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 for executing the DAG schedule. In this paper, we investigate run-time compilation and supporting techniques 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 Cholesky and LU factorizations for which actual speedups have been hard to obtain in the literature. Our experiments on Meiko CS-2 show that the automatically scheduled code achieves scalable performance for these problems and the run-time overhead is small compared to the total execution time.

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 techniques are useful in exploiting irregular parallelism in unstructured computations. A lot of work has addressed these problems, for example, randomized load balancing for tree-structured tasks [CRY94]. For iterative irregular problems in which communication and computation phases alternate, the CHAOS/PARTI system [DUSH94] has used a run-time compilation approach (inspector/executor) to exploit irregular parallelism. 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 problems addressed in the CHAOS system 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) [GJY95]. The DAG schedule can be re-used for a number of iterations before re-scheduling 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 and the topology of the iteration matrix usually remains the same but the numerical values change at each step [Kar91]. Efficient parallelization of sparse factorization requires certain compilation cost, but the optimized solution can be used for many iterations. The sparse matrix factorizations can be modeled as DAG [Sch93]. However 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 while in the DAGs arising from large n-body simulations, most of tasks are coarse-grained.

Most of previous work in DAG scheduling has focused on the algorithmic research for mapping tasks onto multi-processors based on the macro task computation model [Sar89, WG90]. But few researches are conducted on efficient run-time support for executing task schedules. The early work in the PYRROS system [YG92] provides a complete framework for general task computation; however, its run-time support system uses the NX/2 level communication interface and the overhead of message buffer managing and data copying is significant, 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, and they found pre-scheduling 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 lot 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.

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 $6-8\%$ 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 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 automatically parallelized code, especially on distributed memory machines. Recently impressive performance is obtained in factorizing sparse Cholesky factorization [Rot92, RS94]. Our experiments will show that our approach delivers comparable performance. As far as we know, this is the best performance achieved by the automatically scheduled code.

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 experimental results on sparse Cholesky and LU (without pivoting) factorizations and an overhead analysis. Section 7 gives the conclusions and future work.

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, we define a schedule of G on p processors $S(G,p)$ as two components. One part is a mapping Φ between node set T and processor set $P = \{0, 1, 2, \ldots, p-1\}$. And the other part 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. Therefore, we have processor assignment $\Phi(A) = \Phi(B) = \Phi(D) = 0$, $\Phi(C) = \Phi(E) = 1$, and execution orderings $O_0 = \{A, B, D\}$, $O_1 = \{C, E\}$.

Algorithms for static scheduling of DAGs have been extensively studied in the literatures, e.g. [CR92, PP95, 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 ma-

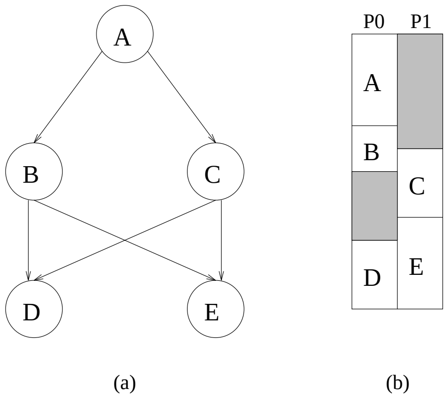

Figure 1: An example of DAG and its schedule on 2 processors.

chines. Static scheduling can adapt to 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 obtain 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 task communication protocol.

Classical data dependence graphs (DDG) for modeling the interaction between computation statements have three types of dependencies: true, anti and output [WB87]. 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 [Sar89, PAM95], i.e., each data item can be written at most once, then there is neither output nor anti dependence. The 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 in 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 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 2 depicts a DDG from which a dependence-compete task graph (See Figure 1) can be derived by deleting all anti and output dependence edges.

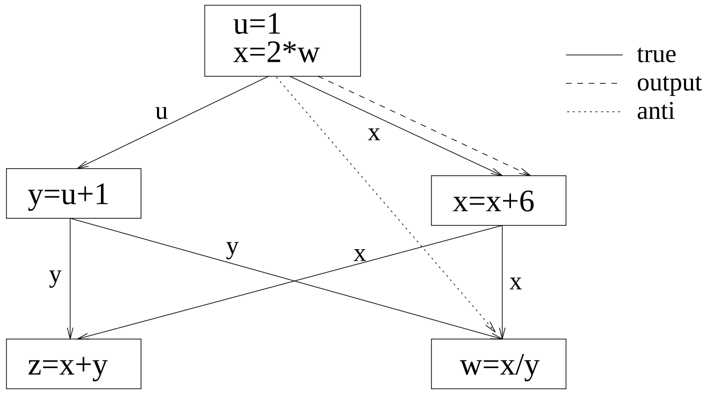

Figure 2: An example of a DDG from which a dependence-complete task graph can be derived.

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 lead to the same semantics as that of the sequential program.

These properties $DA1-4$ impose natural constraints that a task graph should have in representing a 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 [Yan93]. Most task graphs derived from real scientific applications satisfy these properties without renaming. For example, sparse LU graphs discussed in Section 6.2 are dependence-complete.

