Run-time Techniques for Exploiting Irregular Task Parallelism on Distributed Memory Architectures

Cong Fu and Tao Yang
Department of Computer Science
University of California
Santa Barbara, CA 93106
{cfu,tyang}@cs.ucsb.edu

Abstract

Automatic scheduling for directed acyclic graphs (DAG) and its applications for coarse-grained irregular problems such as large n-body simulation have been studied in the literature. However solving irregular problems with mixed granularities such as sparse matrix factorization is challenging since it requires efficient run-time support to execute a DAG schedule. In this paper, we investigate run-time optimization techniques for executing general asynchronous DAG schedules on distributed memory machines and discuss an approach for exploiting parallelism from commuting operations in the DAG model. Our solution tightly integrates the run-time scheme with a fast communication mechanism to eliminate unnecessary overhead in message buffering and copying. We present a consistency model incorporating the above optimizations, and taking advantage of task dependence properties to ensure the correctness of execution. We demonstrate the applications of this scheme in sparse matrix factorizations and triangular equation solving for which actual speedups are hard to obtain. We provide a detailed experimental study on Meiko CS-2 to show that the automatically scheduled code has achieved good performance for these difficult problems and the run-time overhead is small compared to total execution times.

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 [12, 16, 27]. For iterative irregular problems in which communication and computation phases alternate, the CHAOS system [8] has used the inspector/executor approach to exploit irregular parallelism at each iteration of the computation phase. The problems addressed in CHAOS have no loop-carried dependency at the computation phase and processors can run independently before next communication phase. In a more complicated iterative application, the computation phase 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) [12]. The DAG schedule can be re-used for a number of iterations before re-scheduling since particle movement is slow. In solving nonlinear equations using iterative methods, sparse matrix factorization usually dominates the computation time at each iteration step and the topology of the iteration matrix remains the same but the numerical values change at each step [15]. 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 [23]. Compared to large n-body simulations, efficient execution of DAG schedules for sparse matrix problems is more challenging because partitioned graphs contain both coarse and fine grained tasks.

Most of previous work in DAG scheduling has mainly focused on the algorithmic research for task mapping, e.g. [21, 28] and little research has been conducted on efficient run-time support for executing task schedules. The early work in the PYRROS system [29, 30] provides a complete framework for general task computation; however, its run-time support system uses the NX/2 level communication interface and overhead for message management is high, which prevents PYRROS from obtaining good performance for sparse matrix problems.

We have developed an efficient run-time support system called RAPID 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 experiments indicate that the overhead is only $6-10\%$ of the execution time for the tested applications. 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 and consistency model, and ensure the correctness of execution by taking advantage of properties of task graphs.

We demonstrate the applications of our techniques for sparse matrix computations. It has been very difficult to obtain actual speedups by hand-made code for these problems, not to mention automatically parallelized code, especially on distributed memory machines. Recently impressive performance is obtained for sparse Cholesky factorization [19, 20] and sparse triangular solver [6]. We discuss how we can exploit task parallelism in sparse Cholesky factorization and LU without pivoting using our techniques in a general run-time support environment. One difficulty in parallelizing sparse Cholesky factorization is that parallelism also arises from operations that commute. We discuss how to address this issue in our task computation model and present experiments to show that our approach delivers decent performance. As far as we know, this is the best performance achieved by automatically scheduled codes. To demonstrate the applicability of our approach for other sparse matrix problems, we also examine the performance of our run-time techniques for sparse lower triangular solvers and study the impact of granularities on the overall 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 details. Section 5 gives the correctness analysis of our method. Section 6 presents an experimental study on several sparse matrix computations and provides an analysis on overhead and granularity issues. Section 7 summarizes the related work. Section 8 concludes the paper.

2 The task computation model

2.1 Task dependence graphs

Our work is targeted at message passing distributed memory architectures. We use the directed acyclic task graphs (DAGs) as the computation model. We need to study the properties of task dependence so that we can take advantage of those properties in designing the run-time task execution model. Classical data dependence graphs (DDG) for modeling the interaction between computation statements have three types of dependencies: true, anti and output. Task graphs differ from DDG in that they only have true data dependence edges. In a DDG, many of anti or output dependence edges could be redundant if they are subsumed by other true data dependence edges. With the presence of anti and output dependence, synchronization in run-time support becomes more complicated in order to ensure the correctness of execution. In functional languages, the "single assignment" principle has been used in data flow computation and functional languages [17, 21], i.e., each data item can be written at most once, thus there is neither output nor anti dependence. The renaming techniques [7] can eliminate the output and anti dependence in transforming a DDG to a "single assignment" graph. The advantage of such a principle is that dependence enforcement in task execution can be simplified to some 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 the 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 renaming of data items is necessary to remove the remaining anti and output dependence.

Informally we call a task graph *dependence-complete* for a given program if all anti and output dependence relations among tasks of this program are subsumed by edges or paths in this task graph and there exists a sequential program from which this task graph can be derived. Figure 1 depicts a DDG from which a dependence-compete task graph can be derived by deleting all anti and output dependence edges.

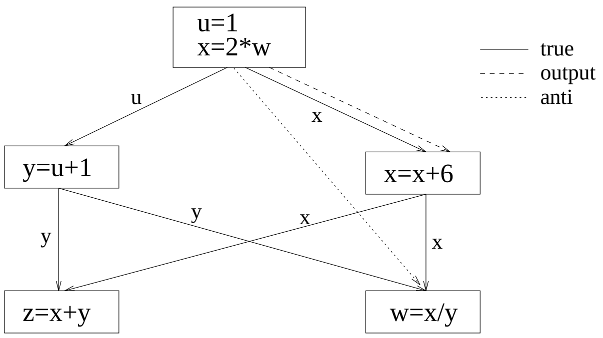


Figure 1: An example of a DDG from which a dependence-complete task graph can be derived.

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, 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 have the same semantics of a sequential program.

These properties $DA1-4$ impose natural constraints that a task graph should have in representing task computation. These constraints provide a basis for us to design highly efficient execution scheme, thus to exploit parallelism residing in the graph as much as possible. This will become clearer as we discuss in the subsequent sections. More discussion on the above properties and their relationship can be found in [29]. The transformation to convert a DDG to a dependence complete task graph is described in [9]. Most task graphs derived from real scientific applications satisfy these properties without renaming. For example, sparse LU graphs discussed in Section 6.1.2 are dependence-complete.

