Limited Resource Scheduling in Sparse Matrix Algorithms

Roldan Pozo

Department of Computer Science
University of Tennessee
Knoxville, TN 37996-1301

Sharon L. Smith

Computer Sciences Division
University of California
Berkeley, CA. 94270

Abstract

Abstract

We present analytic models and simulation techniques that describe the performance of the multifrontal method on distributed memory architectures. We focus on particular strategies for partitioning, clustering, and mapping of task nodes to processors in order to minimize the overall parallel execution time and minimize communication costs. The performance model has been used to obtain estimates for the speedups of various engineering and scientific problems, on several distributed architectures. The result is that the available parallelism of these problems is strongly dependent on the sparsity structure of the input matrices.

1 Problem Statement

The solution of large sparse linear systems constantly arises in many large scale scientific applications, including computational fluid dynamics, structural engineering, weather forecasting, computational physics, circuit simulations, and biomedicine. The resulting matrices are rather large (commonly $10^4 \times 10^4$ to $10^8 \times 10^8$ and larger) and highly sparse (i.e. contain few nonzeroes). Because these problems typically require significant computational resources, there has been much interest to solve large sparse linear systems on parallel architectures. Recent attention has focused on distributed memory architectures that can scale with increasing problem sizes, with particular interest in using small clusters of powerful workstations as a cost-effective means for synthesizing large computational engines.

Various efforts have been directed at solving large sparse systems using direct solvers on distributed memory architectures; a good survey appears in Heath, et. al. [10]. One of the difficulties involved in the distributed implementation of some direct solvers, such as the multifrontal method [8], is that the irregular sparse structure of the matrices makes it difficult to partition and map the sparse matrix to a distributed architecture in a way that minimizes communication costs and minimizes the total execution time of the parallel computation.

In this paper we address the scheduling issues involved in mapping the computational task graphs associated with the multifrontal approach to various distributed memory architectures. In particular, we describe the partitioning, clustering, and mapping strategies used to schedule the algorithm on various architecture configurations.

Our motivation is to study the performance characteristics of the multifrontal method on various parallel architectures and to provide a rough estimate of the speedups that may be observed for various engineering and scientific problems taken from the Harwell-Boeing Test Suite[7]. Our architecture representations include conventional distributed memory parallel machines, such as the Intel iPSC/860, the Intel iPSC/2, the Intel Paragon, the Thinking Machines Corporation's CM-5 and various distributed networks configurations such as a cluster of Sun SPARCstations on standard Ethernet, a dedicated token-ring network of RS/6000s, and SPARCstations on a dedicated high-speed network. **The intent is not to benchmark one architecture against another** (after all, these are only approximations), but to understand how the resource constraints and the communication and computation aspects of a particular architecture affect the obtainable parallelism in distributed sparse matrix methods.

1.1 Approach

The multifrontal method belongs to a class of methods which separate the **LU** factorization into an analysis phase and a numerical factorization phase (for a complete description of the method see [8] and [6]). The analysis phase involves a reordering step, which reduces the fill-in during numerical factorization, and a symbolic factorization step, which identifies the sparsity structure of the matrix factors and builds an elimination tree whose structure indicates the "natural" parallelism of the multifrontal approach and describes the data dependencies between the elimination steps in the numerical factorization. During this step we also amalgamate nodes, so that a single tree node may correspond to several steps of Gaussian elimination.

We focus on the numerical factorization phase of the parallel multifrontal method since it is the most computationally intensive section and generally dominates the overall execution time. By comparison, the ordering, and symbolic factorization phases of the solution are generally smaller and have limited benefits from parallelization [10].

In the numerical factorization phase of the algorithm, the elimination tree is processed from the leaf nodes to the root node. Each node i of the elimination tree corresponds to the Gaussian elimination operations on a full submatrix of size K_i, called a *frontal matrix*. The data dependence, or communication edge, between a node i, and its parent node represents the contribution block that is sent to the parent after the elimination has been performed. The contribution blocks of the children and the subset of variables at node i which can be used as pivots are used to build the frontal matrix of node i. The computation at each node thus consists of an assembly step (for non-leaf nodes), and a factorization step.

In developing a distributed memory implementation of the multifrontal method, the elimination tree can be annotated with information about the estimated communication and computational requirements for each step of the numerical factorization, since the sizes of the frontal matrices and the contribution blocks will be known from the analysis phase. The resulting annotated elimination trees typically resemble unbalanced trees with hundreds or thousands of independent paths, and the granularity of each task increasing towards the root. In our distributed implementation approach, we consider the **partitioning** of large nodes near the tree root, and the **clustering** of small leaf nodes into tasks due to their fine granularity. After the nodes have been partitioned and clustered into *tasks*, we then **map** the resulting tasks onto a parallel distributed architecture. This mapping can be used as a schedule for the tasks in an actual implementation of the distributed algorithm, or it can be used to simulate the behavior of the distributed algorithm, as we have done in this paper.

The rest of the paper is organized as follows. Section 2 presents the analytic performance models used to describe the computation and communication involved at each step of the multifrontal method. Section 3 presents an overview of the performance methodology used, and describes in detail the clustering, mapping, and simulation strategies used. Section 4 gives performance results for several problems from the Harwell-Boeing test suite on different distributed architectures. Finally, Section 5 makes conclusions about this research.