One disadvantage of a dependence-complete task graph (or DDG) is that it cannot model parallelism arising from commuting operations (e.g. summation of n matrices can be performed in any order). Cholesky factorization is one typical example. However in Section 6.1 we will use a dependence-incomplete task graph to model Cholesky factorization before graph scheduling. After the scheduling algorithm exploits the commutativity of operations, we will 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 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 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, the message passing system such as Intel communication layer on nCUBE-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 message 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 and a careful design is needed to avoid such modification.
  - 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 form 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.

All those activities will cause low efficiency in executing DAGs and eat up benefits of task parallelism.

- Secondly, an effort should be put on maintaining the data dependencies among tasks. Each task has to receive the correct copies of the desired data items. Figure 3 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.

  When a buffered communication scheme is used, a message is correctly fetched by matching the desired message ID. A message ID can be constructed using data item ID and task ID. Thus the correctness is easily ensured in such a scheme but the overhead is large due to ID matching, buffering, copying and space recycling.

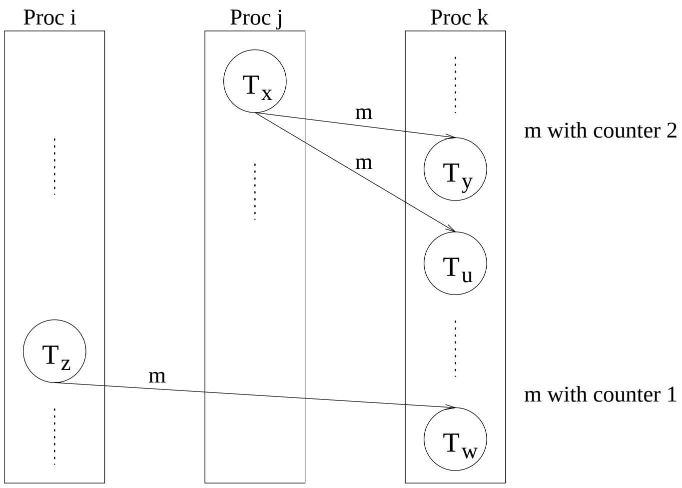

Figure 3: A scenario where data inconsistency could happen.

- 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, 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 [YG92] uses the buffered message-passing mechanism and embodies aggregate communication 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 provide an efficient integrated service for passing data and control on distributed memory machines [vECGS92]. An active message usually contains a function handler and a few short data arguments, and the function at the destination processor can be invoked as soon as this message arrives at the destination. Active messages have been used in [CSBS95] 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 for efficiently implementing active messages as demonstrated in [CSBS95]. And in the situation of task parallelism with mixed granularities, it is not easy to decide where to insert the polling code [BCL+95]. Additionally, active messages usually support short messages[1]. For a general task computation with mixed granularities, efficient support for delivering both short and long messages is required. 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. Presence counter is used to check if the computation for a task is completed. In general, the computation for a task cannot start until all arguments are available.

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 correctness 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 significant amount of overheads, 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 the remote address is given. RMA allows passing data directly from one source location to the destination location, without any copying, packing/unpacking and buffering. Thus user data space and message buffers are integrated together, and there exists no separate message buffer in our scheme.

The model of RMA is defined as follows. When a program in a processor needs to write a data

[1]It should be noted that an extension for long active messages in Meiko CS-2 has been developed in [AISS95], but their *current* implementation only allows one active message outstanding at one time due to hardware restriction and is not quite suitable for the asynchronous execution of tasks in our current work.

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, the data will be transferred asynchronously according to the description. A restriction is that the data transferred through RMA cannot be rewritten until RMA completes, otherwise the remote copy of data will become inconsistent. We need to make sure such inconsistency does not happen in asynchronous execution of tasks.

RMA usually imposes low overhead on both source and destination processors. We have implemented our system on Meiko-CS2 which provides Direct Memory Access(DMA) as the major way to access non-local memory. Each node of Meiko-CS2 is also equipped with a DMA co-processor for handling communication. It takes the main processor 9 microseconds [SS95] 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 machine does not provide DMA support for multicasting a data item. Thus we have to implement the multicasting by repeatedly using DMA feature. For irregular computation where broadcasting happens less frequently than multicasting, such design is reasonable. Sparse Cholesky and LU factorizations use multicasting frequently and our experiment shows the performance is acceptable.

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 processor, 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 in this paper.

4.3 Sending data items

After a task completes its computation, it needs to send 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 4(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 need of waiting for the completion of previous RMA transfer. 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 in 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.

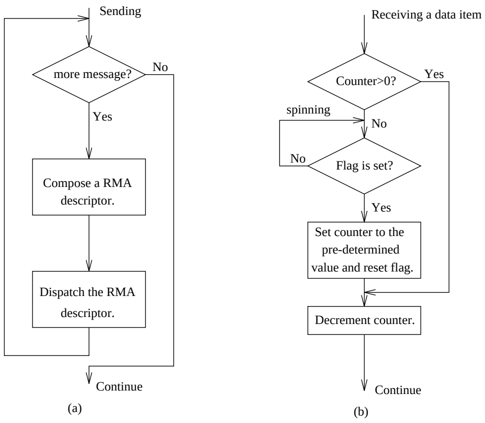

Figure 4: Flow chart for data sending and receiving.

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 "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 the static data flow analysis. We call a data item dead if the corresponding usage counter is zero.

