side of the element do not receive the edge vote from each other, the exchange of votes will in this case not help with selecting an edge that would work for both processors.

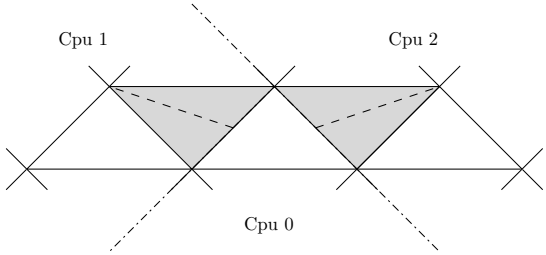


FIG. 4.4. *An example of the case when edge votes could be missed. Shading refers to elements marked for refinement, dashed lines show how a processor want to bisect an element and dash dotted lines refers to the mesh partitioning.*

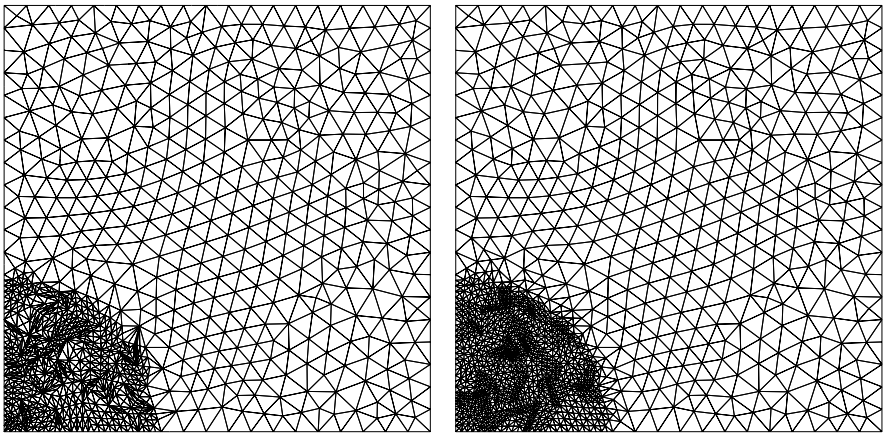
To fix this, an additional exchange step is needed and maybe another and so forth, rendering the perfect fix impossible. Instead, the propagation step ends by exchanging the refined edges which gave rise to a forbidden propagation. All processors could then remove all refinements that these edges introduced, and in the process, remove any hanging nodes on the boundary between processors.

Let $\mathcal{B}$ be the shared boundary between processors in our mesh $\mathcal{T}$. For a set $\mathcal{R}$ of elements c marked for refinement, the algorithm can be expressed as in Algorithm 7 (see Appendix).

4.3. Parallel recursive longest edge bisection. A major drawback of the simple edge bisection algorithm is the poor mesh quality (see Figure 4.5), most notably after several refinements due to the illegal propagation issues. Since the core of an adaptive CFD solver is the repeated use of mesh refinement to improve the solution, poor mesh quality is a severe problem. One solution is to use local edge/face swaps [7]; but this introduces new problems in the parallel setting. In order to cope with this quality problem, our framework also contains a parallel implementation of the longest edge algorithm, which is known to produce good quality refinements.

We follow the work by Castaños and Savage [6] and use a pure recursive longest edge algorithm, decomposed into a serial refinement phase and a parallel propagation phase. As discussed before, the main problem with this approach is to efficiently solve the termination detection problem. Since we aim for algorithms that scale well for thousands of processors, we design a new termination detection method without centralized control.

Our solution is to include termination detection in the propagation step. This can be realized if one uses a collective *all-to-all* exchange algorithm, since each processor would then receive data from all the other processors, hence if it doesn't receive



(a) Simple edge bisection (b) Recursive longest edge

FIG. 4.5. *A comparison between mesh quality for the two different refinement methods.*

any propagations, the refinement has terminated. However, *all-to-all* communication could have a high communication cost $\mathcal{O}(P-1)$, which scales acceptably well if it is done once, but it is not suitable for a recursive algorithm were each recursion ends with an exchange step.

To realize the distributed termination detection on a massively parallel computer the communication pattern must be highly concurrent and memory efficient. The *all-to-all* approach with $\mathcal{O}(P-1)$ communication steps is very memory conservative and concurrent enough, but for a large number of processors even the linear cost is too expensive, and becomes the bottleneck for an efficient implementation.

