Space and Time Efficient Execution of Parallel Irregular Computations*

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

Abstract

Solving problems of large sizes is an important goal for parallel machines with multiple CPU and memory resources. In this paper, issues of efficient execution of overhead-sensitive parallel irregular computation under memory constraints are addressed. The irregular parallelism is modeled by task dependence graphs with mixed granularities. The trade-off in achieving both time and space efficiency is investigated. The main difficulty of designing efficient run-time system support is caused by the use of fast communication primitives available on modern parallel architectures. A run-time active memory management scheme and new scheduling techniques are proposed to improve memory utilization while retaining good time efficiency, and a theoretical analysis on correctness and performance is provided. This work is implemented in the context of RAPID system [5] which provides run-time support for parallelizing irregular code on distributed memory machines and the effectiveness of the proposed techniques is verified on sparse Cholesky and LU factorization with partial pivoting. The experimental results on Cray-T3D show that solvable problem sizes can be increased substantially under limited memory capacities and the loss of execution efficiency caused by the extra memory managing overhead is reasonable.

1 Introduction

People resort to parallel machines for two reasons: 1) a single CPU does not have enough computation power, and/or 2) a sequential machine does not have enough memory to hold the whole data and code for an application. An enormous amount of effort in parallel system research has been spent on time-efficient parallelizations. This paper addresses issues of efficient parallel execution of irregular computation under a limited memory capacity on each processor and investigates the trade-off between time and space efficiency since a time-efficient parallelization may lead to extra memory space requirements. The model for representing parallelism originates from directed acyclic data dependence graphs with mixed granularities which can be derived from partitioned codes. The applications that have been used to demonstrate the effectiveness of this parallelism model are sparse matrix problems [5, 17]. The main difficulties for parallelizing sparse code are that the granularity of computation and communication is non-uniform, dependence structures may change dynamically and code performance is sensitive to the software overhead introduced by system support. The important techniques for dealing with this class of irregular problems are scheduling optimization for exploiting data locality and achieving load balancing, and fast direct remote memory accessing for low-overhead asynchronous communications (e.g., a few microseconds). However all these optimization techniques impose extra memory space requirements. For example, remote addresses need to be known in advance at the time of performing direct memory accesses and the accessible remote space must be allocated in advance. A parallel program would have its per processor space complexity as high as that of a sequential program. Thus using advanced hardware support for fast communication adds difficulties in designing software layers to achieve high utilization for both processor and memory resources.

In this paper, we present a new technique called *active memory management* which incrementally allocates necessary space that each processor needs, notifies collaborating processors of the allocated object addresses and recycles space by deallocating those obsolete data objects. The main challenge is how the allocation activities can be efficiently integrated into an on-going irregular computation execution, without significantly lowering the time efficiency and creating deadlock situations. Since there will be several dynamic space allocation points during a computation, an important optimization is to reduce the number of allocation points, hence to minimize memory management overhead. We present scheduling techniques for optimizing memory utilization and minimizing the potential overhead for space allocation and address notification. The proposed techniques are implemented in the RAPID software tool [5] which parallelizes irregular applications at run-time. With a carefully designed communication protocol that utilizes direct memory access, RAPID delivers good performance for sparse Cholesky and LU with pivoting [5, 6]. We have experienced that the sizes of problems that RAPID can solve are restricted by the available amount of memory, and experiments with incorporating the proposed memory optimizing techniques show that solvable problem sizes with limited memory space can be increased substantially without pay-

*This was supported in part by NSF RIA CCR-9409695 and by ARPA contract DABT-63-93-C-0064.

ing too much extra control overhead.

Our work uses hardware support for directly accessing remote memory, which is available in several modern parallel architectures and workstation clusters [10, 18]. Benefits of the direct remote memory access mechanism are also identified in the fast communication research such as active messages. Thus we expect other software system researchers can also benefit from our results in using fast communication support to design software layers. Most of previous research on scheduling [16, 19, 20] does not address memory issues. In [1], a dynamic scheduling algorithm for directed acyclic graphs is proposed with memory space usage $S_1/p + O(D)$ on each processor, where S_1 is the sequential space requirement, p is the total number of processors and D is the depth of a DAG. This work provides a solid theoretical ground for space-efficient scheduling, and it is still an open research problem how to integrate their techniques in practical systems as indicated by the authors. Their space model is different from ours and assumes a globally shared memory pool. The Cilk [2] run-time system addresses the space efficiency issue and its space complexity is $O(S_1)$ per processor. The RAPID [5] uses at most S_1 space per processor. This paper assumes that each processor has a maximum space limit and the goal is to make the data space cost to be close to S_1/p per processor in order to solve large-scale problems. The scheduling scheme we use is static in the run-time preprocessing stage while [1] and [2] use dynamic scheduling. This is mainly because in practice it is difficult to minimize the run-time control overhead of dynamic scheduling in parallelizing sparse code with mixed granularities.

It should be noted that there exists other space overhead which includes the space for the operating system kernel, hash tables for indexing irregular objects, task dependence graphs etc. This paper focuses on the optimization of space usage dedicated to storing the content of data objects. The rest of the paper is organized as follows. Section 2 describes our parallelism and memory model. Section 3 presents the active memory management scheme. Section 4 presents memory efficient scheduling heuristics that reduce memory requirements. Section 5 gives experimental results.

2 The computation and memory model

The computation model we use consists of a set of *tasks* and a set of distinct *data objects*. Each task reads/writes a subset of data objects. Data dependence graphs (DDG) derived from partitioned code for modeling the interaction among tasks normally have three types of dependencies: true, anti and output [12]. In a DDG, some of anti or output dependence edges could be redundant if they are subsumed by other true data dependence edges. Other anti/output dependence edges can be eliminated by program transformation. A transformed dependence graph contains true dependencies only. An extension to the classical task graph model is that commuting tasks can be marked in a task graph so that it can capture parallelism arising from commutative operations. The details on this parallelism model are in [5, 7] and this paper deals with scheduling and execution of a transformed task graph with an acyclic structure (DAG).

The proposed memory optimizing techniques are intended for executing general task parallelism. The experiments are conducted in the context of RAPID [5] which is a run-time system that uses an inspector/executor approach [15] to parallelize irregular computations by embodying graph scheduling techniques to optimize interleaved communication and computation with mixed granularities. Its API includes a set of library functions for specifying irregular data objects and tasks that access these objects. The system then extracts a task dependence graph from data access patterns, and executes tasks efficiently on a distributed memory machine. The goal is to reduce programmers' job for solving irregular problems on distributed memory machines without compromising on performance. Figure 1 shows the run-time parallelization process in RAPID. Each circle is an action performed by the system and boxes on both sides of a circle represent the input and output of the action. RAPID is targeted at irregular applications which involve iterative computation and have invariant or slowly changed dependence structures, such as those in sparse matrix computation and N-body galaxy simulations [8, 17]. The task communication protocol of RAPID is optimized to have low overhead and as a result, the RAPID is able to deliver good performance for sparse code such as Cholesky factorization and triangular solvers. It also produces reasonable performance for sparse Gaussian Elimination with partial pivoting [6] which is an open parallelization problem in the literature. We have also used this system in parallelizing Newton's method to solve nonlinear systems.

