SAND2007-7113P

A Programming Model for Hybrid Parallelism with Consistent Numerical Results

H. Carter Edwards

Sandia National Laboratory

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

HPC Vision: Networked Manycore Nodes

- **Continue to have distributed memory parallelism**
  - **Well understood programming model**
- **Continue to have multiple CPU sockets per node**
  - **Size memory by number of sockets (#nodes * sockets/node)**
  - **Distributed memory vs. shared memory programming dialogue**
  - **Studies consistently showed equal performance**
  - **Simplicity of using a single distributed memory model**
- **Advent of manycore parallelism**
  - **Many parallel processing cores within a single CPU socket**
  - **Currently have four cores, *anticipating* rapid growth**
  - **Contention: shared access to shared main memory**
  - **Question: size memory by number of cores?**

Re-opened Dialogue on Parallel Programming Model

- Scalability with respect to number of cores per socket
  - Will unmanaged sharing of the socket-to-memory resource limit scalability? (e.g., just doing MPI-model on the cores)
  - Is intentional algorithmic management of this shared resource possible? Will it make a difference?

- Per-core consumption of main memory
  - If just doing MPI-model on the cores:
  - Overhead of handing each core its own executable image
  - Overhead of inter-core shared data
  - Overhead of inter-core communication buffering

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Conclusion: We Need to Investigate Hybrid Parallel Programming Model(s)

- **Two level programming model**
  - Outer distributed memory model (a.k.a. the MPI-model)
  - Inner shared memory / parallel threads model

- **Which parallel threads starting point?**
  - OpenMP: compiler-based standard, defines a model
  - Pthreads: library-based standard, does not define a model
  - Intel TBB: C++ STL-like hiding of Pthreads, defines a model
  - Chapel, Fortress, or X10: language research, no time soon

- **If Pthreads then need to define a programming model**

It is Time to Address and Resolve Non-determinism in Parallel Programming

- The same application solving the same problem *should* yield the same answer – regardless of parallelism
  - Traditionally violated, answers effected by:
  - using a different number of processors
  - using a different decomposition on the same processors
  - sometimes even same decomposition & same processors!

- Non-deterministic behavior is user-hostile
  - Which is the "right" answer? A verification issue
  - How to deal with a bug occurring on 1000s of processors that cannot be repeated when debugging on fewer processors?

It is Time to Address and Resolve Non-determinism in Parallel Programming

- **Parallel thread model can exacerbate this problem**
  - Race condition: unpredictable thread completion order
  - Seemingly random, with factorial(#cores) possible outcomes

- **One common source: parallel summation operation**
  - Finite precision $\sum a(i)$ has an intrinsic error
  - Partitioning yields a different answer (parallelization)
  - Reordering yields a different answer (decomposition)

- **The parallel programming model must encourage, or even insist upon, parallel-insensitive answers**

This Practitioner's Investigation:

A Programming Model for Hybrid Parallelism with Consistent Numerical Results

Two Level Model

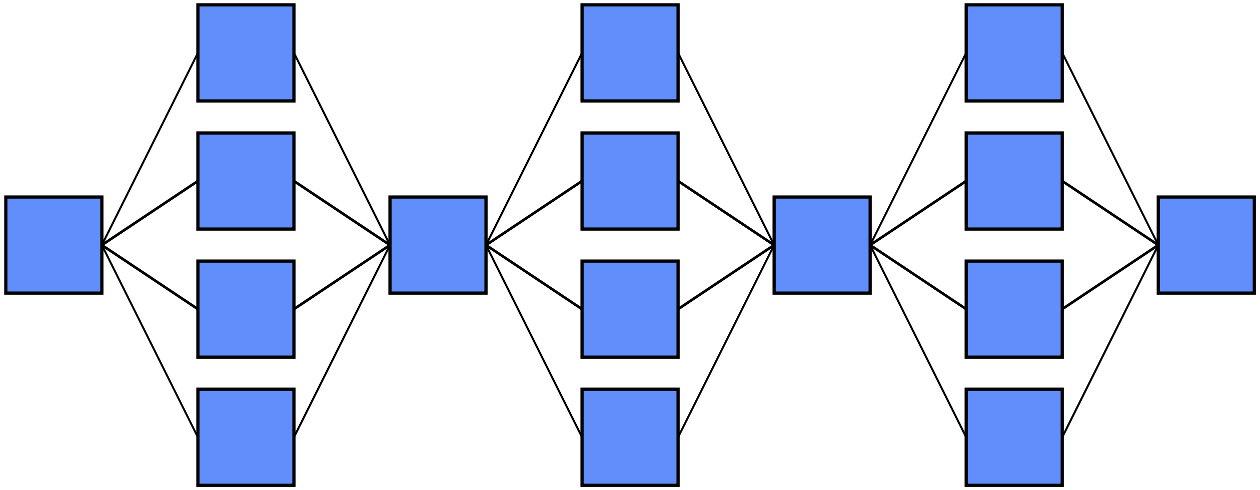
- **Outer level for inter-node parallelism**
  - **Stay with domain decomposition and MPI among nodes**
  - **Tried, true, and familiar model**
- **Inner level for inter-core parallelism**
  - **Address manycore shared resource concerns**
- **Inter-socket parallelism part of outer or inner level**