However, since mesh refinement is a local problem, many processors will not bisect any elements. Thus, they will not send any propagation information. Therefore, we could route propagation information through all processors without consuming too much memory. This could be realized with $\mathcal{O}(\log P)$ cost recursive doubling and hypercube exchange type communication patterns, as in Algorithm 3.

Algorithm 3: Hypercube exchange

```
for i = 0 to log P − 1 do
    dest ← rank ⊕ 2^i
    exchange propagation with dest
    propagation ← merge(received, propagation)
end
```

Another problem to solve for the parallel longest edge refinement is the consistency of mesh entities global numbers. For the simple edge bisection method, this was a minor problem, since refinement only consisted of one bisection step, the only problematic case were if two processors bisected elements around the same edge.

For the parallel recursive algorithm this problem becomes more problematic to solve. Now each processor could have created a sequence of new vertices on the shared edge. Each of these vertices needs an unique global number, which has to be transmitted to the adjacent processors. This could of course be solved by expanding

the recursion with a renumbering step, however this would not scale well.

We solve the consistency problem by always generating unique global numbers (same on all processors) for each bisection. One way to generate these numbers would be to use the already uniquely numbered partitioned, unrefined mesh. For an already partitioned and distributed mesh, each edge e consists of two vertices (v_1, v_2), each with a unique global number. Bisection only occurs around one edge, we could then use the edge's vertex numbers to generate a new unique number v_3 as,

$$\begin{aligned} v_3 &= (v_1 \cdot C + v_2) + \max_v \quad \text{if } v_1 < v_2 \\ v_3 &= (v_2 \cdot C + v_1) + \max_v \quad \text{if } v_2 < v_1 \end{aligned}$$

where $\max_v$ is the largest assigned number in the unrefined mesh and C is a large constant. If the new numbers are unique, the next set of generated numbers (v_1', v_2') will also be unique and so forth. Hence we do not need to explicitly take care of the consistency problem during the recursive refinement.

Let $\mathcal{R} \in \mathcal{T}$ be a set of elements marked for refinement. The parallel variant of the recursive longest edge bisection can be expressed as in Algorithm 4

Algorithm 4: Parallel rivara recursive bisection

```
while R is not empty do
    for each c ∈ R do
        Bisect (c)
    end
    propagate refinement using Algorithm 3
    add received refinements to R
end
```

5. Load balancing. There are mainly two different load balancing methods used today, diffusive and remapping methods. Diffusive methods, like the physical meaning, by finding a diffusion solution of a heavy loaded processor's vertices would move vertices to another processor and in that way smear out the imbalance, described for example in [16, 24]. Remapping methods relies on the partitioner's efficiency of producing good partitions from an already partitioned dataset. In order to avoid costly data movement, a remapping method tries to assign the new partitions to processors in an optimal way. For problems where the imbalance occurs rather localized, the remapping methods seems to perform better [25]. Hence, it fits perfectly to the localized imbalance from local mesh refinement in an adaptive solver.

In this work, we used the load balancing framework of PLUM [21] a remapping load balancer, modified to suite our finite element setting. The mesh is repartitioned according to an imbalance model. Repartitioning is done before refinement, since this would in theory minimize data movement and speedup refinement, since a more balanced number of elements would be bisected on each processor.

5.1. Workload modelling. We model the workload by a weighted dual graph of the finite element mesh. Let $G = (V, E)$ be the dual graph of the mesh, q be one of the partitions and let w_i be the computational work (weights) assigned to each graph vertex. The processor workload is then defined as

$$W(q) = \sum_{\forall w_i \in q} w_i \tag{5.1}$$

where communication costs are neglected. Let W_{avg} be the average workload and $W_{\max}$ be the maximum, then the graph is considered imbalanced if

$$W_{\max}/W_{\text{avg}} > \kappa \tag{5.2}$$

where the threshold value κ is based on problem or machine characteristics.

One could argue that our workload model is too coarse grained since we neglect the communication cost in the graph. However, since mesh refinement only occurs once per adaptive iteration. We believe that the absence of transient refinement reduces the importance of this metric.

5.1.1. Simple edge bisection. This workload model suits the modified longest edge algorithm (section 4.2) perfectly. Since the simplifications reduces the algorithm to only have one propagation and/or synchronization step. If we neglect off processor propagation, a priori workload estimation becomes a local problem. Let each element represent one unit of work, a dry run of the refinement algorithm would produce a dual graph with vertex weights equal to one or two. Each element is only bisected once, giving a computational weight of two elements for each refined element, see figure 5.1.

