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

The RAPID run-time system

RAPID provides an API and run-time support for developing efficient irregular codes [ICS'96]:

1. User specifies shared data objects, tasks and data access patterns.
2. RAPID performs dependence analysis, task scheduling.
3. Its run-time support provides an efficient task execution protocol.

- Use asynchronous fast remote memory access (RMA) for copying/buffering free communication.
- Optimize communication, e.g., eliminating unnecessary synchronizations, and eliminating redundant messages.

Motivation: space optimization

- Original RAPID is not able to solve large-size problems.
- The use of RMA complicates run-time memory management.
- Scheduling trade-off between time and space efficiency.

Related work

- Blelloch et al., SPAA'95, memory bound $S_1/p + O(D)$ using a globally shared memory model.
- Blumfoe et al., PPoPP'95, Cilk, memory complexity $O(S_1)$ on distributed memory machines.

Our goal: per processor space complexity close to S_1/p.

Outline

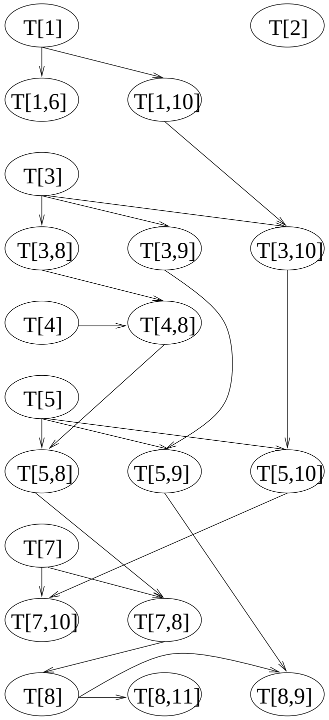
- Memory model.
- Run-time scheme with active memory management.
- Time/space efficient scheduling algorithms.
- Experimental performance evaluation.

The memory model

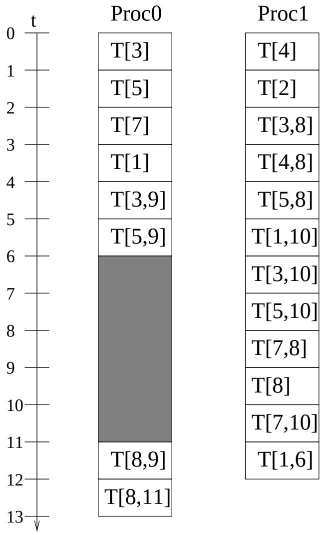
- Each data object has a unique **owner** processor.
- A data object m is called **permanent** on a processor if it owns m, otherwise m is called **volatile**.
- A volatile object m on a processor is **alive** at a position if m is accessed at this position; or if it has been accessed before and will still be accessed after this position. Otherwise m is **dead** (or **obsolete**).
- Memory requirement at a certain point of a schedule: *space of permanent objects* + *space of live volatile objects.*

An example of the memory model

T[i,j]: use d_i to modify d_j; T[i]: modify d_i.



(a)



(b)

$PERM(P_0) = \{d_1, d_3, d_5, d_7, d_9, d_{11}\}$, $VOLA(P_0) = \{d_8\}$,

$PERM(P_1) = \{d_2, d_4, d_6, d_8, d_{10}\}$, $VOLA(P_1) = \{d_1, d_3, d_5, d_7\}$.

For the schedule in (b), MIN_MEM_REQUIREMENT=8.

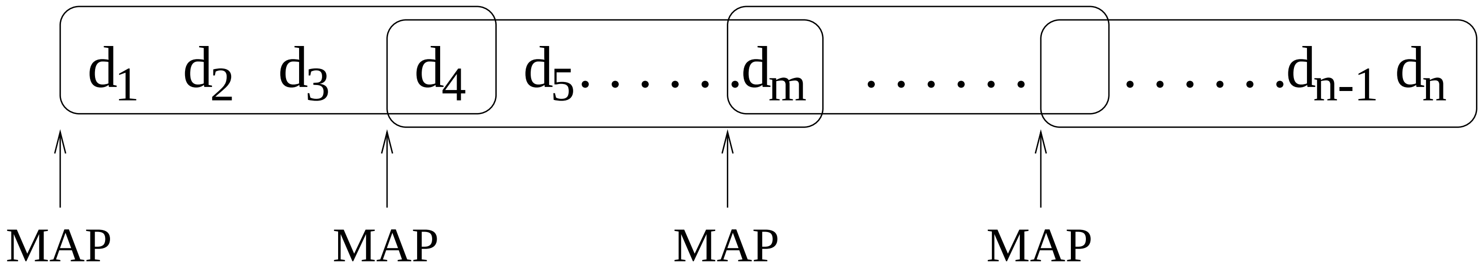
Problem 1: schedule execution

Given a schedule, execute it efficiently under memory constraints.

