Efficient Run-time Support for Irregular Task Computations with Mixed Granularities

Cong Fu and Tao Yang
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
http://www.cs.ucsb.edu/{$\sim$cfu,$\sim$tyang}

Abstract

Many irregular scientific computing problems can be modeled by directed acyclic task graphs (DAGs). In this paper, we present an efficient run-time system for executing general asynchronous DAG schedules on distributed memory machines. Our solution tightly integrates the run-time scheme with a fast communication mechanism to eliminate unnecessary overhead in message buffering and copying, and takes advantage of task dependence properties to ensure the correctness of execution. We demonstrate the applications of this scheme in sparse LU and Cholesky factorizations for which actual speedups have been hard to obtain in the literature because parallelism in these problems is irregular and limited. Our experiments on Meiko CS-2 show the promising results of our approach in exploiting irregular task parallelism with mixed granularities.

1 Introduction

Solving irregular problems, where patterns of computations and communications are unstructured and/or changing dynamically, is very important to many scientific applications. The automatic parallelization of such problems on distributed memory machines is extremely difficult and presents a great challenge. Automatic scheduling and load balancing techniques are useful in exploiting irregular parallelism in unstructured computations [CRY94, LB94, GJY95]. For iterative irregular problems in which communication and computation phases alternate, the CHAOS/PARTI system [DUSH94] has used the inspector/executor approach to exploit irregular parallelism at each iteration of the computation phase. The key idea of this approach is to identify data accessing patterns at run-time and optimize data distribution and communication for the rest of computation.

The CHAOS system handles parallelism with no loop-carried dependence. In a more complicated iterative computation, each iteration may involve task parallelism with irregular dependence structures. For example, in the adaptive n-body simulation using the fast multipole method, parallelism at each time step can be modeled as a directed acyclic task graph (DAG) [GJY95]. A DAG schedule can be re-used for a number of iterations before rescheduling since particles movement is slow. Then good speedups are obtained on nCUBE-2 for large n-body simulation problems after applying PYRROS DAG scheduling techniques [YG92]. Such results demonstrate that graph scheduling can effectively exploit irregular task parallelism to balance loads and overlap computation with communication. In solving nonlinear equations using iterative methods, sparse matrix factorization usually dominates the computation time at each iteration step. The topology of iteration matrices usually remain the same but the numerical values in these matrices change at each step [Kar91]. Efficient parallelization of sparse factorization requires certain compilation cost, but the optimized solution can be used for many iterations. Sparse matrix factorizations can be modeled as DAGs [Sch93]. However compared to large n-body simulations, efficient execution of DAG schedules for sparse matrix problems is more challenging because the partitioned graphs contain both coarse and fine grained tasks while in the DAGs arising from large n-body simulations, most of the tasks are typically coarse-grained [GY93].

The early work in the PYRROS system [YG92] provides a complete framework for general task computation; however, its run-time support system has significant overheads in managing message buffers and copying data between user and system space, which prevents PYRROS from obtaining good performance in executing sparse matrix computations with mixed granularities. In [VNS92], static task and communication scheduling is used for sparse Cholesky factorization, and they found that the prescheduling improves the performance of distributed factorization by 30% to 40%. However the run-time overhead for task execution is still large and there is still a great deal of room for obtaining better absolute performance. Recently the work by [CSBS95] demonstrates that using both effective DAG scheduling and low-overhead communication mechanisms, scalable performance can be obtained on fine-grained DAGs for solving sparse triangular systems.

In this paper, we present an efficient run-time support system for executing *general* DAG computations with mixed granularities. The distinguishing feature of our work is that we *tightly* incorporate a fast communication mechanism into our run-time scheme. Such tight integration allows us to eliminate unnecessary cost in managing message buffers and copying data. The test cases indicate that the overhead is only $6-10\%$ of the execution time. A pitfall in such a scheme is that processors may obtain incorrect data copies without a proper message synchronization. We carefully design the task-level communication protocol and ensure the correctness of execution by taking advantage of properties of task graphs.

We demonstrate the applications of our techniques for sparse matrix factorizations. It has been very difficult to obtain actual speedups by hand-made code for these problems, not to mention the automatically parallelized code. Recently impressive performance has been obtained in sparse Cholesky factorization [Rot92, RS94]. Our experiments will show that our automatically scheduled code delivers acceptable performance.

The paper is organized as follows. Section 2 describes the task computation model. Section 3 identifies critical issues in developing an efficient run-time system. Section 4 presents the design of our system in detail and the correctness analysis of our method. Section 5 presents experimental results on sparse LU without pivoting and Cholesky factorization. Section 6 gives the conclusions and future work.