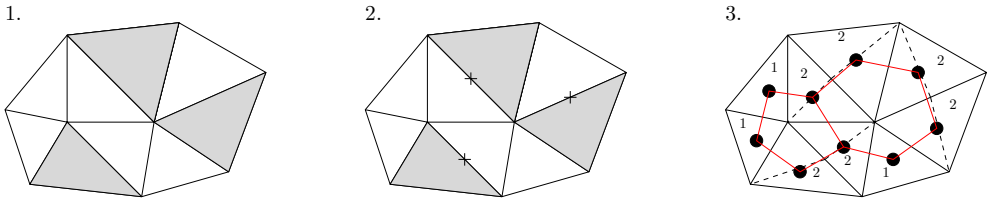


FIG. 5.1. *An example of the workload weights, added to the dual graph prior to refinement with the simple edge bisection algorithm.*

5.1.2. Recursive longest edge. Estimating workload for the recursive longest edge algorithm (section 4.3) involves a bit more work. Neglecting off processor propagations for the modified algorithm does not change the refinement footprint by much (since elements are only bisected once), for the recursive longest edge a refinement could start in one processor and propagate into all the others, hence neglecting propagation could reduce the a priori workload estimation. However, we believe that the cost of modelling this outweights the possible gain of adding them to the model.

In order to estimate the refinement footprint without refining the mesh we use the concept of longest edge propagation paths, LEPP [23]. A LEPP is the set of elements we obtain if we follow the longest edge of an element until the next longest edge is the same as the one we came from. In other words, the LEPP estimates the refinement footprint produced by one element. If we follow the LEPP for each element marked for refinement, increasing the dual graph weight for each element, we would in theory get a good estimate of the workload after refinement since the combination of all LEPPs produces almost the same refined mesh as the recursive algorithm, in two dimensions they are identical [23].

In two dimensions LEPP workload estimation works well, a propagation path does only increase with one element per longest edge. For higher dimensions, LEPP paths can be propagations trees rooted in the element marked for refinement. When a LEPP propagates through a longest edge the tree has to expand into all cells connected to

that edge, thus after a couple of propagation the tree could have grown to such an extent that workload becomes too costly to estimate. To bypass this problem we terminate a path after a specified depth.

Algorithm 5: LEPP workload estimation

```
for each c ∈ T do
    if c marked for refinement then
        e ← longest edge of c
        for each c connected to e do
            Propagate LEPP
        end
    end
end
```

5.2. Remapping strategies. Remapping of the new partitions can be done in numerous ways, depending on what cost metric one tries to minimize. Usually one often talks about the two metrics TotalV and MaxV. MaxV tries to minimize the redistribution time by lowering the flow of data, while TotalV lower the redistribution time by trying to keep the largest amount of data local, for a more detailed description see [21]. We have chosen to focus on the TotalV metric, foremost since it is much cheaper to solve than MaxV, also it produces equally good (or even better) balanced partitions.

Independent of which metric one tries to solve, the result from the repartitioning is placed in a similarity matrix S, where each entry $S_{i,j}$ is the number of vertices on processor i which would be placed in the new partition j. In our case, we want to keep the largest amount of data local, hence to keep the maximum row entry in S local. This could be solved by transforming the matrix S into a bipartite graph where each edge $e_{i,j}$ is weighted with $S_{i,j}$, the problem then reduces to the maximally weighted bipartite graph problem, which can be solved in an optimal way in $\mathcal{O}(V^2 \log V + VE)$ steps [21]. Since the vertices V in the graph are the processors, solving the graph problem quickly becomes a major bottleneck. Since the solution does not need to be optimal, a heuristic algorithm[21] with a runtime of $\mathcal{O}(E)$ is used.

The heuristic algorithm starts by generating a sorted list of the similarity matrix S. It then steps through the list and selects the largest value which belongs to an unassigned partition. Sorting was in the original PLUM paper performed with a serial binary radix sort, gathering the similarity matrix onto one processor. Since the matrix is of size $P \times P$ where P is the number of processors, sorting quickly becomes the bottleneck on a massively parallel machine.