- Address notification in advance to allow pre-sending.
- Maintain address consistency.
- Address buffering.
- Deadlock issue and data consistency.

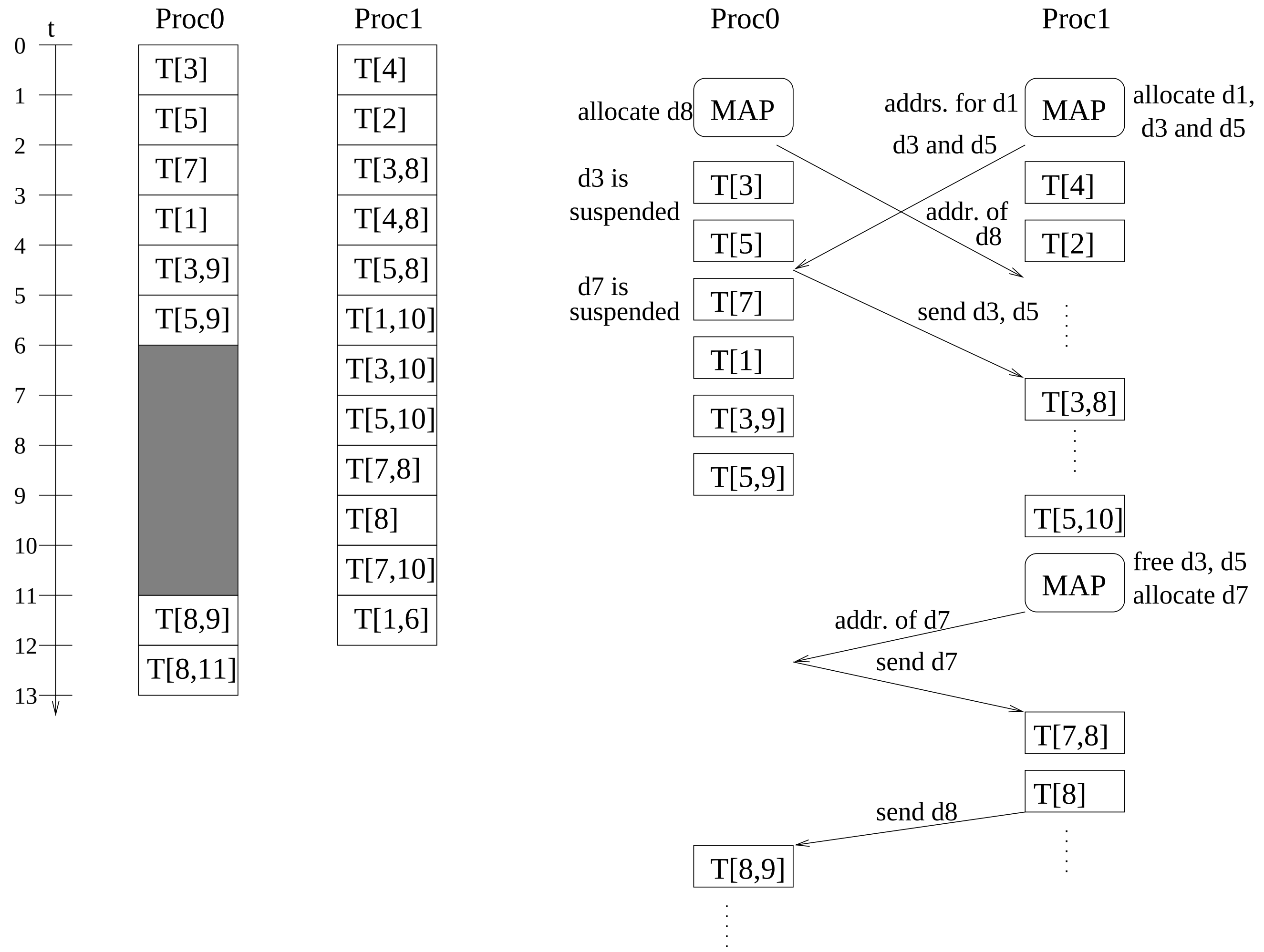
Active memory management

- Execute tasks one by one following the schedule, predict space usage.
- Dynamically insert Memory Allocation Points (MAPs).