2 Analytic Models

The reordering and amalgamation phase produces an annotated tree where each node i is labeled with its number of pivots (P_i) and the frontal matrix size (K_i), where $P_i \leq K_i$. The size of the contribution block is $(K_i - P_i)$ x $(K_i - P_i)$, and the size of the frontal matrix is K_i x K_i. As discussed in Duff & Johnson [5], the sequential multifrontal factorization code consists of the following basic steps. At each node in the elimination tree we assemble the contribution blocks from child nodes; and, for each pivot in the frontal matrix, (1) determine the pivot (within the fully-summed variables 1,2, ..., P_i), (2) divide the pivot column, and (3) update the trailing submatrices in the frontal matrix.

Let n be the number of nodes in the amalgamated elimination tree, and let γ represent the average time to execute a logical or floating point operation. We can characterize the time required to complete the frontal factorization at node i as follows. The assembly portion consists of receiving the contribution block from the child nodes and adding them (using indirect addressing) into the local frontal matrix. Thus the time required for this is given by

$$\mathrm{A_i} = \gamma\Big(\sum_{\mathrm{j}\in \mathrm{S_i}}(\mathrm{K_j} - \mathrm{P_j})^2\Big) \tag{1}$$

where S_i denotes the nodes which are children of node i. Similarly, the factorization time F_i of the frontal matrix is given by

$$\mathrm{F_i} = \gamma \left(\underbrace{\sum_{j=1}^{P_i}(K_i - j + 1)}_{\text{scan for pivot}} + \underbrace{\sum_{j=1}^{P_i}(K_i - j)}_{\text{scaling}} + \underbrace{2\sum_{j=1}^{P_i}(K_i - j)^2}_{\text{rank-1 updates}} \right). \tag{2}$$

The computation at each node consists of the assembly part, A_i, and the numerical elimination part, F_i (step 2 above.) Thus, the sequential time, T_i, spent at elimination tree node i is given by

$$\mathrm{T_i = A_i + F_i}. \tag{3}$$

Assuming that logical comparisons, floating point multiplies, adds, and divides all take roughly the same time we have, as shown in [5],

$$\mathrm{F_i} = \gamma \mathrm{P_i} \left[(1 + 2\mathrm{K_i}(\mathrm{K_i}+1)) - \frac{2}{3}(\mathrm{P_i}+1)(3\mathrm{K_i} - \mathrm{P_i} + 1) \right]. \tag{4}$$

The overall sequential complexity, then, is simply the sum of work on all nodes in the elimination tree:

$$\mathrm{T^*_{seq}} = \sum_{\mathrm{i=1}}^{\mathrm{n}} (\mathrm{A_i + F_i}) \tag{5}$$

2.1 Tree Parallelism

The simplest way to exploit parallelism is to use the natural parallelism of the computational tree and run a sequential version of the frontal factorization code in each node and process several nodes in parallel. From the program development point of view, this approach is quite attractive since it involves little modification of the sequential code. The node subroutines essentially remain unchanged.

Under ideal conditions with zero communication costs and an unbounded number of processors, the time complexity is simply the longest execution path from a leaf node to the root node in the computational tree,

$$\mathrm{T^*}_{\mathrm{root}} = \max_{\ell} \sum_{\mathrm{i} \in \mathrm{path}_\ell} (\mathrm{A_i + F_i}) \tag{6}$$

Unfortunately, since most of the computational bulk lies near the root nodes where parallelism is limited, speedups using only this parallelizing strategy are rather small. In Duff & Johnson [5], for example, this speedup varied from 1.9 to 4.0 for various 30x30 sample problems, even when an unbounded number of processors is allowed. In previous test cases [15], these speedups remain unchanged for larger 200x200 examples. This can also be observed in the performance results in Section 4.

With realistic communication considerations these speedups drop even further. Typically, large clusters of nodes near the leaves get mapped to a single processor. This is necessary because if nodes are small then it takes much longer to ship the contribution block between processors than to compute the frontal factorization. Remember that the communication at node i consists of shipping a $(K_i - P_i) \times (K_i - P_i)$ contribution matrix to the parent node. We assume that the communication cost of moving n words of data between two processors is given by $M(n) = \alpha + w\beta * n$, where α is the startup cost, β is the per-byte transmission cost, and w is the number of bytes per word (typically four for single precision, eight for double precision). This linear model fits many distributed memory machines rather well for point-to-point communication [14]. The communication time required for node i is given by

$$\mathrm{C_i} = \mathrm{M}([\mathrm{K_i} - \mathrm{P_i}]^2) = \alpha + \mathrm{w}\beta * (\mathrm{K_i} - \mathrm{P_i})^2 \tag{7}$$