Instead of sorting the matrix in serial, we improved the heuristic algorithm, implemented an efficient parallel binary radix sort, performing β passes and using 2^r "buckets" for counting. In order to save some memory the sorting was performed per byte of the integer instead of the binary representation. Since each integer is represented by 4 bytes (true even for most 64-bits architectures) the number of passes required was $\beta = 4$. For unsorted data of length N, divided into N/P parts distributed across all processors, the algorithm could be expressed as,

6. Performance analysis.

Algorithm 6: Parallel radix sort

```
for i = 0 to β do
    for j = 0 to N do
        count[i'th byte of data(j)] ← count[i'th byte of data(j)] + 1
    end
    count ← global reduction(count)
    for j = 0 to 2^r do
        index(j) ← ParallelPrefix(count(j))
    end
    Redistribute elements according to index
end
```

6.1. Theoretical. To analyze the experimental result we used a performance model which decompose the total runtime T into one serial computational cost T_{comp} and a communication cost T_{comm}, hence $T = T_{\text{comp}} + T_{\text{comm}}$. In our model, focus lies on refinement and load balancing algorithms and not assembly and linear algebra back-ends, since these operations are handled by external libraries.

6.1.1. Simple edge bisection. The simple edge bisection algorithm has a local computational costs consisting of iterating over and bisecting all elements marked for refinement. For a mesh with N_c elements, this becomes $\mathcal{O}(N_c/P)$. Communication only occurs when boundary elements needs to be exchanged. Our implementation uses an *all-to-all* communication pattern, hence each processor has to communicate with $P-1$ other processors. If we assume that there are N_s shared edges and each edge is on average connected to c elements, the total runtime with communication becomes:

$$T = \mathcal{O}\left(\frac{N_{\text{c}}}{P}\, c\, \tau_{\text{f}}\right) + (P-1)\mathcal{O}\Big(\tau_{\text{s}} + N_{\text{s}}\tau_{\text{b}}\Big) \tag{6.1}$$

Where $\tau_{\text{f}}, \tau_{\text{b}}$ and τ_{s} are the time required to perform a flop, transmit data and the latency of the interconnect respectively. Based on this performance model, more processors would lower the computational time, but in the same time increase the communication cost.

6.1.2. Recursive longest edge bisection. For the longest edge implementation we have a similar local computation cost, and a similar communication cost. The main difference is that refinement occurs γ times, thus we need to perform α propagation steps, transmitting N_{g} propagations each time and the hypercube exchange communication needs only $\log P$ steps, hence.

$$T = \mathcal{O}\left(\frac{N_{\text{c}}}{P}\, \gamma\, d\, \tau_{\text{f}}\right) + \alpha(\log P)\mathcal{O}\left(\tau_{\text{s}} + N_{\text{g}}\tau_{\text{b}}\right) \tag{6.2}$$

where d is the average number of non conforming elements per recursion.

As before, more processors would lower the computational time, but increase communication cost. But for this algorithm we have some extra parameters which make it difficult to state a precise model. But the main difference between (6.1) and (6.2) is the complexity of the communication $P-1$ versus $\log P$.

TABLE 6.1
Theoretical speedup with respect to 32 nodes.

P	Simple edge bisection	Recursive longest edge	Load balancer	Ideal speedup
32	1.00	1.00	1.00	1
64	1.96	1.99	0.99	2
128	3.64	3.97	0.98	4
256	5.64	7.82	0.95	8
512	5.89	14.92	0.88	16
1024	4.08	26.91	0.77	32

6.1.3. Load balancing. The most computationally expensive part of the load balancer is the remapping of new partitions. As discussed earlier we used an heuristic with a runtime of $\mathcal{O}(E)$, the number of edges in the bipartite graph, in the worst case, $E \approx P^2$. The sorting phase is linear, and due to the parallel implementation it runs in $\mathcal{O}(P)$, since the P^2 elements to be sorted are distributed across all P processors.

Communication time for the parallel prefix calculation is given by, for m data it sends and calculates in m/P steps. Since the prefix consists of 2^r elements, it would take $2^r/P$ step, and it is performed for each β sorting phase. In the worst case the reordering (during sorting) needs to send away all the elements, thus P communication steps, which gives the total runtime.

$$T = \mathcal{O}(P^2\tau_{\mathrm{f}}) + \mathcal{O}\left(\beta\tau_s + \beta\left(\frac{2^r}{P} + P\right)\tau_b\right) \tag{6.3}$$

where the $\mathcal{O}(P^2)$ is due to the $\mathcal{O}(E)$ heuristic, since this is the worst case a more realistic average case should be $\mathcal{O}(P)$. If not, these should be fairly easy to observe in the experimental analysis if a large number of processors (128-256) are used.

