Run-time Compilation for Parallel Sparse Matrix Computations*

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

Abstract

Run-time compilation techniques have been shown effective for automating the parallelization of loops with unstructured indirect data accessing patterns. However, it is still an open problem to efficiently parallelize sparse matrix factorizations commonly used in iterative numerical problems. The difficulty is that a factorization process contains irregularly-interleaved communication and computation with varying granularities and it is hard to obtain scalable performance on distributed memory machines. In this paper, we present an inspector/executor approach for parallelizing such applications by embodying automatic graph scheduling techniques to optimize interleaved communication and computation. We describe a run-time system called RAPID that provides a set of library functions for specifying irregular data objects and tasks that access these objects. The system extracts a task dependence graph from data access patterns, and executes tasks efficiently on a distributed memory machine. We discuss a set of optimization strategies used in this system and demonstrate the application of this system in parallelizing sparse Cholesky and LU factorizations.

1 Introduction

Program transformation and parallelization techniques for structured codes have been shown successful in many application domains. However it is still difficult to parallelize unstructured codes, which can be found in many scientific applications [18]. In [4] an important class of unstructured and sparse problems which involve iterative computations is identified and has been successfully parallelized using the inspector/executor approach. The cost of optimizations conducted at the inspector stage is amortized over many computation iterations at the executor stage.

In this paper, we address another class of unstructured problems with loop-carried data dependencies and irregular task parallelism. A typical application is sparse matrix factorization arising from iterative numerical computations such as the Newton's method for solving non-linear equations. Since sparse matrix factorization dominates the computation at each iteration, an effective run-time optimization at the inspector stage could improve the code performance at the executor stage substantially.

Previous results [1, 8, 17, 20] have demonstrated that graph scheduling can effectively exploit irregular task parallelism if task dependencies are given explicitly. We generalize the previous work and discuss a run-time library system called RAPID for exploiting *general* irregular parallelism embedded in unstructured task graphs. The system makes use of graph transformation and scheduling techniques, and an efficient task communication protocol. The design of library functions is based on three concepts: distributed shared data objects, tasks and access specifications. Similar concepts have been proposed in JADE [12] which extracts task dependence and schedules tasks dynamically. Such an approach has a flexibility to handle problems with adaptive structures; however, it is still an open problem to balance the benefits of such flexibility and run-time control overhead in parallelizing applications such as sparse matrix factorization [12]. Our approach extracts dependence and schedules tasks at the inspector stage to trade flexibility for performance.

It should be noted that a full optimization at a preprocessing stage does not suffice to produce efficient code. A careful design in the task communication model is further required to execute the pre-optimized task computation and communication schedule. In [6] we have developed an efficient run-time task communication protocol for executing general irregular task computations with mixed granularities. The distinguishing feature of the protocol is that we *tightly* and *correctly* incorporate several communication optimizations together in one execution framework. They include eliminating message buffering and copying, eliminating redundant messages and unnecessary synchronization. In this paper, we further incorporate techniques of exploiting commutativity and adjusting task granularities, and have observed substantial performance improvements. As a result, RAPID obtains a scalable performance for sparse factorization problems and as far as we know, this is the best performance ever achieved by automatically scheduled code on distributed memory machines.

The paper is organized as follows. Section 2 gives an overview of the RAPID system. Section 3 describes the dependence graph transformation and scheduling performed at the inspector stage, and discusses the optimization that considers task commutativity. Section 4 reviews the run-time task execution protocol and communication optimizations.

*This work is supported by NSF CCR-9409695 and a startup fund from University of California at Santa Barbara.

Section 5 presents experimental results on sparse Cholesky factorization and sparse LU factorization with partial pivoting. It also analyzes the system overhead and presents a performance improvement after increasing task granularities. Section 6 gives the conclusions. A longer version of this paper is [7].

2 System Overview

At the inspector stage, the RAPID system provides a set of C library functions for users to specify shared data objects, and tasks that access these objects. Then it derives the dependence structure from the specification, performs dependence transformation and maps tasks to multi-processors. At the executor stage, the computation of these tasks will be performed. An overview of the system structure is shown in Figure 1 and we explain them in details below.