Model for Inner Level Parallelism

- **Goals: highly portable, simple, minimize overhead, applicable to nontrivial / complicated data structures**
- **Personal preference: C and C++ on Unix-like OS**
- **Use Pthreads, but how?**
  - **Oversubscribe cores or not?**
    - **Answer: at most one thread per core**
    - **Rational: avoid thread context switching overhead**
    - **Concern: thread affinity to cores**
  - **Persistent or local threads?**
    - **Answer: create once and re-use**
    - **Rational: avoid thread creation / destruction overhead**

Model for Inner Level Parallelism

- **Simplicity: only parallel operation are parallel**
  - Sequential operations performed by a single thread
  - Inner level parallel operations performed by all threads
  - Inner level parallel operations have a local and temporary scope



Prototype ‘C’ Interface

```
typedef int (*phdmesh_taskpool_routine)(
  void * routine_data ,
  unsigned p_size , unsigned p_rank );

int phdmesh_taskpool_run(
  phdmesh_taskpool_routine routine ,
  void * routine_data , unsigned number_locks );
```

- Input ‘routine’ is run thread-parallel, called with:
  - Shared ‘routine_data’; don’t use shared global data!
  - How many threads are running
  - The rank of the thread in which this routine is running
- Return the bit-wise ‘or’ of the ‘routine’ return values

Prototype Use in ‘C’: A simple ‘dot’

```
struct TaskXY {
   double         * xy_sum ;
   const double * x_beg ;
   const double * y_beg ;
   unsigned         number ;
};
void txddot( double * s , unsigned n ,
             const double * x , const double * y )
{
  struct TaskXY data = { s , x , y , n };
  phdmesh_taskpool_run( & task_dot_xy_work , & data , 1 );
}
int task_dot_xy_work( void * arg ,
                      unsigned p_size , unsigned p_rank )
{
  struct TaskXY * const t = (struct TaskXY *) arg ;
  /* perform dot over local portion of array */
}
```