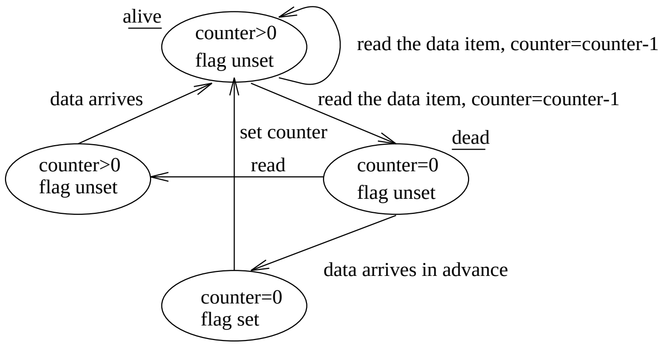

Figure 5: The state transition of a data item during the task graph execution.

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 4(b). The values of the counter and the arrival flag represent the state of a data item. Figure 5 shows how states evolve during the execution of the task graph. The edges in Figure 5 are actions taken to transit from one state to another.

For example, in Figure 3, 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.

One might argue that two fields "counter" and "flag" could be combined together, for example, using the negative counter value to indicate that a data item is not available. The reason we use separate fields is follows. Suppose we combine two fields 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 multicasting 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 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.

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). 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. ■

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.*

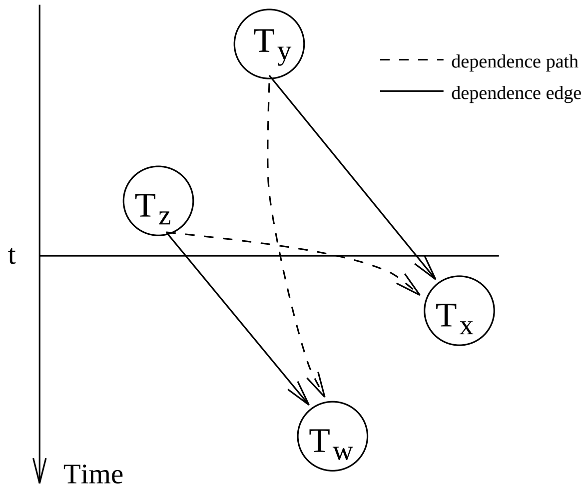

Figure 6: Illustration of the situation in Theorem 1.

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 7, we have two cases.

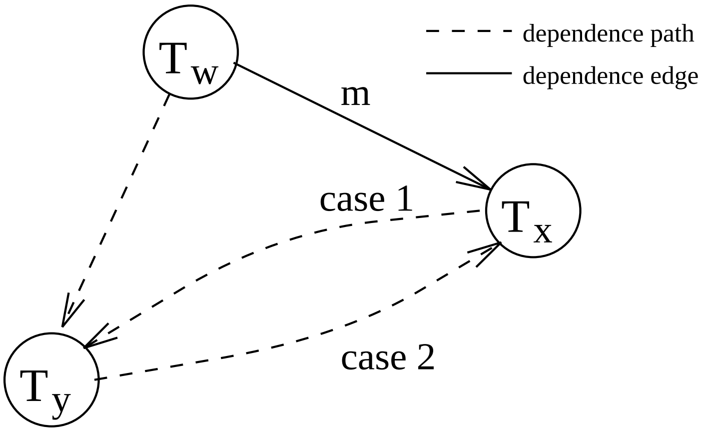

Figure 7: Illustration of the situation in Theorem 2.

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 Experiments results with irregular scientific computations

We will examine the performance of our system on two irregular applications: sparse Cholesky and LU factorizations. We have developed a set of library functions allowing users to specify dependence graph based on a sequential version of the application. Then 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 in 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-CS2 is 40MBytes/Sec. The effective peak bandwidth has been reported as 39MB/Sec and but this is obtained through a ping-pong test [SS95] and message caching improves the performance implicitly. In practice, a message is sent only once. We show the effective bandwidth of a single-message sending or direct memory access in Figure 8 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.

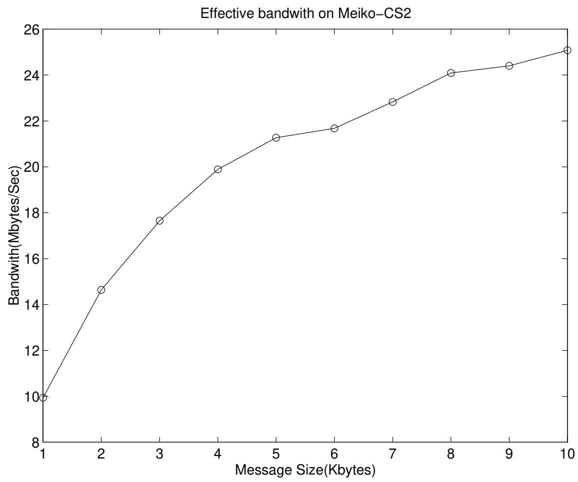

Figure 8: Effective single-message communication bandwidth on Meiko CS-2.

[2]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-CS2.

6.1 Sparse Cholesky factorization

Sequential and parallel sparse Cholesky factorization algorithms have been studied extensively in literature. Rothberg and Schreiber [Rot92, RS94] show that the supernode-based approach can deliver good performance in both shared and distributed memory machines. Specialized scheduling techniques have been studied for sparse Cholesky factorization in [GN89, Liu86]. We will examine the performance of applying general task scheduling and executing techniques to this problem.

