Chapter 1
Performance evaluation of the parallel multifrontal method in a distributed memory environment

Roldan Pozo * Sharon L. Smith †

Abstract

We study, using analytic models and simulation, the performance of the multifrontal methods on distributed memory architectures. We focus on a particular strategy 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.

1 Problem Statement

There have been various efforts directed at solving large sparse systems using direct solvers on distributed memory architectures (see [3] for a survey). One of the difficulties involved in the distributed implementation of some direct solvers, such as the multifrontal method [2], 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. This issue has been addressed by Pothen and Sun [5], who have developed a distributed algorithm for the multifrontal method that uses a proportional mapping scheme for assigning tasks in a clique tree to processors. In this paper we study the performance characteristics of a partitioning and mapping strategy, also based on general tree and graph mapping strategies, for the implementation of the multifrontal method on distributed memory architectures.

We have implemented an analytic model in [6], that captures the communication and computation aspects of the numerical factorization phase of the multifrontal algorithm. The performance results described in [6] are, however, restricted to the unlimited resource case. In this paper we incorporate the partitioning and mapping strategy discussed below that allows us to simulate the performance of matrix computation on actual architectures with limited resources.

2 Approach

The multifrontal method belongs to a class of methods which separate the **LU** factorization into an analysis phase and a numerical factorization phase. The analysis phase involves a reordering step, which reduces the fill-in during numerical factorization, and a symbolic factorization step, which 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.

*Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301
†CERFACS, 42 Avenue Gustave Coriolis, 31057 Toulouse, France.

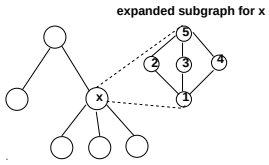

FIG. 1. *Computation graph structure and expansion of a large node*

In the numerical factorization phase of the algorithm, the elimination tree is processed from the leaf nodes to the root node. Each node, i say, 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 contributions 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.

For developing a distributed memory implementation of the multifrontal method, we observe that the elimination tree derived from the analysis phase 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. This annotation is described in detail in [6]. 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 onto a single processor 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 for this paper.

We now describe the particular partitioning, clustering, and mapping strategies used. First, at the same time as the annotation, we perform the partitioning of the larger nodes in the tree. For each node that can be partitioned into more than one frontal matrix of a minimum blocksize, we expand this node into a subgraph that replaces the large node in the elimination tree. For example, in Figure 1, if node x can be partitioned into 3 frontal matrices, it is replaced by the subgraph shown. Node 1 corresponds to the assembly operations of the frontal matrices that are children of x. Nodes 2, 3, and 4 correspond to factorizations on the frontal matrix of x, after it has been partitioned. Node 5 is a synchronization node and performs no additional computation.

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[7]. 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 the 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

TABLE 1
Speedups of Harwell Boeing test problems on various architectures

Architecture Type	Number of Processors	Laplacian 9pt, 200x200	Laplacian 9pt, 70x70	NNC1374	BCSSTK15	BCSSTK24
Suns/Ethernet	20	14.36	8.70	4.09	13.41	10.82
iPSC/2	128	34.65	21.41	5.10	37.94	29.27
SPARC-2/Ethernet	20	9.27	2.46	1.23	7.09	3.48
iPSC/860	128	42.22	17.98	4.97	38.09	23.35
RS/6000 LAN	20	10.80	3.35	1.21	9.10	5.13
SPARCS/HIPPI	20	14.09	11.02	3.99	13.68	13.97
Crays/HIPPI	10	8.52	4.66	1.26	8.28	6.88

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), we perform a similar type of clustering between a child and its parents.

After the computational graph has been clustered in this fashion, we attempt to merge task clusters that are *sequentially dependent*. This is done in a manner similar to that described in [4]. In particular, we first determine for each node the earliest starting time (*est*) that is possible, the latest starting time (*lst*) that is possible, and the level of each node in the graph, relative to the root node. Using this information we can determine task clusters which are sequentially dependent and merge them together. Sequentially dependent clusters should not increase the overall parallel execution of a particular cluster.

