Scaling Transactional Memory Workloads

Bradley C. Kuszmaul

Supercomputing Technologies Research Group
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Requirements for Scaling

Desktop machines will have hundreds of processors. The performance goal will be to keep memory busy, not processors. We’ll need parallel programs. Assume XM.

Requirement: Xactions must not conflict too much. This seems reasonable.

Requirement: Work must be scheduled efficiently. Scheduling causes pain in persistent thread models such as pthreads and java threads.
Cilk

A C language for dynamic multithreading with a provably good runtime system.

Platforms
- AMD Opteron
- Sun UltraSparc
- SGI Altix
- Intel Pentium

Applications
- virus shell assembly
- graphics rendering
- $n$-body simulation
- ★ Socrates and Cilkchess

Cilk automatically manages low-level aspects of parallel execution, including protocols, load balancing, and scheduling.
Fibonacci

```c
int fib (int n) {
    if (n<2) return (n);
    else {
        int x,y;
        x = fib(n-1);
        y = fib(n-2);
        return (x+y);
    }
}
```

**Cilk code**

```c
#include <cilk++>

int fib (int n) {
    if (n<2) return (n);
    else {
        int x,y;
        x = spawn fib(n-1);
        y = spawn fib(n-2);
        sync;
        return (x+y);
    }
}
```

**C elision**

Cilk is a *faithful* extension of C. A Cilk program’s *serial elision* is always a legal implementation of Cilk semantics. Cilk provides *no* new data types.
Dynamic Multithreading

```cilk
int fib (int n) {
    if (n<2) return (n);
    else {
        int x,y;
        x = spawn fib(n-1);
        y = spawn fib(n-2);
        sync;
        return (x+y);
    }
}
```

“The computation dag unfolds dynamically.”

“The processor oblivious.”
Outline

- Theory and Practice
- A Chess Lesson
- Apply to XM
Algorithmic Complexity Measures

\[ T_P = \text{execution time on } P \text{ processors} \]
Algorithmic Complexity Measures

\[ T_P = \text{execution time on } P \text{ processors} \]

\[ T_1 = \text{work} \]
Algorithmic Complexity Measures

\[ T_P = \text{execution time on } P \text{ processors} \]

\[ T_1 = \text{work} \]

\[ T_\infty = \text{critical path} \]
Algorithmic Complexity Measures

\[ T_P = \text{execution time on } P \text{ processors} \]

\[ T_1 = \text{work} \]

\[ T_\infty = \text{critical path} \]

Lower Bounds

- \[ T_P \preceq T_1/P \]
- \[ T_P \preceq T_\infty \]
Algorithmic Complexity Measures

\[ T_P = \text{execution time on } P \text{ processors} \]

\[ T_1 = \text{work} \]

\[ T_\infty = \text{critical path} \]

Lower Bounds

\[ T_P \preceq T_1/P \]

\[ T_P \preceq T_\infty \]

\[ T_1/T_P = \text{speedup} \]

\[ T_1/T_\infty = \text{parallelism} \]
Greedy Scheduling

Theorem [Graham & Brent]:
There exists an execution with $T_P \leq T_1/P + T_\infty$. 
Greedy Scheduling

**Theorem** [Graham & Brent]: There exists an execution with $T_P \leq T_1/P + T_\infty$.

**Proof.** At each time step, ...
Greedy Scheduling

**Theorem** [Graham & Brent]: There exists an execution with $T_P \leq T_1/P + T_\infty$.

**Proof.** At each time step, if at least $P$ tasks are ready, ...
Greedy Scheduling

**Theorem** [Graham & Brent]:
There exists an execution with \( T_P \leq \frac{T_1}{P} + T_\infty \).

**Proof.** At each time step, if at least \( P \) tasks are ready, execute \( P \) of them.
Greedy Scheduling

**Theorem** [Graham & Brent]: There exists an execution with $T_P \leq T_1/P + T_\infty$.

**Proof.** At each time step, if at least $P$ tasks are ready, execute $P$ of them. If fewer than $P$ tasks are ready, …
**Greedy Scheduling**

**Theorem** [Graham & Brent]:
There exists an execution with $T_P \leq T_1/P + T_\infty$.

**Proof.** At each time step, if at least $P$ tasks are ready, execute $P$ of them. If fewer than $P$ tasks are ready, execute all of them.
Greedy Scheduling

**Theorem** [Graham & Brent]: There exists an execution with $T_P \leq T_1/P + T_\infty$.

**Proof.** At each time step, if at least $P$ tasks are ready, execute $P$ of them. If fewer than $P$ tasks are ready, execute all of them.

**Corollary:** Linear speed-up when $P \simeq T_1/T_\infty$. 
Cilk Performance

Cilk’s “work-stealing” scheduler achieves

- $T_P = T_1/P + O(T_\infty)$ expected time (provably);
- $T_P \approx T_1/P + T_\infty$ time (empirically).

Near-perfect linear speedup if $P \approx T_1/T_\infty$.

Instrumentation in Cilk provides accurate measures of $T_1$ and $T_\infty$ to the user.

The average cost of a spawn in Cilk-5 is only 2–6 times the cost of an ordinary C function call, depending on the platform.
Outline

• Theory and Practice
• A Chess Lesson
• Apply to XM
Cilk Chess Programs


- **Socrates 2.0** took 2nd place in the 1995 World Computer Chess Championship running on Sandia National Labs’ 1824-node Intel Paragon.


- **Cilkchess** tied for 3rd in the 1999 WCCC running on NASA’s 256-node SGI Origin 2000.
★ Socrates Normalized Speedup

\[ T_P = T_\infty \]

\[ T_P = T_1/P + T_\infty \]

\[ \frac{T_1/T_P}{T_1/T_\infty} \]

\[ \frac{P}{T_1/T_\infty} \]

*measured speedup*
**Socrates Speedup Paradox**

Original program

\[ T_{32} = 65 \text{ seconds} \]

\[ T_1 = 2048 \text{ seconds} \]
\[ T_\infty = 1 \text{ second} \]
\[ T_{32} = \frac{2048}{32} + 1 = 65 \text{ seconds} \]
\[ T_{512} = \frac{2048}{512} + 1 = 5 \text{ seconds} \]

Proposed program

\[ T'_{32} = 40 \text{ seconds} \]

\[ T'_1 = 1024 \text{ seconds} \]
\[ T'_\infty = 8 \text{ seconds} \]
\[ T'_{32} = \frac{1024}{32} + 8 = 40 \text{ seconds} \]
\[ T'_{512} = \frac{1024}{512} + 8 = 10 \text{ seconds} \]

\[ T_P \approx \frac{T_1}{P} + T_\infty \]
Need More Than XM

Workloads will help, but programmers also need:

• load balancing,
• scalable performance,
• debugging & software release tools, and
• linguistic support.