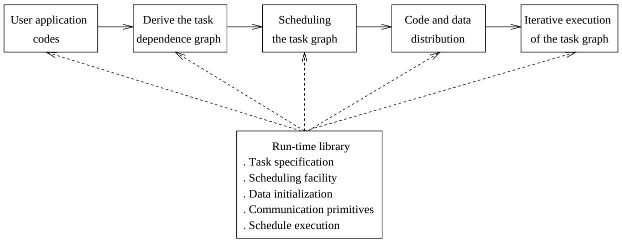

Figure 1: Functional modules of the RAPID system.

2.1 Specification of irregular task computations

The process of specification is that a programmer defines a set of irregular data objects and tasks that read and/or write these objects during computations. An object or a task identification contains a type and indices. Since a change in the execution order of commuting operations does not affect the program correctness, users can also specify commuting operations with respect to a data object and it gives more flexibility for the scheduler to explore parallelism. The specification library functions include:

• $object(data\_type, size, n, index, \cdots)$. This function declares a shared object to be accessed during task graph computation. The dimensionality of the data object is n.

• $task\_function(type, func\_name, n)$. This is to associate tasks referred by *type* with an actual C function to be called at the executor stage. The dimensionality of the task is n, which gives the number of index arguments that will be passed to the task body.

• $task\_begin(task\_type, weight, index, \cdots)$ and $task\_end()$. These functions declare the starting and end point of a task, its computation weight, and indices associated with it.

• $task\_read(task\_type, index, \cdots)$, $task\_write(\cdots)$, and $task\_update(\cdots)$. These functions are called within a task specification body, indicating that this task reads, writes or reads/writes a data object respectively.

• $mark\_commute(data\_type, index, \cdots)$ marks that all the operations which modify this data object will commute after this point. $unmark\_commute(data\_type, index, \cdots)$ indicates that the modifying operations will not commute after this point with respect to this data object.

We will demonstrate the use of these functions for sparse Cholesky factorization. As it can be seen from this example, it is quite easy to specify data accessing patterns following a sequential program. One would argue that it might be cumbersome for users to supply such a specification. We feel that our current objective is to provide a semi-automatic programming tool to reduce burdens of a user in developing *efficient* parallel irregular codes. Such codes are hard to write using existing parallel languages or libraries. The focus of this paper is to demonstrate how we are able to deliver good performance for sparse codes and our future work will address the automatic generation of inspector specification code and automatic task partitioning [2, 11, 15].

Cholesky factorization is performed on a symmetric positive definite matrix A of size $n \times n$. In a block sparse Cholesky algorithm as shown in Figure 2, matrix A is partitioned into $N \times N$ submatrices. A partitioning example is shown in Figure 4. Notice that this submatrix partitioning is not uniform due to supernode partitioning [9, 14]. We assume that the nonzero structure information is available after symbolic factorization and supernode partitioning. These operations are performed before task specification. Each data object is defined as a non-zero sub-matrix of A. The specification of Cholesky tasks and access patterns is in Figure 3.

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

Figure 2: The block-oriented sparse Cholesky factorization algorithm.

2.2 The inspector stage

After a user provides a specification of data objects and tasks, the user can call a C function to invoke the run-time optimization. This function will perform the following operations.

1. Derive dependence structures from the specification. The user's specification may contain loops (e.g. in Cholesky program) and the system will unroll these loops and identify the dependence patterns based on the sequential order given in the specification.

2. Transform the dependence graph to eliminate anti and output dependence, but to preserve parallelism coming from commuting operations. Map the transformed graph onto the given number of processors using graph scheduling techniques. Section 3 discusses more on the dependence model and task scheduling

   The right part of Figure 4 shows a task dependence graph transformed from the graph extracted from data accessing patterns described in Figure 3. The nonzero pattern of the input matrix is shown in the left side of Figure 4. Note that two tasks $M(8,8,2)$ and $M(8,8,5)$ modify the same submatrix object $A_{8,8}$ and they commute. And the corresponding edges are distinguished with dots. Redundant anti and output dependence edges are deleted in this figure.

```
for(i=1;i<=N;i++){
  /* shared data objects to be distributed */
  object("LTinv",datasize(i,i),1,i);
  for(j=1;j<=N;j++)
     if(nonzero(A[i][j])){
        object("A",datasize(i,j),2,i,j);
        mark_commute("A",i,j);
     }
}
/* associate C functions with tasks*/
task_function("F",factor_k,1);
task_function("S",scale_ik,2);
task_function("M",modify_ijk,3);
for(k=1;k<=N;k++){
   /* data accessing patterns of tasks */
   task_begin("F",tasksize_f(k),k);
      unmark_commute("A",k,k);
      task_update("A",k,k);
      task_write("LTinv",k);
   task_end();
   for(i=k+1;i<=N;j++)
       if(nonzero(A[i][k])){ /* scaling */
           task_begin("S",tasksize_s(i,k),i,k);
             unmark_commute("A",i,k);
             task_read("LTinv",k);
             task_update("A",i,k);
           task_end();
       }
   for(j=k+1;j<=N;j++)
     if(nonzero(A[j][k])){
       for(i=j;i<=N;i++) /* updating */
         if(nonzero(A[i][k])){
           task_begin("M",tasksize_m(i,j,k),i,j,k);
             task_read("A",i,k);
             task_read("A",j,k);
             task_update("A",i,j);
           task_end();
         }
     }
}
```