We define some terms used in the task parallelism model as follows.

Definition 1 *Given a DAG G, a static* **schedule** *on p processors defines an execution order of tasks on each processor. Each data object m is assigned to a unique* **owner** *processor.*

Definition 2 *The set of tasks on a processor P_x, denoted as $TA(P_x)$, will read/write a subset of data objects, denoted as $DO(P_x)$.*

Data objects in $DO(P_x)$ are differentiated in the following way.

Definition 3 *For each data object $m \in DO(P_x)$, if P_x owns m, then m is called a* **permanent object** *of P_x; otherwise m is called a* **volatile object** *of P_x. Define $PERM(P_x)$ as the set of permanent data objects on processor P_x and $VOLA(P_x)$ as the set of volatile data objects on processor P_x.*

Permanent objects will stay allocated during a whole computation on their owner processors.

Figure 2(a) shows a DAG with 20 tasks and 11 data objects $d_1, d_2, \cdots, d_{11}$. Each task is notated in a format of either $T[i, j]$, which means a task that reads d_i and updates d_j, or $T[j]$ which means a task that updates d_j. A cyclic mapping of data objects is used, i.e., the owner of data object d_i is processor $(i-1) \ mod \ p$, where $p = 2$, $i = 1, 2, \cdots, 11$. And the owner-compute rule is used to form task clusters. Therefore $PERM(P_0) = \{d_1, d_3, d_5, d_7, d_9, d_{11}\}$, and $PERM(P_1) = \{d_2, d_4, d_6, d_8, d_{10}\}$. It is also easy to see that $VOLA(P_0) = \{d_8\}$, $VOLA(P_1) = \{d_1, d_3, d_5, d_7\}$. Parts (b) and (c) of Figure 2 are two schedules for the DAG in (a). We assume that each task and each message cost one unit of time. Messages are sent asynchronously as illustrated in part (b) and the processor overhead for sending/receiving messages is not included in the Gantt charts. It should be noted that a static scheduling algorithm provides guidances to minimize parallel times, but the prediction may not be accurate depending on weight variations and other overhead in a run-time execution.

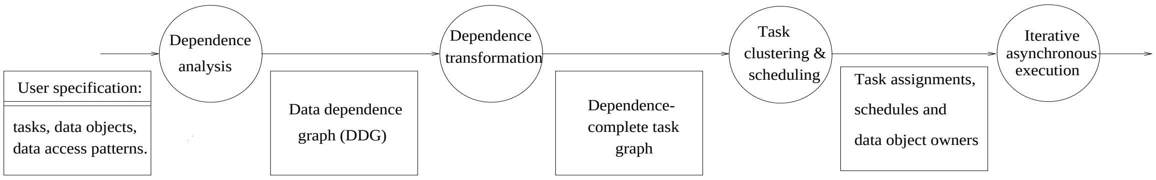


Figure 1: The stages of run-time parallelization in RAPID.

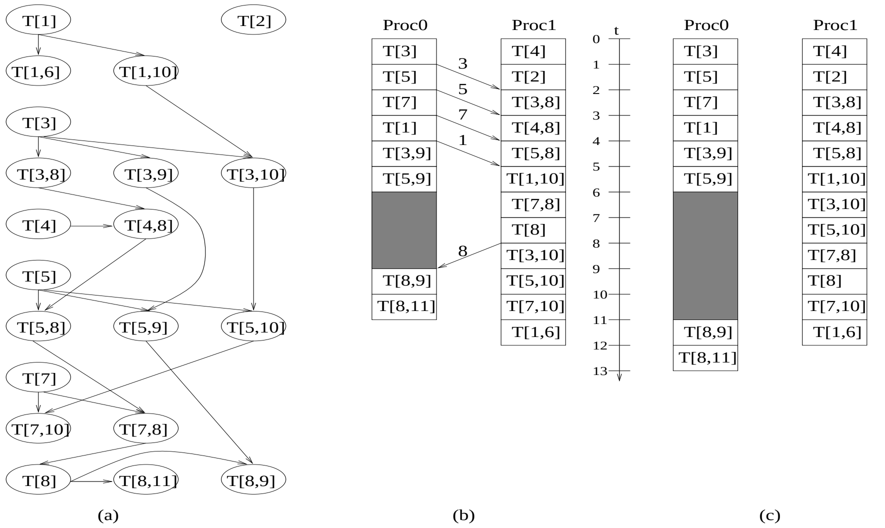


Figure 2: (a) A DAG; (b) A schedule for the DAG on 2 processors; (c) Another schedule.

3 Active memory management

3.1 Basic ideas

Naturally if memory space is not sufficient to hold all data objects, space recycling for volatile data will be necessary. To maintain and reuse data space is not a new research issue, but it is complicated by two supporting techniques for efficient irregular computation with mixed granularities. They are: 1) Integrating buffer space with user space to achieve low overhead communication via direct remote memory access (RMA) and avoid copying/buffering overhead. 2) Exploiting irregular parallelism via scheduler-guided data presending and asynchronous computation execution. The first technique requires hardware support to directly deposit data from user space on one processor to another without buffering and hand-shaking overhead. RMA is available in many modern parallel architectures and workstation clusters [10], but it requires that remote data addresses be known in advance. The second technique for exploiting asynchronous irregular parallelism prevents us from discovering any regularity of space usage. In the original RAPID implementation [5], each processor allocates its volatile space at once and notifies object addresses to collaborating processors. As a result, the volatile space is quite large. For example, in our previous sparse Cholesky factorization experiments [5], the size of volatile object space could be 5 times as high as the size of permanent object space. Table 1 shows some typical ratios of the average amount of space used by both volatile and permanent objects on each processor versus the lower bound of space usage: S_1/p. As the number of processors increases, the ratio is getting larger because more inter-processor messages are needed and each processor owns less permanent objects.

#processor	2	4	8	16
ratio	1.88	3.19	4.64	5.72

Table 1: Average ratios of per processor memory usage over S_1/p for sparse Cholesky factorization experiments.

We propose an approach called *active memory management* to reduce the space needed for volatile objects. The basic idea is simple. During an execution, each processor repeats the following steps: predicts data usage in the near future, allocates necessary space, notifies new data addresses to collaborating processors and deallocates data objects when they become obsolete. The addresses are usually notified prior to when the corresponding data objects are needed so that data pre-sending is allowed. In order to support direct remote memory access and asynchronous execution, the following difficulties must be addressed:

- **Address consistency.** An address for a data object at one processor may become stale when this copy of the data object becomes dead at that processor. The address then must be invalidated on other processors since this data object may have a new address at that processor. Data renaming would avoid this problem [4], but it creates more complexity in indexing data objects and memory optimization.
- **Address buffering.** We will also use the RMA fea-

ture to transfer addresses. Without providing address buffering, a processor cannot re-send address information unless the destination processor has read the previous address package. With address buffering, additional overhead is needed for managing address buffers.