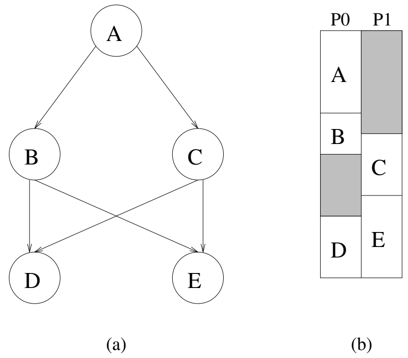

Figure 1: An example of DAG and its schedule on 2 processors.

2 The Task Computation Model

2.1 Definitions and assumptions

Our work is targeted at message passing distributed memory architectures. The task computation model consists of a set of tasks $T = \{T_1, T_2, \ldots, T_n\}$ and a set of distinct data items $D = \{d_1, d_2, \ldots, d_m\}$. Each task can read, define and modify data items. A dependence edge from task T_x to T_y indicates that task T_x writes a data item which T_y reads. The task dependence graph is a DAG.

Given a DAG G, a schedule for it derived from the scheduling algorithms usually contains the starting time for each task on each processor. This symbolic schedule will strictly satisfy all the dependence constraints contained in G. However for execution purposes, the symbolic schedule can not be used directly. Instead, two pieces of information from the schedule will be enough to execute the DAG. One is a mapping Φ between task set T and processor set $P = \{0, 1, 2, \ldots, p-1\}$. And the other is p static task orderings $O_0, O_1, \ldots, O_{p-1}$, one for each processor. For example, Figure 1(a) shows a DAG, and Figure 1(b) is its schedule on 2 processors. We have the processor assignment $\Phi(A) = \Phi(B) = \Phi(D) = 0$, $\Phi(C) = \Phi(E) = 1$, and the execution orderings $O_0 = \{A, B, D\}$, $O_1 = \{C, E\}$.

Algorithms for the static scheduling of DAGs have been extensively studied in the literature, e.g. [CR92, KB88, Sar89, YG92]. Dynamic scheduling which is adaptive to run-time change of dependence structures and weights is more attractive [Pol88], but it is still difficult to balance its benefits and the run-time control overhead in executing tasks with mixed grains on message-passing machines. Static scheduling can tolerate a certain degree of run-time weight variation [GJY95, YFGS95] and its run-time control mechanism is relatively simple. It is important to minimize run-time overhead in order to gain good speedups for executing tasks of non-uniform grain sizes.

2.2 Properties of task dependence

We study the properties of task dependence so that our run-time support can take advantage of those properties in designing the task communication protocol.

Classical data dependence graphs (DDG) for modeling the interaction between computation statements have three types of dependencies: true, anti and output. With the presence of anti and output dependence, run-time synchronization becomes more complicated in order to ensure the correctness of execution. In functional languages, the "single assignment" principle has been used for data flow computation [Sar89]. Renaming techniques [CF87] can eliminate the output and anti dependence in transforming a DDG to a "single assignment" graph. The advantage of such principle is that dependence enforcement in task execution can be simplified to a certain degree. The disadvantage is that memory usage and data copying overhead increase accordingly.

The task graph model we use is less restrictive than "single assignment" functional data flow graphs in the sense that data items can be modified more than once. Such a model more naturally fits into many scientific problems, for example, linear algebra algorithms usually update matrix elements gradually. Thus unnecessary data renaming and copying can be avoided when we transform a DDG to a task graph. In transforming a DDG of a program to a task graph, we first delete all redundant output and anti dependence edges. If the resulting graph contains true data dependence only, then this graph is a task graph. Otherwise other transformations are necessary to remove the remaining anti and output dependence [FY95]. We call this kind of task graph a *dependence-complete* task graph.

Formally a task graph G is dependence-complete if it satisfies the following four properties:

- **DA1:** *A task only uses distinct data items.* A task T_x does not receive data items with the same ID from different predecessors.
- **DA2:** *Write-read and read-write data dependence.* If task T_x produces data item m and another task T_y reads data item m, then there must be either a path from T_x to T_y or a path from T_y to T_x in G.
- **DA3:** *Write-write data dependence.* If task T_x produces data item m, and another task T_y also produces data item m, there must be either a path from T_x to T_y or a path from T_y to T_x in G.

- **DA4:** *A DAG is sequentializable with respect to a topological ordering of tasks.* An execution following any dependence order imposed by edges in G should lead to the same semantics as that of the sequential program.

These properties impose natural constraints that a task graph should have in representing a task computation. Most task graphs derived from real scientific applications satisfy these properties without any transformation. For example, the sparse LU graphs discussed in Section 5.1 are dependence-complete. One disadvantage of a dependence-complete task graph (or DDG) is that it cannot model parallelism arising from commutative operations. Cholesky factorization is one typical example. In [FY95] we have proposed an approach that uses a dependence-incomplete task graph to model the Cholesky factorization before graph scheduling. After the scheduling algorithm exploits the commutativity of the operations, we transform the scheduled Cholesky graph into a dependence-complete graph and then use our run-time system to execute it.