Figure 3: Task specification of sparse Cholesky factorization. In practice, nonzero patterns of a sparse matrix are stored in a compressed format and the specification code can be changed accordingly.

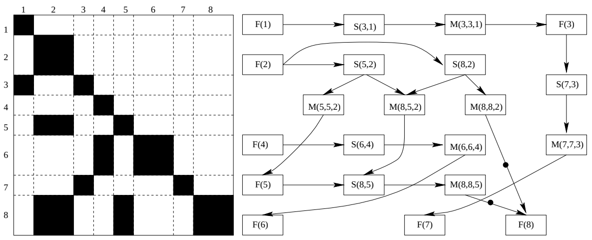

Figure 4: The left is a sparse matrix after partitioning. The right is the corresponding sparse Cholesky task dependence graph.

3. Determine the processor ownership of each data object in a distributed memory system. In order for a program to initialize and re-access data objects after each iteration at the executor stage, the system will assign an owner processor to each data object. At the completion of each task graph computation phase, the value of an object is retained on the owner of this object.

2.3 Task execution and auxiliary functions

After the inspector stage, the program is ready to be executed. Auxiliary functions are provided to assist with data initialization. For example, a user program can call the following function.

• *execute_tasks*(): It will execute the specified computation. We will provide a more detailed description on run-time support for efficiently executing a task schedule in Section 4.

• *get_dataaddr*(*data_type*, *index*, $\cdots$). This function will return the local address of the given data object at the executor stage. For example, a C function *factor_k*() in Figure 5 specifies the computation of task F_k in Figure 3. It uses *get_dataadddr*() to obtain data object addresses before factorizing submatrix A_k^k.

```
factor_k(k){
    SUBMATRIX *Akk,*LTinv;
    Akk=(SUBMATRIX *)get_dataaddr("A",k,k);
    LTinv=(SUBMATRIX *)get_dataaddr("LTinv",k);
    /* Factorize submatrix A(k,k) pointed by
       Akk, put the inverse of L in LTinv */
     ...
 }
```

Figure 5: An example of using get_dataaddr().

3 Task Dependence and Scheduling

Three classical types of data dependence relations between tasks are possible: true, anti, output. We also include an additional relation for operations that commute, i.e., these operations can be performed in an arbitrary order. Figure 6 shows an example of a task dependence graph. $T3$ and $T4$ commute because any order between them can still lead to a correct solution. $T6$ is the task that issues function *unmark_commute*("*z*"), which is the end point of commuting operations $T3$ and $T4$. Notice that for commuting operations, we do not impose any anti/output dependence among those operations unless they are caused by other data objects.

3.1 Pre-transformation of a task dependence graph

After examining the task specification, deterministic dependence is revealed. Before applying graph scheduling, we need to consider necessary dependence to be enforced in the execution. The true dependence can be enforced easily since one task cannot start to execute until the required data objects produced by its predecessors arrive at the local processor. With the presence of anti and output dependencies, run-time synchronization becomes more complicated. We can use renaming techniques [3] to remove these dependencies, however it needs additional memory optimization. We use the following simple strategy to remove output and anti dependence. 1) First we delete all the redundant edges for output and anti dependencies if they are subsumed by a dependence path in the graph. For example, in Figure 6(a), the following anti/output dependence edges are

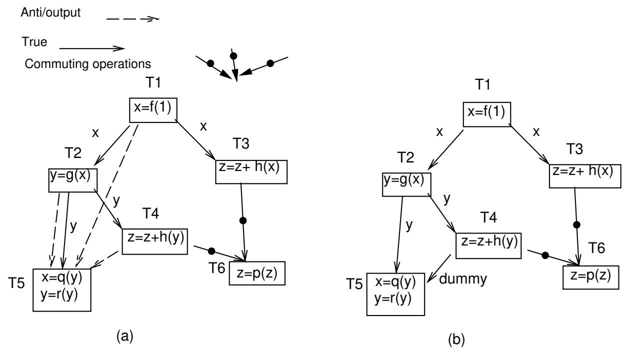

Figure 6: (a) An example of task dependence graph derived from a data accessing specification. (b) The result of dependence graph transformation.