- **Deadlock.** Because a message sending may be suspended if the destination address is not available, the remote processor expecting this message may enter a spinning cycle, which may create a deadlock. The traditional send/receive deadlock issue is complicated by the address availability problem.
- **Data consistency.** When a remote message sending is suspended, the local content of the message may be modified by a subsequent computation or by other processors before it is actually sent out. At the receiving site, the processor only checks the availability of the data object based on its name. Is it possible to receive a wrong copy?

3.2 Our approach

We briefly discuss our approach in dealing with the aforementioned difficulties.

- For address consistency, we could use a classical cache coherence protocol such as write-invalidate. But it introduces a substantial amount of overhead to maintain the correctness. We have taken a simple approach in which a volatile object is considered as obsolete if no task will use this object with the same name any more. In this way, a volatile object at each processor will be only allocated once. When it becomes obsolete, no other processors will use the address of this data object anymore. This criterion is weaker than the criterion based on data copies and could lead to a slightly larger memory requirement, but it reduces the complexity of maintaining address consistency.
- For address buffering, since address packages are sent infrequently, in our implementation we will not support address buffering in order to avoid the overhead of buffer managing. Each processor has one buffer space for every other processor in order to receive addresses from them. If a previous address package has not been consumed by a destination processor, the source processor will not be able to send a new address package to this destination processor.
- For deadlock and data consistency, we carefully design the execution protocol and will prove there is no deadlock and data inconsistency.

For the above approach, the space requirement on each processor in our scheme is estimated as follows.

Definition 4 *A volatile object m on a processor is* **alive** *at a position if m is accessed at this position, or if it has been accessed before and will still be accessed after this position. Otherwise m is dead (or called obsolete).*

Definition 5 *For any task T_w on processor P_x (i.e., $T_w \in TA(P_x)$), we compute the* **memory requirement** *at T_w on P_x as:*

$$MEM\_REQ(T_w, P_x) = \sum_{m \in PERM(P_x)} sizeof(m) + \sum_{\substack{m \text{ is alive at } T_w \\ m \in VOLA(P_x)}} sizeof(m).$$

Then the minimum memory requirement of a schedule is:

$$MIN\_MEM = \max_{\forall P_x}\{\max_{T_w \in TA(P_x)}(MEM\_REQ(T_w, P_x))\}.$$

Definition 6 *If the MIN_MEM value for a given schedule is greater than the available memory space per processor, then this schedule is called a* **non-executable schedule** *under the memory constraint.*

In the schedule of Figure 2(b), on processor P_1, the volatile object d_3 is dead after task $T[d_3, d_{10}]$, d_5 is dead after $T[d_5, d_{10}]$. If we assume each data object is of unit size, it is easy to calculate that $MEM\_REQ(T[d_8, d_9], P_0) = 7$, $MEM\_REQ(T[d_7, d_8], P_1) = 9$ and $MIN\_MEM = 9$. However for the schedule in Figure 2(c), the MIN_MEM equals to 8 because the lifetime of volatile objects d_7 and d_3 are disjoint on P_1 so that they can share the same space.

3.3 The execution model

We present our active memory management scheme integrated with the execution model as follows. First we introduce the concept of Memory Allocation Point (MAP). MAPs are positions between two consecutive tasks in the partial task schedule of a processor and are inserted dynamically based on memory space availability. The first MAP is always at the beginning of schedule execution on each processor. Each MAP does the following:

- Deallocate space for those volatile objects that will not be accessed after the current execution point. Instead of dynamically checking if a data object is dead at a certain point, which is quite expensive at run-time, the dead point information can be statically calculated by performing a data flow analysis on a given DAG with a complexity proportional to the size of the graph.
- Allocate volatile space for tasks to be executed after the current point following the execution chain. Assuming that $T_1, T_2, \cdots$ are the rest of tasks at that processor, the allocation will stop after T_k if space for T_{k+1} cannot be allocated. The next MAP will be at the position right before T_{k+1}.
- Assemble address packages for other processors. The address packages for different processors could be different depending on what objects are to be accessed at other processors.

Figure 3(a) illustrates MAPs and address notification in executing the schedule of Figure 2(c). If the available amount of memory is 8 for each processor, then there are 2 units of memory for volatile objects on P_1. In addition to the MAPs at the beginning of two task chains, there is another MAP right after task $T[5, 10]$ on P_1 at which space for d_3 and d_5 will be freed and space for d_7 is allocated. The address for d_7 on P_1 is then notified to P_0. P_0 will send the content of d_7 to P_1 using RMA after it receives the address of d_7.

After being augmented with the MAP device, the system now has five different states of execution:

- **REC.** Waiting for receiving desired data objects. If a processor is in the REC state, it can not proceed unless all the data objects the current task needs are available locally.

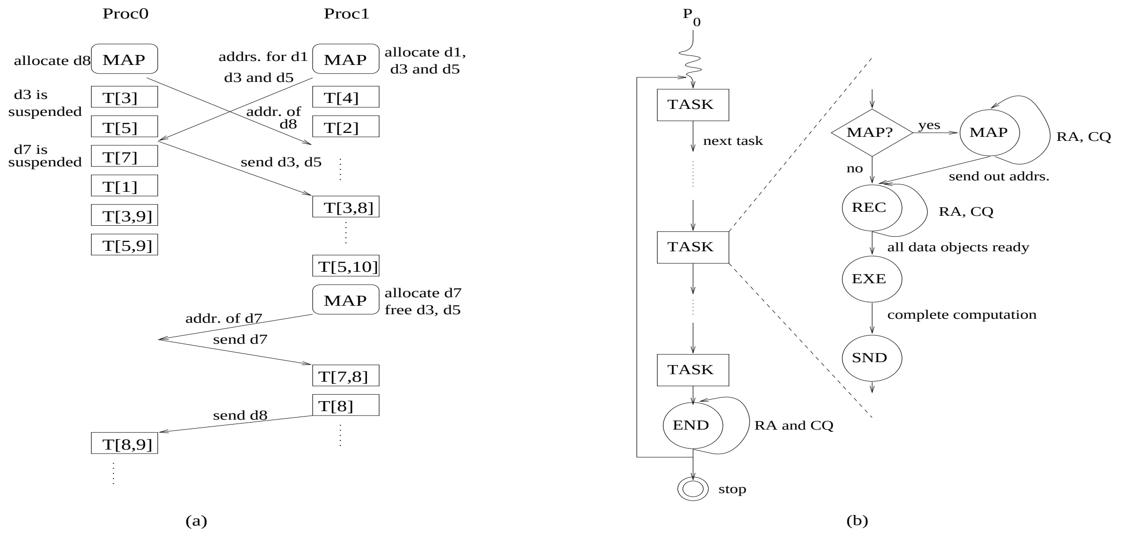


Figure 3: (a) Illustration of memory allocation points for volatile objects in executing the schedule of Figure 2(c); (b)The execution flow with memory allocation points.

- **EXE.** Executing a task computation. EXE is apparently a nonblocking state.
- **SND.** Sending messages. In the SND state, if the remote address of a message is not available, this message is enqueued and the control returns to the normal execution. Of course it is under one assumption that the suspended sending queue should be never overflowed. In the worst case, the queue length is $O(e)$, where e is the number of edges in a task graph.
- **MAP**. Performing an MAP's actions. A processor could be blocked in the MAP state when it attempts to send out address packages to other processors but a previous address package has not been consumed by a destination processor.
- **END**. Entering an ending stage when all the tasks assigned to a processor complete execution. However this processor still needs to clear up the sending queue in order not to hold back any message. A processor in the END state might be blocked if it does not receive addresses for those messages suspended in the sending queue.