3 Issues in run-time support for executing task graphs

Techniques involved for executing a task graph schedule could be very simple. Essentially, each processor follows the task execution order, issues a receiving operation for each message a task needs from its predecessors and sends data items produced by this task to its successors. However without a carefully designed communication scheme, the execution efficiency could be very poor. We discuss the major performance issues as follows.

- In order to support asynchronous communication, the message passing system such as Intel NX/2 uses the system buffer space to manage incoming and outgoing messages. Such a mechanism imposes a significant amount of overheads on both source and destination processors. We discuss several types of overhead as follows.
  - At the sending site, a message has to be packed and copied from the user space to the system buffer before sending it out to the interconnection network. The system buffer can not be reused before the content has been completely delivered to the network. In NX/2, isend() does not require the pre-copying of data, but the involved user data space can not be modified until the message is actually sent out.
  - At the receiving side, the system has to dynamically allocate buffer space for incoming messages, and the program has to pick up the correct data items from the system buffer because the arriving order of messages could be totally different from the receiving order. Usually this is done by assigning each message a distinct identification number. Additional cost involves message unpacking, data copying from the system buffer to the user space, and buffer space management.
- Secondly, effort is required to maintain data dependencies among tasks. Each task has to receive the correct copies of the desired data items. Figure 2 shows a situation in which a task could receive a wrong copy of the desired data item. Task T_w should receive the copy of data m from task T_z, not the copy of m from task T_x. There should be a way to distinguish different copies of the same data item. Notice that the design of the task communication protocol affects the method that assures the correctness of execution.

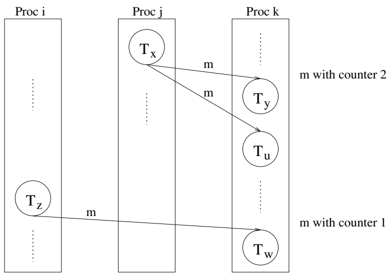

Figure 2: A scenario where data inconsistency could happen.

- Thirdly, a task may send the same message to several successors and some of these successors could be assigned to the same processor. In this case, it is enough to send this message once to the processor on which those successors reside. Optimizations have to be performed to eliminate redundant messages. Correspondingly, at the destination only one receive operation is needed to pull out data from the network. Subsequent receives for this data item should be re-directed to read from the local memory instead.

PYRROS [YG92] uses a buffered message-passing mechanism and aggregates messages as much as possible. The overhead is tolerable with coarse grain computation, but the lesson learned from the PYRROS project is that these techniques are not sufficient to make sending/receiving overhead low enough to efficiently execute task schedules arising from sparse matrix computation with mixed granularities. Active messages [vECGS92] have been used in [CSBS95] for executing fine-grained triangular solving DAGs. We find that active messages offer more functionality than what is needed to support the DAG schedule execution, thus with the disadvantage of imposing extra overhead. Additionally, a careful network polling is required for efficiently implementing active messages as demonstrated in [CSBS95]. In the situation of task parallelism with mixed granularities, it is not easy to decide where to insert the polling code [BCL+95]. The implementation in [CSBS95] takes advantage of commutativity in triangular solving and as soon as one data item arrives at the destination, an arithmetic operation can be performed. A presence counter is used to check if the computation for a task is completed.

4 Design of a run-time supporting system

We show how we can address all the aforementioned issues in a single run-time support framework. The key ideas of our approach are to integrate the low level communication primitives with carefully designed data space management strategies and ensure the correctness by using the properties of dependence-complete task graphs.

4.1 Remote memory access

Since the existence of system buffers cause significant amount of overhead, in order to achieve low latency communication, our approach is to use Remote Memory Access(RMA) as our basic message passing method. RMA can be implemented in modern multi-processor architectures such as Cray-T3D and Meiko CS-2 [SS95]. With RMA, a processor can write to memory of any other processor if a remote address is given. RMA allows passing data directly from one source location to a destination location, without any copying, packing/unpacking and buffering. Thus user data space and message buffers are integrated together, and there is no separate system buffer.

The model of RMA is defined as follows. When a program in a processor needs to write a data item to the remote processor, the user only needs to provide a remote service descriptor which specifies the source/destination addresses, length of data to be transferred, and some other control information. This can be done with a few instructions. After dispatching the RMA descriptor, data will be transferred asynchronously according to the descriptor. A restriction is that the data transferred through RMA cannot be rewritten until the RMA completes, otherwise the remote data copy will become inconsistent. We need to make sure such inconsistency does not happen in the asynchronous execution of tasks.