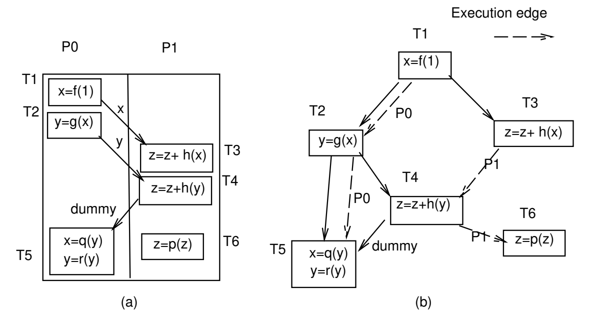

Figure 7: (a) The scheduling result represented as a Gantt chart. (b) The scheduled task graph.

redundant: $(T2, T5)$ and $(T1, T5)$. 2) For a remaining anti or out dependence edge, we replace it with a true dependence edge using a dummy data object of size zero. For example, in Figure 6(b), a dummy variable is introduced for anti-dependence edge $(T4, T5)$.

3.2 Graph scheduling

The transformed task graph contains true dependencies and commuting dependencies. We use a graph scheduling algorithm to exploit task parallelism and also determine the execution order of commuting operations so as to minimize parallel time.

Algorithms for static scheduling of DAGs have been extensively studied in the literatures, e.g. [10, 15, 20]. The main optimizations are eliminating unnecessary communication to exploit data locality, overlapping communication with computation to hide communication latency, and exploiting task concurrency to balance loads among processors. A global performance monitoring for minimizing the overall execution time is needed to guide these optimizations. To do that, task and communication weights are first estimated. In our scheme, users provide a rough estimation of task weights for each irregular task in the *task_begin* function. The system provides an estimate of communication cost between processors based on startup latency, effective communication bandwidth and the size of data objects.

Our scheduling algorithm using weight and dependence information has two stages. At the first stage, we cluster tasks to a set of threads (or directly call them clusters) to reduce communication and exploit the data locality. Two clustering strategies are used: 1) Use the DSC algorithm [21]. 2) Form clusters based on the data accessing patterns. If tasks write or modify the same data object, they will be assigned into one cluster. This data-driven approach is essentially following the owner-computes rule. Currently we use this strategy when a task graph contains commuting operations. The reason is that if commuting operations are assigned to different processors, a global reduction is needed to aggregate the results from different processors, then the reduction operator must be recognized by the system, which is not possible unless a user provides such information. At the second stage, clusters are mapped to a fixed number of physical processors available at the run-time by using the PYRROS algorithm [20] to balance loads and overlap computation with communication.

Figure 7(a) shows the result of scheduling for Figure 6(b). Notice that task $T3$ has to wait for the arrival of data object x sent from $T1$ at processor 0. Computation of $T2$ can be overlapped with this sending. Task $T5$ needs to wait for the dummy data sent from $T4$ at processor 1. One would argue that this dummy dependence may not be necessary since after $T2$ sends out data object y, the modification of y at processor 0 does not affect y at processor 1. This is true if a blocking synchronous communication scheme is used. Since we want to use asynchronous communication to maximize overlapping computation with communication, after $T2$ issues an asynchronous sending, the communication subsystem has to copy y to a system buffer so that the modification of y by $T5$ will not cause inconsistency. However, in our task communication scheme discussed later in Section 4, buffering and copying free primitives are used. Hence the dummy true dependence should still be maintained to ensure the correctness. For sparse matrix computation, such inter-processor anti-dependence or output dependence occurs rarely (e.g. none in Figure 4). Thus the total overhead for handling dummy dependence is insignificant. If two end tasks of a dummy dependence edge are assigned to the same processor, then this edge is redundant and can therefore be deleted since the execution order within a processor enforces the anti or output dependence automatically.