- A volatile object at each processor is only allocated once.
- Each processor has one address buffer for every other processor.
- A message is suspended if address unavailable.

A sample execution



(a) (b)

The augmented execution model

States of execution:

∗ **REC:** Receiving;

∗ **EXE:** executing task;

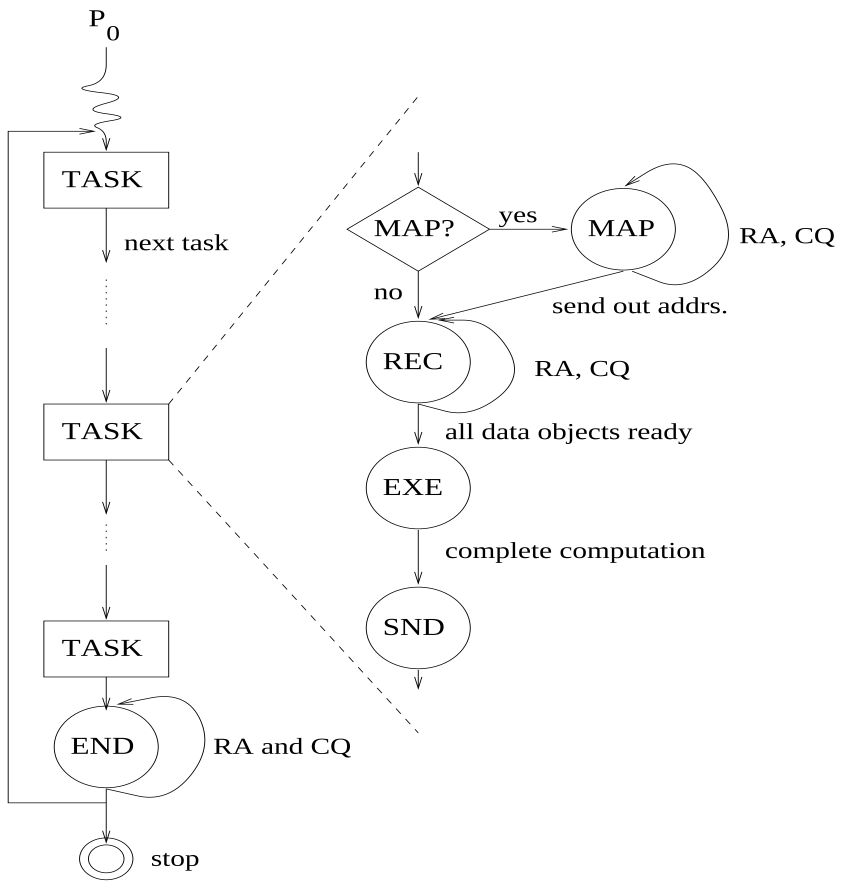
∗ **SND:** Sending;

∗ **MAP:** Does the following;

- Deallocate space of those obsolete volatile objects.
- Allocate for volatile objects to be used in the near future.
- Assemble and pass address packages to collaborating processors.

∗ **END:** Ending stage.

State transition during execution



(b)

* **RA**: Reading Addresses
* **CQ**: Checking the suspended sending queue (CQ).

Correctness

- Address availability issue complicates the traditional deadlock problem.
- Suspended messages may cause data inconsistency.

Theorem 1 *The RAPID execution model with the active memory management scheme is deadlock free and data-consistent.*

Problem 2: schedule generation

Given a DAG, how to generate a schedule that has small memory requirements and good time efficiency as well.

- Reducing number of MAPs.
- Trade-off between time and space efficiency.

Time/space efficient scheduling algorithm

- **Task clustering and processor assignment.** Cluster tasks using owner-compute rule, and map clusters using load balancing criteria.
- **Ordering task execution.**
  - RCP[Yang/Gerasoulis'92]: time-efficient ordering.
  - New algorithms (MPO, DTS): space/time-efficient ordering. Minimize the memory requirement and reduce number of MAPs.

Memory-priority guided ordering (MPO)

Once a volatile object local, reference it ASAP.