2.2 Handling commutativity

One disadvantage of a dependence-complete task graph (or DDG) is that it cannot model parallelism arising from commuting operations. For example, in sparse Cholesky factorization all the updating tasks for the same block commute. And this is a potential source of parallelism [1].

We define our commutativity model as follows. For a given data item m, there are a set of commuting tasks CT_m^1, CT_m^2, $\cdots$, CT_m^n in modifying data item m, and a reduction task RT_m uses the result of these n tasks. This set of commuting tasks can be executed in an arbitrary order. Therefore, depending on whether we take this characteristic into consideration or not, we can have two different versions of task graphs. One is based on pure sequential semantics, and the other is based on the commutativity. Figure 2(a) shows a dependency chain derived from a sequential code, and Figure 2(b) shows the dependency structure reflecting the commutativity. We call this type of task graphs *commutative* task graphs. Apparently scheduling a commutative task graph would gain more performance because of more flexibility in scheduling.

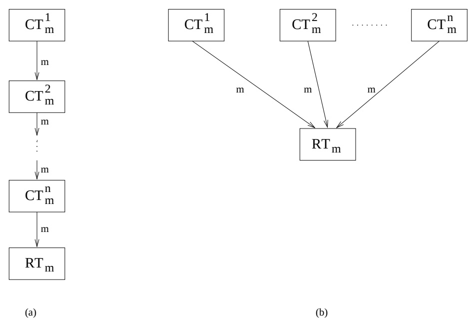


Figure 2: (a) The dependence structure based on the sequential semantics. (b) The dependence structure that exposes commutative operations.

However while the task graph derived from sequential semantics satisfies DA1-4, the task graph considering commutativity *is not* dependence-complete since all commuting tasks (e.g., in Figure 2(b)) modify the same data item which violates assumptions DA1 and DA3. In order to correctly execute a commutative task graph, we have to alter the task graph after the scheduling. This is done by renaming data items and changing edges as depicted in Figure 3. We explain the procedure as follows. For each reduction task

[1]This is called fan-in parallelism in the literature [14]. Fan-out parallelism in Cholesky factorization is naturally included in the task graph model as submatrix multi-casting edges.

RT_m in a commutative task graph, we apply the following transformations for its commuting predecessors CT_m^i's in each processor P_x. 1) We delete all edges from these commuting tasks CT_m^i's to the reduction task RT_m and create a new data item m_x for these commuting tasks in processor P_x; 2) Add a chain of dependence edges between these commuting tasks following the ordering at P_x in the DAG schedule and label each edge using the newly-created data item. 3) Add a dependence edge from the last predecessor commuting task on P_x to task RT_m and also label this edge using the data item m_x. It is easy to verify that the transformed graph is dependence-complete.

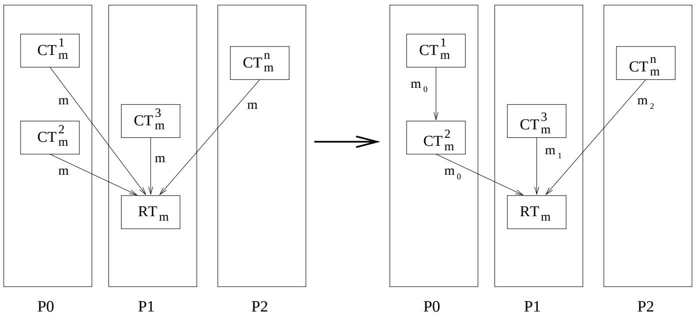


Figure 3: Dependence graph alteration after scheduling.

3 Issues in run-time support of 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.

First of all, sending and receiving overheads should be low enough compared to the amount of computing work each task does. Usually in order to support asynchronous communication, a message passing system such as Intel NX/2 uses system buffer space to manage incoming and outgoing messages. It is well known that message-buffering imposes a significant amount of overheads for copying and space management on source and destination processors [26].

Secondly, effort is required to maintain data dependencies among tasks. Each task has to receive the correct copies of the desired data items. Figure 4 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.

Thirdly, due to the nonblocking nature of asynchronous communication, returning from a sending operation does not guarantee the message is actually copied or sent out. Thus a proper mechanism is needed to ensure that the message will not be overwritten before it is copied or sent out. For example in Figure 4, a task T_x on processor j sends a data item m to processor k. Another task T_h will update data m as well. It is necessary to make sure that task T_h will not be executed until the data m is sent out.

Finally, 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

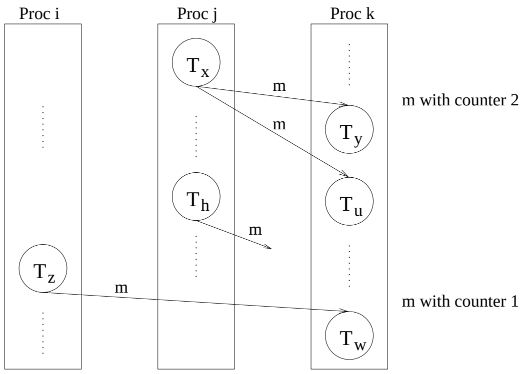


Figure 4: A scenario where data inconsistencies could happen.

which those successors reside. Optimizations have to be performed to eliminate redundant messages. Correspondingly, in the destination only one receiving is needed to pull out data from the network. Subsequent receiving operations for this data item should be re-directed to read from the local memory instead.

PYRROS [30] uses a buffered message-passing mechanism and aggregates communication as much as possible. The overhead is tolerable with medium 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. In the next section, we will show how we can address all these issues in a single run-time support framework by carefully designing the message-passing layer, and the message/data management strategy.

4 Design of a run-time supporting system

The key ideas of our approach are to integrate the low level communication primitives with carefully designed data space management and ensure the consistency by using the properties of dependence-complete task graphs. We discuss aspects of this system in details as follows.

4.1 Remote memory access

Since the existence of system buffers causes a significant amount of overheads, in order to achieve low latency communication, our approach uses Remote Memory Access(RMA) as our basic message passing method. RMA can be implemented in modern multi-processor architectures such as Cray-T3D (SHMEM) and Meiko CS-2 (DMA). 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. The model of RMA is defined as follows. When a program in a processor needs to write a data item to another processor, it 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, the data will be transferred asynchronously according to the description. A restriction is that the data transferred through a RMA cannot be rewritten until the RMA completes, otherwise the remote copy of data will become inconsistent. We need to make sure such inconsistency does not happen in the asynchronous execution of tasks.