Redistributing the elements is done with a communication pattern that groups processors together as in an *all-to-all* operation, but with a data exchange performed on a *point-to-point* basis (see Appendix, Algorithm 8). In the worst case each processors has to redistribute elements to $P-1$ other processors, hence a run time of.

$$T = \mathcal{O}\left(\frac{N_c}{P}\tau_{\mathrm{f}}\right) + (P-1)\mathcal{O}\left(\tau_s + \frac{N_c}{P}\tau_b\right) \tag{6.4}$$

To conclude the analysis we present the theoretical speedup obtained from these models. Machine specific parameters τ_f, τ_s and τ_b where choosen to match a Blue Gene/L [12], the others were determined empirically. The result, presented in Table 6.1 is less then encouraging, but it should be noted that these results are worst case scenarios, for the average case we except far better scalability.

6.2. Experimental. The adaptive solver described in this paper has successfully been tested on two completely different architectures. First the 1024 node Blue Gene/L, *Hebb* at PDC/KTH. Secondly a regular 805 dual quad core node cluster, *Neolith* at NSC/LiU.

Our experimental analysis of the solver's performance has been divided into two different parts. First we have tested smaller components with synthetic data, matching different worst cases for the component. Secondly we have measured the performance of the entire solver, solving flow past a circular cylinder on large unstructured meshes.

6.2.1. Mesh refinement. To measure performance of our mesh adaption routines we defined a set of cells to refine, and measured the total runtime for refinement and load balancing. For this analysis we marked all cells inside a small region for refinement, foremost since this model the behavior of error indicators from a real solver and secondly, if we define a large set of cells for refinement communication cost would be less important since the local computation time will be dominant. A downside of this method is that most of the processors would be idling during refinement, hence we would gain less from additionally processors.

We used *Hebb* for our performance measurements, using a mesh with $25 \cdot 10^6$ cells. The small mesh size was dictated by the small local memory capacity for each node, and in order to have a mesh that would have a reasonable amount of local elements for larger number of processors, we chose 32 nodes as our baseline.

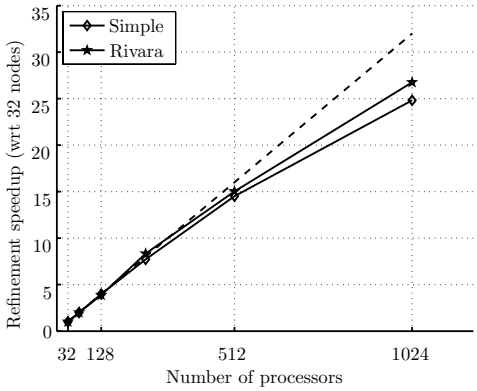


FIG. 6.1. *Strong scaling result for refinement, including both mesh refinement and load balancing. The dashed line refers to ideal speedup.*

As seen in Figure 6.1, the mesh refinement part of the solver scales well up until the point were a majority of the processors don't engage in any refinement activity. The interesting part here is that both methods scale equally well, even if the recursive rivara refinement routes propagation information through all nodes, and performs several communication steps.

Our a priori workload estimation were measured by counting both maximum and minimum number of cells per node after the refinement. As seen in Figure 6.2, workload modelling together with our scratch and remap load balancer does a good job distributing the workload. As mentioned in section 5.1, workload modelling for the simple bisection is more or less exact while for the recursive rivara it is a rough estimation. Therefore it is interesting that the experimental results indicates that our estimation does a fairly good job (Figure 6.2(b)), and in some case even better than the more exact workload modelling (compare with Figure 6.2(a)).

Furthermore, the results clearly shows that the communication cost in (6.1)-(6.4) are less than the computational cost, hence the good scalability compared to the theoretical results in Table 6.1. Also the assumption $\mathcal{O}(P^2) \approx \mathcal{O}(P)$ in (6.3) seems to be reasonable. Otherwise the speedup would have been much less, or even negative as observed in the theoretical results.

6.2.2. Solver. The entire CFD solver was tested on both *Hebb* and *Neolith*, solving a flow problem with the largest possible unstructured mesh that we could fit