The execution flow and the interaction among different states in our scheme are depicted in Figure 3(b). There are three blocking states and two important operations must be conducted in those blocking states: 1) **RA** reads address packages and 2) **CQ** checks the suspended sending queue. Whenever a processor is idle in one of the blocking states, it will invoke RA to see if any new address package arrives and then invoke CQ to dispatch ready messages, i.e., those of which remote addresses are now available. RA and CQ must be conducted frequently to clear the sending queue as soon as possible so that the whole execution can evolve quickly.

3.4 Correctness and deadlock issue

One of the necessary deadlock conditions is circular waiting. In the following theorem, we can show that our execution scheme can prevent deadlocks by breaking up possible circular waiting chains.

Another possibility of incorrectness could be caused by message suspending. Some data objects may not be sent out because their remote addresses are not available and this brings up a question: could they be overwritten by another task, either locally or remotely. We analyze this issue and prove that this will never happen by using the property of a transformed graph called dependence-completeness [5].

Theorem 1 *The execution with the active memory management is deadlock free and has no data inconsistency.*

Proof: First we observe the following fact.

Fact I: If a processor is waiting for receiving a data object, then the local address for this data object must have already been notified to its predecessors.

Now we will prove by induction.

Induction base: At the beginning of each processor's schedule, it must be an MAP. Since at the beginning all the address buffers are available, no processor will be blocked. Also all the entry tasks (i.e., without any predecessor) will start executing.

Induction assumption: There exists a task T_x such that all its parents have been already executed and all tasks scheduled before T_x on the same processor have been executed. We need to show that task T_x will be executed, i.e., it will not be trapped in a deadlock situation.

Suppose not, i.e., a deadlock situation happens, the involved processors must be blocked in a circular waiting chain. According to the state transitions shown in Figure 3(b), there are three possible blocking states in the execution scheme. Therefore, a processor in a deadlock situation must enter one of the three blocking states. Let P_x be T_x's processor. The state of P_x can be either MAP or REC. First let's assume P_x is in the REC state. We discuss the following cases based on possible states of the other involved processors.

Case 1: All the other processors are also in the REC state. According to the induction assumption, all T_x's parents have been executed. The only reason that P_x can not receive a data object for T_x is that this data object has not been sent out from a remote processor P_y. Since all T_x's predecessors are finished, the only cause for P_y not to send a data object out is the un-availability of its remote address on P_x. According to **Fact I**, the address must be already sent out to P_y if P_x is waiting to receive that object. Hence P_y will eventually read that address through RA and deliver the message to P_x. Therefore task T_x will be able to execute.

Case 2: All the other processors are in the MAP state. Since each processor (including P_x which is in the REC state) will invoke RA which frees local address buffers, the remote processors waiting to send addresses to these buffers should be able to continue and transit to the REC state (refer to Figure 3(b)). Therefore the situation will evolve into the same one discussed in **Case 1**.

Case 3: Each involved processor is either in the MAP state or in the REC state. Similar to **Case 2**, a processor in the MAP state will proceed to the REC state eventually, whether other processors in the circular waiting chain are either in the MAP or in the REC state because in both cases, they will invoke RA to release their address buffers. Therefore the situation evolves into the same one as **Case 1**.

Case 4: One or more involved processors are in the END state. For any processor in the END state, it will send all the suspended messages out because of **Fact I** and get out of the circular waiting chain. Therefore, the remaining chain only includes processors in either the MAP state or the REC state. And from the previous cases, we know that the chain will be eventually broken, and the execution will evolve forward.

If P_x is in the MAP state, since it will proceed to REC state eventually, the proof is similar to the above.

As for the data consistency, the proof is similar to the one in [7] given the task graph is dependence complete. ∎

Apparently the main overhead for execution under memory constraint in this scheme is caused by the insertion of MAPs. Minimizing memory requirements can reduce the number of MAPs, which would reduce the execution overhead. However it requires reordering tasks in an execution. In the next section we will discuss the trade-off between memory and time-efficient scheduling optimizations.

4 Space and time efficient scheduling

In [5] a time-efficient scheduling algorithm which contains a two-stage mapping process has been used. At the first stage tasks are clustered to exploit data locality using DSC [21] or the owner-compute rule, i.e., all the tasks that modify the same object are assigned to the same cluster. Clusters are then mapped to physical processors using a load balancing criterion. For simplicity of the description, *we assume that each task modifies only one object in this section.* The second stage is to order tasks on each processor to overlap communication with computation so that maximum inter-processor parallelism is explored. This ordering algorithm is called RCP [20]. A RCP schedule is time efficient, but may not be space-efficient because it executes tasks in the order of importance based on the critical path information, which may require more memory to hold volatile objects.

We discuss two heuristics to improve memory utilization while maintaining a good time efficiency as much as possible. The idea is to have volatile objects referenced as early as possible once they are available in the local memory. This shortens the lifetime of volatile objects and potentially reduces the memory requirement on each processor.

4.1 MPO: Memory-priority guided ordering

In this approach, tasks are first clustered and mapped to processors according to the strategies described above. At the second stage, tasks are ordered on each processor by using the MPO heuristic described below. This heuristic simulates the execution of tasks following task dependencies. A task during scheduling is called *ready* if its predecessors have been executed already and the needed data objects can be received at this point. At the beginning, no volatile object is allocated. When a task is chosen to be scheduled, all volatile objects this task needs are allocated. At each scheduling cycle, each processor selects and executes a ready task with the highest *memory priority*. We define the memory priority of a task as the number of objects which have been allocated for this task versus the total number of objects needed to execute the task. If there is a tie, the task with the highest critical path priority, i.e., the length of the longest path from this task to an exit task, is selected. The goal of the heuristic is to produce a schedule that reuses volatile objects as soon as possible. The ordering algorithm is listed in Figure 4. The main difficulty in designing this algorithm is to keep a low complexity in updating the memory priorities of unscheduled tasks. At line `(5)`, it is enough to update the memory priorities of the children and the siblings of the newly scheduled task because only those tasks are possible candidates to become ready tasks in the next round. The memory priorities of the children of these candidate tasks are to be updated in the future after the candidate tasks are scheduled. The total time complexity of the scheduling algorithm is $O(ve)$ where v is the number of tasks and e is the number of dependence edges.

For example, Figure 2(b) is a schedule produced by RCP while (c) is a schedule produced by MPO. The ordering difference between Figure 2(b) and (c) is that on processor 1, $T[7,8]$ is executed at time 6 by RCP while $T[3,10]$ is chosen instead at time 6 by MPO. The reason is that for RCP, $T[7,8]$ has a longer path from this task to an exit task (the path is $T[7,8], T[8], T[8,9]$ with length 4 because communication delay is also included) than other unscheduled tasks on P_1. For MPO, $T[3,10]$ has a higher memory priority 1 because data d_3 and d_{10} are all available locally at time 6, and $T[7,8]$'s memory priority is 0.5 because the space for d_7 has not been allocated before time 6. As a result, the MPO schedule has a less memory requirement but leads to a longer parallel time.