Prototype Use in ‘C’: locking for shared updates

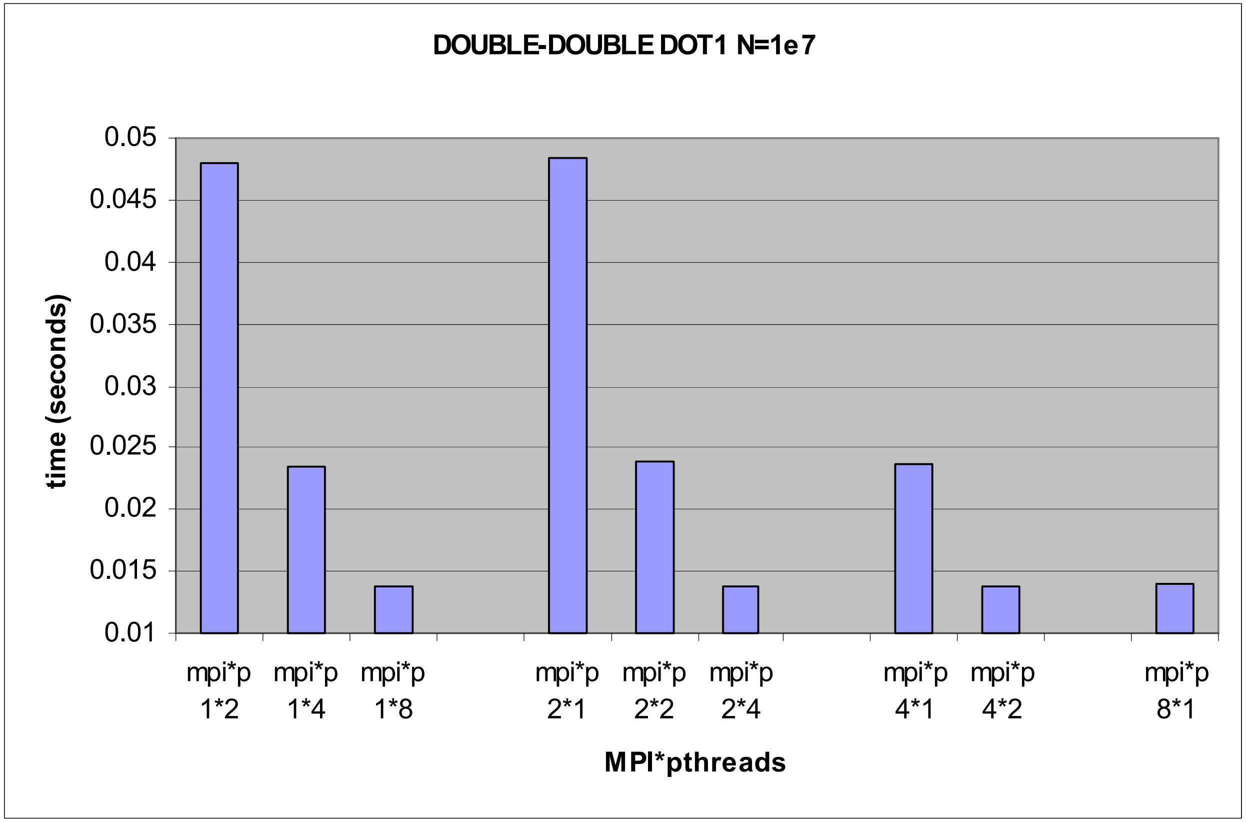
```
/** Issue a lock, provide a string to print for an error */
void phdmesh_taskpool_lock(   unsigned, const char * const );
void phdmesh_taskpool_unlock( unsigned, const char * const );


int task_dot_xy_work( void * arg ,
                      unsigned p_size , unsigned p_rank )
{
  struct TaskXY * const t = (struct TaskXY *) arg ;

  /* ... perform dot over local portion of array ... */

  phdmesh_taskpool_lock(0,NULL);
  xdsum_add_dsum( t->xy_sum , local_partial );
  phdmesh_taskpool_unlock(0,NULL);
  return 0 ;
}
```

Prototype Use in ‘C’: Deterministic ‘dot’

- **Sources of parallel non-determinism in parallel dot(x,y)**
  - Race condition to contribute thread’s partial sum
  - Number of threads yields different partial sums
- **Restore determinism: high-accuracy accumulation**
  - Error in $\sum$ a(i) is O(n*ε), given 0 $\leq$ a(i) and ε is precision (~1e-16)
  - Make $\sum$ x(i) * y(i) “error free”, i.e. error < ε
    - Accumulate positive & negative contributions separately
    - Accumulate in double-double precision
    - Error is now O(n*ε*ε) < O(ε) for n < 1,000,000,000,000,000 < 1/ε
  - Cost? 1 multiply, 7 adds, and 2 branches per term
    - “Free” in-register flops, ‘dot’ is a bandwidth-limited operation
- **Applicable to MPI_Allreduce for the outer parallel ‘dot’**

Scaling of High-Accuracy 'dot(x,x)'

- MacPro: Dual quad-core Intel Xeon, 3GHz
- Hybrid parallel: #Processes = MPI*Pthreads
- Pure Pthreads negligibly faster ⇐ no MPI_Allreduce
  - Very small test executable



Model for Inner Level Parallelism, Load Balancing

- **Dot product example is trivially load balanced**
  - Partition array into #threads subarrays, one per thread
- **Work partitioning not always so easy**
  - Set of items with irregular work load
  - Could perform a load-partitioning pre-algorithm OR
  - Use a "task pool" approach (approx, automatic load balancing)
- **Task pool: threads share a pool of items to be worked**
  - Lock task pool iterator
  - Grab next item(s) of work to do, advance the iterator
  - Unlock task pool iterator (release lock ASAP)
  - Perform work on item(s)
  - Repeat until task pool is done
- **Trying task pool in phdMesh geometric search**

Summary

- **Networks of manycore nodes are coming, ready?**
  - Scalability with increasing cores per socket

- **Pure MPI or hybrid MPI / thread programming model?**
  - Hybrid may be necessary to address memory access contention
  - If hybrid, what programming model?
  - Preference: simple, highly portable, applicable to non-trivial data structures and using task pool pattern

- **Time (past time) to address non-determinism**
  - Same application, same data, give same answer
  - Independent of #nodes, #sockets, #cores, domain decomposition
  - Will require greater discipline in algorithms and data structures