Scalability evaluation of barrier algorithms for OpenMP

Ramachandra Nanjegowda, Oscar Hernandez,
Barbara Chapman and Haoqiang H. Jin

High Performance Computing and Tools Group (HPCTools)

Computer Science Department

University of Houston, Texas

Supported by NSF
under grant CCF – 0702775 and CCF - 0833201

University *of* Houston

Agenda

- Motivation
- Overview of different barrier implementation.
- Methodology
- Experiment Results
- Conclusion
- Future Work

Motivation

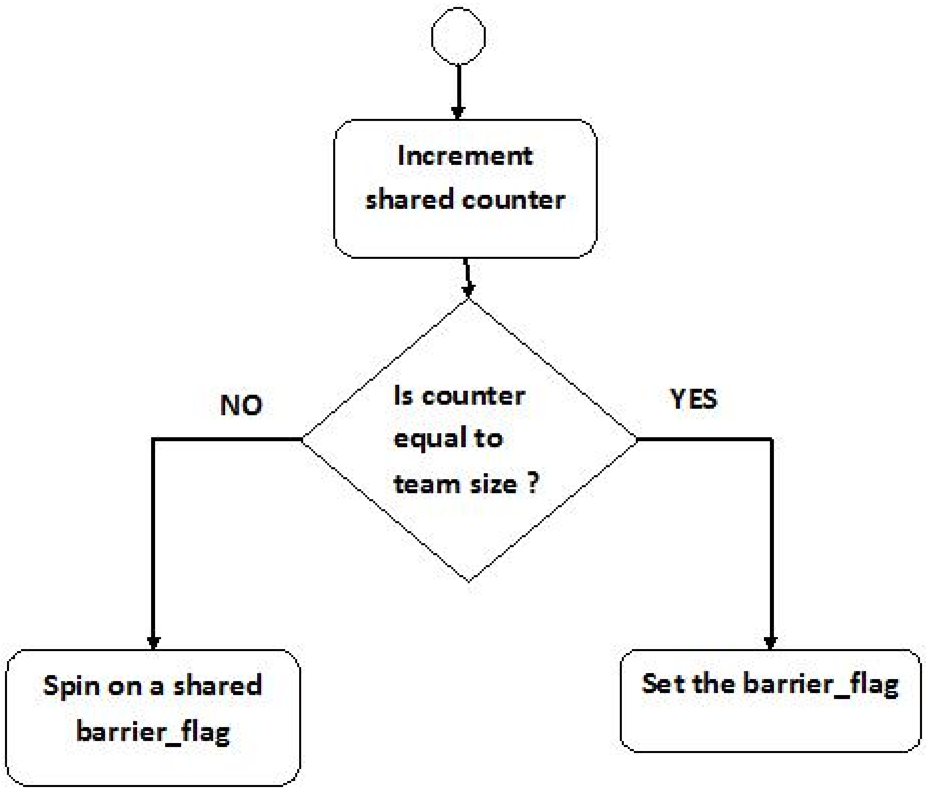
- The number of cores continues to grow…..
    - *OpenMP application scalability is critical.*
- Diverse architecture, memory hierarchy, network interconnect, applications…..
    - *Needs an adaptive runtime.*

OpenMP and Barriers

- OpenMP
  - Relies heavily on *explicit* barrier synchronization to coordinate the work of threads.
  - Most OpenMP constructs involve *implicit* barriers, and incur overhead because of it.