The communication cost is incurred only if the parent node is located on a different physical processor. Thus, if $\mathbf{f}(i)$ denotes the processor assigned to node i, the actual communication time is then given by

$$\mathrm{C^*_i} = (1 - \delta_{\mathbf{f}(\mathrm{i}),\mathbf{f}(\mathrm{S_i^{-1}})})\, (\alpha + \mathrm{w}\beta * (\mathrm{K_i} - \mathrm{P_i})^2) \tag{8}$$

where δ is the Kronecker delta symbol ($\delta_{i,j} = 1$ if $i = j$, or 0 otherwise) and S_i^{-1} is the parent node of i. The total time required to process the complete computational tree, therefore, depends on the processor mapping, $\mathbf{f}()$, and the optimal time for T^*_{tree} is given by

$$\min_{\mathbf{f}()} \left[\max_{\ell} \sum_{\mathrm{i} \in \mathrm{path}_\ell} \begin{array}{l} (A_i + F_i) \\ +(1 - \delta_{\mathbf{f}(i),\mathbf{f}(S_i^{-1})})(\alpha + w\beta * (K_i - P_i)^2) \end{array} \right]. \tag{9}$$

2.2 Tree and Node Parallelism

As was noted in the previous section, the speedups that can be achieved using only the parallelism from the tree are limited. We can, however, combine tree parallelism with node parallelism by partitioning the larger nodes towards the root of the elimination tree. In effect we parallelize the frontal matrix factorization on large nodes by performing a close variant of the right-looking LU factorization algorithm. The performance results in [5, 1, 15] have indicated that this approach realizes better performance than if only tree parallelism is employed.

This approach is characteristic of shared-memory paradigms, where the inner loops of computational kernels are parallelized. It is still effective in the distributed memory environment because nodes near the root of the computation tree will dominate the execution time and will have to be decomposed in parallel to achieve significant speedups. Modeling the dense LU parallel factorization of a single node, however, can be rather complicated since the best parallel algorithms rely on the effective overlapping of communication with computation for their successful implementation.

The assembly process is more difficult to parallelize and model since it involves combining results from child nodes. Here we assume that only the factorization phase ($O(K^2 * P)$) is parallelized, i.e. the assembling of contribution blocks ($O(K^2)$) occurs sequentially. We also make the simplifying assumption that most of the internal communication is involved in the original distribution of the frontal matrix over various processors, since most of the communication can be effectively overlapped with computation in modern architectures [3].

In order to model node parallelism, we slightly alter the structure of the elimination tree. If a node is large enough to be partitioned into subtasks of some fixed granularity, we expand these nodes into subgraphs to represent the parallel factorization. The resulting data structure is a series-parallel graph.

Let p denote the number of partitions for a large node i. We expand node i into a subgraph with a fork-join structure, as shown in Figure 1. The "fork" node in the graph corresponds to the sequential assembly phase. We use the notation A_i to denote this node in the example graph, as well as to denote the cost of the assembly for this node. The computation nodes that fork from A_i correspond to the partitioned nodes that perform the factorization. We denote these nodes as $F_{i.1}, F_{i.2}, \ldots, F_{i.p}$. The edges between the fork node and the parallel nodes represent the communication of the frontal matrix after the assembly phase. These edges are denoted as OUT_i in the figure. Finally, the "join" node (denoted as J_i in Figure 1) represents a collection node where all the results from the parallel factorization are gathered. The edges going from the parallel factorization nodes into the join node (labeled as IN_i) represent the communication involved in gathering the frontal matrices at the join node.

The annotation of the nodes in the subgraph is as follows. First, the assembly phase is sequential so its annotation is as given in equation 1. For the parallel factorization phase, the $K_i \times K_i$ matrix must be divided into p pieces and sent to respective processors. This may be done, for example, by partitioning the matrix into block rows that are to be used in a row wise elimination routine, as is described in [1]. The factorization step is split among p processors, and for simplicity we assume that the cost incurred by each processor at this step is the same. (Note that this is only an approximation. In practice it may be achieved if the block rows are partitioned and mapped to processors in a way that balances the computation among processors.)

$$\mathrm{F_i(p) = F_{i.k}} = \frac{\mathrm{F_i(1)}}{\mathrm{p}}, \quad \mathrm{k} = (1, 2, \ldots, \mathrm{p}) \tag{10}$$

In this equation $F_i(1)$ is F_i, or the cost of factorization for an unpartitioned node.

The communication cost of sending the partitioned frontal matrix to each of the factorization nodes, along the communication edges OUT_i is

$$\mathrm{OUT_i} = \left[\alpha + \mathrm{w}\beta * \frac{(\mathrm{K_i})^2}{\mathrm{p}}\right]. \tag{11}$$

In this communication, the fork node sends $(p-1)$ submatrices to each of the processors involved in the parallel factorization step. As described in [14], this type of communication is effectively sequentialized. Therefore, the *effective* time required for *all* nodes to communicate the frontal matrix is

$$(\mathrm{p}-1)\left[\alpha + \mathrm{w}\beta * \frac{(\mathrm{K_i})^2}{\mathrm{p}}\right]. \tag{12}$$

Since the factorization must be performed on all the nodes at the same time to realize the best possible performance in the dense factorization, this delay is enforced upon all nodes involved in the computation (even if only one node actually observes the delay as a result of communication).