4.2 DTS: Data-access directed time-slicing

DTS is a more aggressive task ordering algorithm in optimizing space when memory usage is of primary importance. The design is based on the fact that the memory usage of a processor can be optimized if each volatile object has a short life-span on this processor, namely the time period from the allocation of this object to the deallocation of this object is short. According to this principle, the heuristic for ordering tasks on the same processor is to execute tasks that access the same volatile object as close as possible. The basic idea of DTS is to slice the computation graph based on data accessing patterns of tasks so that all tasks within the same slice access a *small* group of volatile objects. Tasks are scheduled on physical processors slice by slice, and tasks within each slice are ordered using dependence and critical path information.

We briefly discuss the algorithm as follows. For a given DAG G in which a set of tasks $V = \{T_1, T_2, \cdots, T_v\}$ operates on a set of data objects $D = \{d_1, d_2, \cdots, d_m\}$, we construct a

```
(1) while there is at least a un-scheduled task
(2)     Find a processor P_x that has the earliest idle time;
(3)     Schedule a ready task T_x that has the highest priority on processor P_x;
(4)     Allocate all volatile objects that T_x uses and that have not been allocated yet on processor P_x;
(5)     Update the priorities of T_x's children and siblings on processor P_x;
(6)     Update the ready task list on each processor;
(7) end-while
```

Figure 4: The MPO algorithm.

data connection graph (DCG) in which each node represents a distinct data object, and each edge represents a temporal order of data accessing during a task computation. A cycle may occur if accesses of two data objects are interleaved. For simplicity, we use the same name for a data object and its corresponding data node when no confusion will be caused. To construct a DCG, the following rules are applied.

- If a task T_x uses but does not modify data object d_i, or T_x only modifies object d_i and does not use any other objects, we associate task T_x with data object node d_i.
- It is possible that a task is associated with multiple data nodes and in this case doubly directed edges are added among those data nodes to make them strongly connected.
- A directed edge is added from data node d_i to data node d_j if there exists a task dependence edge (T_x, T_y) such that T_x is associated with data node d_i and T_y is associated with data node d_j.

The last two rules reflect the temporal order of data accessing during a computation. Then we construct strongly connected components from a DCG and the edges among the components constitute a DAG. A task only appears in one component. Each component is associated with a set of tasks that use/modify data objects in this component, and is considered for scheduling in one slice. At run-time, each processor will execute tasks slice by slice following a topological order of slices imposed by dependencies among corresponding strongly connected components.

It should be noted that a topological order of slices only imposes a constraint on task ordering. A processor assignment of tasks following the owner-compute rule must be supplied before using the DTS to produce an actual schedule. When referring to a DTS schedule below, we assume a task assignment R is implicitly included. In producing a DTS schedule, we use a priority based precedence scheduling approach. Priorities are assigned to tasks based on the slices they belong to. For two ready tasks in the same slice, the task with a higher critical path priority is scheduled first. If there is a ready task that has a slice priority lower than some other unscheduled tasks on the same processor, this task will not be scheduled until all the tasks that have higher slice priorities on this processor are scheduled. By this way we can guarantee that on each processor tasks are executed slice by slice according to the derived slice order. The overall complexity of the DTS algorithm is $O(e(\log v + \log m))$, where e is the number of edges in the original DAG G.

Figure 5 shows an example of DTS ordering for the DAG in Figure 2(a). Part (a) is the DCG and we mark the data name for each node. Since the DCG itself is a DAG, each node is a maximal strongly connected component and is treated as one slice. A topological order of those nodes produces a slice order: $d_1 \to d_3 \to d_4 \to d_5 \to d_7 \to d_8 \to d_2$. Each processor will execute tasks following this slice order as shown in Part (b). The memory requirement MIN_MEM is 7, compared to 9 in Figure 2(b) produced by RCP and 8 in Figure 2(c) by MPO. But the schedule length increases from RCP, through MPO to DTS because less and less critical path information is used.

DTS can lead to good memory utilization. The following theorem gives a memory bound for the DTS algorithm. The corollaries from this theorem on an important class of irregular problems firmly support our design. First a definition is introduced.

Definition 7 *Given any processor assignment R for a slice L, the volatile space requirement on processor P_x, denoted as $V_{P_x}(R, L)$, is defined as the amount of space needed to allocate for the volatile objects used in executing tasks of L on P_x. The maximum volatile space requirement for L under R is then defined as $H(R, L) = \max_{\forall P_x} V_{P_x}(R, L)$.*

Assuming that a task assignment R following the owner-compute rule produces an even distribution of data space for permanent data objects among processors, we can show the following results.

Theorem 2 *Given a DTS schedule on p processors which consists of k slices L_i, $i = 1, 2, \cdots, k$, (assume it is also a valid topological order) and a task assignment R, the schedule is executable under $S_1/p + h$ space per processor, where S_1 is the sequential space complexity, $h = \max_i H(R, L_i)$.*

Proof: First of all, since R leads to an even distribution of permanent data objects, the permanent data space needed on each processor is S_1/p. Suppose a task T_x in slice L_i needs to allocate space for a volatile data object d. If $i = 1$, there should be enough space for d according to the definition of h. If $i > 1$, then we claim that all the space allocated to the volatile data objects associated with slices $L_1, \cdots, L_{i-1}$ can be freed. Therefore the extra h space will be enough for executing tasks in L_i.

Now we need to show the above claim is correct. Suppose not and there is a volatile data object d' that can not be deallocated after slice L_{i-1}, and d' is associated with L_a, $a < i$. Then there is at least one task $T_y \in L_j$, $j \geq i$, that uses d'. If T_y also modifies d', then d' is a permanent data object; if T_y does not modify d', then according to the DTS algorithm, T_y should belong to slice L_a instead of L_j. Thus there is a contradictory. ■

Corollary 1 *If the DCG of a task graph is a DAG and each object is of unit size, the DTS produces a schedule which can be executed on p processors using $S_1/p + 1$ space per processor, where S_1 is the sequential space complexity.*

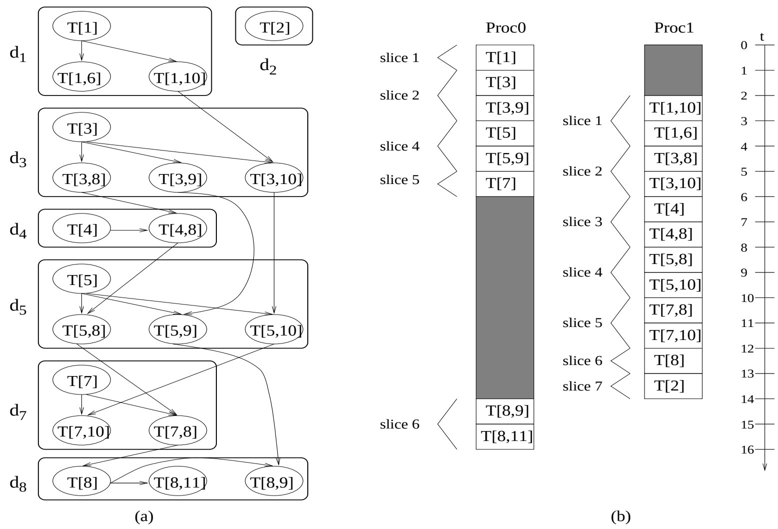


Figure 5: (a) A sample DCG derived from a DAG; (b) A DTS schedule for the DAG on 2 processors.

