Case Studies in Asynchronous, Message-Driven Shared Memory Programming

Pritish Jetley
Parallel Programming Laboratory
pjetley2@illinois.edu

Outline

- Types of parallelism
- Programming in parallel
- Charm++
- Case studies
- Performance considerations
- Preliminary results

Parallelism in its various forms

- Multiple levels
  - Individual bits
  - Instructions
  - Data elements
  - Independent tasks

Parallel programming, *ca.* 2010 A.D.

- Different models
- Different languages
- Different applications
- This is actually a good thing!

Parallel programming paradigms

- Data flow
  - Linda
  - Charisma
  - CnC

```
while(err > TOL){
 foreach x,y in Jacobi
  (l[x,y], r[x,y],
  t[x,y], b[x,y]) ←
      Jacobi[x,y].prodBdries();
  (+err) ← Jacobi[x,y].consumeBdries(
                l[x+1,y], r[x-1,y],
                t[x,y-1], b[x,y+1]);
 end-foreach
}
```

Parallel programming paradigms

- Functional
  - Parallel Haskell
  - NESL

```
F(i:int) :
if (c < THRESH)
 g(c)
else
 let
  x = f(i,A),
  y = g(x),
  z = h(c),
  w = p(z)
 in
  F(w+y);
```

Parallel programming paradigms

- Streaming
  - Brook
  - StreamIt
  - CUDA

Parallel programming paradigms

- Multithreading
  - OpenMP
  - Various thread packages
  - TBB

Parallel programming paradigms

- Message passing/driven
  - MPI
  - Charm++

Charm++

- Object-oriented
  - Algorithms written in terms of natural components
  - More objects than processors: *virtualization*
    - Grain size control
    - Cache friendliness
  - Objects are migratable: dynamic load balancing

Charm++

- Communication via asynch. method invocations
  - Implicitly synchronizes objects
  - Explicitly specifies data/control dependencies
  - Parameters or messages

Charm++

- Computation is message-driven
  - Receipt of message associated with task
  - Methods are non-preemptible
  - Reason about ordering of messages, not individual instructions
    - *Structured dagger* (SDAG) helps manage event ordering in reactive programs

Expressing Parallelism in Charm++

- Task parallelism
  - Individual, dynamically created objects (*chares*)
  - Medium-grained computation
- Data parallelism
  - Indexed collections of objects (*chare arrays*)

Case studies

How does all this translate into elegant programs?

Barnes-Hut Algorithm

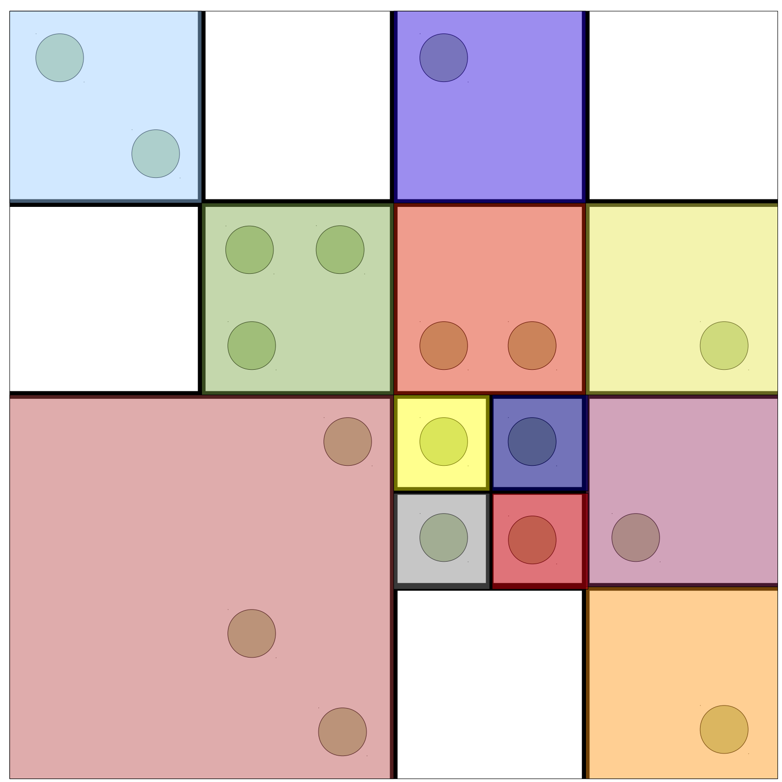
- Efficient, approximate computation of particle trajectories in self-gravitating systems

Barnes-Hut Algorithm

- Domain decomposition and tree building
  - Partition space into compact, disjoint regions containing approximately equal numbers of particles
  - Regions should be arranged in an octree
  - Independent subtrees: **task parallel**
  - Shuffle particles into child bins: **data parallel**
- Force calculation
  - Objects own non-intersecting sets of particles, and calculate forces on them