Finally, the cost of collecting the results at the synchronization node, J_i, involves only the communication of the factorized sub-matrices to the join node. Thus $J_i = 0$, and the p collection edges, IN_i, have the same cost as the p edges OUT_i for distributing the frontal matrices to the factorization nodes.

Note that the OUT_i and IN_i terms refer only to communication within a subgraph. The communication between the subgraph replacing node i, and the parent node of this subgraph is still denoted by C_i. This is the cost of sending the contribution block to the parent of node i, as given in equation 7.

Thus the total time required to perform the assembly and parallel factorization at each node i is given by

$$\mathrm{T_i} = \underbrace{2(p-1)M(\frac{K_i^2}{p})}_{\text{communication}} + \underbrace{\frac{F_i(1)}{p} + A_i}_{\text{computation}} . \tag{13}$$

The analytical expression for the total time involved in the computation looks similar to equation 9, except that the F_i term has been replaced by $F_i(p)$ and the optimization space is over two dimensions:

$$\min_{\substack{\mathbf{f}() \\ 1 \le i \le K_i}} \quad \max_{\ell} \left(\sum_{\mathrm{i} \in \mathrm{path}_\ell} \begin{array}{c} 2(p-1)M(\frac{K_i^2}{p}) + \frac{F_i(1)}{p} \\ +A_i + (1 - \delta_{\mathbf{f}(i), \mathbf{f}(S_i^{-1})})C_i \end{array} \right) \tag{14}$$

In this case the $\delta_{i,j}$ symbol is 1 if the fork and join nodes of subgraph i are assigned to the same processor as their parent node, S_i^{-1}. Otherwise, $\delta_{i,j}$ is 0.

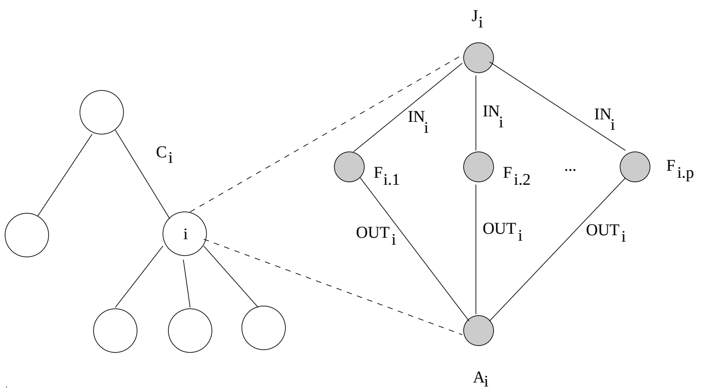

Figure 1: Computation graph structure and expansion of a large node

The mapping procedure presented in the next section describes how this is determined.

Computing this combinatorical optimization is computationally infeasible due to the large search space; in Section 3 we describe a heuristic solution to this problem.

3 Performance Modeling Methodology

This section describes how the annotated computational graph can be used for determining the performance of various problems for different architecture configurations.

3.1 Example Parallel Architectures

We characterize distributed computer architectures by the four classical parameters: γ, the average time to perform a logical or floating point operation; α, the communication startup cost; β, the communication bandwidth, or the per-byte cost of sending a large message; and p, the number of processors. Having a simple parameterization of a parallel architecture $(\alpha, \beta, \gamma, p)$ allows us to study the effects of each component and experiment with various hypothetical situations. Rather than study random combinations of $(\alpha, \beta, \gamma, p)$, however, we have focused on combinations that characterize existing architectures. The list of example architectures used in this study is shown in Table 1 and includes examples from a wide range of computing resources. Megaflop numbers are for double precision operations. Some of the performance numbers, such as those for the iPSC/860 and the iPSC/2, were taken from direct experimentation [14], while others were based on reasonable approximations.

3.2 Example Sparse Matrix Problems

To make our study relevant to practical situations, we have utilized matrices taken from real scientific and engineering applications. These test matrices are part of the Harwell-Boeing Test Suite [7], and are indicative of the medium-scale computing problems that scientists and engineers encounter on a daily basis. The Harwell-Boeing Test Suite was chosen because it is a widely used reference for the design, validation, and benchmarking of sparse matrix solvers and allows for comparison between various machine architectures. A list of the various problems used in this study is shown in Table 2. We have included 9-point Laplacian discretization examples which allow us to construct sparse matrix problems of arbitrary size. Finally, we have purposely included a small-scale problem, NCC1374, to illustrate how problem size limits effective parallelism.

3.3 Simulating the Computational Task Graph

The simulation process consists of several steps. First we create the elimination tree using the MUPSAD() subroutine from the MUPS [2] software package for shared-memory parallel multifrontal solvers. Given the original matrix, we perform minimum degree reordering, amalgamation, and symbolic factorization to generate an amalgamated elimination tree representing the data dependencies between the elimination variables. Second, using the analytical models developed in the previous sections, together with an $(\alpha, \beta, \gamma, p)$ architecture description, we annotate the nodes in the elimination tree with the respective computation and communication times. At this time we also partition larger nodes in order to exploit node

Table 1: Example parallel architectures.