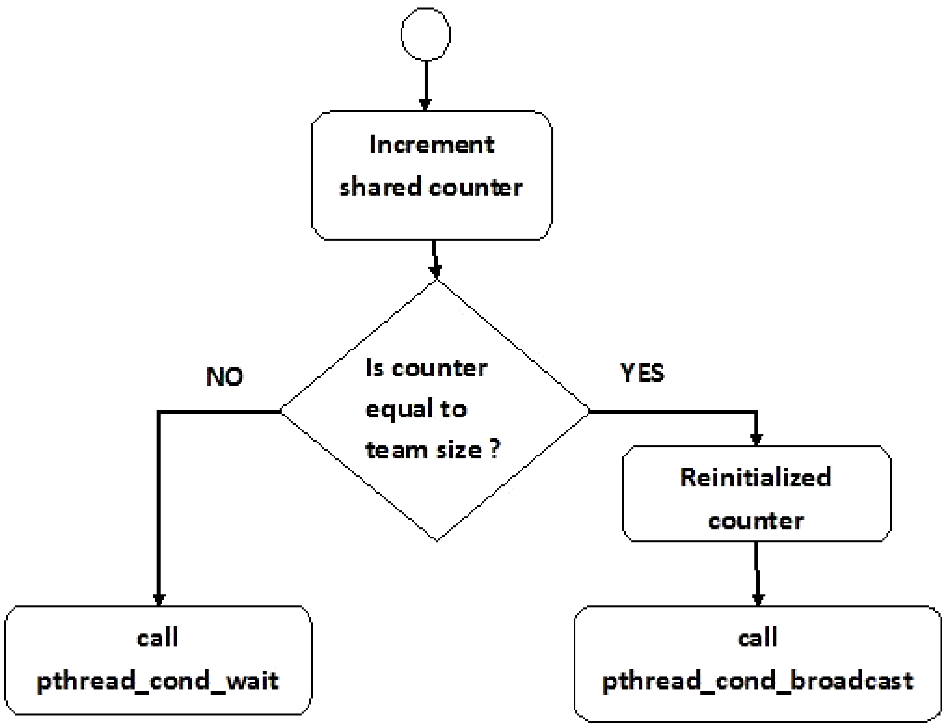
- Barrier algorithms classification:
  1. Centralized busy-wait : busy-wait on shared variable.
  2. Blocking : pthread call that de-schedules the waiting thread.
  3. Distributed busy-wait : busy-wait on a separate locally accessible variable.
     1. Dissemination barrier.
