P-Threads and Shared Memory Programming in PETSc

Kerry Stevens
APAM Research Conference
February 3, 2012
Software Needs To Utilize Resources

- Most HPC systems consist of many, many, many nodes networked together in a suitable topology
  - A node consists (most often) of multiple processors (sockets) that all have multiple cores

- Utilizing only 1 MPI process per node utilizes only 1 core of 1 processor

- P-Threads/OpenMP designed for shared memory parallel programming, coordinating work intra-node

Your basic desktop

Your basic supercomputer
Communication Model

- P-Threads provides 2 routines to inform waiting threads that the state of some shared data structure has changed

  - `pthread_cond_broadcast` awakens all the threads
    - initiates a 'thundering herd' all contending for the lock

  - `pthread_cond_signal` awakens only 1 thread waiting
    - Can specify which thread is to be awoken by using a different condition variable for each thread

  - `pthread_cond_wait` causes the thread to unlock the held mutex and sleep AUTOMATICALLY
The Track Race Analogy

• Guy with the gun – the 'main' thread
• Runners – the posix threads

• The Event
  • All runners line up and get into position
  • The guy with the gun fires it off
  • Runners perform their task: run

• How can the above procedure be accomplished in software?
The Track Race Analogy Continued …

• Common track race events include the 100m, 200m, 400m, and 800m

• Track race events in software we term thread kernels and include vector dot product, matrix-vector multiply, etc.

  • Just as the 100m takes less time to complete than the 800m, different thread kernels take different amounts of time to complete
The Firing of the Starting Gun

• In real life the Starter fires the gun, the runners all hear it simultaneously, and everyone starts running

• The Starter firing the gun analogous to the 'main' thread issuing a pthread_cond_broadcast instruction
Implicit Synchronization
The Starter

• Must ensure all the runners are lined up and in position

• Must ensure all the runners have finished the race before commencing to set up a new race

• Must collect race results (sometimes)
Implicit Synchronization
The Runners

• In real life, unless running a relay, runners are completely independent of one another

• In software, a baton (mutex) enters
  • Depending on the implementation, threads may solely coordinate with the 'main' thread or could coordinate with multiple other threads

• The need for runner synchronization makes thread pool use much less appealing
Why can't all runners (threads) start simultaneously?

- Associated with the `pthread_cond_wait()` is a condition variable AND a mutex
- Any way that wake up & run can truly be simultaneous, not sequential?
  - NO! Unlock and wait needs to be atomic
Thread Pool & Use

• Collection of threads
• Used often in producer/consumer model
  • Producer: add task to some shared data structure
  • Consumer: remove task from shared data structure and complete task
• Lock must protect the shared data structure
Thread Pool Implementation in PETSc

• Thread pools designed for programs comprised of many short tasks
• Run time consists of combining computation and P-Thread overhead time
• Small jobs can take longer using P-Threads due to (comparatively) large amount of time for thread creation
• Thread creation occurs on startup, thread termination occurs at shutdown
Thread Pool Implementations
“True Pool”

- 1 mutex coordinates/synchronizes everyone
- “Thundering herd” for mutex control
- Last worker to finish signals Main
Thread Pool Implementations

“Chain”

- Explicit ordering of worker threads
- Many mutexes, 1 for each worker
- Sequentiality explicit
Thread Pool Implementations “Dictatorship”

- Many mutexes, 1 for each worker thread
- Main sends signals sequentially
- Each worker thread signals Main
- Workers truly independent
Thread Pool Implementations
“Corporate” or “Tree”

- Threads tell their subordinates to get to work
- Subordinates inform their bosses that they're done
- “Parallelization” of wake up procedure
Threads And Core Affinity

• Which core does Main run on? Which do my threads run on?

• Process-to-processor and thread-to-core mappings greatly affect performance
  • “Task mapping problem” heavily researched
Parallel Performance

- All tests conducted on MCS's PETSC machine (see schematic)

- What's the Bottleneck?
  - Compute bound: program execution time determined by speed at which floating point operations can be retired
  - Memory bound: program execution time determined by speed at which cores can obtain data from memory

- Important questions to ask:
  - Can we double sequential performance by using 2 threads?
  - Can we quadruple sequential performance by using 4 threads?
  - Can we octuple sequential performance by using 8 threads?
Sequential (Uni-core)
Performance Hardware Limited

- Argonne “PETSC” machine
  - 32KB L1 Data & Instruction
  - 6144KB L2 Unified

- Working Set
  - Amount of physical memory needed by program
  - As size increases, L2 cannot hold it all and must utilize DRAM
Vector Dot Product Results
Vector of Size 100,000 Elements

- Sequential
  - Runs made using different cores
    - As expected, the particular core utilized did not make a difference
  - Results ranged from 1868 to 1983 Mflops/sec with 1960 Mflops/sec as the median

- Thread Creation/Destruction (2 Threads)
  - Results varied widely between 860 to 1300 Mflops/sec
    - Indicative of current inability to control where threads are placed
Thread Pool Vector Dot Product Results
Vector of Size 100,000 Elements

- Performance shows how much data movement matters
Vector Norm Results
Vector of Size 1,000,000 Elements

- Sequential
  - Median of 850 Mflops/sec
  - Working set overflow of L2 means core has to wait for data to arrive from DRAM

- Thread Creation/Destruction (2 Threads)
  - Comparable performance to sequential
Thread Pool Vector Norm Results
Vector of Size 1,000,000 Elements

Thread 0, Core 0; Thread 1, Core 4
Thread 0, Core 0; Thread 1, Core 2
Thread 0, Core 0; Thread 1, Core 1
Matrix-Vector Multiplication
SNES ex19, dmmg_nlevels = 6

- Sequential implementation sees 585 Mflops/sec performance
References

• Documentation: http://www.mcs.anl.gov/petsc/docs
• PETSc Users manual
• Manual pages
• Many hyperlinked examples
• FAQ, Troubleshooting info, installation info, etc.
• Publications: http://www.mcs.anl.gov/petsc/publications
• Research and publications that make use PETSc
• Programming with POSIX Threads, by Butenhofer