Proof: If a DCG is acyclic, then each data node in the DCG constitutes a strongly connected component itself. Therefore each slice is associated with only one data object which implies that the h defined in Theorem 2 is 1. Therefore the corollary is proven. ■

Corollary 2 *For the 1-D column-block based sparse LU task graphs [6], a DTS schedule is executable under $S_1/p + w$ space per processor at run-time. For the 2-D block based sparse Cholesky graphs [5], a DTS schedule is executable under $S_1/p + w$ space per processor. Here w is the size of the largest column block of the partitioned input matrix and normally $w << S_1/p$.*

Proof: (sketch) The DTS algorithm produces acyclic DCGs for the 1-D column-block based sparse LU task graphs. Each data object is a column block of size at most w. According to Corollary 1, each processor will at most need w volatile space to execute a DTS schedule for sparse LU.

In the 2-D block based sparse Cholesky approach [5], task graphs can be structured layer by layer. Each layer represents the elimination process on the submatrix starting from column block k. Let N be the number of blocks in both dimensions of the input matrix. At elimination step k, the Cholesky factor computed from the diagonal block A_{kk} will be used to scale all the blocks in the kth column block, i.e., A_{ik}'s ($k < i \leq N$). And the A_{ik}'s will then be used to update the rest of matrix, i.e., A_{ij}'s, $k < i \leq N$, $k < j \leq i$. All these updating tasks at step k belong to the same slice associated with data objects A_{ik}'s ($k < i \leq N$). Hence the extra space needed to execute a slice is the summation of the blocks in column block k. According to Theorem 2, a DTS schedule for the Cholesky can execute with $S_1/p + w$ space on each processor. ■

If the available memory space for each processor is known, say $AVAIL\_MEM$, the time efficiency of the DTS algorithm can be further optimized by merging several consecutive slices if memory is sufficient for those slices, and then applying the priority based scheduling algorithm on the merged slices. Assuming there are k slices and a valid slice order is $L_1, L_2, \cdots, L_k$, for a given task assignment R, the merging strategy is summarized in Figure 6. A set of new slices $L'_1, L'_2, \cdots, L'_{k'}$ will be generated. The complexity of the merging process is $O(v)$.

```
space_req = H(R, L_1);  k' = 1;
Let slice L'_1 be L_1;
for i=2 to k do
     if space_req + H(R, L_i) <= AVAIL_MEM
          Merge L_i to L'_{k'};
          space_req = space_req + H(R, L_i);
     else
          k' = k' + 1;
          Let slice L'_{k'} be L_i;
          space_req = H(R, L_i);
     endif
end-for
```

Figure 6: The DTS slice merging algorithm.

5 Experimental results

We have implemented the proposed active memory management scheme and the memory efficient scheduling heuristics in the context of RAPID system on Cray-T3D and Meiko CS-2. In this section we will report the performance of our approach on T3D for two irregular codes. 1) Sparse Cholesky factorization. The task graph has a static dependence structure if the nonzero pattern of a sparse matrix is given at the run-time preprocessing stage. A 2-D block data mapping is used, which can expose more parallelism and give better scalability [14]. 2) Sparse LU (Gaussian Elimination) with partial pivoting. This problem has an unpredictable dynamic dependence structure. Its parallelizations on shared memory platforms are addressed in [11]. But its efficient parallelization on *distributed memory machines* still remains an open problem in the scientific computing literature. We use a static symbolic factorization approach to avoid the data structure variation and 1-D data mapping to

Percentage	100%	75%		50%		40%	
	PT increase	#MAPs	PT increase	#MAPs	PT increase	#MAPs	PT increase
P=2	3.8%	3.75	7.7%	∞	∞	∞	∞
P=4	12.0%	2.00	18.5%	7.38	33.6%	∞	∞
P=8	12.4%	2.00	25.3%	3.44	33.7%	5.32	51.4%
P=16	17.6%	2.00	39.0%	2.97	45.7%	3.85	56.8%
P=32	22.0%	1.98	42.1%	2.35	61.3%	3.16	65.1%

Table 2: Effectiveness of the run-time execution scheme for sparse Cholesky on Cray-T3D.

Percentage	100%	75%		50%		40%	
	PT increase	#MAPs	PT increase	#MAPs	PT increase	#MAPs	PT increase
P=2	0%	∞	∞	∞	∞	∞	∞
P=4	0.4%	3.50	15.5%	∞	∞	∞	∞
P=8	1%	2.00	11.1%	5.63	37.5%	∞	∞
P=16	1.4%	2.00	18.3%	2.94	18.1%	4.00	32.2%
P=32	2.1%	1.72	13.8%	2.38	15.6%	3.06	16.7%

Table 3: Effectiveness of the run-time execution scheme for sparse LU on Cray-T3D.

eliminate communication in partial pivoting and row swapping operations [6]. Each node of T3D has 64 MB memory and can reach 103 MFLOPS with the BLAS-3 DGEMM routine. The RMA primitive SHMEM_PUT can achieve $2.7\mu s$ overhead with 128 MB/s bandwidth.

We will examine how the memory managing scheme impacts the parallel performance if the available amount of memory space is short and multiple MAPs are needed, and study the effectiveness of scheduling heuristics in reducing memory requirements. We have conducted experiments on a number of testing sparse matrices. The representative matrices are Harwell-Boeing matrices BCSSTK15 and BCSSTK24 arising from structural engineering analysis, and "goodwin" matrix from a fluid mechanics problem. These matrices are of medium sizes (dimensions ranging from 3500 to 7320) and solvable with any one of the three scheduling heuristics so that we can compare their performance. Experiments with other matrices reach similar conclusions.

5.1 Overhead of the active memory management scheme

We examine the overhead of the memory managing scheme under different memory constraints by manually controlling the available memory space on each processor to be 75%, 50%, 40% and 25% of TOT, where TOT is the total memory space needed for a given task schedule without any space recycling. To obtain TOT, we first calculate the summation of the space for permanent and volatile objects accessed on each processor, and then let TOT be the maximum value among all processors.