RMA imposes low overhead on both source and destination processors. We have implemented our system on Meiko CS-2 which provides Direct Memory Access(DMA) as the major way to access non-local memory. Each node of Meiko CS-2 is also equipped with a DMA co-processor for handling communication. It takes the main processor 9 microseconds [22] to dispatch the 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. It should be noted that the Meiko does not provide DMA support for multi-casting a data item. Thus we have to implement the multi-casting by repeatedly using the DMA feature. For irregular computation where broadcasting happens less frequently than multi-casting, such design is reasonable.

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.

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. And the memory space will be 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 constraint is not considered.

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 or more 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 is shown in Figure 5(a). When a task tries to send a data item m to another processor, 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 can be initiated consecutively without waiting for the completion of previous RMA transfers. After the processor issues all the RMAs needed 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 associated with that data item will be set to indicate its arrival.

4.4 Receiving data items and preserving data dependencies

Data consistency is the major concern when a task receives data items. Two issues need to be addressed. Firstly, how to decide whether a data item should be retrieved from the local memory or from the remote

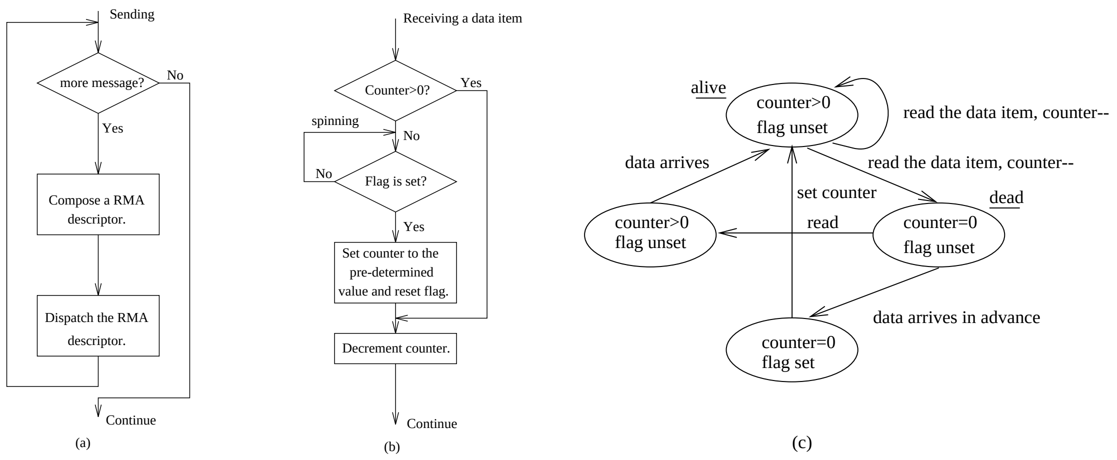


Figure 5: (a) Flow chart for data sending and receiving; (b) Flow chart for data receiving; (c) The state transition of a data item during the execution of a task graph.

processor? Secondly, 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 "counter" field of a data item represents the number of tasks at processor p_x that use this copy of the data item. For the "flag" field, it indicates the data item has arrived at the local memory if it is set. The counter information for each copy of a data item at processor p_x can be determined during a 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 data item. If the counter is greater than zero, the task decrements the counter and proceeds. 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. The task will then 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 5(b). The values of the counter and the arrival flag represent the state of a data item. Figure 5(c) shows how states evolve during the execution of a task graph. The edges are actions taken to transit from one state to another.

For example, in Figure 4, 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. This time the usage counter is 1.

We also explain the reason of using two separate fields "counter" and "flag" as follows. Suppose they are combined as one, then when data item m is sent from processor p_x to p_z via RMA, the counter must be set to a proper value before RMA starts and it cannot be changed until RMA completes. Since multi-casting of the same data item happens frequently in applications (e.g., in sparse Cholesky and LU factorizations) and the usage counters of this data item may differ for different destination processors. Hence sending data item to one processor cannot be overlapped with sending the same copy of the data item to another processor.

5 Consistency analysis

In this section, we address the correctness of our asynchronous execution scheme. Two data inconsistencies could arise in the following situations: **1. Data inconsistency caused at receiving sites**. Before a data item m on processor p is dead, a new copy of data m could come in and overwrite it. Thus the subsequent reading of data m will not get the correct copy. **2. 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 in our execution scheme.

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

Proof: Suppose there is another task T_z which sends a new copy of data m to the processor of T_x before T_x reads data m. There must be another task T_w that receives data m from T_z. If T_x is the same as T_w, then task T_x would receive the same data from T_y and T_z, which is impossible according to DA1. Thus T_x is different from T_w, and T_y and T_z finish their execution before T_x and T_w.

According to DA2, there are a dependence path from T_y to T_w and a path from T_z to T_x as well (shown in Figure 6(a)). And DA3 implies that there must be a path between T_y and T_z. If the path is from T_y to T_z, then T_x uses m from T_y and T_z redefines m, which contradicts DA4 and the same is true if the path is from T_z to T_y. ■

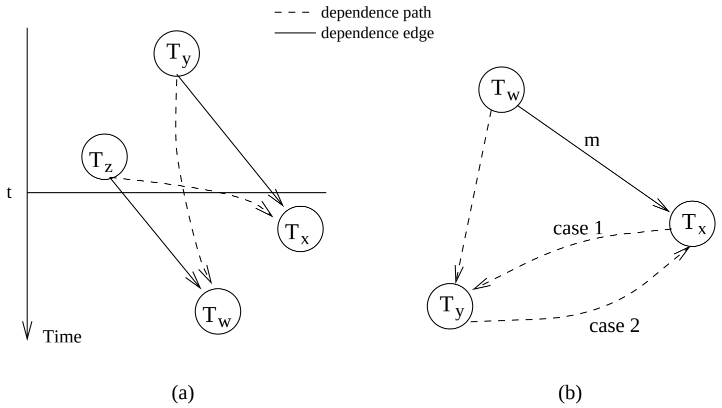


Figure 6: (a) Illustration for Theorem 1; (b) Illustration for Theorem 2.

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 m is actually sent out, no other task will redefine this item on processor p_w, i.e., processor p_x will receive the the correct copy of data item m.*

Proof: Suppose there is another task T_y that will redefine data item m after task T_w, then according to

DA2 and DA3, there must be a dependence path between T_w and T_y, and a path between T_x and T_y as well. As it is shown in Figure 6(b), we have two cases.