For a dense matrix, the BLAS-3 level Cholesky factorization algorithm[3] is shown in Figure 9. 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.

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

Figure 9: The block Cholesky factorization algorithm for dense matrices.

Given a sparse matrix, we first use the multiple minimum degree algorithm to decide the ordering and then calculate fill-ins. For instance, Figure 10 is a 21×21 sparse matrix with 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 the advantage of caching and reduce indexing overhead. In the literature, tasks F_{kk} and S_{ik} are also called *DIV* operations. And tasks

[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 multi-frontal method. In this paper we are not considering parallelizing multi-frontal method.

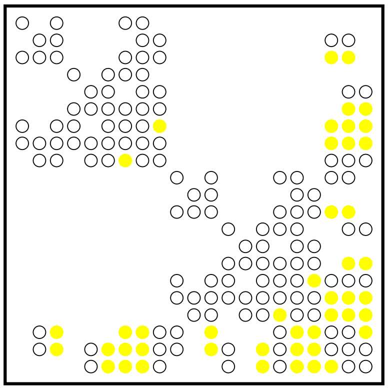

Figure 10: A sparse matrix with fill-ins.

M_{ij}^k are called *MOD* operations.

An important property of sparse Cholesky factorization is that all the *MOD* operations for updating the same block commute. And this is a potential source of parallelism [4]. 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 of *MOD* operations. Figure 11(a) shows a dependency chain derived from sequential code. But in fact different orders of performing those updates to the destination block will not affect the final result. Hence if we schedule the graph containing fan-in parallelism shown in Figure 11(b), we would gain more performance because of more flexibility in scheduling. In Figure 12 we also show the two different Cholesky task graphs for the example matrix in Figure 10. From this small example it can be seen that exploiting operation commutativity does expose more parallelism.

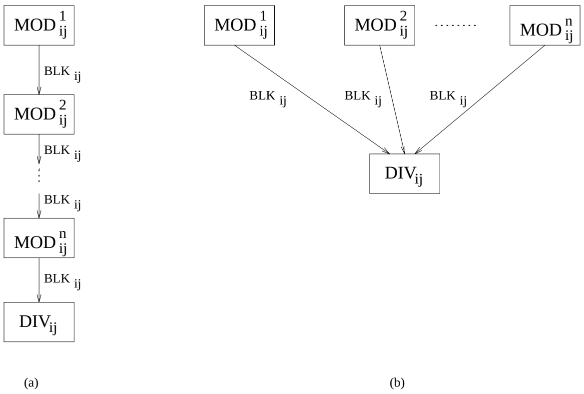

Figure 11: (a) Dependence structure based on the sequential algorithm. (b) Dependence structure exposing commutative operations.

However while the task graph derived from sequential semantics (such as the one in the left of Figure 12) satisfies DA1-4, the task graph considering commutativity (such as the one in the right

[4]This 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.

of Figure 12) *is not* dependence-complete since all updating operations in Figure 11(b) modify the same data item which violates Assumptions DA1 and DA3. In order to correctly execute the fan-in task graph, we have to alter the task graph following the scheduling result. This is done by renaming data items and changing edges as depicted in Figure 13. We explain the procedure as follows. For each DIV task (call it DIV_{ij}) in the fan-in task graph, we apply the following transformations for its MOD task predecessors at each processor P_x:

- We delete all edges from these MOD tasks to this task DIV_{ij} and create a new data item for these MOD tasks on processor P_x;
- Add a chain of dependence edges between these MOD tasks following the ordering at P_x in the DAG schedule and label each edge using the newly-created data item.
- Add a dependence edge from the last MOD predecessor task of DIV_{ij} on P_x to task DIV_{ij} and also label this edge using the new data item.

It is easy to verify that the transformed graph is dependence-complete, which ensures the correctness of execution in our scheme.