The RMA usually imposes low overhead on both source and destination processors. We have implemented our system on the Meiko CS-2 which provides Direct Memory Access(DMA) as the major way to access non-local memory. Each node of the Meiko CS-2 is also equipped with a DMA co-processor for handling communication. It takes the main processor 9 microseconds [SS95] to dispatch a remote memory access descriptor to the co-processor. The co-processor afterwards will be responsible for sending data without interrupting the main processor so that the opportunity of overlapping communication and computation is maximized.

4.2 Maintaining data items and addresses

In order for each processor to initialize data items, we assign an owner processor to each data item. There could be several choices for the owner of each data item: the processor that first modifies the data item; the processor that uses the data item most frequently; or the last processor that modifies the data item. We adopt the last strategy because it is easy to collect the final results. Each processor will use some data items other than those it owns and a static data flow analysis is performed to determine a set of data items each processor will use.

The RMA requires remote addresses for each processor. If a processor wants to receive a data item from other processors, it has to let all the senders know its address of that data item in advance. For this purpose, on each processor we maintain a hash table which contains addresses of all data items that this processor needs. The memory space is allocated in advance for all these data items. The goal of this paper is to evaluate how efficient a run-time system can be and the memory constraints are not considered in this paper.

4.3 Sending data items

After a task completes its computation, it needs to send the defined or updated data items to its successors. If two successors which receive the same message are in the same processor, then this processor only needs to receive one copy of the message. The sending procedure at a processor P_x is shown in Figure 3(a). When a task is trying to send a data item m to processor P_y, it obtains the remote address of this item m from the local hash table. After that a RMA descriptor is composed and a RMA transfer is initiated. Notice that multiple RMAs may be initiated consecutively without the need of waiting for the completion of the previous RMA transfer. After the processor issues all the RMAs required for the current task, it can continue to execute the next task. Since the startup overhead of initiating a RMA is relatively small, computation can be overlapped with communication to a maximum degree. When this data item arrives at the destination processor, a flag in the destination processor associated with that data item will be set automatically to indicate its arrival.

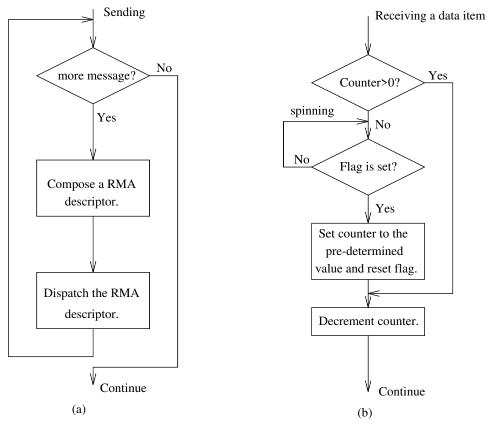

Figure 3: Flow chart for data sending and receiving.

4.4 Receiving data items and preserving data dependencies

Data consistency is the major concern when a task receives data items. Two points must be considered.

- Should a data item be retrieved from the local memory or from the remote processor?
- How can a processor find out if a data item from the remote processor has arrived at the local memory.

In our scheme, each data item at processor p_x is associated with a usage counter and a flag. The "usage counter" field of a data item represents the number of tasks at processor p_x that use this copy of the data item. And the "flag" field indicates if the data item has arrived at the local memory. The counter information for each copy of a data item at processor p_x can be determined during the static data flow analysis. We call a data item dead if the corresponding usage counter is zero.

When task T_y at processor p_x needs to read data item m, it first checks the usage counter of this item. If the counter is greater than zero, the task decrements the counter. If the data item is dead, the task has to wait until the associated flag is set, which indicates that a new copy of this data item arrives. Then the task will clear the flag and set the new counter to be the number of times that this copy will be used on this processor. The receiving procedure is illustrated in Figure 3(b). The values of the counter and the arrival flag represent the state of a data item.

For example, in Figure 2, when task T_y first receives data m from processor j, it will set the usage counter of m to 2 because another task T_u in processor k also needs it. Then T_y will decrement the counter. After T_u uses it, the counter will be decremented to zero. Thus when task T_w tries to receive m, it will find the counter to be zero. Then it has to wait for the arrival of a new copy of data m which is from task T_z in processor i. At this time the usage counter is 1.

4.5 Correctness analysis

In this section, we address the correctness of our asynchronous execution scheme. Two data inconsistencies could arise in the following situations:

- **Data inconsistency caused at receiving sites**.
  Before a data item m on processor p is dead, a new version of data m could come in and overwrite it. Thus the subsequent reading of data m will not get the correct copy.
- **Data inconsistency caused at sending sites**.
  Since the initiation of a RMA request for a data item m does not guarantee that data m is actually sent out, another task, either at the same processor or at the different processor, might overwrite m before the RMA data transfer is completed. Then the destination processor will not have the correct copy it needs.