Case1: Suppose there is a dependence path from T_x to T_y, then task T_y will not start executing until the task T_x finishes. So task T_x will receive the correct version of data item m;

Case2: Suppose there is a dependence path from T_y to T_x, since T_y will modify the data m, it is not sequentializable, which contradicts the DA4. ■

6 Experimental studies

We will examine the performance of our system on two types of irregular applications: sparse factorization (Cholesky and LU without pivoting) and sparse triangular solvers. We have developed a set of library functions [9] which allow users to specify data access patterns based on a sequential version of the application. We use the PYRROS scheduling algorithms to produce a task schedule and our run-time system will execute such an irregular schedule. We will analyze the overall performance and itemized cost distribution of the run-time execution.

All our experiments are conducted on the distributed memory machine Meiko CS-2. The single-node LINPACK performance we have tested is 8.04MFLOPS for double-precision floating operations[2]. The cost to dispatch a RMA request is 9 microseconds. The communication peak bandwidth of Meiko CS-2 is 40MBytes/Sec. The effective peak bandwidth has been reported as 39MB/Sec, but this is obtained through a ping-pong test [22] and message caching improves the performance implicitly. In practice, a message is sent only once at a time. We find that the effective bandwidth of a single-message sending or direct memory access is in the range of 10–25 MB/Sec when the sizes of messages vary from $1K$ to $10K$. Since our experiments use matrices as messages, for small messages with sizes smaller than $1K$ we list the bandwidths by matrix sizes in Table 1. Each matrix element is a double-precision number, which is 8 bytes long. In this test, we assume the whole message is pre-fetched into the cache. This assumption is justifiable because each task usually sends out the submatrix it just updated. The individual message sizes in our experiments are all less than 6K.

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.

In the rest of this section, we will report our experiments and examine the overall performance of the run-time system for executing task graphs arising from sparse Cholesky factorization, LU without pivoting and sparse triangular solvers. We will study different aspects of system performance such as the benefit of exploiting commutativity of operations and the overhead of run-time support.

[2]All codes mentioned in this paper are compiled with options -O4 -fast -Bstatic. The source code of LINPACK is obtained from the public domain.

6.1 Sparse matrix factorizations and run-time overhead analysis

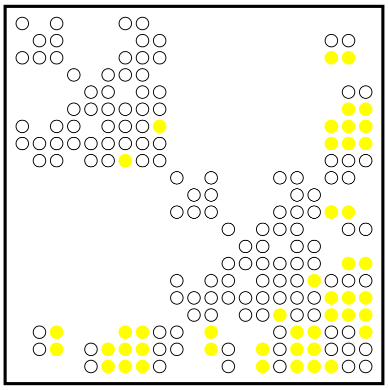
6.1.1 Sparse Cholesky factorization

Sequential and parallel sparse Cholesky factorization algorithms have been studied extensively in literature. Rothberg and Schreiber [19, 20] show that the supernode-based approach can deliver good performance on both shared and distributed memory machines. Specialized scheduling techniques have been studied for sparse Cholesky factorization in [11]. We will examine the performance of applying general task scheduling and executing techniques to this problem.

For a sparse matrix, the BLAS-3 level Cholesky factorization algorithm [3] is shown in Figure 7(a). The $n \times n$ matrix is divided into $N \times N$ submatrices and some submatrices are zero due to the sparsity. The task graph only operates on those nonzero submatrices.

for $k = 1$ **to** N
  F_{kk}: $\boxed{\text{Factorize } A_{kk} \text{ as } L_k * L_k^T}$
  for $i = k+1$ **to** N with $A_{ik} \neq 0$
    S_{ik}: $\boxed{A_{ik} = A_{ik} * (L_k^T)^{-1}}$
  end
  for $j = k+1$ **to** N with $A_{jk} \neq 0$
    for $i = j$ **to** N with $A_{ik} \neq 0$
      M_{ij}^k: $\boxed{A_{ij} = A_{ij} - A_{ik} * A_{jk}^T}$
    end
  end
end

(a)



(b)

Figure 7: (a) The block-oriented sparse Cholesky factorization algorithm; (b) A sparse matrix with fill-ins.

Given a sparse matrix, we first use the multiple minimum degree algorithm to decide the ordering and then calculate fill-ins. For instance, Figure 7(b) is a 21×21 sparse matrix with fill-ins. After that we use supernode partitioning [14] 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 the advantage of caching and reduce indexing overhead. In the literature, tasks F_{kk} and S_{ik} are also called *DIV* operations. And tasks M_{ij}^k are called *MOD* operations. An important property is that all the *MOD* operations with respect to the same block commute. This is a potential source of parallelism as we discussed in Section 2.2. The reduction task here is the *DIV* operation. In Figure 8 we show two different Cholesky task graphs for the example matrix in Figure 7(b) depending on whether we consider commutativity or not. From this small example it can be seen that exploiting operation commutativity does expose more parallelism.

We have tested a set of benchmark matrices from the Harwell-Boeing collection. BCSSTK15 arises from structural engineering analysis for Module of an Offshore Platform, BCSSTK16 is for Corp. of Engineers Dam, BCSSTK17 is for Elevated Pressure Vessel, and BCSSTK24 is for Calgary Olympic Saddledome

[3] This algorithm is presented in a right-looking style. Another presentation is called left-looking. Our task model can capture parallelism arising from both styles. Another algorithm for sparse Cholesky is the multi-frontal method. In this paper we are not considering parallelizing the multi-frontal method.