Simulated Architecture	Comp. Rate. (Mflops)	Comm. Startup ($\mu secs$)	Comm. Bandwidth (MB/sec)
Sun-3 on Ethernet	0.4	5,000	0.1
Intel iPSC/2	0.3	321.8	2.8
SPARC-2s on Ethernet	4.0	5,000	0.1
Intel iPSC/860	4.0	321.8	2.8
SPARC-2s on HIPPI	4.0	2,000	50.0
RS/6000 Dedicated LAN	20.0	5,000	1.0
Intel Paragon	20.0	300.0	40
Thinking Machines CM5	50.0	100.0	8

Table 2: Example matrices from the Harwell-Boeing Test Suite.

#	Name	Size	nz	Description
1	9pt70x70	4900	43264	9-point of Laplacian operator (70x70 grid)
2	9pt200x200	40,000	198802	9-point of Laplacian operator (200x200 grid)
3	BCSSTK24	3562	159910	Calgary Olympic Arena
4	BCSSTK33	8738	591904	automobile steering mechanism
5	NNC1374	1374	8606	Nuclear reactor core modelling

parallelism. Third, given the new annotated graph, we perform clustering of the nodes to form tasks, and the contraction (if necessary) of tasks to fit a given architecture. Finally, with each processor in the architecture assigned a pool of tasks we can then simulate the execution of the tasks. The results of the simulation are the performance measures that characterize the behavior of the multifrontal method for a particular test problem, architecture, and mapping. In addition, the simulation phase produces a schedule of tasks that could be used in a distributed implementation of the algorithm.

3.3.1 Annotating and Partitioning.

The annotation of the elimination tree is performed as described in section 2. At the same time as the annotation, we perform the partitioning of the larger nodes in the tree, in order to gain more parallelism from the computation. Each node is partitioned into several pieces based upon some fixed granularity that can be set by the user. (For example, in Section 4 we examine several granularities and show the results of these experiments). The granularity of the resulting tasks is a crucial issue in the resulting performance of this phase of the computation. The primary factor influencing the choice of granularity for this model is the communication speed of a particular architecture. There are other considerations such as the hardware characteristics of the architecture (for example, vector register length and the pipeline startup time) that may also affect the performance, however, we do not address these issues here.

When a node is partitioned into a subgraph during this phase of the computation, we limit the number of node partitions to the number of physical processors available. This is done so that all the nodes of a partitioned subgraph will execute simultaneously, in order to efficiently parallelize the factorization step and overlap communication of dense factors and computation.

3.3.2 Clustering.

After annotation and partitioning, the nodes of the resulting computational graph are clustered into tasks. There are two steps involved in this clustering. The first step uses a clustering algorithm similar to the *dominant sequence clustering* (DCS) algorithm of Yang and Gerasoulis [17]. The algorithm is based on stepwise refinement, and begins with the computation tree nodes mapped onto distinct processors. For each node in the graph, we sort its child nodes in decreasing order of combined (communication + computation) time and attempt to cluster them (zero out the communication edge) with the parent node if it does not increase the overall parallel execution time. We repeat this process recursively for each child in the computation graph with a unique parent, beginning with the leaf nodes.

For those nodes with multiple parents (resulting from a subgraph expansion), each node is assigned to a different processor. Since all the nodes of a parallel factorization phase must execute at the same time in parallel, it is important that the distribution of nodes to processors be balanced. This is insured in the partitioning phase, where it is assumed that each factorization node will execute on a different processor. Thus the clustering algorithm does not attempt to remap these types of nodes.

After the computational graph has been clustered in this fashion, we attempt to merge task clusters that occur sequentially in the graph. The algorithm used is similar to the linear clustering algorithm described in [11], although we relax the constraint that clusters must be linear.

In order to merge clusters, it is necessary to know the following quantities for each node i in a cluster: the level of the node ($LVL(i)$), the earliest possible start time ($EST(i)$), the latest possible start time ($LST(i)$), and the latest completion time, ($LCT(i)$). These quantities are defined below and can be determined either by a depth-first search order traversal of the computation graph, as done for the EST information, or by a breadth first search order traversal, as done for the LVL, LST, and LCT information. Recall that $S(k)$ is the set of children of node k, S_k^{-1} is the set of parents for node k, and that T_k and C_k refer to the computation and communication times of node k, as defined in equations 3 and 7.

$$LVL(i) = \begin{cases} 1 \quad \text{if } i \text{ is a root node} \\ (\max_{j \in S_i^{-1}}(LVL(j)) + 1 \quad \text{otherwise.} \end{cases}$$

$$EST(i) = \begin{cases} 0 \quad \text{if } i \text{ is a leaf node} \\ \max_{j \in S(i)}(EST(j) + T_j + (1 - \delta_{\mathbf{f}(i),\mathbf{f}(j)})C_j) \\ \text{otherwise.} \end{cases}$$

$$LST(i) = \begin{cases} EST(i) \quad \text{if } i \text{ is a root node} \\ \min_{j \in S_i^{-1}}(LST(j) - T_i + (1 - \delta_{\mathbf{f}(i),\mathbf{f}(j)})C_i) \\ \text{otherwise.} \end{cases}$$

$$LCT(i) = LST(i) + T_i.$$

Once these quantities have been determined for each node in the graph, clusters can be merged together if they appear to be sequentially dependent. The following conditions are used to test if two clusters, i and j can be merged. Let L_i and L_j be the set of level numbers for the tasks in cluster i and cluster j.