Fortunately, in the following theorems, we can show that if a task graph is dependence-complete, the above situations will never happen. The proofs can be found in [FY95].

Theorem 1 No receiver site inconsistency. *For any dependence-complete graph, if a task T_y defines m and sends it to task T_x, then before T_x reads it, no other task will overwrite it.*

Theorem 2 No sender site inconsistency. *For any dependence complete task graph, if there is a task T_x on processor p_x that receives data item m from task T_w on processor p_w, then before the data item is actually sent out, no other task will redefine this data item on processor p_w, i.e., processor p_x will receive the the correct copy of data item m.*

5 Experiment results

We will examine the performance of our system on two irregular applications: sparse LU and Cholesky factorizations. We have developed a set of library functions allowing users to specify data access patterns in a sequential program. Then the system derives a dependence graph and uses the PYRROS scheduling algorithms to produce a task schedule which will be executed by the run-time system. We will analyze the overall performance and itemized cost distribution of the run-time execution.

All our experiments are conducted in the distributed memory machine Meiko CS-2. The single-node LINPACK performance we have tested is 8.04MFLOPS for double-precision floating operations[1]. The communication peak bandwidth of Meiko CS-2 is 40MBytes/Sec. The effective bandwidth has been reported as 39MB/Sec and this is obtained through a ping-pong test [SS95]. In practice, a message is sent only once at a time. We have measured the effective bandwidth of a single-message sending or direct memory access. When the sizes of messages vary from $1K$ to $10K$, the bandwidth is about $10 - 25$MBytes/Sec. Since our experiments use matrices as messages, for messages smaller than $1K$ we list the bandwidths by matrix sizes in Table 1.

5.1 Sparse LU factorization without pivoting

We first give the definition of the LU factorization without pivoting and demonstrate the irregularity of sparse LU graphs. LU without pivoting can be used for positive definite or strictly diagonally dominant matrices. Many of algebraic systems arising from the discretization of diffusion and convection problems in fluid dynamics[PT83] are nonsymmetric and diagonally dominant. The boundary conditions usually make the first row of matrices strictly diagonally dominant. For such matrices, it can be shown [Cai95] that stable solutions can be obtained by LU without pivoting. The proof in [Cai95] is based on a modification to Theorem 6.19 in [BF93]. It should be noted that parallel sparse LU with global pivoting is more complicated.

For a dense matrix, the BLAS-3 level LU factorization without pivoting is shown in Figure 4. The $n \times n$ matrix is divided into $N \times N$ submatrices and each submatrix is of size $r \times r$ where $N = n/r$. The task graph operates on those N^2 submatrices.

Given a sparse matrix, we first use the multiple minimum degree algorithm to decide an ordering and then calculate fill-ins. After that we use supernode partitioning [HNP91] so that every basic task operation involves only dense matrix or vector operations, which can be implemented using BLAS and LAPACK routines. We also partition supernodes into smaller blocks. And for those supernodes smaller than the block size used, they will remain unchanged. In this way we can take advantage of caching and reduce indexing overhead. Figure 5 depicts an unstructured LU task graph from a 21×21 sparse matrix. It can be shown that a LU graph is dependence complete.

We have tested two benchmark matrices from Boeing-Harwell collection: BCSSTK15 and BCSSTK24. Both matrices have symmetric structures. The purpose of using these matrices is to compare with the performance of Cholesky factorization and analyze the impact of algorithm characteristics on the code performance. We summarize some statistics of these two matrices in Table 2. Block size 25 is used to partition large supernodes.

[1] All codes mentioned in this paper are compiled with options -O4 -fast -Bstatic.

Matrix Size	1x1	2x2	3x3	4x4	5x5	6x6	7x7	8x8	9x9	10x10	11x11
Bandwidth(MB/S)	0.09	0.38	0.83	1.50	2.28	3.22	4.14	5.52	6.55	8.10	9.73

Table 1: Communication bandwidth for small matrices on Meiko CS-2.