     2. Tournament barrier.
     3. Tree barrier.

Centralized barrier



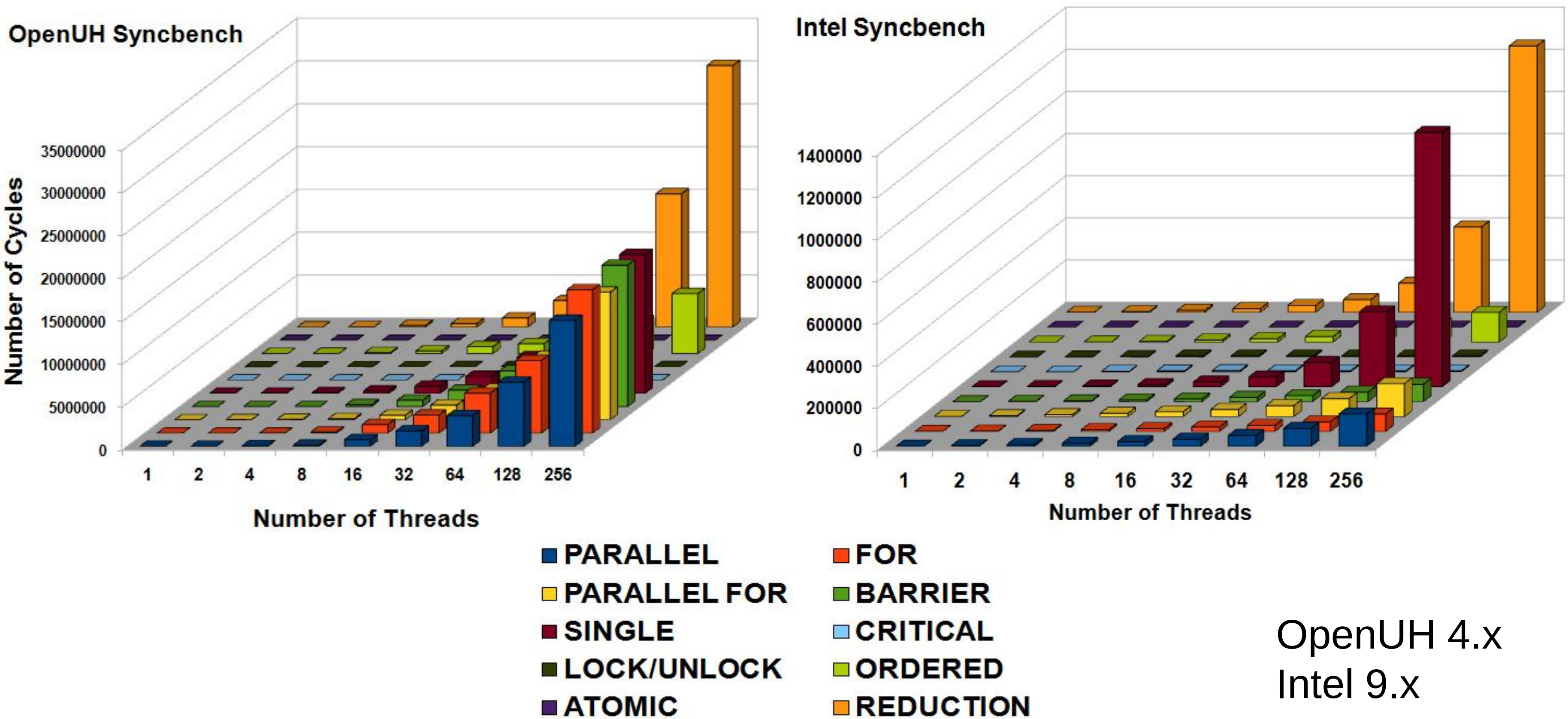
- The shared counter accessible by all threads is incremented atomically.
- Busy waiting on a single flag causes hot spots and large amounts of memory, and interconnect traffic.
- Effect is pronounced as application scales.