(1) **while** there is at least a un-scheduled task
(2)   Find a processor with the earliest idle time;
(3)   Select a ready task with the highest memory priority on that processor;
(4)   Allocate all volatile objects needed by the task;
(5)   Update priorities of sons/siblings;
(6)   Update ready task lists on each processor;
(7) **end-while**

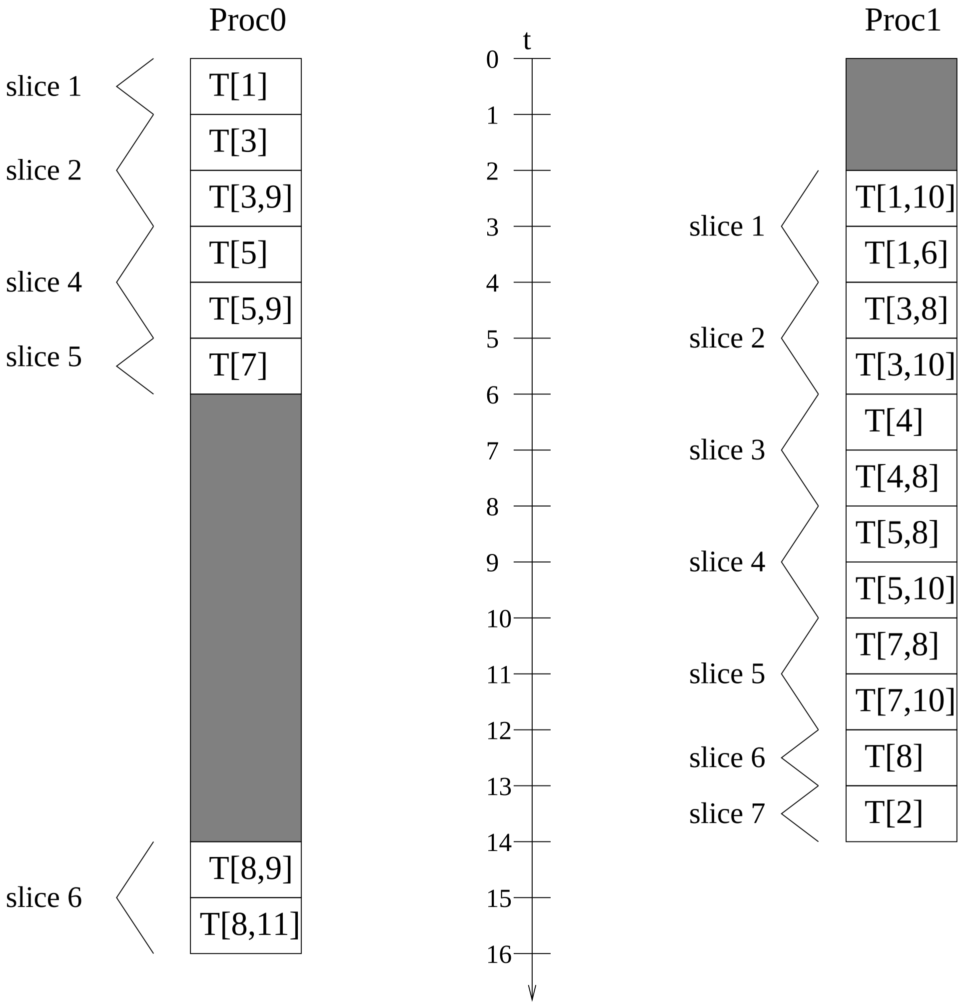
Define **memory priority:** $\frac{\text{no. of objects allocated}}{\text{total no. of objects needed}}$.

Data-access directed time-slicing (DTS)

Basic heuristic: to execute the tasks that access the same volatile object as close as possible.

- Slice the computation such that tasks in the same slice only access a small group of data objects.
- Schedule tasks slice by slice following a slice ordering;
- Within each slice, tasks are ordered based on critical path priority.