1. $L_i \cap L_j = \phi$
2. EITHER $\max_{k \in \text{cluster } i} LCT(k) < \min_{t \in \text{cluster } j} LST(t)$,
   OR $\max_{k \in \text{cluster } j} LCT(k) < \min_{t \in \text{cluster } i} LST(t)$.

Note that to insure that two clusters i and j are *strictly* sequentially dependent, the second condition should replace the LST by the EST in the equation. In our application, however, most of the clusters that are candidates for sequential dependence have an EST of 0, since they are leaf nodes in the computation. Thus, we relaxed this condition in order to be able to merge clusters.

Finally, after merging all possible clusters satisfying these conditions, we map the task clusters to a parallel architecture. We do this by determining the load or the amount of computation at each task cluster, sorting the clusters by their load, and then mapping the clusters in a wrapped fashion to the processors. This type of mapping has also been used in practice by [9, 16] and in the PYRROS scheduling tool, developed by Yang and Gerasoulis [17].

Note that at this level of modeling, it makes little sense to model the topology, so we assume a flat topology and are concerned with reducing the number of clusters to be the same as the number of processors. This type of topology is used in the iPSC/860 and the Paragon machines.

The above mapping procedure results in an assignment of tasks to processors but it does not provide an ordering by which tasks should execute on a processor. To obtain this information, we determine a schedule via simulation as discussed next.

3.3.3 Simulating the execution of the task graph.

The performance measures and the task schedule for a particular task graph are obtained by performing a discrete event simulation of the execution of the tasks in the computation graph.

In the discrete event simulation, the tasks from the computational graph are placed into the ready list of their assigned processor when their child tasks have completed execution. The ready list is ordered by the *ready time* of a task, which is the earliest time that a task may begin execution. The ready time takes into account the communication time of data transferred from a child to its parent(s). A task k is ready to execute when the simulation time of its processor is equal to or greater than the ready time of task k. The execution of task k is then simulated by advancing the simulation clock of its processor by its computation time, T_k.

For the execution of partitioned nodes, it is necessary to insure that each processor involved in the parallel factorization executes the tasks of the partitioned node at the same time. In order to model this in the simulation, the subgraph task nodes associated with the parallel factorization are each inserted into their appropriate ready lists at the first possible time that *all* tasks can begin execution. The ready lists of these processors are then *locked* at each of the task nodes. Locking a ready list at a task i insures that no task with a smaller ready time than task i can be inserted into the processor ready list if its execution will delay the start of task i. Note that without locking, the ready time of a task i only guarantees the position of task i in the ready list. A lock on task i insures that task i will begin execution at its ready time.

At the completion of the simulation, we can calculate the overall time of the computation, based on the value of the largest simulation processor clock. In addition, by recording the order of execution of tasks on a processor, we have a task schedule for an actual implementation.

4 Performance Results

For each of the test problems described above, we considered different sized partitions of frontal matrices for the nodes in the elimination tree that were large enough to be partitioned. The performance results for the multifrontal method using the partitioning, clustering and mapping algorithms described, are shown in Tables 3 and 4. As mentioned in Section 3.3, the choice of granularity is an important factor in the performance of the algorithm. In the experiments presented here, we assume that frontal matrices are partitioned for use with a row oriented solver, such as the KIJ-SAXPY algorithm, where, at the kth step of this block form, a block row of factors is computed and the corresponding transformations, based on Level-3 BLAS kernels [4], are applied to the remaining reduced matrix.

Table 3 illustrates, for a partition size of 10 rows, the best possible speedup that can be achieved for each test problem and the number of processors needed to achieve that speedup. The speedups are shown for architectures with limited resources and compared to the speedups obtained for the same architectural characteristics without a fixed number of processors. In the unlimited resource case computational nodes are clustered together to reduce communication, and the number of resulting clusters is given in the table as the number of processors. In the limited resource case, the number of processors shown is the smallest number of processors needed to achieve the highest possible speedup for a given architecture.

Also illustrated in the table is the discrepancy between the speedups that are achievable with and without limitations imposed on the resources. One reason for this is that the parallelization of the large nodes requires that all processors assigned tasks in this phase be available. This type of synchronization often causes processors to be idle. The largest differences occur in the biggest problems; for example it is possible with unlimited resources to achieve speedups of about 71 for the problem BCSSTK33 on the HiPPI network of SPARC-2s, whereas only half of that is actually realized with limited resources. Table 3 also indicates that the processor efficiency (speedup divided by the number of processors) for even the large problems is not so good. This is true even on the best balanced systems, such as the HiPPI network and the iPSC/2, where the best speedups are observed.

Table 3 indicates that the particular partition size of 10 rows does not always yield suitable performance. In Table 4 we present comparisons for the performance of the multifrontal method using several different granularity sizes: 5 rows, 10 rows, 20 rows, 30 rows, 40 rows, and an infinite number of rows. The latter case is equivalent to no partitioning, that is, we do not subdivide the frontal matrices and only the resulting tree parallelism, as described in Section 2.1, is exploited. The tables contrast the behavior of the Intel Paragon and the CM5 on a large problem with good potential for speedup. The balance between the computation and communication speeds of the Paragon allow it to realize both better execution times and speedups in this problem, with smaller partition sizes producing the best results.

