Semantically Ordered, Parallel Execution of Multiprocessor Programs

Gagan Gupta and Gurindar S. Sohi
{gagang, sohi}@cs.wisc.edu
WISCONSIN UNIVERSITY OF WISCONSIN-MADISON

Summary

Multiprocessors are ubiquitous, but programming them continues to be challenging

Our Goal: Simplify multiprocessor programming without compromising performance

Conventional Wisdom	Our Approach
Order in programs obstructs parallelism	Order can help to expose parallelism!
Use non-deterministic programs, or make dataflow in programs explicit	Use **ordered** programs; maintain precise **program-order** execution **semantics**
Programmer should expose parallelism	Use **run-time dataflow** and **speculative** techniques to expose parallelism

Benefits:

- Simplified programming; Simplified system design; Better reliability
- Performance at par or better (5% to 288%) than conventional methods

Programmer's Role

Conventional

Develop Parallel Algorithm → Schedule Execution of Tasks → Reason About Inter-task Data Access Conflicts → Ensure Independence between Parallel Tasks

Program text ≠> Order

Our Approach

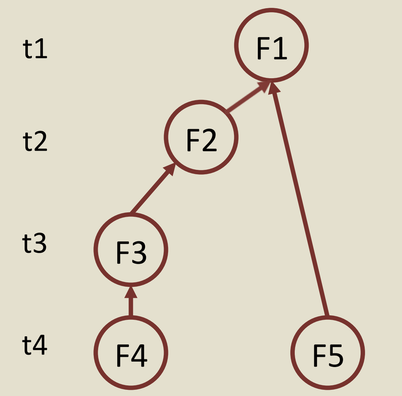
Develop Parallel Algorithm → Reason About Task-local Data Accesses

Program text => Order

Exploiting Parallelism

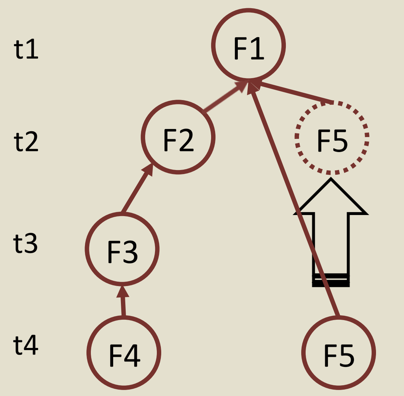
Task Dependence Graph of Cholesky Decomposition

Conventional



- Execution has to respect programmer-exposed parallelism
- Cannot schedule F5 in t2

Our Approach



- If dependences are known, distant parallelism can be exposed
- Can schedule F5 in t2

Speculation

Three key patterns of parallelism:

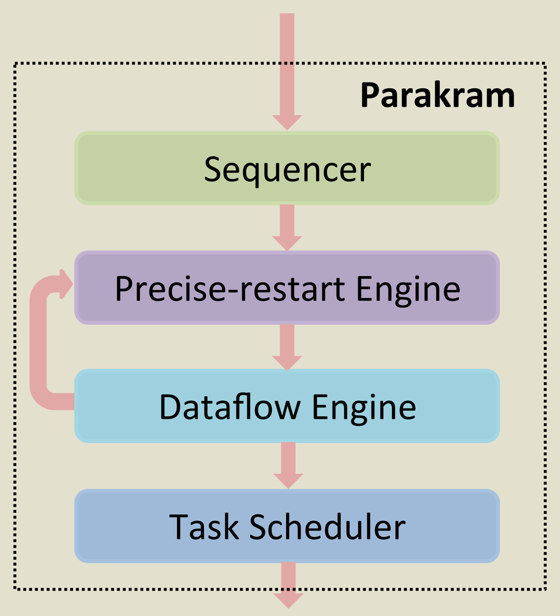
- Exposing work
  - Recursion
  - Loop
- Task's dataset
  - Data independent
  - Data dependent
- Data structures
  - Static
  - Dynamic

Speculative execution for

- Recursion
- Data dependent dataset
- Dynamic data structure

Parakram

- Run-time parallel execution manager (C++ library)
- Performs out-of-order superscalar processor-like execution
  - Dataflow scheduling
  - Speculation
  - Globally precise-interruptible



Parakram Block Diagram

Parakram Operations

Multiprocessor Program

Sequencer

- Unrolls dynamic instances of tasks
- Computes data set dynamically (user assisted)
- Advances over stalled tasks

```
for (i=0; i<n; i++) {
   call F (wr, rd)
}
```

Fn	Write Set	Read Set
F1:	{B, C}	{A}
F2:	{D}	{A}
F3:	{?}	{?}
F4:	{B}	{D}
F5:	{B}	{D}
F6:	{G}	{H}