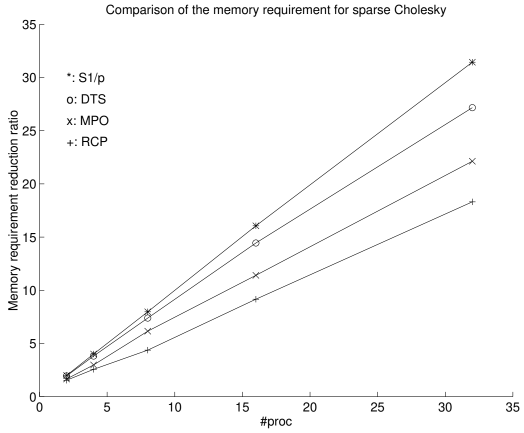
The average results for sparse Cholesky factorization with BCSSTK15 and 24 are listed in Table 2. The RCP ordering is used in this case. Column "PT increase" is the ratio of parallel time increase after using our memory management scheme under different memory constraints. The comparison base is the parallel time of a RCP schedule with 100% memory available and without any memory managing overhead. The same setting applies to the sparse LU experiments in Table 3. Entries marked with "∞" imply that the schedule is non-executable under that memory constraint. And the numbers in the column of #MAP are average numbers of MAPs on each processor. In cases they are fractional, it is because different processors may have different number of MAPs. The results basically show the trend that the performance degradation increases as the number of processors increases and as the available memory space decreases, but the degradation ratio is reasonable comparing to the amount of memory saved. For example, the memory managing scheme can save 60% memory space while the parallel times are degraded by 51-65%.

It can be observed that a schedule is more likely to be executable under reduced memory capacity when the number of processors increases. This is because more processors lead to more volatile objects on each processor, which gives the memory scheme more flexibility to allocate and deallocate. That is why even with 40% of the maximum memory requirement, the schedules are still executable on 8, 16 and 32 processors.

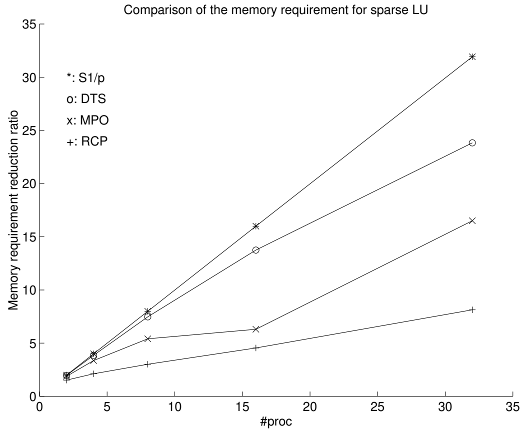
Table 3 lists the run-time performance for sparse LU with pivoting on the "goodwin" matrix. The overall parallel time slowdown of sparse LU is better than that of sparse Cholesky (refer to Table 2). The parallel times only increase about 17-32% when only 40% of the total needed memory is available. The reason is that the 1-D data mapping results in a less number of data objects and computation tasks, which leads to less memory managing overhead because the memory managing overhead increases with the number of data objects and tasks. The 1-D data mapping also produces relatively coarser gained computation. Therefore the overall time is less sensitive to the overhead caused by memory managing activities. The 1-D data mapping also accounts for more "∞" entries appeared in this table because it produces larger data objects which reduces the freedom of allocation within a certain amount of memory space.

5.2 Effectiveness and comparisons of the memory scheduling heuristics

We examine the memory efficient scheduling heuristics from two aspects. First we examine how much memory can be saved by using MPO and DTS. We define *memory scalability* (or *memory reduction ratio*) as S_1/S_p^A, where S_1 is the sequential space requirement, S_p^A is the per processor space requirement for a schedule produced by algorithm A on p processors. Figure 7 shows the memory reduction ratios for the three scheduling algorithms and the upper-most curve in each graph is for S_1/p, which is the perfect memory scalability. It can be seen that the MPO significantly reduces the memory requirement while DTS has the memory requirement close to the optimum. This is consistent with Corollaries 1 and 2. On the other hand, although RCP is very time efficient, it is not memory scalable, particularly for sparse LU.



(a)



(b)

Figure 7: Memory scalability comparison of the three scheduling heuristics. (a) Sparse Cholesky; (b) Sparse LU.

The second issue is how time-efficient the memory-efficient schedules are. Tables 4, 6 and 7 compare parallel execution times between different scheduling algorithms under different memory constraints. In these tables, if algorithm A is being compared with B (i.e., A vs. B), each entry is calculated as $PT_B/PT_A - 1$. In case an entry is marked by an "*", it implies that the corresponding B schedule is executable under that memory constraint while the corresponding A schedule is not executable. If an entry is marked by a "-", it implies that both A and B schedules are non-executable under that memory constraint.

Table 4 shows that the actual parallel time increases when switching from RCP scheduling to MPO scheduling. The result is surprising. The difference is negligible and sometimes MPO schedules even outperform RCP schedules while the predicted parallel times of RCP are better than those of MPO. This is because even though MPO does not use as much critical path information as RCP does, it reduces the number of MAPs needed (Table 5 shows an example of the reductions of number of MAPs by MPO scheduling heuristic) which in turn can improve the execution efficiency. Furthermore, reusing a data object as soon as possible such as in MPO potentially improves caching performance because it tends to improve temporal locality. These factors are mixed together, making the final parallel execution times of MPO schedules still competitive to those of RCP schedules.

Mem. Perce.	75%	50%	40%	25%
P=2	0.9%	-	-	-
P=4	1.4%	-5.1%	*	-
P=8	2.4%	-0.7%	-1.9%	-
P=16	-1.2%	3.4%	0%	*
P=32	-5.6%	-6.3%	-0.8%	-6.1%

(a)

Mem. Perce.	75%	50%	40%	25%
P=2	*	-	-	-
P=4	-3.8%	-	-	-
P=8	7.2%	-8.8%	*	-
P=16	1.0%	0.1%	5.1%	-
P=32	10.5%	-7.7%	10.0%	*

(b)

Table 4: Parallel execution time comparisons between scheduling algorithms: RCP vs. MPO. (a) For sparse Cholesky; (b) For sparse LU;

Mem. Perce.	75%	50%	40%	25%
P=2	4/3	∞/∞	∞/∞	∞/∞
P=4	2/2	7.8/4	∞/7.3	∞/∞
P=8	2/2	3.3/3	5.3/4	∞/∞
P=16	2/2	3/2.9	3.9/3.3	8.3/6.6
P=32	2/2	2.22/2.19	3/3	5.6/5.2

Table 5: Average number of MAPs for sparse Cholesky: RCP vs. MPO.

DTS is a more aggressive memory-saving algorithm, but it does not utilize the critical path information in slicing the computation. In Table 6, we show the parallel time slowdown between MPO and DTS schedules. It is pretty clear that MPO outperforms DTS substantially in terms of parallel time, even though DTS is more efficient in memory usage. The difference is especially significant when the number of processors is large. This is because MPO not only optimizes the memory usage, but also optimizes parallel time. But there are times we do need DTS, for example, in the LU cases when only 25% memory is available, the DTS schedules are executable on 16 processors, while the MPO schedules are too space costly to run. In addition, the performance difference between two algorithms for LU are bigger than the difference for Cholesky. Again this is because the sparse LU tasks are relatively more coarse grained and they are more sensitive to different task orderings.