3.3 Properties of a scheduled task graph

We study the properties of a task schedule in terms of a *scheduled* task graph, which is defined as follows. Since the scheduling produces a static task execution order for each processor, we can mark the execution edges in the task graph as follows: if task T_x executes immediately after task T_y at processor P_z and then we add an execution edge between them, annotated with the processor ID P_z in the task graph. Redundant dummy dependence edges within a processor are deleted and the dependence edges from individual commuting tasks to the common end point task are also deleted. We call the new graph as *a scheduled task graph.* Figure 7(b) shows the scheduled task graph for Figure 6(b) based on the DAG schedule in Figure 7(a). The links $(T3, T6)$ and $(T4, T6)$ are deleted.

We study the properties of a scheduled task graph so that our run-time support can take advantage of those properties in designing the task communication protocol. The following properties are satisfied. 1) A task only uses distinct data objects. 2) If two tasks access the same object and one of them writes it, there must be a path between them. 3) The task graph executed on one processor with any order following dependence edges has the same semantics as the sequential code. We call above properties as *dependence-*

complete. Using these properties, we can design an efficient task execution protocol.

4 Design of run-time support for executing scheduled task graphs

A carefully designed communication scheme is important to efficiently execute a task graph schedule. We discuss the major optimizations and performance issues as follows.

4.1 Optimization strategies

Buffering and copying free communication. Low sending/receiving overhead is key to getting good performance for irregular problems on distributed memory architectures. The overhead is mainly caused by buffering and copying incoming/outgoing messages in order to support asynchronous communication. In our approach we use Remote Memory Access(RMA) to communicate a data object between two processors. It does not incur any copying/buffering during the transfer. The functionality of RMA suffices to fit our task protocol and the overhead is kept as small as possible. The only requirement is that a processor needs to allocate a dedicated space for each data object it needs. RMA can be implemented in modern multi-processor architectures such as Cray-T3D and Meiko CS-2 [16]. We have implemented our system on Meiko CS-2 which provides Direct Memory Access(DMA) as the major way to access non-local memory.

Eliminating redundant communication and synchronization. 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 [20].

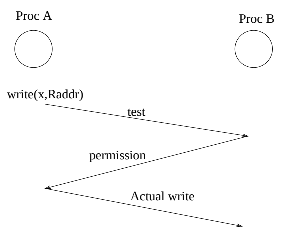

Figure 8: Synchronization for remote data writing.

Another optimization is to eliminate unnecessary synchronization. Since the RMA directly writes data to a remote address, it is possible that the copy at the remote address is still being used by other tasks and then the execution at the remote processor is incorrect. Thus for a general computation, a permission to write the remote address should be obtained before issuing such writing, as depicted in Figure 8. Such synchronization causes extra overhead for our task execution model. After carefully studying the dependence complete properties of a scheduled task graph, we find such explicit synchronization can be eliminated with a simple implicit synchronization scheme introduced below.

4.2 Task Execution Protocol

Our goal is to provide a copying and buffering free task communication protocol that utilizes the RMA facility. In addition to this, we need to incorporate some other optimization strategies as explained above. The ideas of these optimization strategies can be more or less found in some previous work. But it is challenging to put all of them together in a single run-time support framework and design a *correct* execution protocol. For example, we need to make sure messages will not be overwritten before actually being sent out. Thus 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 dependence-complete properties.

Data sending. In sending data objects, the optimization of eliminating redundant messages is incorporated. It is also worthy of noticing that multiple RMAs can be initiated in a row without waiting for the previous RMAs to complete transferring. After a 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 object arrives at the destination processor, a flag in the destination processor associated with that data object will be set automatically to indicate its arrival.

Data receiving. In receiving data objects, we need to make sure that all the data dependencies are preserved. In our scheme, each data object at processor p_x is associated with a usage counter and a flag. The "counter" field of a data object represents the number of tasks at processor p_x that use this copy of the data object. For the "flag" field, it indicates the data object has arrived at the local memory if it is set. The counter information for each copy of a data object at processor p_x is determined by a static data flow analysis. We call a data object dead if the corresponding usage counter is zero.