5 Conclusions

We have derived analytic and simulation performance models for the numerical factorization phase of the multifrontal method and have applied these models to several large examples from the Harwell-Boeing Sparse Matrix Suite to give an indication of the speedups obtainable on various distributed memory architectures.

The results from these simulations suggest that multifrontal methods share similar characterizations with other parallel direct solvers –they are quite effective at achieving reasonable speedups on shared memory architectures and small clusters of workstations, but they may prove more challenging to adapt to massively parallel distributed systems with thousands of processors. Various test examples on realistic problems have shown that the available parallelism is strongly dependent on the sparsity structure and fill-in characteristics of the input matrix.

One source of difficulty lies in the imbalance of elimination trees for sparse matrices with irregular patterns. By comparison, one typically sees better performance for similar algorithms on structured prob-

Table 3: Best limited resource speedup (S) and number of processors (P) used vs. the unlimited resource speedup (S) and number of processors (P) used (Row size of 10).

Architecture	**LA70x70**				**LA200x200**				**BCSSTK24**				**NNC1374**			
	Limited		**Unlimited**		**Limited**		**Unlimited**		**Limited**		**Unlimited**		**Limited**		**Unlimited**	
	S	P	S	P	S	P	S	P	S	P	S	P	S	P	S	P
Sun-3/Ethernet	2.34	80	2.37	1760	6.96	100	7.43	17020	3.84	40	3.84	1909	0.74	20	0.75	320
iPSC/2	14.47	512	23.67	2074	45.61	512	71.53	19521	28.58	128	38.92	1909	6.56	512	7.25	518
SPARC-2/Ethernet	0.26	10	0.26	1753	0.81	20	0.81	17005	0.41	10	0.41	1909	0.08	10	0.08	238
iPSC/860	5.38	512	5.68	1776	16.62	512	18.20	17063	9.30	256	9.30	1909	1.68	256	1.69	325
RS/6000 LAN	0.37	10	0.37	1753	1.43	10	1.43	17005	0.69	10	0.69	1909	0.07	10	0.07	241
SPARCS/HIPPI	6.97	80	8.34	1755	24.67	100	46.33	17007	17.35	80	18.50	1909	1.06	10	1.22	283
Paragon	6.84	512	7.27	1757	26.78	512	31.19	17008	13.98	256	14.09	1909	1.32	256	1.32	289
CM5	1.54	128	1.54	1753	4.81	256	4.85	17005	2.48	32	2.42	1909	0.45	32	0.46	294

Table 4: Comparison of performance parameters (execution time in seconds and speedup) for different partition sizes for problem BCSSTK33 on the Intel Paragon and the Thinking Machines CM5

Paragon	**Partition Size**											
# procs	infinity		40		30		20		10		5	
32	58.50	2.03	8.85	13.41	8.90	13.33	7.81	15.19	6.89	17.23	6.41	18.49
64	55.38	2.14	8.85	13.41	6.76	17.55	5.56	21.33	4.49	26.41	3.89	30.48
128	55.38	2.14	6.95	17.07	4.83	24.57	4.50	26.37	3.51	33.79	3.20	37.07
256	55.38	2.14	5.95	19.93	4.53	26.18	3.77	31.46	3.11	38.14	3.03	39.16
512	55.38	2.14	5.78	20.51	4.70	25.24	3.62	32.72	2.94	40.35	2.89	41.00
CM5												
# procs	infinity		40		30		20		10		5	
32	25.20	1.88	8.41	5.64	8.70	5.45	8.24	5.76	7.87	6.03	7.76	6.12
64	22.20	2.14	8.08	5.87	8.25	5.75	7.75	6.12	7.46	6.36	7.43	6.38
128	22.20	2.14	8.03	5.91	7.95	5.97	7.64	6.21	7.40	6.41	7.34	6.47
256	22.20	2.14	8.02	5.91	7.91	6.00	7.62	6.22	7.39	6.42	7.33	6.47
512	22.20	2.14	8.02	5.91	7.91	6.00	7.62	6.23	7.40	6.41	7.34	6.47

lems [12, 13]. The coarse grain parallelism available just from the elimination tree (with no partitioning) is limited. The tree parallelism can, in some instances be augmented by partitioning the larger frontal matrices. In many cases, such as with the large problems on well balanced architectures this approach yields moderate speedups. However with small problems and poorly balanced systems, such as network clusters of workstations, the results can be discouraging. Certainly the most significant factor affecting performance is that the high communication costs on distributed architectures limit the amount of parallelism available to decompose single frontal factorization into finer grained tasks. It should also be noted that the multifrontal method is already highly efficient on a single processor, since it employs dense linear algebra in its computational kernel. This efficient computation section, however, also serves to make the communication bandwidth limitation even more visible.