However, when the available amount of memory space is known, DTS schedules can be further optimized by the slice merging process discussed in Section 4.2. We have compared

Mem. Perce.	75%	50%	40%	25%
P=2	4.4%	-	-	-
P=4	9.0%	10.9%	3.9%	-
P=8	22.1%	25.4%	28.3%	*
P=16	45.5%	45.6%	47.9%	31.9%
P=32	89.6%	87.4%	86.9%	71.7%

(a)

Mem. Perce.	75%	50%	40%	25%
P=2	2.7%	-	-	-
P=4	20.4%	-	-	-
P=8	43.9%	34.4%	14.4%	-
P=16	60.9%	59.9%	44.7%	*
P=32	115%	116%	94.5%	75.5%

(b)

Table 6: Parallel execution time comparisons between scheduling algorithms: MPO vs. DTS. (a) For sparse Cholesky; (b) For sparse LU.

the RCP to the DTS with slice merging in Table 7. The results are very encouraging. Not only are there many cases in which DTS schedules are executable while RCP schedules are not, but also the parallel times of DTS schedules are very close to those of RCP schedules. This is because merged slices give the scheduler more flexibility in utilizing critical path information and DTS is also effectively improving cache performance. Thus the DTS algorithm with slice merging is very valuable when the problem size is big and the available amount of memory space is known.

Mem. Perce.	75%	50%	40%	25%
P=2	3.5%	-	-	-
P=4	1.0%	-1.1%	*	-
P=8	4.8%	5.5%	6.1%	*
P=16	5.3%	13.8%	9.4%	*
P=32	9.7%	12.1%	13.1%	19.8%

(a)

Mem. Perce.	75%	50%	40%	25%
P=2	*	-	-	-
P=4	-10.3%	-	-	-
P=8	3.9%	-12.7%	*	-
P=16	-9.6%	-2.4%	1.9%	*
P=32	21.6%	12.8%	22.4%	*

(b)

Table 7: Parallel execution time comparisons between scheduling algorithms: RCP vs. DTS with slice merging. (a) For sparse Cholesky; (b) For sparse LU.

5.3 Solving large problems

We demonstrate the performance of the new scheme for solving the previously-unsolvable problem instances on Cray-T3D. In this experiment, we use nonzero patterns of an Harwell-Boeing matrix BCSSTK33. The original RAPID sparse LU code can solve BCSSTK33 if we take data from column/row 1 up to 5600, and produces 419 MFLOPS on 64 processors. The number of nonzero elements (after fill-in, the same below) involved is 3.88 million. The MFLOPS calculation does not include extra floating point operations introduced by the over-estimation of the static factorization scheme [6]. The new system can now solve BCSSTK33 if we take data from column/row 1 up to 6080 with 9.49 millions of non-zeros (i.e., the problem size is increased by 145%). The absolute performance is listed in Table 8. In terms of single node performance, we get 22 MFLOPS per node on 16 nodes and 9.9 MFLOPS per node on 64 nodes. These numbers are pretty good for Cray-T3D considering that the code has been parallelized by a software tool.

#proc	PT(Seconds)	Ave. #MAPs	MFLOPS
16	41.8	5.63	353.1
32	25.9	4.09	569.2
64	23.3	3.78	634.0

Table 8: Performance of sparse LU with partial pivoting on Cray-T3D.

6 Conclusions

Optimizing memory usage is important to solving large parallel scientific applications and the software support is complicated when applications have irregular computation and data access patterns. The main contribution of this work is the development of an efficient active memory managing scheme and scheduling optimization techniques in supporting the use of low-overhead communication primitives available on modern processor architectures. The proposed techniques integrated with the RAPID run-time library system can simplify parallel programming of irregular code to achieve both good time and space efficiency. The theoretical analysis on the correctness and memory performance corroborates the design of our techniques. Experiments with overhead-sensitive sparse matrix codes show that the overhead introduced by the memory managing activities is acceptable comparing to the amount of space saved. The MPO heuristic is competitive to the critical path scheduling algorithm and it delivers good memory and time efficiency. The DTS is more aggressive in memory-saving and achieves competitive time efficiency when slice merging is conducted. DTS is suitable for the situation when space resource is the most critical factor. Another possible application of the DTS algorithm is to guide data placement in memory for shared data objects to improve both spatial and temporal locality. One future work is to extend our results for more complicated task dependence structures [3, 9, 13].

The experiments have also revealed that other space limiting factors, such as the storage of dependence information in our current implementation, affect the ability to process larger problem instances. For the examples we have tested, dependence structures can take from 18% to 50% of the total memory space. Although a complete dependence structure is needed for scheduling at the inspector stage, it is possible to distribute the dependence structure during the executor stage. However we found that space freed from irregular dependence structures usually contains many small pieces and is hard to be re-utilized. To address this fragmentation problem, it is necessary to develop a special memory allocator.

References

[1] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably Efficient Scheduling for Languages with Fine-Grained Parallelism. In *Proceedings of 7th ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, July 1995.

[2] R. Blumfoe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An Efficient Multi-threaded Runtime System. In *Proceedings of Fifth ACM Symposium on Principles and Practice of Parallel Programming*, pages 207–216, July 1995.

[3] S. Chakrabarti, J. Demmel, and K. Yelick. Modeling the Benefits of Mixed Data and Task Parallelism. In *Proceedings of 7th ACM Symposium on Parallel Algorithms and Architectures*, pages 74–83, July 1995.

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

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

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

[7] C. Fu and T. Yang. Run-time Techniques for Exploiting Irregular Task Parallelism on Distributed Memory Architectures. *Journal of Parallel and Distributed Computing*, 1997. Accepted for publication. Also as UCSB technical report TRCS97-03.

[8] A. Gerasoulis, J. Jiao, and T. Yang. Scheduling of Structured and Unstructured Computation . In D. Hsu, A. Rosenberg, and D. Sotteau, editors, *Interconnections Networks and Mappings and Scheduling Parallel Computation*, pages 139–172. American Math. Society, 1995.

[9] M. Girkar and C. Polychronopoulos. Automatic Extraction of Functinal Parallelism from Ordinary Programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–178, 1992.

[10] M. Ibel, K. E. Schauser, C. J. Scheiman, and M. Weis. Implementing Active Messages and Split-C for SCI Clusters and Some Architectural Implications. In *Sixth International Workshop on SCI-based Low-cost/High-performance Computing*, September 1996.

[11] X. Li. *Sparse Gaussian Elimination on High Performance Computers*. PhD thesis, CS, UC Berkeley, 1996.

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

[13] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A Convex Programming Approach for Exploiting Data and Functional Parallelism. In *Proceedings of International Conference on Parallel Processing*, pages 116–125, 1994.

[14] E. Rothberg and R. Schreiber. Improved Load Distribution in Parallel Sparse Cholesky Factorization. In *Proceedings of ACM/IEEE Supercomputing*, pages 783–792, November 1994.

[15] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-Time Scheduling and Execution of Loops on Message Passing Machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

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

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

[18] T. Stricker, J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross. Decoupling Synchronization and Data Transfer in Message Passing Systems of Parallel Computers. In *Proceedings of ACM International Conference on Supercomputing*, pages 1–10, Barcelona, July 1995.

[19] R. Wolski and J. Feo. Program Parititoning for NUMA Multiprocessor Computer Systems. *Journal of Parallel and Distributed Computing*, 1993.

[20] T. Yang and A. Gerasoulis. List Scheduling with and without Communication Delays. *Parallel Computing*, 19:1321–1344, 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 in Proceedings of Supercomputing'91.