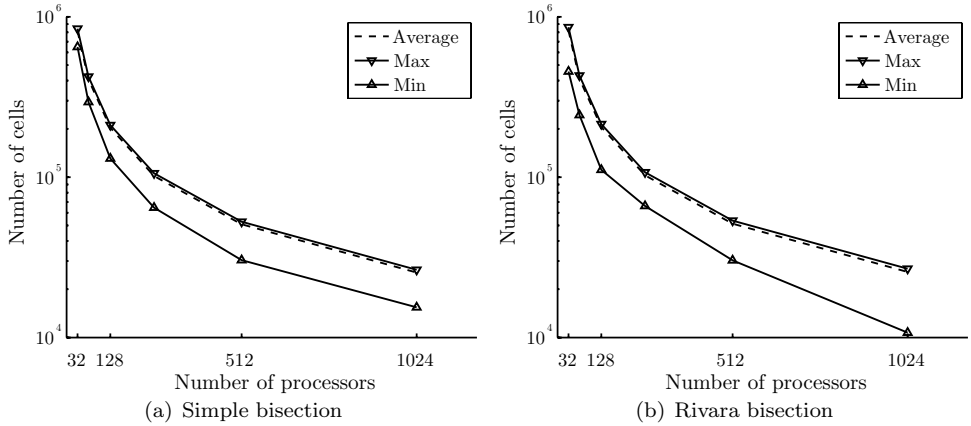


FIG. 6.2. *Number of cells per processor after load balancing.*

into 32 nodes. For *Hebb* the entire solver could solve a problem with $6 \cdot 10^6$ cells, while the bigger memory capacity of *Neolith* allowed us to use a mesh with $50 \cdot 10^6$ cells. We measured the time to assemble the momentum matrix and how long it takes to compute a full time step. Here we neglect the refinement since most of the time in our solver is spent on solving the primal and dual problems. Often the refinement cost around $1-10$ flow solution time steps, which is negligible compare to the primal/dual solvers thousands of time steps.

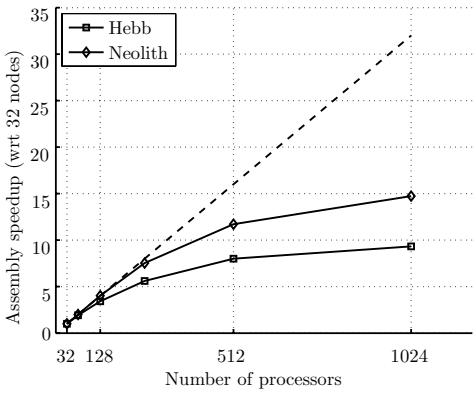


FIG. 6.3. *Strong scaling result for matrix assembly, three dimensional momentum equation. The dashed line refers to ideal speedup.*

As seen in Figure 6.3, assembly performance drops fairly quickly. When the number of processors becomes large, the communication cost seems to dominate, and the initial almost linear scaling suffer. It should be noted that $85-90\%$ of all matrix entries are local in the computation, and care has been taken to reorder the MPI communicator in such way that it maps to the different network topologies provided by the different architectures.

In the solver, each time step is computed by a fixed point iteration which assembles and solves the coupled continuity and momentum equation, for a more detailed description see [15]. Initially, each time step involves several fixed point iterations,

but after a short startup phase, each time step converges in only one iteration.

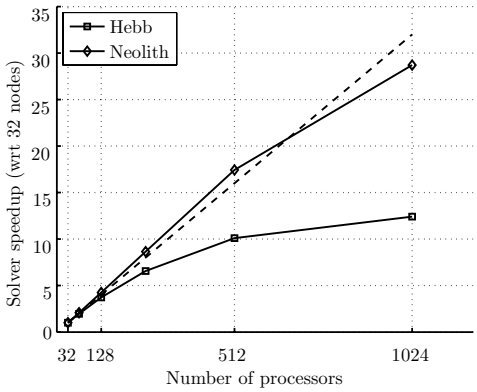


FIG. 6.4. *Strong scaling result for entire solver, measuring the entire fixed point iteration in the flow solver. The dashed line refers to ideal speedup.*

As seen in Figure 6.4, the solver performs fairly good for larger problem sizes on *Neolith*, while the smaller problem shows decent scaling on *Hebb*. Furthermore, the decent scaling shows that matrix assembly is not the main bottleneck for larger problems. In our case it seems to be the linear solvers, more specifically solving the continuity system.

An interesting observation that can be made in Figure 6.4 is the super linear scaling. From theory we known that this can only be observed if a suboptimal execution time is used as the baseline. However, this is only true if one solves the exact same problem for all number of processors. In our case, due to a communication optimization routine we reorder the matrix entries, so we do not solve the exact same problem in each run.