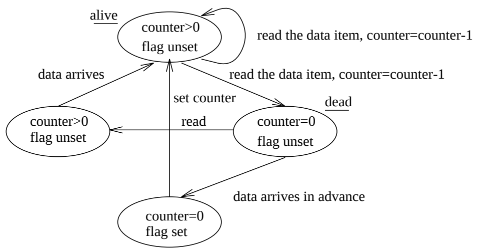

Figure 9: The state transition of a data object during the task graph execution.

When task T_y at processor p_x needs to read data object m, it first checks the usage counter of this object. If the counter is greater than zero, the task decrements the counter. If the data object is dead, the task has to wait until the associated flag is set, which indicates that a new copy of this data object 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 values of the counter and the arrival flag represent the state of a data object. Figure 9 shows how states evolve during the execution of the task graph. The edges in Figure 9 are actions taken to transit from one state to another.

4.3 Correctness analysis

Since we have skipped the synchronization phase before a remote write operation, two data inconsistencies could arise due to the nature of RMA. One is the receiver side inconsistency illustrated in Figure 10(a). Task T_x needs to read data m produced by task T_y. But T_x could read the copy from T_z because this copy of m could come in and overwrite the old copy from T_y. The other one is the sender side inconsistency as shown in Figure 10(b). Task T_w sends a data object m to task T_x at another processor by issuing a RMA request. The initiation of a RMA request for a data object m does not guarantee that m is actually sent out. Before the actual RMA data transferring is completed, another task T_y could redefine this object on the same processor. Then the copy of m received by T_x may not be correct.

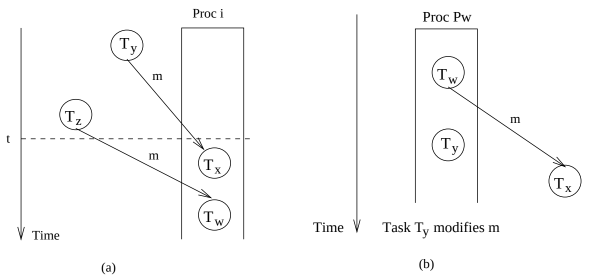

Figure 10: (a) Illustration of receiver site inconsistency. (b) Illustration of sender site inconsistency.

Based on the properties of a scheduled task graph discussed in Section 3.3, we can show the above inconsistency situations will never happen in RAPID. The detailed proofs can be found in [7]. Therefore, the execution of a scheduled task graph is correct in the RAPID system.

Theorem 1 *Given a scheduled task graph, the executor stage will have no receiver site inconsistency and no sender site inconsistency.*

Corollary 1 *The execution of a scheduled task graph in RAPID is correct.*

5 Experimental studies

We will examine the performance of our system on two irregular applications: sparse Cholesky and LU factorizations, on Meiko CS-2. The single-node LINPACK performance we have tested is 8.04MFLOPS for double-precision floating operations. The effective communication bandwidth of Meiko CS-2 is about $1 - 5$MBytes/sec for a double-precision matrix of size less than 6×6 under the assumption that the message is sent only once at a time. The bandwidth would be higher if a ping-pong type of test is conducted [16]. Also it takes the main processor 9 microseconds [16] to dispatch a remote memory access descriptor to the communication co-processor.

5.1 Sparse Cholesky factorization

The Cholesky program in Figure 2 is presented in a right-looking style. Another presentation is called left-looking. Our task model can capture parallelism arising from both styles. We have reported good results based on 2-D block partitioning of a sparse matrix in [6] using our run-time support. Here we further incorporate the advantage of commuting tasks. The choice of block size depends on the cache size of the target machine. But a block size that is too large will reduce available parallelism. A Meiko node has a 20K data cache, and in the worst case, three full size matrices will be needed in performing a M_{ij}^k task. Based on these considerations, we choose 25 as the maximum block size in all our experiments.

We have tested a set of benchmark matrices from Harwell-Boeing collection: BCSSTK15, BCSSTK16, BCSSTK17 and BCSSTK24. Task graphs derived from these matrices are very big and unstructured. We summarize some statistics of those task graphs and the sequential performances in Table 1. Our sequential sparse Cholesky code has achieved 73%–92% of dense matrix LINPACK performance[1]. All parallel speedups are compared directly to this sequential performance.

Matrix	order	#tasks	flops($\times 10^6$)	MFLOPS
B15	3,948	67,621	157.4	5.89
B16	4,884	36,639	153.5	7.38
B17	10,974	69,557	147.8	6.05
B24	3,562	9,124	32.4	5.94

Table 1: Statistics of benchmark matrices for Sparse Cholesky factorization with maximum block size 25.