Blocking barrier algorithm



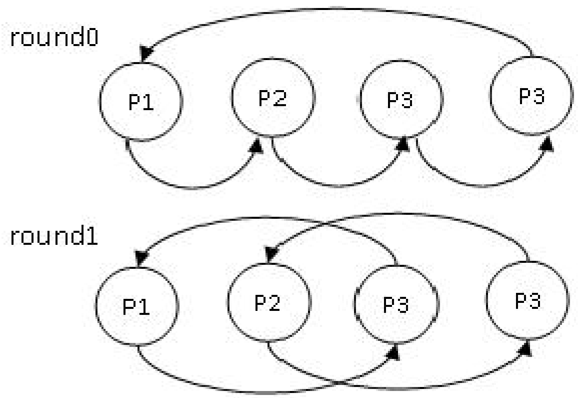
- Improves upon centralized shared counter algorithm.
- Involved rescheduling of threads.
- Suitable when there is frequent oversubscription of threads as in case of,
  - multiuser environment.
  - task support.
- Poor scalability because of,
  - Mutex and condition variables.
  - Context switching.

Overheads of blocking barrier (EPCC benchmark)



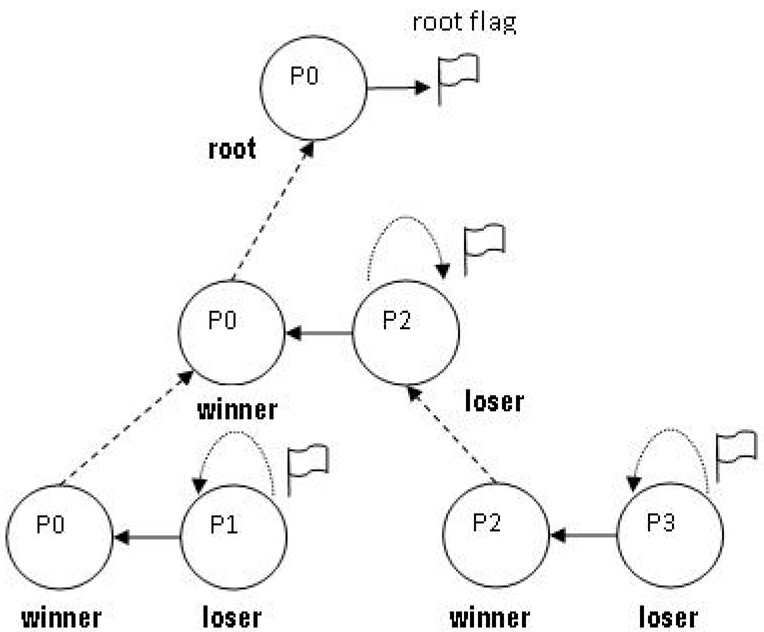
OpenUH 4.x
Intel 9.x

Dissemination Barrier



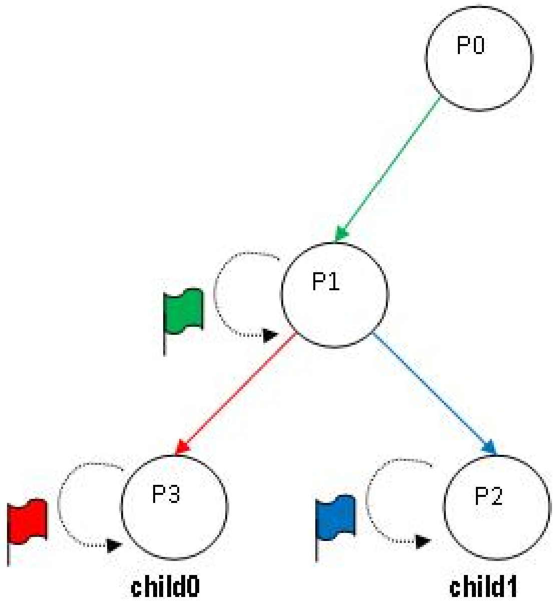
- Originally implemented in MPI environment.
- For each round n, thread i, waits for flag setting from i-2k and sets flag of thread i+2k
- Results
  - Gives good results for < 16 threads because of the simplicity.
  - Not as good on 16+ (constant interconnect communication in each round) and directory based cache (sun niagara)

Tournament barrier



- Based on the idea of tournament game.
- For each round n, loser sets the flag of winner before spinning on root flag.
- Better performance on higher number of threads because of faster initialization as compared to tree algorithm.
- The interconnect traffic caused by root flag can cause performance bottleneck in which case the tree implementation would be better.