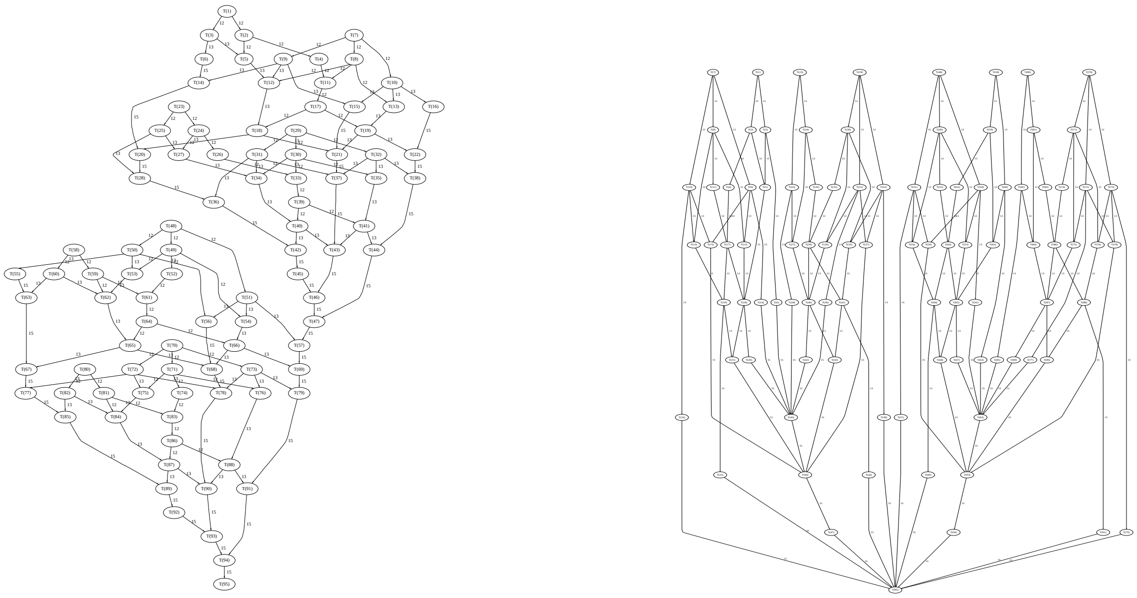


Figure 8: The left is a sparse Cholesky DAG (for the matrix in Figure 7(b)) based on sequential semantics and the right is the DAG for the same problem but it exposes the commutativity of *MOD* operations.

Matrix	#tasks	#edges	#data items	Floating ops	Seq. time(sec.)	MFLOPS
BCSSTK15	67,621	177,179	12,167	157,398,044	26.72	5.89
BCSSTK16	36,639	90,348	6,523	153,527,463	20.81	7.38
BCSSTK17	69,557	173,452	16,252	147,834,028	24.44	6.05
BCSSTK24	9,124	19,911	2,487	32,432,756	5.46	5.94

Table 2: Statistics of benchmark matrices for Sparse Cholesky factorization with maximum block size 25. Note that LINPACK performance is 8.05 MFLOPS.

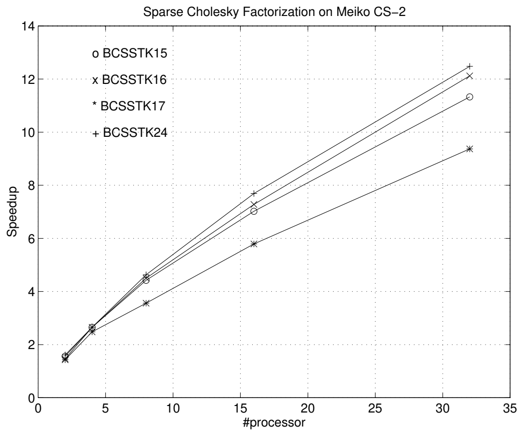
Arena. Task graphs derived from these matrices are quite large and unstructured. We summarize some statistics of those task graphs in Table 2. We have also implemented a sequential version of the sparse Cholesky factorization code to assess the speedups. Its performance on a single Meiko CS-2 node is also listed in Table 2. The sparse matrix performance on sequential machines is usually poor because of the indexing of irregular data items. Our sparse Cholesky implementation has achieved $73\% - 92\%$ of the dense matrix LINPACK performance. We think this is the best sequential implementation we can have[4]. All parallel speedups are compared directly to this sequential performance.

The parallel performance is shown in Figure 9(a) and all the speedups are obtained with consideration of commutativity. In average, we have obtained speedup 7 on 16 processors and 11.3 on 32 processors. We will discuss the overhead of run-time execution in Section 6.1.3. The speedup on 32 processors is 11.33 for BCSSTK15 and processor utilization efficiency is 35.4%. In [20], efficiency 25% has been reported for BCSSTK15 on 64 processors(Intel Paragon) after improving the load balancing strategy. This indicates

[4]In comparison, Rothberg [19] reported that the sequential performance of his code achieved 77% of LINPACK for BC-SSTK15 and 71% in average for other matrices on IBM RS/6000 Model 320.

that the parallelism is limited in the sparse problems and our scheduled code has achieved reasonable performance. To the best of our knowledge, this is the highest performance obtained for sparse Cholesky factorization by automatically scheduled code.

We examine how much improvement is obtained by exploring commuting operations. Figure 9(b) shows the improvement ratio in terms of execution time. In general, we have observed $1 - 15\%$ performance improvement.



Matrix	P=2	P=4	P=8	P=16	P=32
BCSSTK15	4%	4%	10%	9%	15%
BCSSTK16	6%	8%	4%	3%	7%
BCSSTK17	8%	15%	4%	-9%	3%
BCSSTK24	6%	1%	4%	1%	4%

(a) (b)

Figure 9: (a) Performances of sparse Cholesky; (b) Execution time improvements after exploiting commutativity in sparse Cholesky factorization.

6.1.2 Sparse LU factorization without pivoting

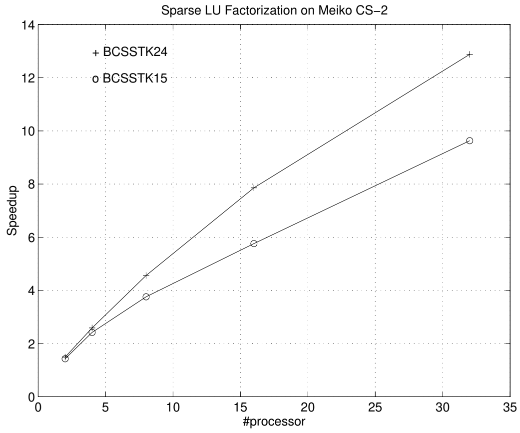
LU factorization without pivoting can be used for positive definite or strictly diagonal dominant matrices. Many of algebraic systems arising from the discretization of diffusion and convection problems in fluid dynamics[18] are nonsymmetric and diagonal dominant. The boundary conditions usually make the first row of matrices strictly diagonal dominant. For such matrices, it can be shown [5] that stable solutions can be obtained by LU (Gaussian Elimination) without pivoting. It should be noted that parallel sparse LU with pivoting is more complicated [10].

We list the BLAS-3 level sparse LU factorization algorithm in Figure 10(a). Again we use supernode partitioning to divide a sparse matrix and construct the sparse task graph from the partitioned matrix. It can be shown that a LU task 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 and their sequential performance on a single Meiko CS-2 node in Table 3. The sequential performance reaches 85% of the LINPACK performance.

