What is MTOOL?

- **Performance profiler**
  - Shared memory bottlenecks, synchronization overhead, parallelization overhead
  - At least 2 profiled executions required
- **Supported platforms**
  - MIPS based architectures (+ others?)
    - SGI 380 (8x33 MHz processors and 256M shared mem)
  - C + ANL macros
  - Fortran with loop level parallelism
Overview of paper

- **Instrumentation**
  - Timers
  - Basic block counters
- **Efforts to minimize instrumentation overhead**
- **Description of memory/synchronization bottlenecks**
- **2 case studies**

Timers

- `start_timer/stop_timer` added to begin/end of procedures
- Bloat is minimized by scanning initial execution profile to exclude fast/frequently executed regions
  - Minimum of 5x the overhead of start/stop timer
- Alternative to timers is pc-sampling
Basic block

- A sequence of one or more consecutive, executable statements containing no branches

```c
for(i=0;i<10;i++)
{
    if(f(i) != 0)
        x=1/f(i);
    else
        x=0;
}
```

Minimum Cost Basic Block Counting

- Minimize overhead while collecting block counts during program execution
- Only place counters on independent control paths
  - Derive dependent counts during post processing
  - Eg: Don’t count both blocks of if/then/else
- Use loop counters to avoid counting each iteration
Basic block counting

- Capture block counts during initial execution
  - Counting cost 379
- Eliminate edges on maximal path
  - \{(a,b),(b,d),(e,b),(a,f)\}
  - Counting cost 125
- Examine loop variables
  - \{(a,b),(e,f)\}
  - Counting cost 4

Memory bottlenecks

- Identify bottlenecks by comparing actual execution time to an estimated execution time that assumes optimal memory access
- Use initial profile run to select target regions
  - Contain large amount of global memory access
  - Low timer overhead
  - Reasonable number of lines of code
Estimating optimal memory

- Estimated compute time for basic block * basic block count

- RISC architecture allows for estimation of compute time except in:
  - Data dependent stalls
    - Memory accesses
  - Stalls between instructions

Synchronization bottlenecks

- Overhead is any time spent idle/spin-waiting
  - Low perturbation timers used

- Bottlenecks examined
  - Load imbalance
    - Waiting at barrier
  - Critical sections
    - Lock contention
  - Starvation
    - Sequential executions in master process

- User defined locks are ignored but can be specified in a config file
Case study 1

- Significant memory bottleneck
- Suspect subroutine contains pointer swap that is replaced with a copy to take advantage of cache
  - $\rightarrow 50\%$ decrease in memory overhead

Case study 2

- Shared vector (Ready) used to synchronize processes exchanging computed values
- Non-linear speedup indicates a bottleneck
MTOOL displays code block responsible for the bottleneck
UI allows for reclassification of user spin-wait as synchronization overhead
Code indicates that numerous global memory references may be saturating the shared bus and causing the bottleneck

Summary

- MTOOL profiling can identify memory and synchronization bottlenecks on a shared memory architecture with as few as 2 program executions
- MTOOL timer and basic block count instrumentations minimize overhead and program perturbation