Tree barrier

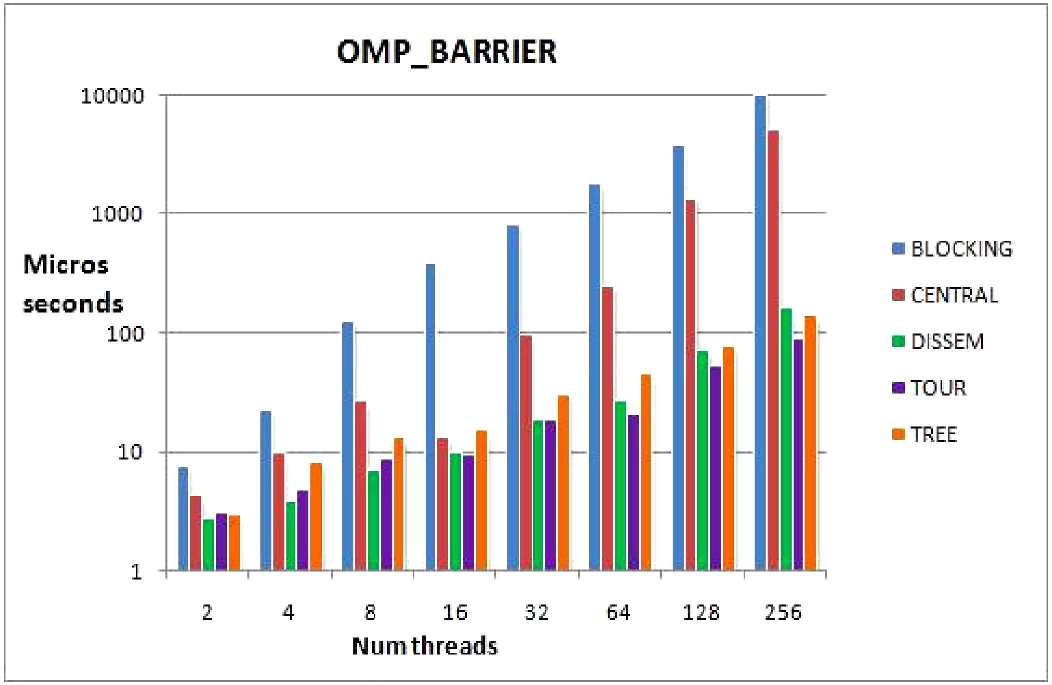


- Binary tree communication between threads.
- All non-root threads wait on local flags during arrival phase. (leaves to root)
- The parent sets the flag of the children during wakeup phase (root to leaves).
- Better performance in some cases because it eliminates the root flag.
- Involves complexity of building the tree structure.

Methodology

- Platforms:
  - SGI 4700 Altix: 1024 dual core Intel processor.
  - Sun Fire X4600: 8 dual core AMD Opteron.
  - Fujitsu-Siemens RX600S4: 4 quad core Intel Xeon processors.
  - Sun T5120: 8 processor cores each with 8 hardware threads.
- Benchmarks
  - The EPCC microbenchmark suite.
  - NAS Parallel Benchmark 3.0, LU-HP benchmark.
  - OpenMP application ASPCG kernel and GENIDLEST.
  - An hand-crafted pthread application.
- Compilers and Tools
  - OpenUH, OpenMP Collector API tool, GCC

EPCC Results



OMP_BARRIER

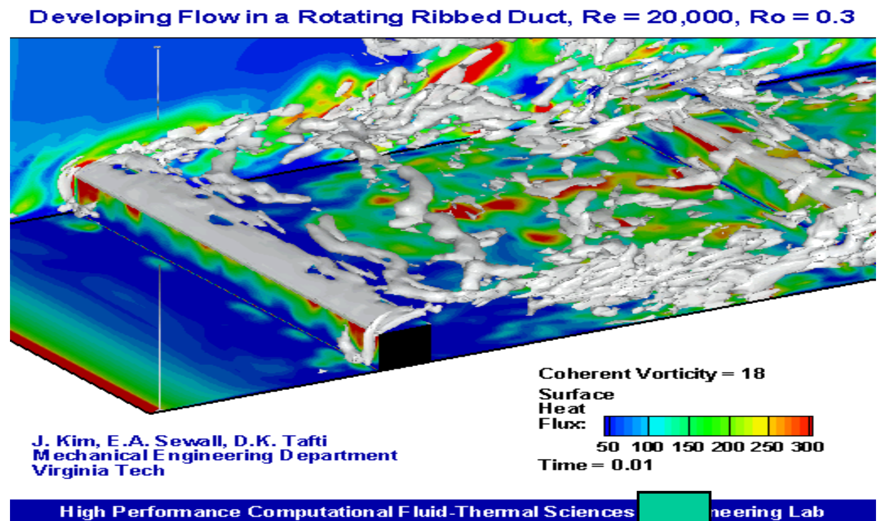
- The centralized and blocking barrier do not perform as well as other algorithms.
- Dissemination barrier has the least overhead on 16 threads or less.
- Tournament barrier resulted in lowest overhead for more than 16 threads