Example code and dynamic task instances

Precise-restart Engine

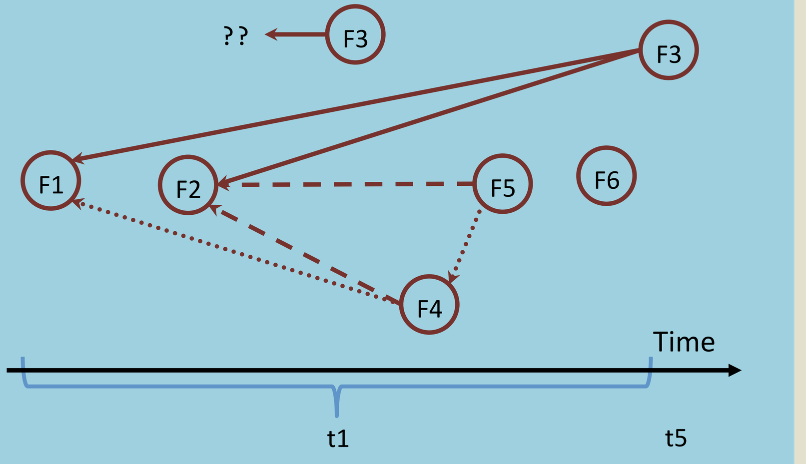
- Tracks tasks and their order in a Reorder List
- Checkpoints mod set in History Buffer
- Retires task in (total) program order

⇨ F6 F5 F4 F3 F2 F1 ⇨

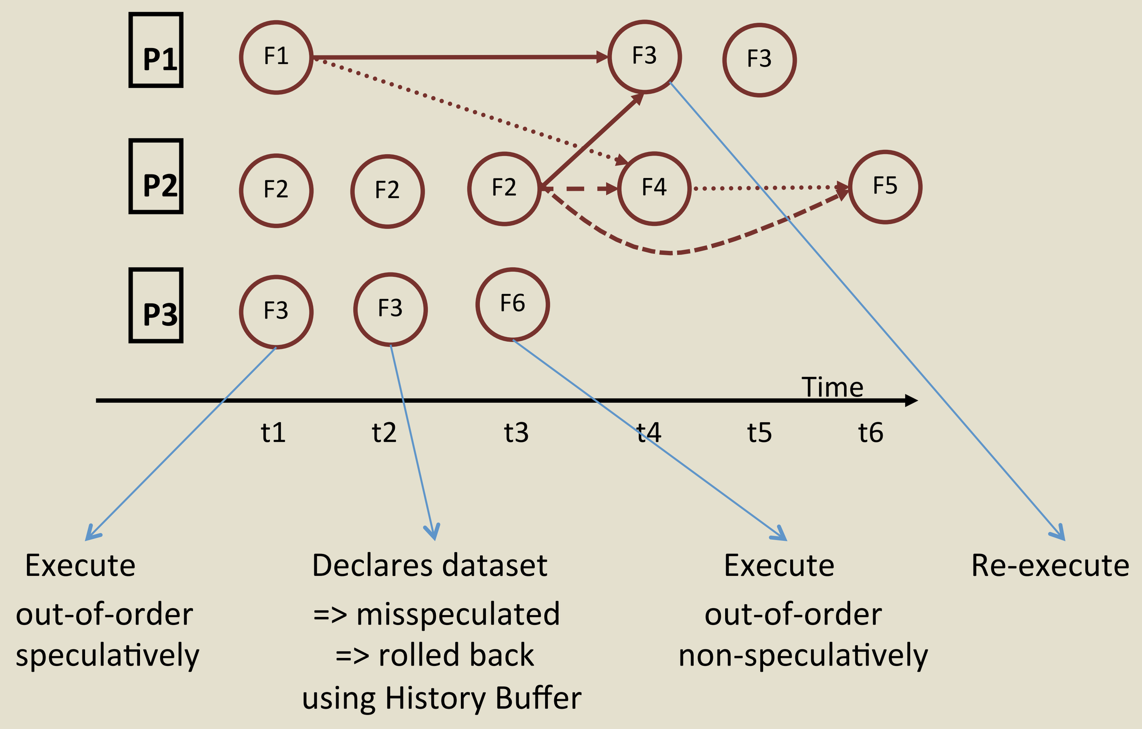
Dataflow Engine

Uncovers parallelism past blocked tasks in the program

- Constructs dynamic data dependence graph using write and read sets
- Executes tasks out-of-order
- If task dependences/order are unknown, speculates tasks are independent
- Detects and rectifies misspeculation