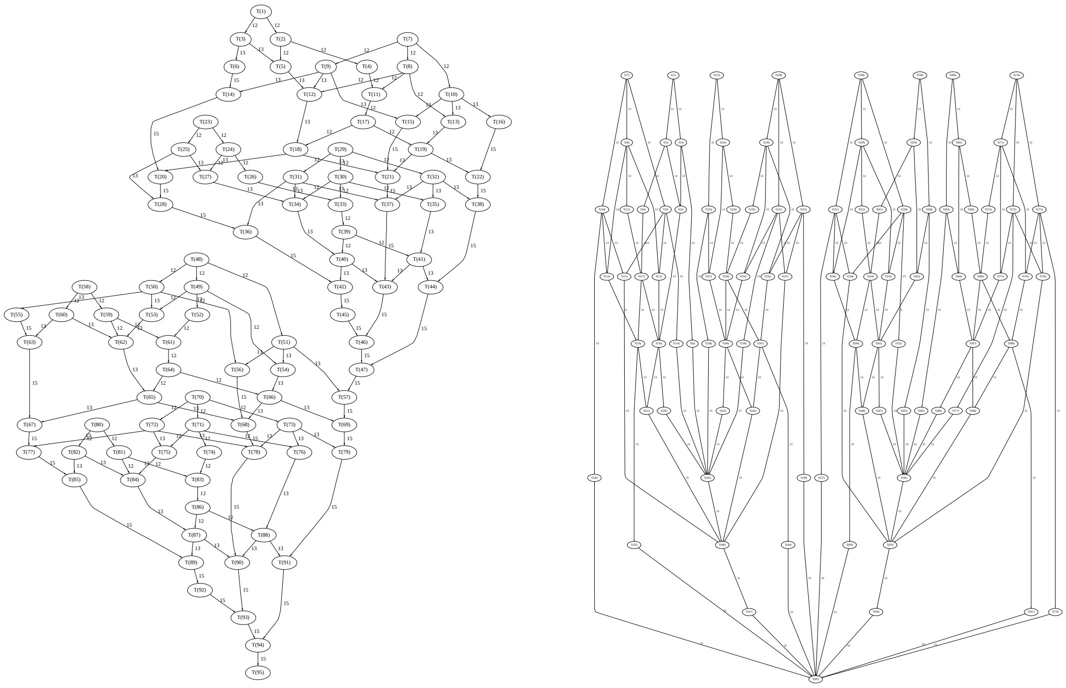

Figure 12: The left is a sparse Cholesky DAG (for the matrix in Figure 10) based on pure sequential semantics and the right is a sparse Cholesky DAG exploring the commutativity of *MOD* operations.

The above transformation introduces new data items at each processor that owns the predecessors of DIV_{ij}. This considerably increases the cost of managing data items and memory requirement. We

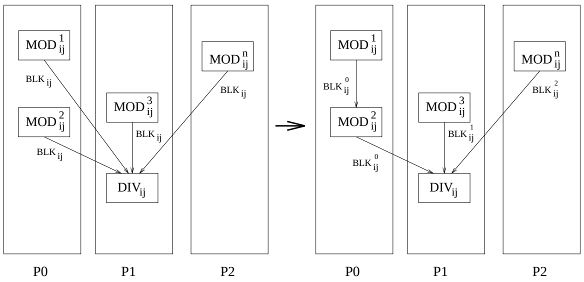

Figure 13: Dependence graph alteration after scheduling.

have used a simple mapping heuristic for clustering based on well-known "owner-compute rule". We cluster all tasks which modify the same data item to the same processor. In this way, we can avoid creating new data items, but still keep the flexibility of scheduling those commuting MOD operations in arbitrary order. The PYRROS scheduling algorithm further merges clusters to processors and performs task ordering. We will show in the experiment results that the above scheduling strategy works well.

Matrix	#tasks	#edges	#data items	Floating ops
BCSSTK15	67,621	177,179	12,167	157,398,044
BCSSTK16	36,639	90,348	6,523	153,527,463
BCSSTK17	69,557	173,452	16,252	147,834,028
BCSSTK24	9,124	19,911	2,487	32,432,756

Table 2: Statistics of benchmark matrices for Sparse Cholesky factorization with block size 25.

Matrix	Sequential time(seconds)	MFLOPS
BCSSTK15	26.72	5.89
BCSSTK16	20.81	7.38
BCSSTK17	24.44	6.05
BCSSTK24	5.46	5.94

Table 3: Sequential performance of sparse Cholesky factorization for benchmark matrices with block size 25. Note that LINPACK performance is 8.05 MFLOPS.

We have tested a set of benchmark matrices from Harwell-Boeing collection [DGL86]. 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 Arena. Task graphs derived from these matrices are very big 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 listed in Table 3. The sparse matrix performance on sequential machines is usually poor because of indexing of irregular data items. Our sparse Cholesky implementation has achieved 73% − 92% of dense matrix LINPACK performance. We think this is the best sequential implementation we can have[5]. All parallel speedups are compared directly to this sequential performance.

The parallel performance is shown in Figure 14 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 overhead of run-time execution in Section 6.3. The speedup on 32 processors is 11.33 for BCSSTK15 and processor utilization efficiency is 35.4%. In [RS94], efficiency 25% has been reported for BCSSTK15 on 64 processors(Intel Paragon) after improving the load balancing strategy. Thus our scheduled code has achieved comparable performance. To the best of our knowledge, this is the highest performance obtained for Cholesky factorization by automatically scheduled code.

We examine how much improvement is obtained by exploring commuting operations. Table 4 shows the improvement ratio in terms of execution time. In general, we have observed 1 − 15% performance improvement. Only in one case the performance actually degrades. This is mainly because the change of clustering strategies. We are further investigating this issue.

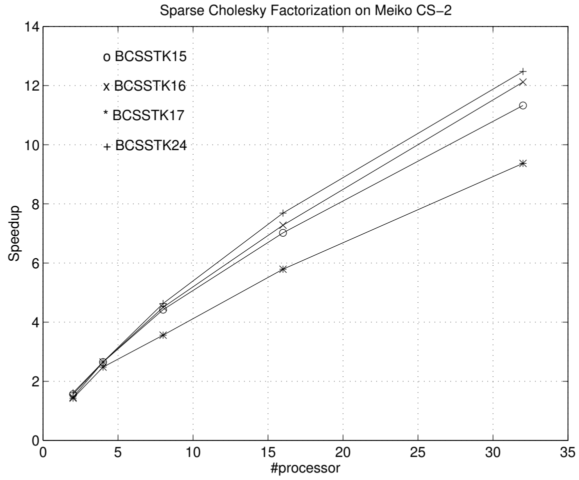

Figure 14: Performances of sparse Cholesky with block size 25.