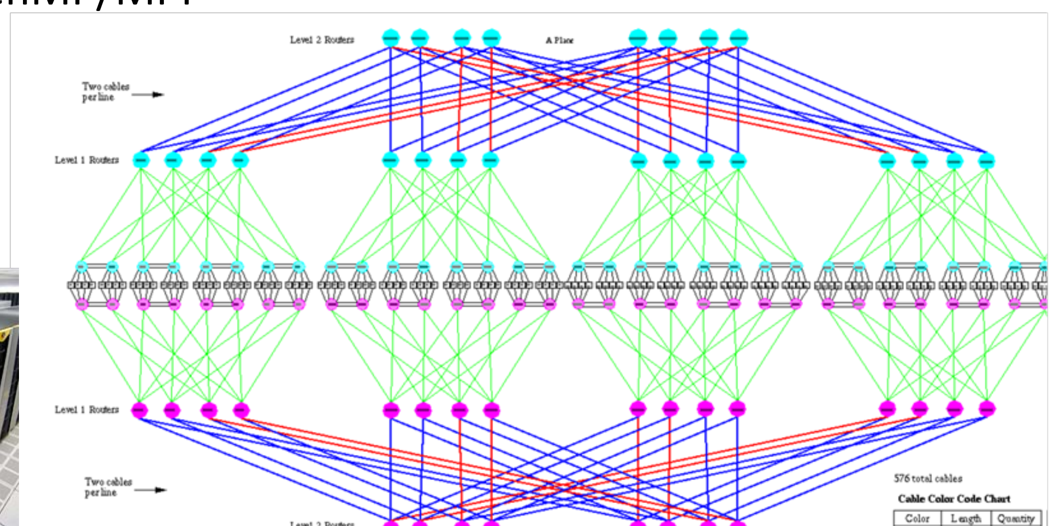
```
for (k=0; k<=OUTERREPS; k++){
  start  = getclock();
#pragma omp parallel private(j)
  {
   for (j=0; j<innerreps; j++){
     delay(delaylength);
     #pragma omp barrier
   }
  }
  end = getclock();
}
```

GenIDLEST / ASPCG

- Generalized Incompressible Direct and Large-Eddy Simulations of Turbulence
- Solves the incompressible Navier-Stokes and energy equations
- Overlapping multi-block body-fitted structured mesh topology combined with an unstructured inter-block topology.
- ASPCG is a small CG kernel of GenIDLEST
- OpenMP, MPI and OpenMP/MPI

  Versions of code.