Order-aware Load-balancing Task Scheduler

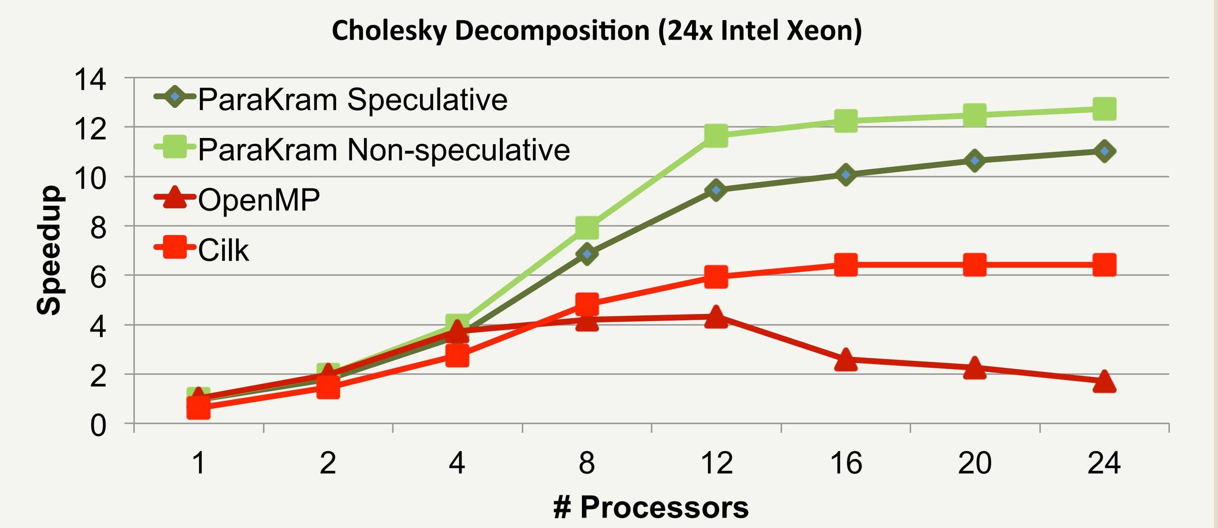


Example speculative dataflow execution on 3 processors

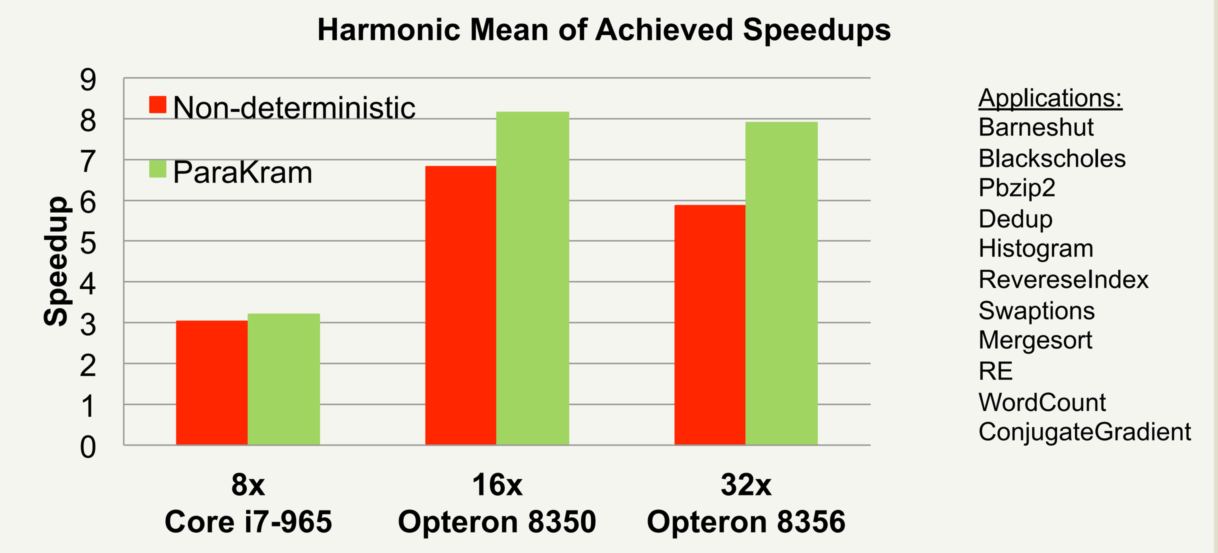
Epoch	Reorder List Entries	Completed	Retired
t1	F6 F5 F4 F3 F2 F1		
t2	F6 F5 F4 F3 F2	F1	F1
t3	F6 F5 F4 F3 F2	F1	
t4	F6 F5 F4 F3	F1, F6, F2	F2