Finally, after merging all possible sequentially dependent clusters, we map the task clusters to a parallel architecture. (In this paper, our model does not account for the topology of the network of a particular architecture, so we are only concerned with reducing the number of clusters to be the same as the number of processors). We do this by determining the "load" 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 been used in practice (see for example [7]).

After the mapping, we have an assignment of task clusters to processors. We then simulate the execution of the parallel algorithm using this task assignment.

3 Performance Results

Table 1 show some performance results of our study for several test problems from the Harwell Boeing test suite [1]. Two distributed multiprocessors are represented among the architectures, as well as several system arrangements using clusters of various types of workstations, connected either by an Ethernet, a dedicated Ethernet (LAN), or by a 50 MB/sec HIPPI interface. In this table, the problems with the largest number of nodes (the 200x200 9-point Laplacian and BCSSTK15) show the best speedups.

Figures 2 and 3 show the effect of the different blocksizes on speedup, for both workstation configurations and for the multiprocessors. The smaller blocksize displays the best performance speedups in all cases, since it increases the potential parallelism of a large node. Because of the high cost of communication startup in the network systems, the small granularity of the partitioned nodes is not as advantageous for systems in which the communication and computation is not well balanced. This relationship is apparent in Figure 2, where the Sparc/HIPPI cluster is much better balanced than the Sparc-2/Ethernet

cluster.

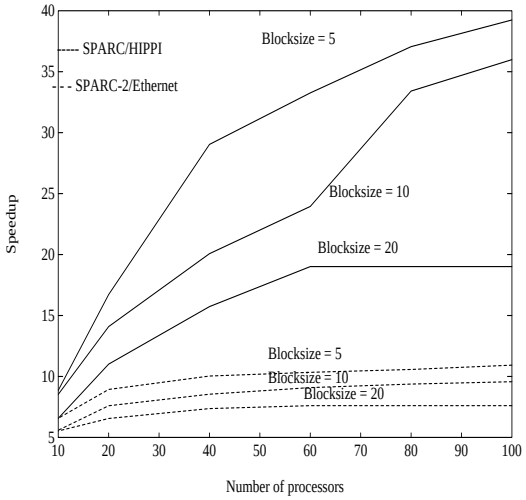

FIG. 2. *Speedup for problem BCSSTK33 for different blocksizes on network systems*

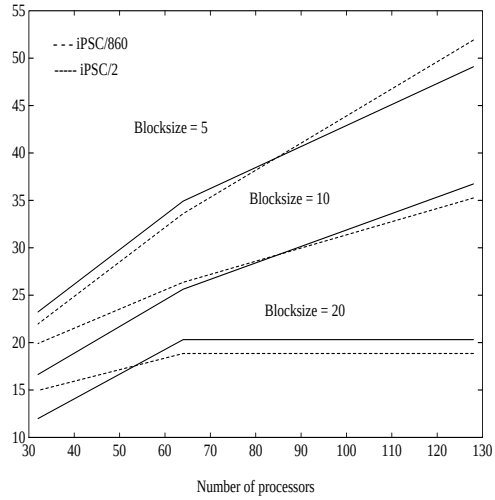

FIG. 3. *Speedup for problem BCSSTK33 for different blocksizes on hypercube systems*

4 Summary

Our results indicate that task partitioning and mapping is an important consideration for the performance of the multifrontal method on distributed memory architectures. There is promise for reasonable speedups, particularly for large sparse problems on clusters of workstations that have a reasonable communication network. We intend to continue experimenting with different task partitioning and mapping strategies for this algorithm, as well as with dynamic strategies for load balancing that will enable the redistribution of frontal matrices to improve upon an initial static partitioning and mapping of the elimination tree.

References

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

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

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

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

[5] A. Pothen and C. Sun, *Distributed multifrontal factorization using clique trees*, in Proceedings of 5th SIAM Conference on Parallel Processing for Scientific Computing, J. Dongarra, K. Kennedy, P. Messina, D. C. Sorenson, and R. G. Voigt, eds., SIAM, Philadelphia, 1992, pp. 34–40.

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

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