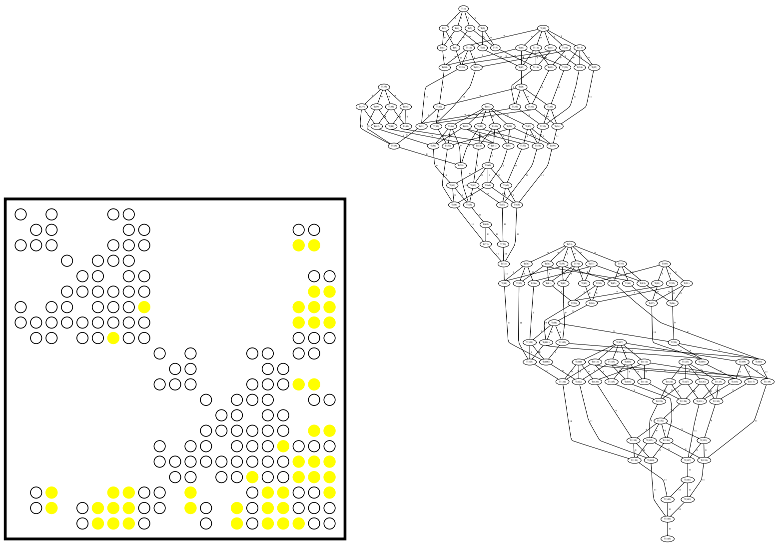

Figure 5: A sparse matrix with fill-ins and the sparse LU DAG derived from it.

for $k = 1$ **to** N
  F_{kk}: $\boxed{\text{Factorize } A_{k,k} \text{ as } L_k * U_k}$
  for $i = k+1$ **to** N
    S_{ik} : $\boxed{A_{i,k} = A_{i,k} * U_k^{-1}}$
    S_{ki} : $\boxed{A_{k,i} = L_k^{-1} * A_{k,i}}$
  end
  for $j = k+1$ **to** N
    for $i = k+1$ **to** N
      M_{ij}^k : $\boxed{A_{i,j} = A_{i,j} - A_{i,k} * A_{k,j}}$
    end
  end
end

Figure 4: The block LU factorization algorithm for a dense matrix.

We are in the process of getting more test data including matrices with nonsymmetric structure in which our sparse LU algorithm can be tested.

We have implemented a sequential version of the sparse LU factorization to assess speedups. The sequential performance on a single Meiko CS-2 node is also listed in Table 2. The sparse matrix performance on sequential machines is usually poor because of indexing of irregular data items. Our sparse LU implementation has a performance close to 85% of LINPACK performance for dense matrices. We think that is the best sequential implementation we can have. All parallel speedups are compared directly to this sequential performance.

Matrix	#tasks	#edges	Floating ops	MFLOPS
B15	123,075	321,907	312,886,996	6.77
B24	15,761	37,776	64,032,308	6.84

Table 2: Statistics of B15 and B24 for Sparse LU.

We use the PYRROS mapping algorithm to generate the schedule for LU graphs and use the run-time system discussed in this paper to execute this schedule. Speedups are shown in Figure 6. On 32 processors, we have obtained 12.9 speedup for B24, and 9.7 for B15. As far as we know, this is the best performance achieved by automatically scheduled code. We will discuss the overhead of the run-time execution in Section 5.3.

5.2 Sparse Cholesky factorization

Sequential and parallel sparse Cholesky factorization algorithms have been studied extensively in [Rot92, RS94]. They show that the supernode-based approach can deliver good performance in both shared and distributed memory machines. We will examine the performance of general task scheduling and executing techniques applied to this problem. An important property

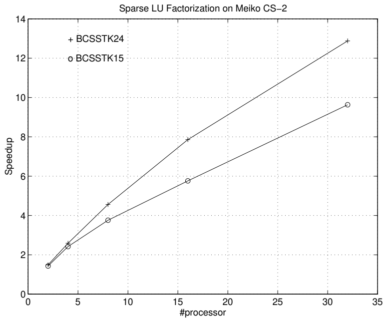

Figure 6: Performances of sparse LU for B24 and B15.

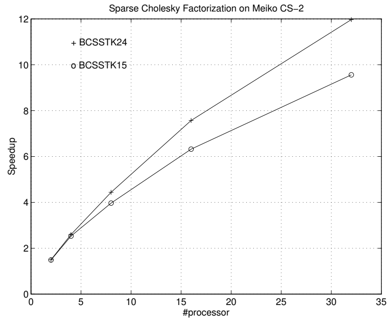

Figure 7: Performances of sparse Cholesky for B24 and B15.

of sparse Cholesky factorization is that all the operations for updating the same block commute. Parallelism from this property is called fan-in parallelism in the literature [HNP91]. Fan-out parallelism in Cholesky factorization is naturally included in the task graph model as submatrix multicasting edges. In [FY95] we have given a method that exploits parallelism from commuting operations so that better parallel performance can be achieved.

Matrix	#tasks	#edges	Floating ops	MFLOPS
B15	67,621	177,179	157,398,044	5.89
B24	9,124	19,911	32,432,756	5.94

Table 3: Statistics of B15 and B24 for Sparse Cholesky.