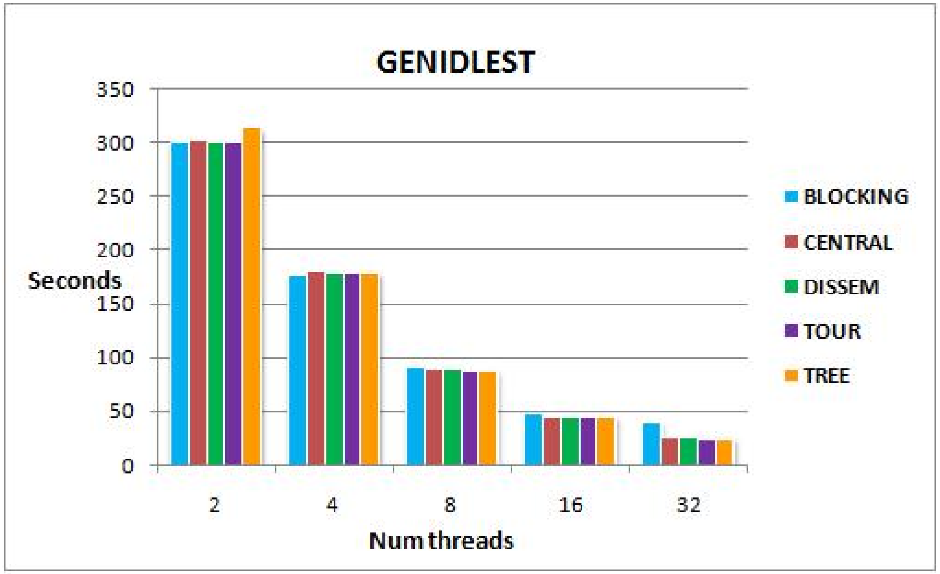




The Altix 3700 512-processor system

University *of* Houston

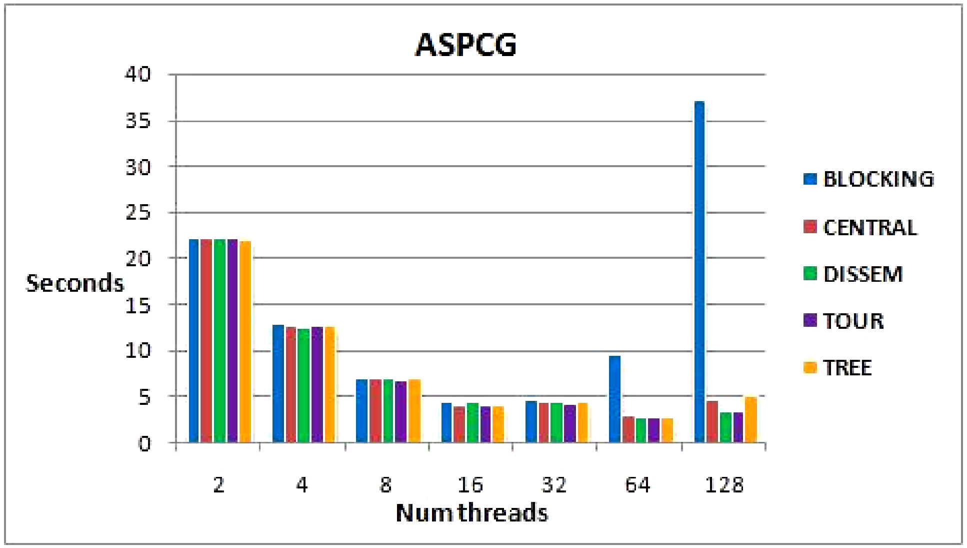
Application Results



GENIDLEST

- GENIDLEST solves incompressible Navier-stokes and energy equations.
- Total 35% improvement in the execution time on 32 threads.
- Blocking barrier performs equally well because of nature of the application, which is memory intensive.

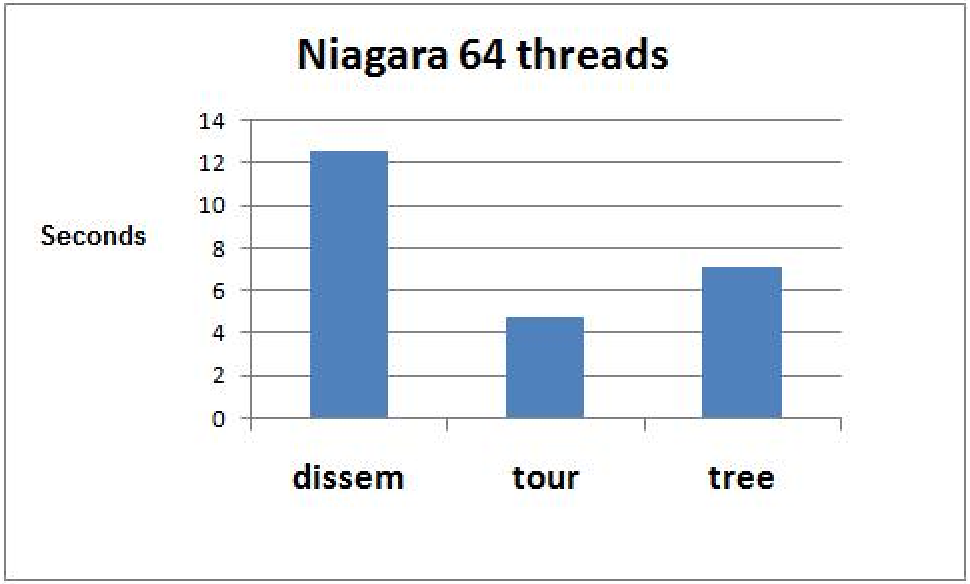
Application Results (cont…)