Another difficulty arises from the need to explicitly deal with task mapping and scheduling issues. In shared-memory environments, such concerns can be handled quite effectively by simple task-queue scheduling techniques. In distributed memory environments with considerably higher communication costs and fragmented memory pools, such approaches are not as effective. The types of clustering and mapping strategies that we have used in this performance analysis are needed in order to perform the scheduling. In general, such mapping problems have been shown to be NP-complete and can be quite expensive to calculate, although the ones we have used are heuristic based strategies and much more efficient, but not without cost. The performance of the cluster merging algorithm, for example, is $O(k^2)$, where k is the number of clusters. For problems comparable in size to the LA200x200, the number of clusters can easily be in the thousands, depending on the architecture.

The extent to which a static mapping strategy can be effective is also questionable. As the comparison of the unlimited and limited resource models indicated, there is still some room for improvement in the clustering, mapping, and scheduling of tasks in order to better utilize the limited resources. Further-more, we have assumed that numerical pivoting occurs within the pivot variables designated in the symbolic factorization. Because of this we have been able to estimate communication and computation costs accurately enough to perform a static scheduling of tasks to processors. In practice suitable pivot variables might not be found within the frontal matrix as predicted. In this case, additional fill-in of the matrix may be created, resulting in additional computation. This may alter the schedule of a static mapping strategy. Because of this, it may be necessary to consider hybrid approaches that combine both static and dynamic scheduling techniques.

Finally, although we have tried to accurately model the computation and communication characteristics of the algorithm, these performance models are not intended to be the definitive word on the performance aspects of the multifrontal method on distributed memory architectures. There have been various simplifying assumptions (e.g. the restriction of full pivoting, strict data flow dependencies between nodes of the amalgamated elimination tree, fixed γ during lifetime of computation, etc.) imposed to make the analytical and simulation modeling possible. At best, they can be used to predict the performance of various problems, and to expose potential difficulties in the implementation of the method for distributed memory architectures.

6 Acknowledgements

The authors would like to thank Iain Duff and Patrick Amestoy for numerous helpful discussions on the multifrontal method. This work was conducted while Sharon Smith had a postdoctoral research position at CERFACS, 42 Avenue Gustave Coriolis, 31057 Toulouse, France.

References

[1] Amestoy, P. R. Factorization of large sparse matrices based on a multifrontal approach in a multiprocessor environment. PhD Thesis TH/PA/91/2, CERFACS, Toulouse, France, 1991.

[2] Amestoy, P. R. Mups: Subroutine library specification. internal documentation, CERFACS, Toulouse, France, 1991.

[3] Choi, J., Dongarra, J. J., Pozo, R., and Walker, D. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computing* (October 1992).

[4] Dongarra, J. J., Du Croz, J., Duff, I. S., and Hammarling, S. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw. 16* (1990), 1–17.

[5] Duff, I., and Johnsson, L. Node orderings and concurrency in structurally-symmetric sparse problems. In *Parallel Supercomputing: Methods, Algorithms, and Applications*, G. Carey, Ed. John Wiley & Sons Ltd., 1989, pp. 177 – 189.

[6] Duff, I. S., Erisman, A., and Reid, J. K. *Direct Methods for Sparse Matrices.* Oxford University Press, London, 1986.

[7] Duff, I. S., Grimes, R. G., and Lewis, J. G. Sparse matrix problems. *ACM Transactions on Mathematical Software 14* (1989), 1–14.

[8] Duff, I. S., and Reid, J. K. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software 9* (1983), 302–325.

[9] Geist, G. A., and Heath, M. T. Matrix factorization on a hypercube multiprocessor. In *Hypercube Multiprocessors* (Philadelphia, 1986), SIAM, pp. 161–180.

[10] Heath, M. T., Ng, E., and Peyton, B. W. Parallel algorithms for sparse linear systems. *SIAM Review 33* (1991), 420–460.

[11] Kim, S., and Browne, J. A general approach to mapping of parallel computation upon multiprocessor architectures. In *Proceedings of the International Conference on Parallel Processing* (1988), pp. 1–8.

[12] Lucas, R., Blank, T., and Tiemann, J. A parallel solution method for large sparse systems of equations. *IEEE Transactions on Computer-Aided Design CAD-6* (1987), 981–991.

[13] Ostrouchov, L. S., Heath, M. T., and Romine, C. H. Modeling speedup in parallel sparse matrix factorization. Tech. Rep. ORNL/TM-11786, Oak Ridge National Laboratory, Oak Ridge, TN., 1990.

[14] Pozo, R. *Performance Modeling of Parallel Architectures for Scientific Computing.* PhD thesis, University of Colorado, Boulder, 1991.

[15] Pozo, R. Performance modeling of sparse matrix methods for distributed memory architectures. In *CONPAR 92 - VAPP V* (September 1992).

[16] Saad, Y. Gaussian elimination on hypercubes. In *Parallel Algorithms and Architectures*, M. Cosnard and et. al, Eds. Elsevier Science Publishers, North Holland, 1986.

[17] Yang, and Gerasouslis, A. Pyrros: Static scheduling and code generation for message passing multiprocessors. In *Proc. of the 6th ACM Inter. Conf. on Supercomputing* (July 1992).