6.2 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

[5]In comparison, Rothberg [Rot92] reported that the sequential performance of his code achieved 77% of LINPACK for BCSSTK15 and 71% in average for other matrices on IBM RS/6000 Model 320.

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%

Table 4: Execution time improvements after exploiting commutativity in sparse Cholesky factorization.

problems in fluid dynamics[PT83] 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 [Cai95] that stable solutions can be obtained by LU (Gaussian Elimination) 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 $k = 1$ **to** N
  F_{kk}: Factorize $A_{k,k}$ as $L_k * U_k$
  for $i = k + 1$ **to** N
    S_{ik} : $A_{i,k} = A_{i,k} * U_k^{-1}$
    S_{ki} : $A_{k,i} = L_k^{-1} * A_{k,i}$
  end
  for $j = k + 1$ **to** N
   for $i = k + 1$ **to** N
    M_{ij}^k : $A_{i,j} = A_{i,j} - A_{i,k} * A_{k,j}$
   end
  end
end

Figure 15: The block LU factorization algorithm for a dense matrix.

We list the BLAS-3 level LU factorization algorithm for a dense matrix in Figure 15. For a sparse matrix, we also use supernode partitioning to divide the 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 [DGL86]. 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 5. The sequential performance reaches 85% of the LINPACK performance. Block size 25 is used to partition large supernodes.

We use the PYRROS mapping algorithm to generate the schedule for LU graphs and use the runtime system discussed in this paper to execute this schedule. Speedups are shown in Figure 16. On

Matrix	#tasks	#edges	#data items	Floating ops	Seq. time(secs)	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 5: Statistics of BCSSTK15 and BCSSTK24 for Sparse LU factorization with block size 25.

32 processors, we have obtained 12.9 speedup for BCSSTK24, and 9.7 for BCSSTK15.

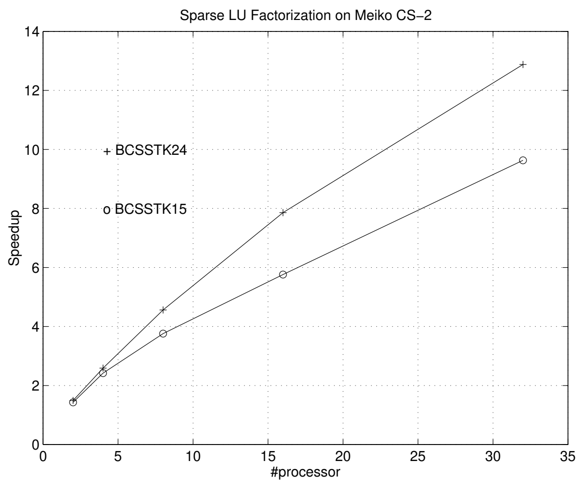

Figure 16: Performances of sparse LU for BCSSTK24 and BCSSTK15 with block size 25.

6.3 Run-time overhead analysis

We classify the time spent at each processor into four parts: computations (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 wanted 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 two sources: one is the network communication delay, and another is 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 17 and Figure 18 show the overhead distribution in 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

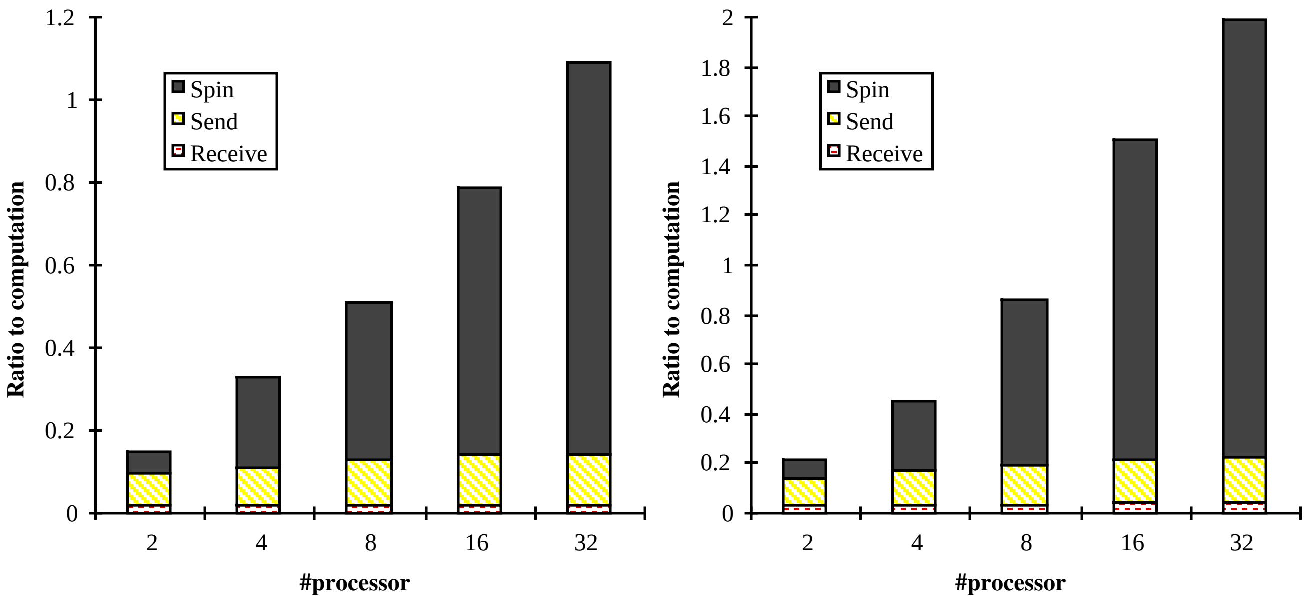

Figure 17: Distributions of the communication overheads for sparse LU factorization. The left part is for BCSSTK24 and the right is for BCSSTK15.

overhead is about 10% of the computation time. Thus the total run-time sending/receiving overhead contributes about 6−8% 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 9 microseconds cost for each message.