We use the PYRROS mapping algorithm to generate schedules for LU graphs and use the run-time system discussed in this paper to execute the schedules. Speedups are shown in Figure 10(b). On 32 processors, we have obtained 12.9 speedup for BCSSTK24, and 9.7 for BCSSTK15.

for $k = 1$ **to** N
  F_{kk}: $\boxed{\text{Factorize } A_{kk} \text{ as } L_k * U_k}$
  for $i = k+1$ **to** N
    if $A_{ik} \neq 0$, S_{ik}: $\boxed{A_{ik} = A_{ik} * U_k^{-1}}$
    if $A_{ki} \neq 0$, S_{ki}: $\boxed{A_{ki} = L_k^{-1} * A_{ki}}$
  end
  for $j = k+1$ **to** N with $A_{kj} \neq 0$
    for $i = k+1$ **to** N with $A_{ik} \neq 0$
      M_{ij}^k: $\boxed{A_{ij} = A_{ij} - A_{ik} * A_{kj}}$
    end
  end
end

(a)



(b)

Figure 10: (a) The block-oriented sparse LU factorization algorithm; (b) Performances of sparse LU for BCSSTK24 and BCSSTK15.

Matrix	#tasks	#edges	#data items	Floating ops	Seq. time(sec.)	MFLOPS
BCSSTK15	123,075	321,907	22,984	312,886,996	46.21	6.77
BCSSTK24	15,761	37,776	4,533	64,032,308	9.36	6.84

Table 3: Statistics of BCSSTK15 and BCSSTK24 for Sparse LU factorization.

6.1.3 Run-time overhead analysis

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. The waiting time is caused by both the network communication delay and the idle time in the pre-determined schedule. We calculate the ratio of sending, receiving and spinning time over the computation time at each processor. Figure 11 and Figure 12 show the overhead distributions in the LU and Cholesky factorizations. 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 from both LU and Cholesky cases that when the number of processors increases, the receiving overhead is relatively small, which is about 2% of the computation time. The sending overhead is about 10% of the computation time. Thus the total run-time sending/receiving overhead contributes about $6 - 10\%$ of the total execution time at each processor. The reason that the receiving cost is much less than the sending cost is as follows. Messages are directly written to the user data space so that tasks can access them as soon as they arrive. However in the sending site, there is at least 9 microseconds cost for each message.

The major loss of the performance goes to the spinning time. The main reason is that a pre-determined schedule does not match actual run-time situation very well. This is because computation weights vary at run-time due to different caching behaviors. Also, the scheduler assumes that the sending communication

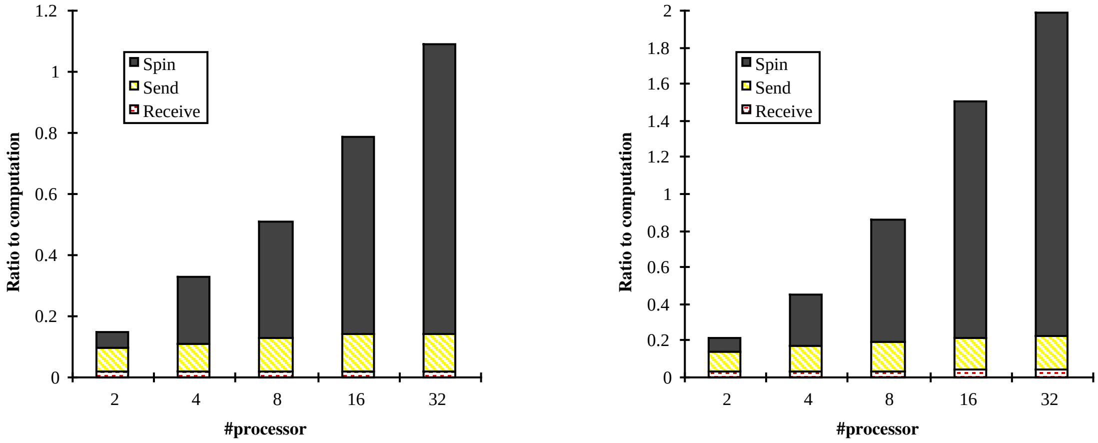


Figure 11: Distributions of the communication overheads for sparse LU factorization. The left part is for BCSSTK24 and the right is for BCSSTK15.

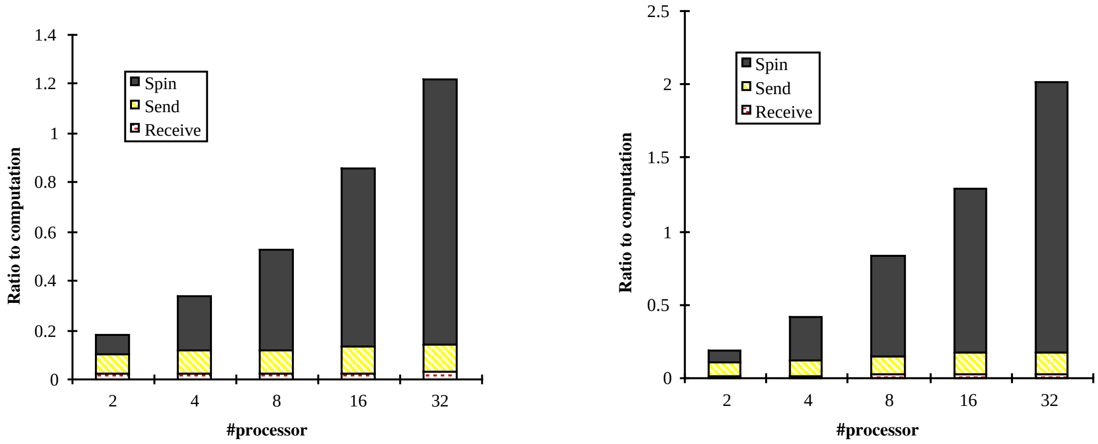


Figure 12: Distributions of the communication overheads for sparse Cholesky factorization. The left part is for BCSSTK24 and the right is for BCSSTK15.

overhead is zero, which is not the case in reality. These two issues make timing among tasks different from what is expected at the static time and tend to increase the processor idle time. The performance becomes even more sensitive when task granularities are small [12]. We have examined the distribution of the sizes of supernodes (after blocking) for the tested matrices and foubd that there are a lot of blocks with sizes less than 6, especially for BCSSTK15, 78% of the blocks are of size 1, which makes the corresponding tasks just a few floating operations. In order to improve the performance further, our future work is to consider techniques of increasing average supernode sizes, e.g., supernode amalgamation [2, 19].

6.2 Sparse triangular solver and performance impact of granularities

From the discussion in the last section we know that the task granularity of an application is important to the performance of irregular code. In this section we further examine how the granularity of an application impacts both the performance and the overhead derived from the system itself. This study is important because we want to make sure that our run-time scheme performs stably when granularity varies.