Another interesting observation we made during our performance evaluation was the impact of different preconditioners for the overall execution time. During our experiments we used two different preconditioner. First the most simple, and default preconditioner in PETSc *block Jacobi*, where each sub block is solved with ILU(0). Secondly we tested with the parallel algebraic multigrid solver *BoomerAMG* from Hypre [10]

As seen in Figure 6.5 there is a major impact on scaling when different preconditioners are used. Interestingly, Hypre scales better than bjacobi for a small number of processors. However, for a large number of processors bjacobi scales better, which is strange since BoomerAMG has previously shown to excelent weak scaling for thousand of processors [9, 11]. Speedup results do not give any information of the overall execution time, hence a slow preconditioner can scale well, we also present actually execution time. As seen in Figure 6.6, Hypre has far better execution time up until 128-256 processors. After that, Hyper's performance appear to suddenly drop, and the entire solver's speedup becomes negative.

It should be noted that during these experiments we did not have a large enough time allocation on Neolith to tune all of Hypre's parameters. Hence, all reported results are with the default parameters provided by PETSc.

7. Summary. We have presented an efficient general adaptive FEM framework for parallel computation on unstructured meshes, with fully distributed algorithms for assembly, error estimation, mesh adaption and load balancing.

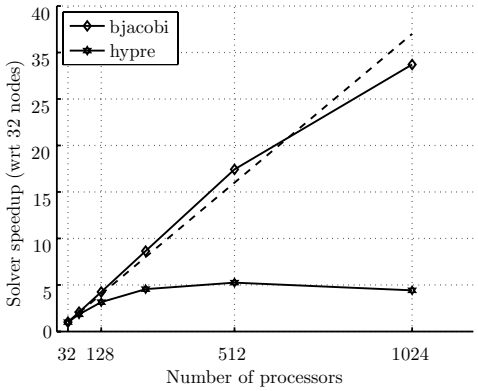


FIG. 6.5. *Comparison of the solvers scaling when using two different preconditioners.*

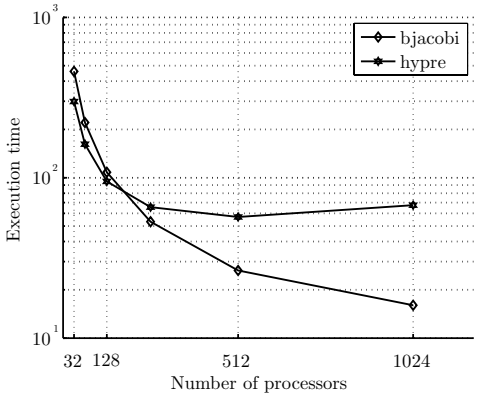


FIG. 6.6. *Comparison of the execution time for one time step using different preconditioners.*

Our framework has shown to perform well, with strong linear scaling up to one thousand processors for an incompressible flow solver. Furthermore, we have improved an earlier parallelization strategy for the recursive longest edge bisection and key parts of an intelligent remapping load balancing framework. With these improvements we also obtained strong linear scaling for our mesh adaption routines up to one thousand processors, allowing us to perform adaptive large eddy simulation simulations of industrial flow problems for realistic geometries [8]. We have thus shown that a general FEM framework can be parallelized efficiently for unstructured meshes.

8. Acknowledgement. The authors would like to acknowledge the financial support from the Swedish Foundation for Strategic Research and the European Research Council. This research was conducted using the resources provided by the Swedish National Infrastructure for Computing (SNIC) at National Supercomputer Centre in Sweden (NSC) and PDC - Center for High-Performance Computing.

Appendix.

Algorithm 7: Parallel refinement of shared entities

for each $c \in \mathcal{B} \cup \mathcal{R}$ **do**
  find longest edge e
  if $e \neq \emptyset$ *and* e *is on the boundary* **then**
    vote(e) $\leftarrow$ vote(e) + 1
  end
end
Exchange votes between processors, mark $c \in \mathcal{B}$ with maximum number of edge votes for refinement
for each $c \in \mathcal{B}$ **do**
  mark e with $\max_{e \in c}$ (vote(e)) for refinement
end
Exchange refinement between processors
for each *received refinement* **do**
  if e *is not refined and not part of a refined element* **then**
    mark edge and propagate refinement
  else
    send back illegal propagation
  end
end
for each *received illegal propagation* **do**
  remove refinements and hanging nodes
end

Algorithm 8: All-to-all like exchange

for $j = 1$ *to* $P - 1$ **do**
  $src \leftarrow (rank - i + P) \mod P$
  $dest \leftarrow (rank + i) \mod P$
  sendrecv(send buffer(dest) to dest and recv from src)
end