The major loophole of the performance goes to the spinning time, which is mainly caused by the limited parallelism in these problems. Furthermore, 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 the sending communication 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 [GJY95, YFGS95]. If we look at the distribution of the sizes of supernodes (after blocking) as shown in Figure 19, we can see that for both matrices, 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 [AG89, Rot92]. Dynamic scheduling will be more adaptive to run-time weight variation, but it also increases run-time overhead significantly.

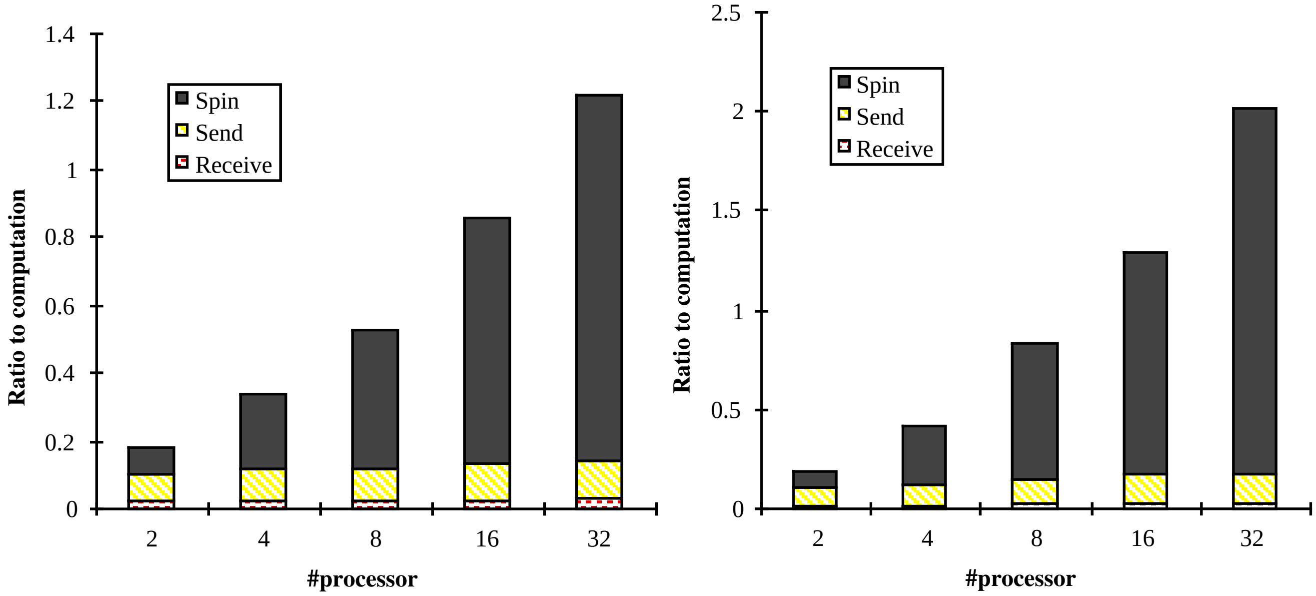

Figure 18: Distributions of the communication overheads for sparse Cholesky factorization. The left part is for BCSSTK24 and the right is for BCSSTK15.

6.4 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 more important role. Our experiments show that the overall overhead of RAPID code is substantially smaller that that of the PYRROS code.

Figure 20 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 Figure 20 indicate that the buffered communication scheme has a poor performance for supporting fine-grained computations.

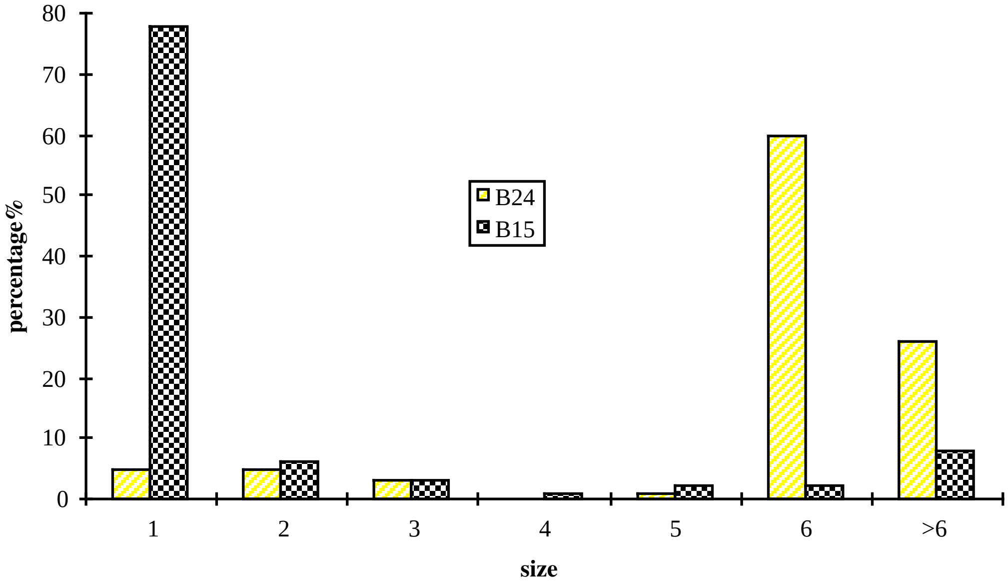

Figure 19: Distributions of the block sizes.