The parallel performance is shown in Figure 11 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 5.3. A further improvement over this performance by using supernode amalgamation is discussed in Section 5.4. To the best of our knowledge, this is the highest performance obtained for Cholesky factorization by automatically scheduled code.

We have also examined how much improvement is obtained by exploring commuting operations. This is done by comparing the above code with the Cholesky code without marking commuting tasks. Table 2 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 and this could be caused by changing the clustering strategy.

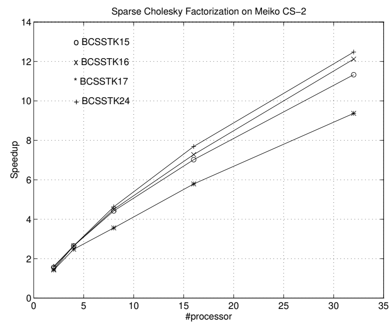

Figure 11: Performance of sparse Cholesky.

5.2 Sparse LU factorization with partial pivoting

LU factorization without pivoting can be used for positive definite or strictly diagonally dominant matrices. We have reported good performance for it in [6]. However in many cases pivoting is needed to maintain the numerical stability. Because partial pivoting operations make the data structures change dynamically, it has been an open problem to

[1] In comparison, Rothberg [13] 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 2: Execution time improvements after exploiting commutativity in sparse Cholesky factorization.

efficiently parallelize sparse LU code with partial pivoting, particularly on message passing machines. We report our preliminary results on this tough problem using the RAPID system. The performance for a set of unsymmetric Harwell-Boeing matrices is listed in Table 3. The single node performance is reasonable since it has reached $25 - 45\%$ of LINPACK performance and the multi-processor speedups are good considering the current status of parallel sparse LU factorization research. This work is still preliminary and we are comparing it with other sparse LU code [5].

Matrix	P=1	P=2	P=4	P=8	P=16
sherman5	2.55	4.41	7.93	14.1	17.2
lnsp3937	2.33	4.08	7.04	13.6	15.0
lns3937	2.01	3.60	6.17	12.0	14.6
sherman3	3.77	6.50	11.1	19.3	20.3
jpwh991	3.01	5.15	9.18	16.7	16.0
orsreg1	3.78	6.77	11.2	20.5	21.6
saylr4	3.64	6.62	11.3	19.9	22.3

Table 3: Preliminary performance for Sparse LU factorization with partial pivoting. All numbers are in MFLOPS.

5.3 A comparison with PYRROS and overhead analysis

We have 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 between 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. The results in Table 4 show that the overall parallel time of RAPID code is substantially smaller than that of the PYRROS code, which indicates that the buffered communication scheme has a poor performance for supporting fine-grained computations.

#processor	P=2	P=4	P=8	P=16
PT reduction	26%	35%	49%	52%

Table 4: Parallel time reduction of RAPID over PYRROS.

To further investigate the run-time overhead distribution, we classify the time spent at each processor into three parts: computations (CPU time used for executing tasks), sending/receiving overhead, and the idle time that a processor spends on waiting for the arrival of the desired data object. If the desired data object 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.

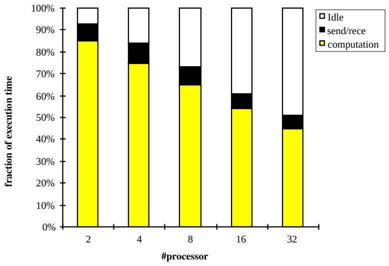

Figure 12: Distributions of the communication overheads for sparse Cholesky factorization on BCSSTK15.

We calculate the ratios of computation, sending/receiving overhead and the idle time over the total execution time at each processor. Figure 12 shows the result for Cholesky factorization on matrix BCSSTK15. For other matrices, similar breakdowns have been observed. It can be seen that sending/receiving overhead only contributes about $6 - 10\%$ of the total execution time.