REFERENCES

[1] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang, *PETSc Web page*, 2009. http://www.mcs.anl.gov/petsc.
[2] W. Bangerth, R. Hartmann, and G. Kanschat, *deal.II — a general-purpose object-oriented finite element library*, ACM Trans. Math. Softw., 33 (2007).
[3] Eberhard Bänsch, *An adaptive finite-element strategy for the three-dimensional time-dependent navier-stokes equations*, J. Comput. Appl. Math., 36 (1991), pp. 3–28.
[4] J. Bey, *Tetrahedral grid refinement*, Computing, 55 (1995), pp. 355–378.
[5] Lin bo Zhang, *A Parallel Algorithm for Adaptive Local Refinement of Tetrahedral Meshes Using Bisection*, Tech. Report Preprint ICM-05-09, Institute of Computational Mathematics and Scientific/Engineering Computing, 2005.
[6] Josè G. Castaños and John E. Savage, *Parallel refinement of unstructured meshes*, in IASTED PDCS, 1999.

[7] Gaëtan Compère, Jean-François Remacle, Johan Jansson, and Johan Hoffman, *A mesh adaptation framework for dealing with large deforming meshes*, Int. J. Numer. Methods Eng., 82 (2010), pp. 843–867.

[8] R. Vilela de Abreu, N. Jansson, and J. Hoffman, *Adaptive computation of aeroacoustic sources for rudimentary landing gear*, in Proc. Workshop on Benchmark Problems Airframe Noise Computation (BANC-I), 2010.

[9] RD Falgout, JE Jones, and UM Yang, *Pursuing scalability for hypre's conceptual interfaces*, ACM Trans. Math. Softw., 31 (2005), pp. 326–350.

[10] RD Falgout and UM Yang, *hypre: A library of high performance preconditioners*, in Computational Science-ICCS 2002, PT III, Proceedings, vol. 2331 of Lecture notes in Computer Science, 2002, pp. 632–641.

[11] Robert D. Falgout, *An introduction to algebraic multigrid*, Comput. Sci. Eng., 8 (2006), pp. 24–33.

[12] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas, *Overview of the Blue Gene/L system architecture*, IBM J. Res. Dev., 49 (2005), pp. 195–212.

[13] Johan Hoffman, Johan Jansson, Murtazo Nazarov, and Niclas Jansson, *Unicorn*. http://launchpad.net/unicorn.

[14] ———, *Unicorn: A unified continuum mechanics solver*, in Automated Scientific Computing, Springer, 2011.

[15] Johan Hoffman and Claes Johnson, *Computational Turbulent Incompressible Flow*, vol. 4 of Applied Mathematics: Body and Soul, Springer, 2007.

[16] Y. Hu and R. Blake, *An optimal dynamic load balancing algorithm*, Tech. Report DL-P95-011, Daresbury Laboratory, Warrington, UK, 1995.

[17] Niclas Jansson, *Adaptive Mesh Refinement for Large Scale Parallel Computing with DOLFIN*, master's thesis, Royal Institute of Technology, School of Computer Science and Engineering, 2008. TRITA-CSC-E 2008:051.

[18] Benjamin S. Kirk, John W. Peterson, Roy H. Stogner, and Graham F. Carey, *libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations*, Eng. Comput., 22 (2006), pp. 237–254.

[19] Anders Logg and Garth N. Wells, *DOLFIN: Automated finite element computing*, ACM Trans. Math. Softw., 37 (2010), pp. 1–28.

[20] Anders Logg, Garth N. Wells, et al., *DOLFIN*. http://launchpad.net/dolfin.

[21] Leonid Oliker, *PLUM parallel load balancing for unstructured adaptive meshes*, Tech. Report RIACS-TR-98-01, RIACS, NASA Ames Research Center, 1998.

[22] Maria-Cecilia Rivara, *Mesh refinement processes based on the generalized bisection of simplices*, SIAM J. Numer. Anal., 21 (1984), pp. 604–613.

[23] ———, *New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations*, Int. J. Numer. Meth. Eng., 40 (1997), pp. 3313–3324.

[24] Kirk Schloegel, George Karypis, and Vipin Kumar, *Multilevel diffusion schemes for repartitioning of adaptive meshes*, J. Parallel. Distr. Com., 47 (1997), pp. 109–124.

[25] Kirk Schloegel, George Karypis, Vipin Kumar, Rupak Biswas, and Leonid Oliker, *A performance study of diffusive vs. remapped load-balancing schemes*, in 11th Intl. Conference on Parallel and Distributed Computing Systems, 1998.