Matrix	#tasks	#edges	#data items	Floating ops	Seq. time(secs)	MFLOPS
BCSSTK15	6,181	10,824	22,984	1,275,360	0.41	3.11
BCSSTK24	2,092	3,424	2,092	557,844	0.16	3.48

Table 4: Statistics of BCSSTK15 and BCSSTK24 for the sparse triangular solver with one right-hand side.

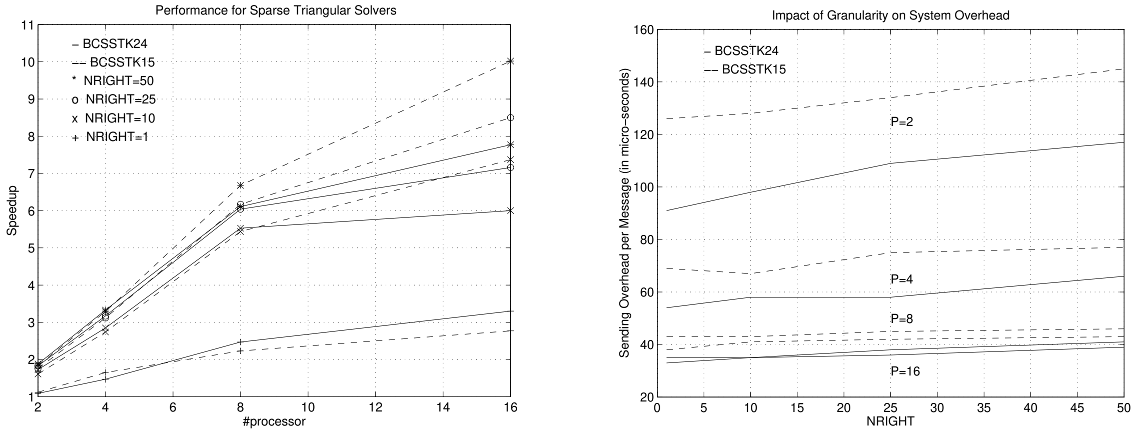
We choose the sparse triangular solver as an example application. Generally a triangular solver is much less time consuming than factorization. A triangular system with multiple right-hand sides is useful in many scientific applications [13]. An interesting property with the triangular solver is that it is very easy to adjust the granularity of tasks by using different number of right-hand sides. Again we first use the supernode partitioning techniques to divide a triangular sparse matrix into $N \times N$ 2-D blocks. The right-hand sides are divided into N submatrices accordingly. We describe the block-oriented triangular solve algorithm in Figure 13, where X_k and B_k are the unknown vectors and right-hand side vectors respectively. They could be matrices instead of vectors if there are multiple right-hand sides.

for $k = 1$ **to** N
  S_{kk}: $\boxed{\text{Do a local triangular solve: } A_{kk} * X_k = B_k}$
  for $i = k+1$ **to** N with $A_{ik} \neq 0$
    P_{ik}: $\boxed{B_i = B_i - A_{ik} * X_k}$
  end
end

Figure 13: The block-oriented sparse triangular solving algorithm.

Notice that the tasks P_{ik}'s for a given i are commutative. Similar to the sparse Cholesky case, we can also take advantage of the commutativity in triangular solvers. In order to examine the impact of task granularity, we choose instances of triangular solvers with the number of right-hand sides being 1, 10, 25 and 50 respectively. The sequential performances for these cases range from 3.1 to 6.1MFLOPS. We list the MFLOPS obtained for one right-hand side in Table 4. The speedups are shown in Figure 14(a). For those cases with the number of right-hand sides greater than 1, decent speedups are obtained. There is not much speedup for the case with 1 right-hand side. But its sequential time is very tiny, less than half second for the tested cases.

To examine the impact of granularity on the run-time overhead, we plot the sending overhead per message for these cases in Figure 14(b) since the sending overhead dominates the run-time overhead. For each matrix, the curves from the bottom to the top are for $P = 16, 8, 4, 2$ respectively, where P is the number of processors used. It can be seen that the sending overhead per message varies very little for a given number of processors even the granularity changes significantly. Notice that in addition to the start-up cost for initiating a RMA request, the sending overhead also includes the costs for eliminating redundant sendings and indexing irregular data objects. Remember when a task tries to send a data item to another processor P_x, it first has to look up the remote address for the data item from a hash table. That is where the indexing cost comes from. Apparently, the number of data items distributed on each processor is counter proportional to the number of processors used, i.e., the hash table size on each processor is smaller if more processors are used. That is why the sending overhead per message on small number of processors is greater than that on a larger number of processors. However, when the number of processors is large



(a) (b)

Figure 14: (a) Speedups for the triangular solver with different numbers of right-hand sides; (b) The run-time sending overhead per message for different numbers of processors.

enough, this indexing cost will become less significant.

We explain the relation between the sending overhead per message with the granularity as follows. From Table 4, we can see that the average computation cost per task for one right-hand side is $66.3\mu s$ for BCSSTK15 and $76.5\mu s$ for BCSSTK24. This average computation cost is relatively close to the sending overhead per message shown in Figure 14(b). Notice that each task may have several outgoing messages and therefore the cost of computation is relatively smaller than the associated communication cost in those graphs. This is the reason that we do not see good performance for those cases. When increasing the number of right-hand sides to 10, the average computation task cost increases in an order of 10, the computation cost dominates and good speedups are obtained even with limited irregular parallelism. It shows that our system overhead has been kept low enough to gain good performance for sparse computation.

6.3 Overhead comparison with a buffered communication scheme

We further compared our approach with a buffered communication scheme to evaluate benefits of the run-time techniques proposed in this paper. The current implementation of PYRROS on Meiko CS-2 uses NX/2 buffered communication primitives. We examine the performance difference of the PYRROS code and the RAPID code for a column-block based Gaussian elimination. Both codes use the same scheduling result. The test cases we choose have small granularities since for such settings the run-time communication overhead plays a more important role. Our experiments show that the overall overhead of RAPID code is substantially smaller than that of the PYRROS code.

#Proc	2	4	8	16
PT reduction	26%	35%	49%	52%
Overhead reduction	74%	60%	65%	57%

Table 5: Improvement of RAPID over PYRROS.