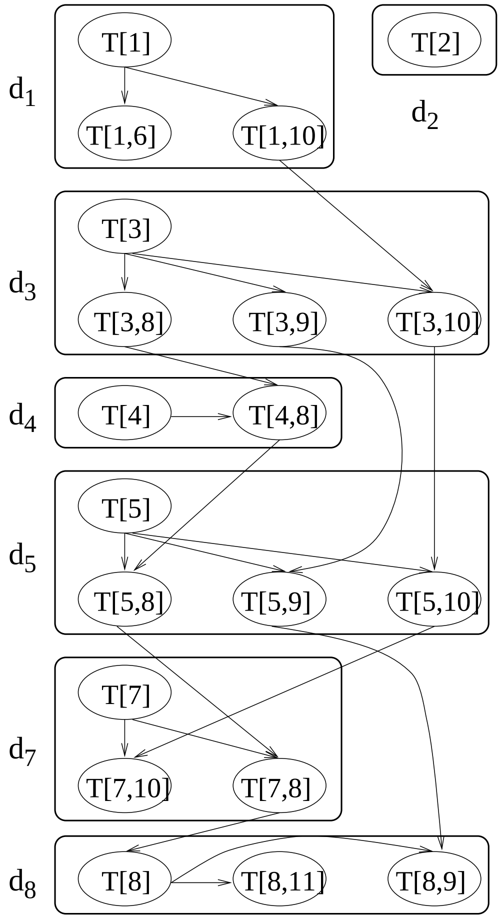
An example of a DTS schedule

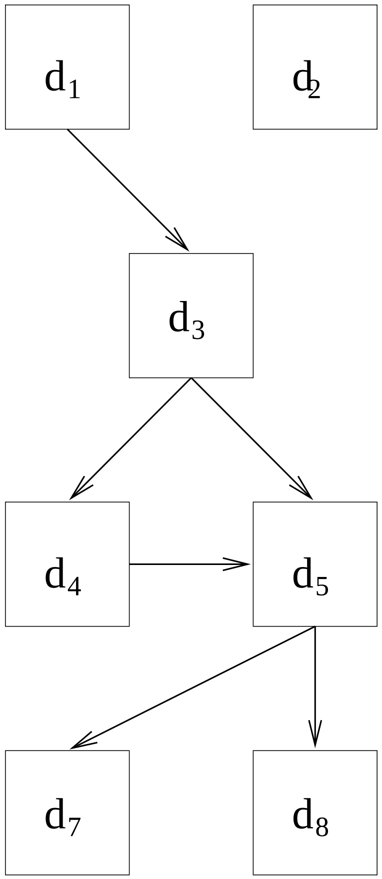


Slicing task computation

1. Build a data connection graph (DCG). Each node in DCG represents a data object;
2. Mapping tasks to data nodes based on the data accessing patterns;
3. Calculate precedence among data nodes based on the temporal accessing order during the computation.
4. Identify the Strongly Connected Components (SCCs) in the DCG. Each SCC is a slice and dependencies among these SCCs define the precedence constraints among slices.
5. Any topological order of SCCs defines a slice ordering.

An example of computation slicing





(a) Task mapping; (b) Data connection graph.

Memory bound analysis of DTS

Assuming permanent data space is evenly distributed on p processors.

Theorem 2 *Given a DTS schedule on p processors, it is executable under $S_1/p + h$ space per processor, where h is the maximum volatile space requirement among all slices.*

<u>Apply to sparse LU and Cholesky</u>, $h << S_1/p$.

Slice merging for DTS

Goal: improve the time efficiency of DTS.

Given $AVAIL\_MEM$, and k slices $(L_1, L_2, \cdots, L_k)$, output k' new slices: $(L'_1, L'_2, \cdots, L'_{k'})$.

Basic idea:

If the memory requirements of several consecutive slices does not exceed $AVAIL\_MEM$, merge them as a new slice.

Advantages:

1. Fully utilize the available memory space.
2. More flexibility to order tasks, better time efficiency.

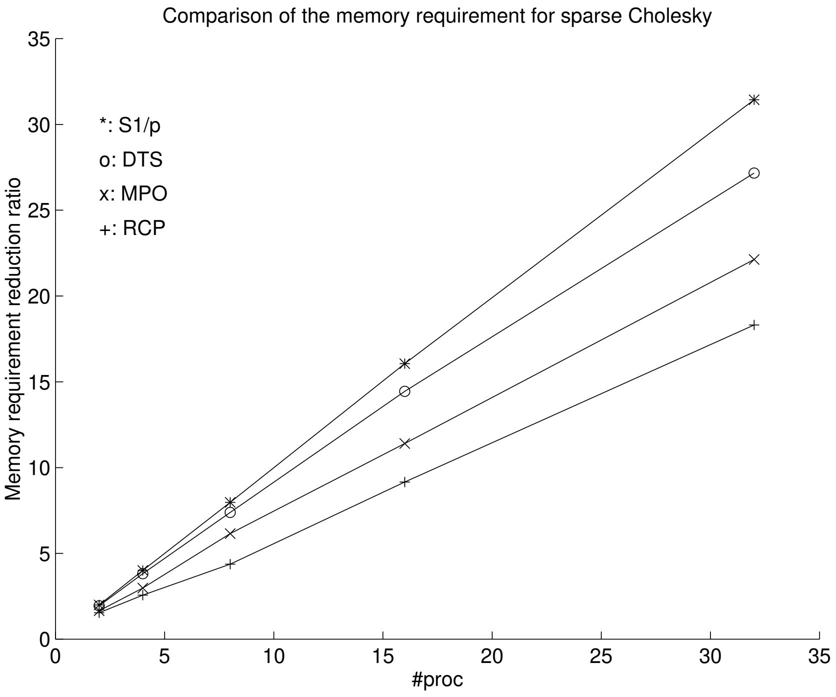
Experimental results

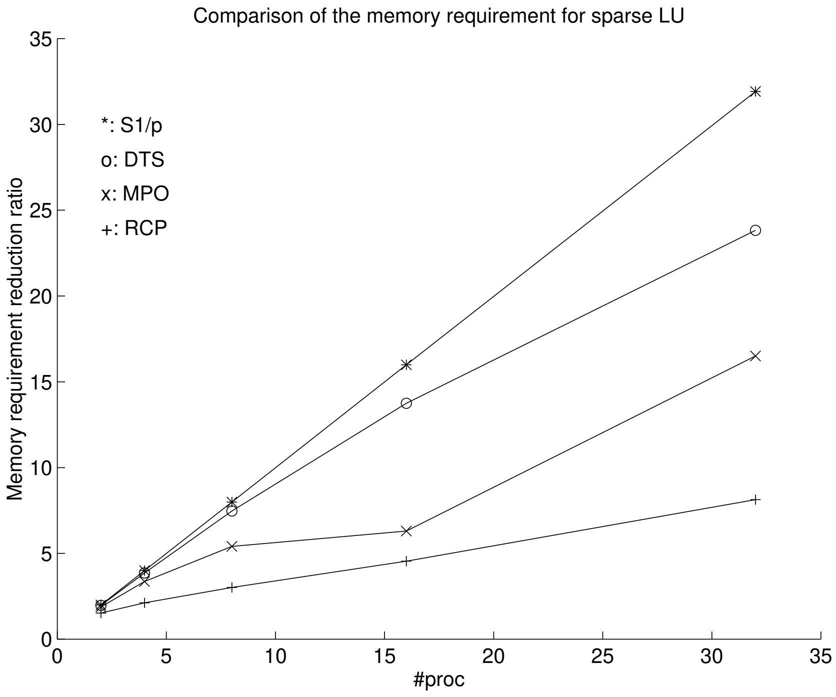
Implemented on T3D/T3E and Meiko CS-2.

- Application codes: sparse Cholesky and sparse LU with partial pivoting.
- Benchmark matrices from Harwell-Boeing and other sources.
- Comparison of space efficiency of scheduling algorithms.
- Comparison of time efficiency of scheduling algorithms.
- Overhead of active memory management.
- Solving large problems on T3D and T3E.

Space efficiency: DTS, MPO and RCP

Memory scalability (reduction ratio): S_1/S_p (memory space per processor).





Time efficiency: DTS vs. RCP

Parallel time increases: $PT_{DTS}/PT_{RCP} - 1$.

For sparse Cholesky

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%

For sparse LU

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%	*

Time efficiency: MPO vs. RCP

Parallel time increases: $PT_{MPO}/PT_{RCP} - 1$.

For sparse Cholesky

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%

For sparse LU

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%	*

Memory management overhead

Cholesky:

Perce.	100%	75%		50%		40%	
	PT incr.	#MAPs	PT incr.	#MAPs	PT incr.	#MAPs	PT incr.
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%

Sparse LU:

Perce.	100%	75%		50%		40%	
	PT incr.	#MAPs	PT incr.	#MAPs	PT incr.	#MAPs	PT incre.
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%

"∞": non-executable.

Solving large problems

Using active memory management scheme and memory scheduling, the solvable problem size is increased by **145%**. Before: the maximum MFLOPS achieved on 64 nodes of T3D is: 419 MFLOPS.

#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

Recent results on T3E

Without active memory management: MFLOPS achieved on 64 nodes of T3E is: 699.4 MFLOPS.

With active memory management:

#proc	PT(Seconds)	Ave. #MAPs	MFLOPS
16	43.3	8	619.3
32	33.5	5	800.5
64	30.5	3.6	879.2

Conclusions

- Provide a run-time space recycling scheme in conjunction with the use of a fast communication mechanism.
- Proposed scheduling algorithms to optimize both CPU and memory resources at the same time.
- The scheduling algorithms can significantly reduce the memory requirements and improve the execution efficiency as well.
- The techniques effectively improved the solvable problem sizes.