ASPCG

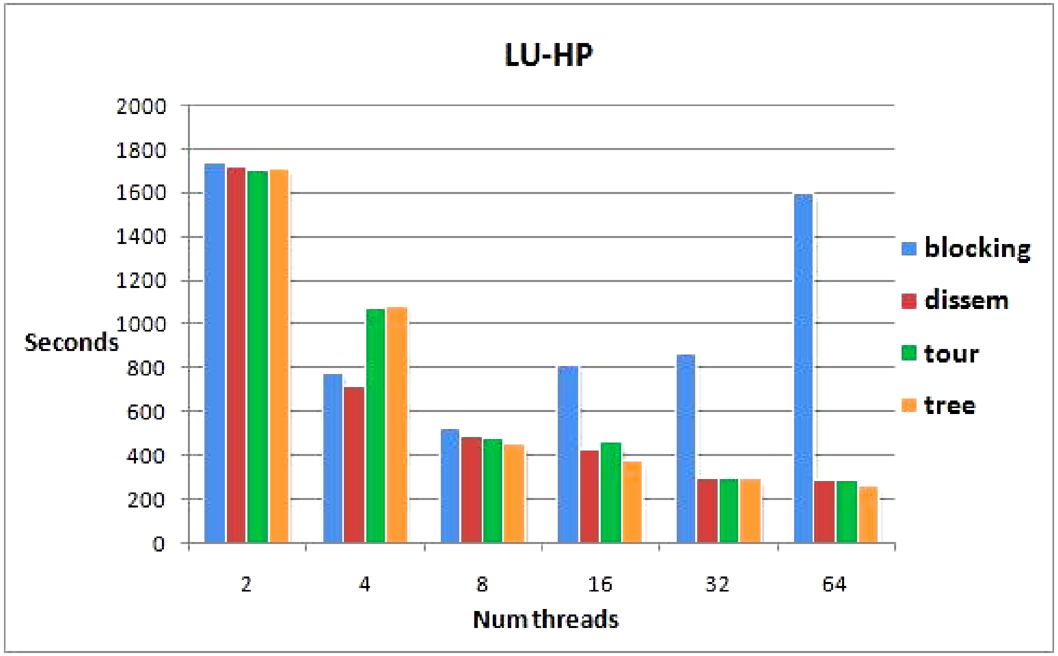
- ASPCG solves a linear system of equations generated by a Laplace equation in cartesian coordinates.
- All busy wait algorithms scale, some with better performance than other.
- Nearly 12x improvement when compared to blocking barrier on 128 threads.
- Tree barrier incurs nearly 20% more execution time than the tournament and dissemination.

Sun Niagara Results



- Employed an hand crafter pthread application for testing, due to lack of OpenUH implementation on Solaris.
- The cache coherence protocol causes poor performance of dissemination.

NAS PB results



- Explored the behavior of LU-HP NAS Parallel benchmark
- LU-HP is CFD application that invokes 739426 implicit barriers on 4 threads. (Data obtained using OpenMP collector API tool)
- Shows considerable performance benefit of tree barrier over others (10% better).

Conclusion

- Achieved good scalability and performance using the new algorithms.

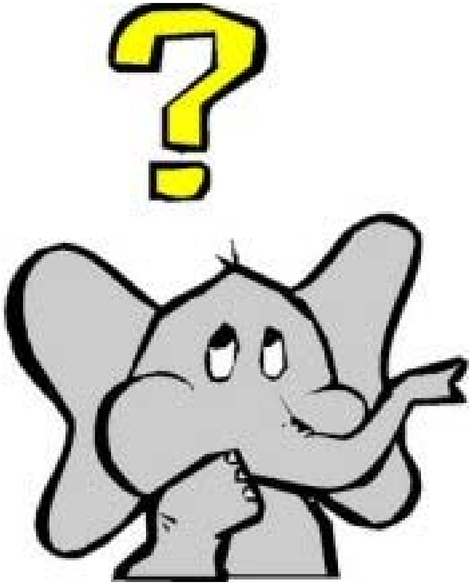
- In general, the performance of a given barrier implementation is dependent on
  - The number of threads used.
  - The number of barriers.
  - The application.
  - The OpenMP directive.
  - The architecture (memory layout/interconnect)
  - And possibly system load (on small scale multicore).

- An OpenMP runtime library should ideally, adapt to different barrier implementations during runtime.

Future Work

- Further testing of these algorithms on even larger system (in particular, with 2048 cores)
- As OpenMP 3.0 implementation becomes available in OpenUH we would like to evaluate how these algorithms affect the tasking feature.
- Implement similar enhancements for the reduction operation.
- Cost model for adaptive runtime.
- Integrate these work with the Adaptive OpenMP and Dynamic compiler work.

Questions?



Backup slides

Dynamic Optimizer and OpenMP

- DO checks if collector is present
- DO requests collector for initialization.
- Thread states are kept by the OpenMP RTL
- DO registers thread events with callbacks to the OpenMP RTL
- DO requests for parallel region replacement, change of schedulers, implementations.