The computation of the block Cholesky factorization is similar to that of the block LU factorization. Table 3 shows the statistics and the sequential performance of the Cholesky DAGs. The performance of Sparse Cholesky factorization without exploring the commutativity is shown in Figure 7. The speedups on 32 processors are 9.6 for B15 and 12 for B24. Therefore, processor utilization efficiency are 30% and 37.5% respectively. In [RS94], efficiency of 25% has been reported for B15 on 64 processors(Intel Paragon) after improving the load balancing strategy, which indicates parallelism in these problems is limited. Our code produced by automatic scheduling has achieved acceptable performances.

5.3 Run-time overhead distribution

We classify the time spent at each processor into four parts: computation (CPU time used for executing tasks), sending overhead, receiving overhead, and waiting (or spinning) time which is the idle time that the processor spends on waiting for the arrival of the desired data item. If the desired data item arrives before a task issues the receiving primitive, the waiting time will be zero. This is an ideal situation in which communication is totally overlapped with computation. We calculate the ratio of sending, receiving and spinning time over the computation time at each processor. Figure 8 shows the overhead distribution in LU factorization. The Y axis lists the average ratio of overhead type X (X is sending, receiving or spinning) over the computation time.

It can be seen that when the number of processors increases, the sending/receiving overhead remains relatively small, about $10 - 18\%$ of the computation time. Thus the total run-time sending/receiving overhead contributes about $6 - 10\%$ of the total execution time at each processor.

block size	1	2	3	4	5	6	>6
B15	78%	6%	3%	1%	2%	2%	8%
B24	5%	5%	3%	0%	1%	60%	26%

Table 4: Distributions of the supernode sizes.

The major loss of the performance goes to the spinning(or waiting) time. This is partly because of the limited parallelism in these problems. Another important reason is that the predetermined schedule does not match actual run-time situation very well because task computation weights vary at run-time. The performance becomes even more sensitive when task granularities are small [GJY95]. If we look at the distribution of the sizes of supernodes (after blocking) as shown in Table 4, we can see that for both matrices, there are a lot of blocks with sizes less than 6×6. For BCSSTK15, 78% of the blocks are of size 1×1, which makes the corresponding tasks just a few floating point operations. In order to improve the performance further, our future work is to consider techniques of increasing supernode sizes, e.g., supernode amalgamation [Rot92]. Dynamic scheduling will be more adaptive to run-time weight variation, but it also increases run-time overhead significantly.

6 Conclusions

In this paper, we have presented an efficient run-time scheme for supporting task computations and shown its applications in irregular sparse LU and Cholesky factorization. The experiments indicate that overhead of run-time data communication is small compared to the total execution time ($6 - 10\%$). The performance of sparse computation is limited by available parallelism, but not the overhead of the system. Our current work is to develop a

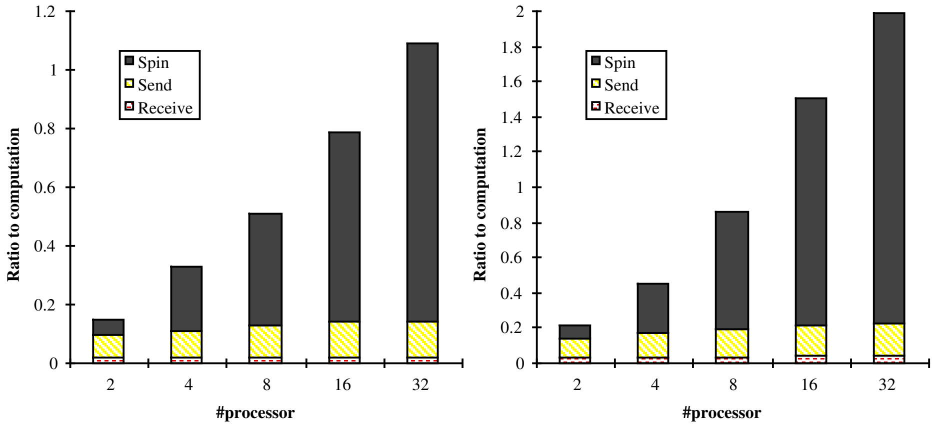

Figure 8: Distributions of the communication overheads for sparse LU. The left part is for B24 and the right is for B15.

complete run-time system called RAPID which provides a programming interface for users to parallelize their irregular codes using our techniques. We will also investigate the applications of our techniques in a wider range of irregular problems.

Acknowledgement

This work is supported by NSF CCR-9409695 and a startup fund from UCSB. We thank Ed Rothberg and Rob Schreiber for their insightful comments on this work; Dan Andresen, Martin Rinard, Klaus Schauser, Chris Scheiman, and the anonymous referees for useful comments and helps.