Precisely restarting misspeculated task (F3 from above)

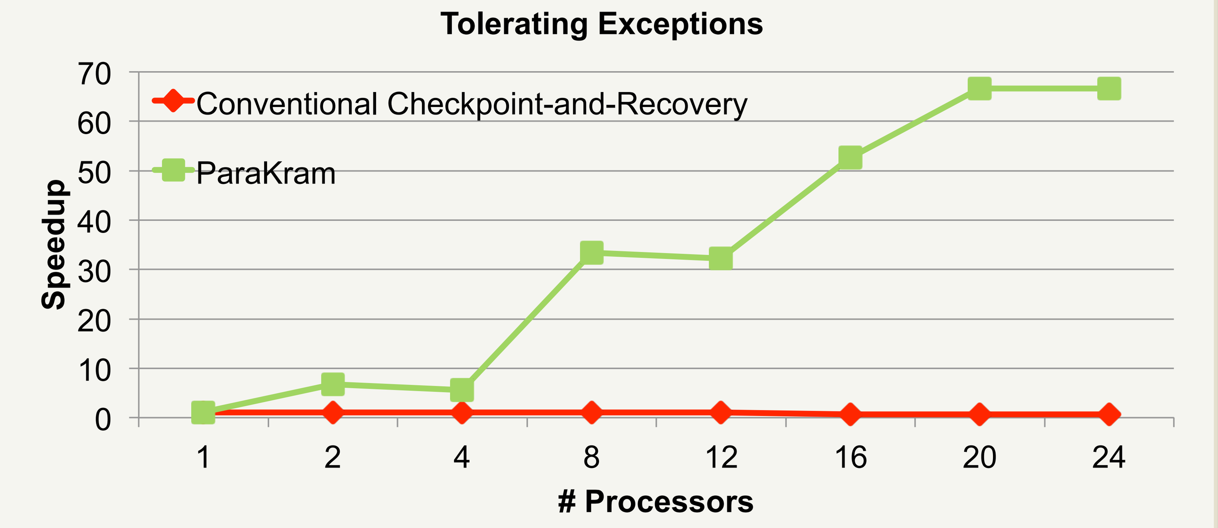
Evaluation



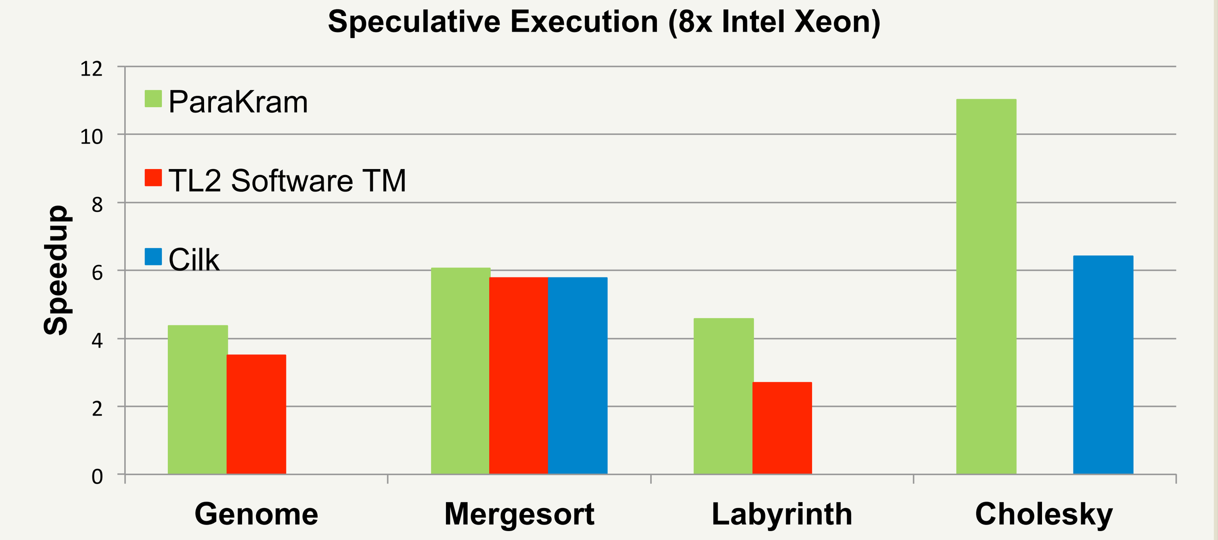
Better speedups (75%, ~3X) than OpenMP, Cilk



Better speedups (20%) than Pthreads



Scales fault-tolerance with system size



Better speedups than Cilk, TL2 STM