5.4 Increasing task granularity for further improvement

From the overhead analysis in the previous section, we can see that though the communication overhead, i.e., sending/receiving overhead, of the run-time system is small, each processor still spends a lot of time waiting for messages to arrive, and not doing any useful computation. Part of the reason is the limited parallelism in these problems. However, another important reason is that pre-determined schedule does not match actual run-time situation very well. This is mainly caused by run-time computation weights variation. When the variation is within a small range, the static schedule can still deliver good performance [8, 19]. But when the task granularities are small, the performance will become very sensitive to the weights variation. We have examined distribution of supernode sizes after splitting large supernodes for matrices BCSSTK15 and BCSSTK24. BCSSTK15 has the smallest average supernode sizes which implies the small average task granularity (the ratio of computation over communication). BCSSTK24 has the biggest average supernode sizes among all tested matrices. As shown in Figure 13, we can see that for both matrices, there are a lot of supernodes with sizes less than 6, especially for BCSSTK15, 78% of the supernodes are of size 1, which makes the corresponding tasks just a few floating operations.

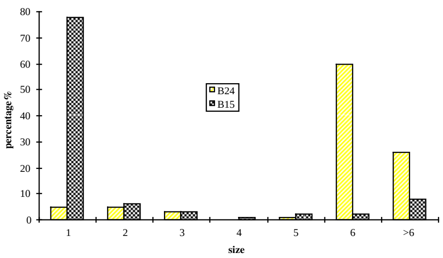

Figure 13: Distributions of the block sizes.

In order to improve the performance further, we need to increase the average task grain size. One way is to use the supernode amalgamation [13] technique. The idea behind the supernode amalgamation is that we can merge several small supernodes together to form a big one so that the corresponding task becomes coarser. This merging process is done by relaxing the restriction that all the columns in the same supernode have identical nonzero pattern. We have used this technique on matrices BCSSTK15 and BCSSTK24 for Cholesky factorization, and the execution time improvements are listed in Table 5.

Matrix	P=2	P=4	P=8	P=16	P=32
BCSSTK15	3%	8%	17%	19%	36%
BCSSTK24	1%	5%	2%	9%	20%

Table 5: Performance improvements by supernode amalgamation.

From this result, we can see that although some zero entries have been introduced by amalgamation, the overall performance gain is very significant. We can reach a speedup of 17 for BCSSTK15 and 15 for BCSSTK24 on 32 processors. The reasons are explained as follows. First amalgamation creates bigger chunks of data objects which improves the cache performance. Second, it also reduces the number of tasks and number of messages. An interesting phenomenon is that the more processors are used, the bigger the improvement is. A partial explanation is that the more processors are used, the more inter-processors communication edges occur in the schedule and the more chance contention will happen in a wormhole routed network. Reducing the number of messages potentially alleviates the network contention problem.

6 Conclusions

In this paper, we have presented a run-time system for parallelizing sparse/irregular code and discussed its library primitives and run-time optimization techniques. Our experiments with sparse Cholesky and LU factorizations show that the system can successfully explore irregular parallelism, utilize the flexibility of commuting operations, and obtain good performance for these challenging problems. Adjusting task granularities can further improve the performance substantially.

Acknowledgment

We thank Ed Rothberg for his insightful comments and providing supernode amalgamation code, Vivek Sarkar and Rob Schreiber for their encouragement and valuable discussions on this work, Xiaoye Li for providing the unsymmetric matrices. We also thank Yizhou Yu and anonymous referees for useful comments and helps.

References

[1] 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.

[2] M. Cosnard and M. Loi. Automatic Task Graph Generation Techniques. *Parallel Processing Letters*, 5(4):527–538, December 1995.

[3] 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.

[4] 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.

[5] J. Demmel, S. Eisenstat, J. Gilbert, X. Li, and J. Liu. A Supernodal Approach to Sparse Partial Pivoting. Technical Report CSD-95-883, UC Berkeley, September 1995.

[6] C. Fu and T. Yang. Efficient Run-time Support for Irregular Task Computations with Mixed Granularities. In *Proceedings of IEEE International Parallel Processing Symposium*, Hawaii, April 1996.

[7] C. Fu and T. Yang. Run-time Compilation for Parallel Sparse Matrix Computations. Technical Report TRCS96-01, UCSB, 1996.

[8] 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.

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

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

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

[12] M. Rinard. Communication Optimizations for Parallel Computing Using Data Access Information . In *Proceedings of Supercomputing*, San Diego, December 1995.

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

[14] E. Rothberg and R. Schreiber. Efficient Parallel Sparse Cholesky Factorization . In *Proc. of Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 407–412, February 1995.

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

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] T. Yang and A. Gerasoulis. DSC: Scheduling Parallel Tasks on An Unbounded Number of Processors . *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994. A short version is Proc. of Supercomputing '91.