7 Conclusions

In this paper, we have presented an efficient run-time scheme for supporting task computations and shown its applications in irregular sparse Cholesky and LU factorizations. The experiments indicate that overhead of run-time data communication is small compared to the total execution time ($6-8\%$). The performance of sparse computation is limited by available parallelism, but not the overhead of the system. Our future work is to examine the applications of our techniques in wider ranges of irregular problems and provide programming interface for users to parallelize irregular code using our techniques.

Acknowledgement

This 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; Chris Scheiman, Martin Rinard and Klaus Schauser for useful discussions on Meiko; Yizhou Yu for his help in the implementation of part of the system; Dan Andresen for numerous help in preparing this presentation.

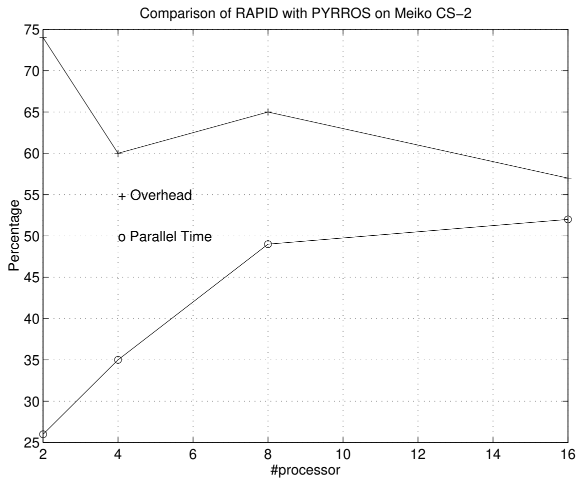

Figure 20: Improvement of RAPID system over PYRROS system.

References

[AG89] Cleve Ashcraft and Roger Grimes. The Influence of Relaxed Supernode Partitions on the Multifrontal Method . *ACM Transactions on Mathematical Software*, 15(4):291–309, 1989.

[AISS95] A. Alexandrov, M. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP model - One step closer towards a realistic model for parallel computation. In *7th Annual Symposium on Parallel Algorithms and Architectures*, pages 95–105, July 1995.

[BCL+95] Eric A. Brewer, Frederic T. Chong, Lok T. Liu, Shamik D. Sharma, and John 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 Publishing Company, fifth edition, 1993.

[Cai95] Wei 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 Rajiv K. Kalia and Priya Vashishta, editors, *Toward Teraflop Computing and New Grand Challenge Applications.*, New York, 1995. Nova Science Publishers.

[DGL86] I. S. Duff, R. G. Grimes, and J. G. Lewis. User's guide for the harwell-boeing sparse matrix collection. Technical report, 1986.

[DUSH94] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures . *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[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.

[GN89] G. A. Geist and E. Ng. Task Scheduling for Parallel Sparse Cholesky Factorization . *International Journal of Parallel Programming*, 18(4):291–314, 1989.

[HNP91] Michael Heath, Esmond Ng, and Barry 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.

[Liu86] Joseph W. H. Liu. Computational Models and Task Scheduling for Parallel Sparse Cholesky Factorization . *Parallel Computing*, 18:327–342, 1986.

[PAM95] 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.

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

[PP95] Santosh Pande and Kleanthis Psarris. *Scheduling Acylic Task Graphs on Distributed Memory Parallel Architectures*, volume 22 of *Parallel Processing of Discrete Optimization (Edited by P. Pardalos, M. Resende and K. G. Ramakrishnan)*, pages 289–304. AMS, 1995. DIMACS Series.

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

[Rot92] Edward Rothberg. *Exploiting the Memory Hierarchy in Sequential and Parallel Sparse Cholesky Factorization* . PhD thesis, Dept. of Computer Science, 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, November 1994.

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

[Sch93] Robert 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.

[SS95] Klaus E. Schauser and Chris J. Scheiman. Experience with active messages on the meiko cs-2. In *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] Sesh Venugopal, Vijay Naik, and Joel Saltz. Performance of Distributed Sparse Cholesky Factorization with Pre-scheduling. In *Proc. of Supercomputing'92*, pages 52–61, Minneapolis, November 1992.

[WB87] M. Wolfe and U. Banerjee. Data Dependence and Its Application to Parallel Processing. *International Journal of Parallel Programming*, 16(2):137–178, 1987.

[WG90] 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.

[Yan93] Tao Yang. *Scheduling and Code Generation for Parallel Architectures* . PhD thesis, Dept. of Computer Science, Rutgers University, New Brunswick, May 1993.

[YFGS95] Tao Yang, Cong Fu, Apostolos Gerasoulis, and Vivek Sarkar. Mapping Iterative Task Graphs on Distributed-memory Machines . In *International 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.