Table 5 reports two performance improvement ratios: the execution time reduction ratio ($1 - \frac{PT_{rapid}}{PT_{pyrros}}$) and the overhead reduction ratio ($1 - \frac{Overhead_{rapid}}{Overhead_{pyrros}}$). The dense submatrix has a size of 3 (we have similar improvement for other cases) and each task is relatively fine-grained. The matrix dimension is 192 and thus there are 64 blocks in each dimension of the matrix. The overhead for each processor is estimated as parallel time minus computation time. The large improvement ratios shown in Table 5 indicate that the buffered communication scheme has a poor performance for supporting fine-grained computations.

7 Related work

We discuss several categories of related work as follows.

Application of graph scheduling for solving scientific problems. As we discussed before, application of graph scheduling has been used in large n-body simulations [12]. In [25], it is found that pre-scheduling improves the performance of distributed sparse Cholesky factorization by 30% to 40% and there is still a lot of room for obtaining better absolute performance. Recently the work by [6] demonstrates that using both effective DAG scheduling and low-overhead communication mechanisms, scalable performance can be obtained for solving fine-grained sparse triangular systems.

Run-time support for parallel computations. The Charm [24] system adopts a message driven approach for general asynchronous computation using dynamic scheduling. The Cilk [3] multi-threaded system aims at applications that have "strict" dependence structures. Randomized load balancing and work stealing techniques are used to execute a dynamic DAG. Compared to our work, Charm and Cilk are suitable for relatively coarser grain computation because dynamic load balancing on distributed memory machines has relatively high control overhead. There are also other run-time systems developed for irregular computations. We have mentioned Chaos in the introduction. Multipol [27] is a run-time library system which supports distributed data structures for several kinds of scientific applications.

Consistency models. The issue of executing a program correctly has been studied in the context of distributed shared memory systems (DSM), e.g. TreadMarks [1]. The main difference between our work and DSM is that a DSM system does not have any knowledge of a program to be executed. A DSM system maintains the consistency of shared objects for all processors without knowing if those objects will be further accessed or not. For example, two common protocols used in DSM are the write-invalidate and write-update. The first one does not allow the prefetching of data, but in our model, we allow processors to send data items to tasks before those tasks need them. The second protocol supports prefetching, but the system may send data items to processors which do not need them. Hence DSM normally has higher overhead than our scheme and has a difficulty for obtaining good performance for sparse matrix problems.

Low overhead communication mechanism. The lower-level communication primitives of our system could be implemented using active messages [26] which have been used in [6] for executing fine-grained triangular solving DAGs. We find that it is not easy to integrate active messages with a general task graph execution scheme because careful network polling is required as demonstrated in [6]. In the situation of task parallelism with mixed granularities, it is not easy to decide where to insert the polling code [4].

8 Conclusions

In this paper, we have presented an efficient run-time scheme for supporting task computations and shown its applications in irregular sparse Cholesky, LU factorization without pivoting and triangular solvers. The experiments indicate that overhead of run-time data communication is relatively small. The performance of sparse computation is limited by available parallelism, but not the overhead of the system. In [9, 10] we have examined the application of these techniques in sparse LU factorization with partial pivoting, which is a more challenging problem and provided an application programming interface (API) for users to parallelize irregular code using our run-time scheme.

Acknowledgement

This research is supported in part by NSF CCR-9409695 and ARPA Contract DABT-63-93-C-0064. We thank Ed Rothberg and Rob Schreiber for their insightful comments on this work; Dan Andresen, Apostolos Gerasoulis, Santosh Pande and anonymous referees for their valuable feedbacks to improve the presentation.

References

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[2] C. Ashcraft and R. Grimes. The Influence of Relaxed Supernode Partitions on the Multifrontal Method . *ACM Transactions on Mathematical Software*, 15(4):291–309, 1989.

[3] R. Blumfoe, M. Frigo, C. Joerg, C. Leiserson, and K. Randall. Dag-Consistent Distributed Shared Memory. In *Proceeding of 10th Inter. Parallel Processing Symposium*, pages 132–141, April 1996.

[4] E. Brewer, F. Chong, L. Liu, S. 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.

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

[6] F. T. Chong, S. D. Sharma, E. A. Brewer, and J. Saltz. Multiprocessor Runtime Support for Fine-Grained Irregular DAGs. *Parallel Processing Letters*, 5(4), December 1995.

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

[8] 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.

[9] C. Fu and T. Yang. Run-time Compilation for Parallel Sparse Matrix Computations. In *Proceedings of International Conference on Supercomputing*, pages 237–244, Philadelphia, May 1996.

[10] C. Fu and T. Yang. Sparse LU Factorization with Partial Pivoting on Distributed Memory Machines. In *Proceedings of Supercomputing'96*, Pittsburgh, November 1996.

[11] G. A. Geist and E. Ng. Task Scheduling for Parallel Sparse Cholesky Factorization . *International Journal of Parallel Programming*, 18(4):291–314, 1989.

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

[13] G. Golub and J. M. Ortega. *Scientific Computing: An Introduction with Parallel Computing Compilers* . Academic Press, 1993.

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

[15] 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.

[16] 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.

[17] S. Pande, D. P. Agrawal, and J. Mauney. A Scalable Scheduling Scheme for Functional Parallelism on Distributed Memory Multiprocessor Systems . *IEEE Transactions on Parallel and Distributed Systems*, 6(4):388–399, April 1995.

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

[19] E. Rothberg. *Exploiting the Memory Hierarchy in Sequential and Parallel Sparse Cholesky Factorization* . PhD thesis, Dept. of Computer Science, Stanford, December 1992.

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

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

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

[23] R. Schreiber. *Scalability of Sparse Direct Solvers*, volume 56 of *Graph Theory and Sparse Matrix Computation (Edited by Alan George and John R. Gilbert and Joseph W.H. Liu)*, pages 191–209. Springer-Verlag, New York, 1993.

[24] W. Shu and L. Kale. Chare Kernel - A Runtime Support System for Parallel Computations. *Journal of Parallel and Distributed Computing*, 11(3):198–211, March 1991.

[25] 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.

[26] 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.

[27] C.-P. Wen, S. Chakrabarti, E. Deprit, A. Krishnamurthy, and K. Yelick. *Runtime Support for Portable Distributed Data Structures*, chapter 9. Languages, Compilers, and Runtime Systems for Scalable Computers. Kluwer Academic Publishers, May 1995.

[28] M. Y. Wu and D. Gajski. Hypertool: A Programming Aid for Message-passing Systems . *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, 1990.

[29] T. Yang. *Scheduling and Code Generation for Parallel Architectures* . PhD thesis, Dept. of Computer Science, Rutgers University, New Brunswick, May 1993.

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