References

[BCL+95] E. A. Brewer, F. T. Chong, L. T. Liu, S. D. Sharma, and J. Kubiatowicz. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 42–53, 1995.

[BF93] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. PWS, fifth edition, 1993.

[Cai95] W. Cai. Personal Communication, 1995.

[CF87] R. Cytron and J. Ferrante. What's in a name? The Value of Renaming for Parallelism Detection and Storage Allocation. In *Proc. of International Conf. on Parallel Processing*, pages 19–27, February 1987.

[CR92] Y-C. Chung and S. Ranka. Applications and Performance Analysis of a Compile-time Optimization Approach for List Scheduling Algorithms on Distributed Memory Multiprocessor. In *Proc. of Supercomputing '92*, 1992.

[CRY94] S. Chakrabarti, A. Ranade, and K. Yelick. Randomized Load Balancing for Tree-structured Computation. In *Proc. of Scalable High-Performance Computing Conference*, pages 666–673, February 1994.

[CSBS95] F. T. Chong, S. D. Sharma, E. A. Brewer, and J. Saltz. Multiprocessor Runtime Support for Fine-Grained Irregular DAGs. In *Toward Teraflop Computing and New Grand Challenge Applications.*, New York, 1995. Nova Science Publishers.

[DUSH94] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures . *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[FY95] C. Fu and T. Yang. Run-time Techniques for Exploiting Irregular Task Parallelismon Distributed Memory Architectures. Technical Report TRCS95-21, Dept. of Computer Science, UCSB, 1995. http://www.cs.ucsb.edu/Research/rapid_sweb/RAPID.html.

[GJY95] A. Gerasoulis, J. Jiao, and T. Yang. Scheduling of Structured and Unstructured Computation . In Dominique Sotteau D. Frank Hsu, Arnold Rosenberg, editor, *Interconnections Networks and Mappings and Scheduling Parallel Computation*. American Math. Society, 1995.

[GY93] A. Gerasoulis and T. Yang. On the Granularity and Clustering of Directed Acyclic Task Graphs . *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, June 1993.

[HNP91] M. Heath, E. Ng, and B. Peyton. Parallel Algorithms for Sparse Linear Systems . *SIAM Review*, 33(3):420–460, September 1991.

[Kar91] N. Karmarkar. A New Parallel Architecture for Sparse Matrix Computation Based on Finite Project Geometries . In *Proc. of Supercomputing '91*, pages 358–369, February 1991.

[KB88] S.J. Kim and J.C. Browne. A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures, . In *Proc. of Inter. Conf. on Parallel Processing*, pages 318–328, 1988.

[LB94] A. Lain and P. Banerjee. Techniques to Overlap Computation and Communication in Irregular Iterative Applications. In *Proc. of ACM Inter. Conf. on Supercomputing*, pages 236–245, July 1994.

[Pol88] C. D. Polychronopoulos. *Parallel Programming and Compilers* . Kluwer Academic Publishers, 1988.

[PT83] R. Peyret and T. D. Taylor. *Computational Methods for Fluid Flow* . Springer-Verlag, 1983.

[Rot92] E. Rothberg. *Exploiting the Memory Hierarchy in Sequential and Parallel Sparse Cholesky Factorization* . PhD thesis, Stanford University, December 1992.

[RS94] E. Rothberg and R. Schreiber. Improved Load Distribution in Parallel Sparse Cholesky Factorization. In *Proc. of Supercomputing'94*, pages 783–792, 1994.

[Sar89] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors* . MIT Press, 1989.

[Sch93] R. Schreiber. *Scalability of Sparse Direct Solvers*, volume 56 of *Graph Theory and Sparse Matrix Computation*, pages 191–209. Springer-Verlag, New York, 1993.

[SS95] K. E. Schauser and C. J. Scheiman. Experience with Active Messages on the Meiko CS-2. In *Proc. of 9th International Parallel Processing Symposium*, pages 140–149, 1995.

[vECGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[VNS92] S. Venugopal, V. Naik, and J. Saltz. Performance of Distributed Sparse Cholesky Factorization with Pre-scheduling. In *Proc. of Supercomputing'92*, pages 52–61, Minneapolis, November 1992.

[YFGS95] T. Yang, C. Fu, A. Gerasoulis, and V. Sarkar. Mapping Iterative Task Graphs on Distributed-memory Machines . In *Proc. of 24th Inter. Conf. on Parallel Processing*, pages 151–158, August 1995.

[YG92] T. Yang and A. Gerasoulis. PYRROS: Static Task Scheduling and Code Generation for Message-Passing Multiprocessors . In *Proc. of 6th ACM Inter. Confer. on Supercomputing*, pages 428–437, 1992.