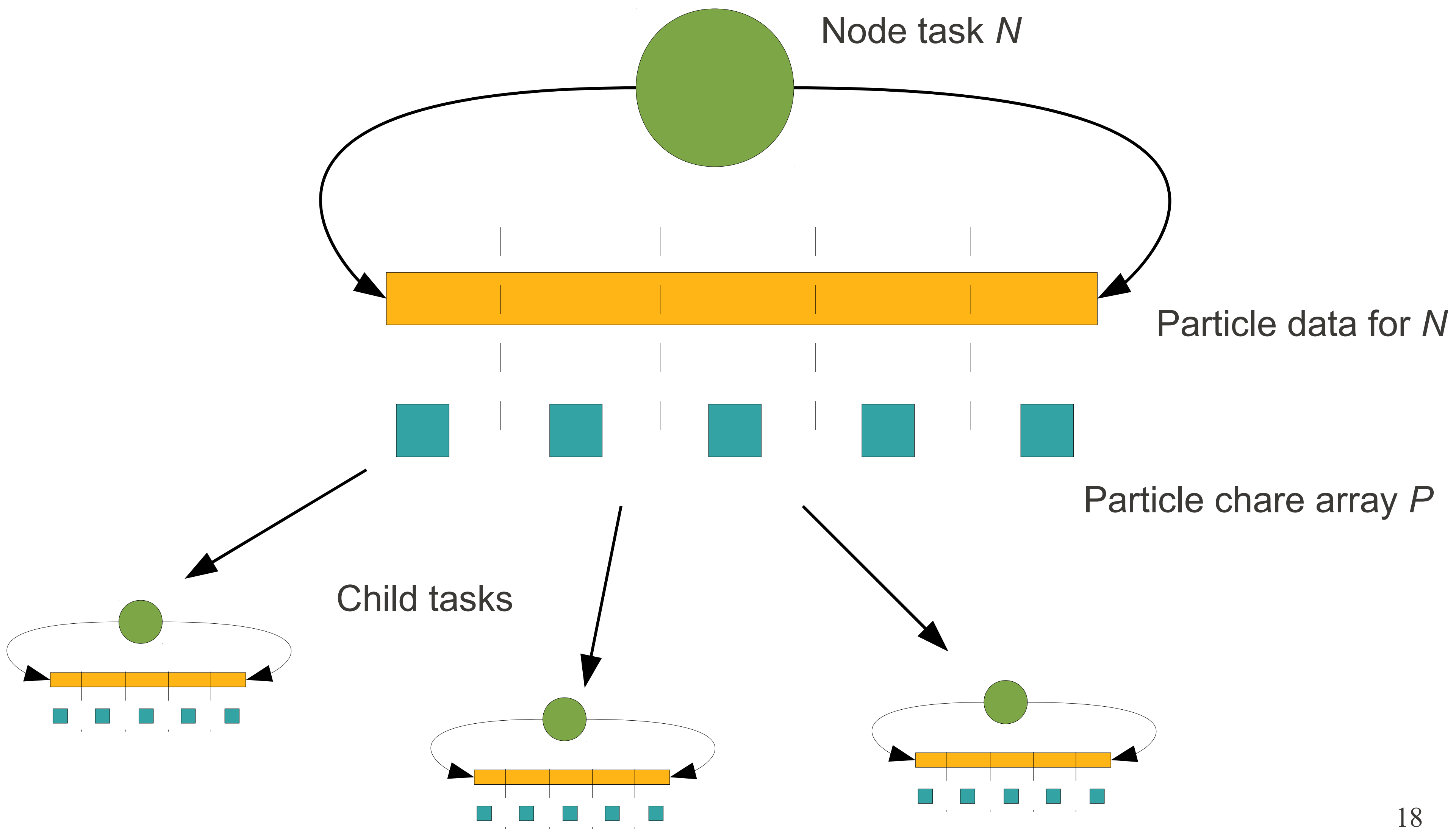
Decomposition

- Recursively divide partition into quadrants if more than τ particles within it



$\tau = 3$

Octree construction



Charm++ pseudocode

```
entry void Worker::shuffleParticles(NodeTaskID parent){
 for (I in 0 to NCHILD-1){
  childKey[I] = (parentKey<<lg(NCHILD)|I);
 }
 for(J in myStartIdx to myEndIdx){
  Find K such that childKey[K] ‡‡ particle[J].key
                   && particle[J].key ‡ childKey[K+1];
  childParticles[K].add(particle[J]);
 }
 reduce(childParticles,parent,NodeTask::recvParticles);
}
```

```
entry NodeTask::NodeTask(Particles parts){
 if(parts.size() <= TAU) sequentialBuild(parts);
 workers.shuffleParticles();
}

entry void NodeTask::recvParticles(Particles *childParts){
 for(I in 0 to NCHILD-1){
  if(childParts[i].size() > 0) NodeTask::ckNew(childParts[i]);
 }
}
```

Charm++ pseudocode

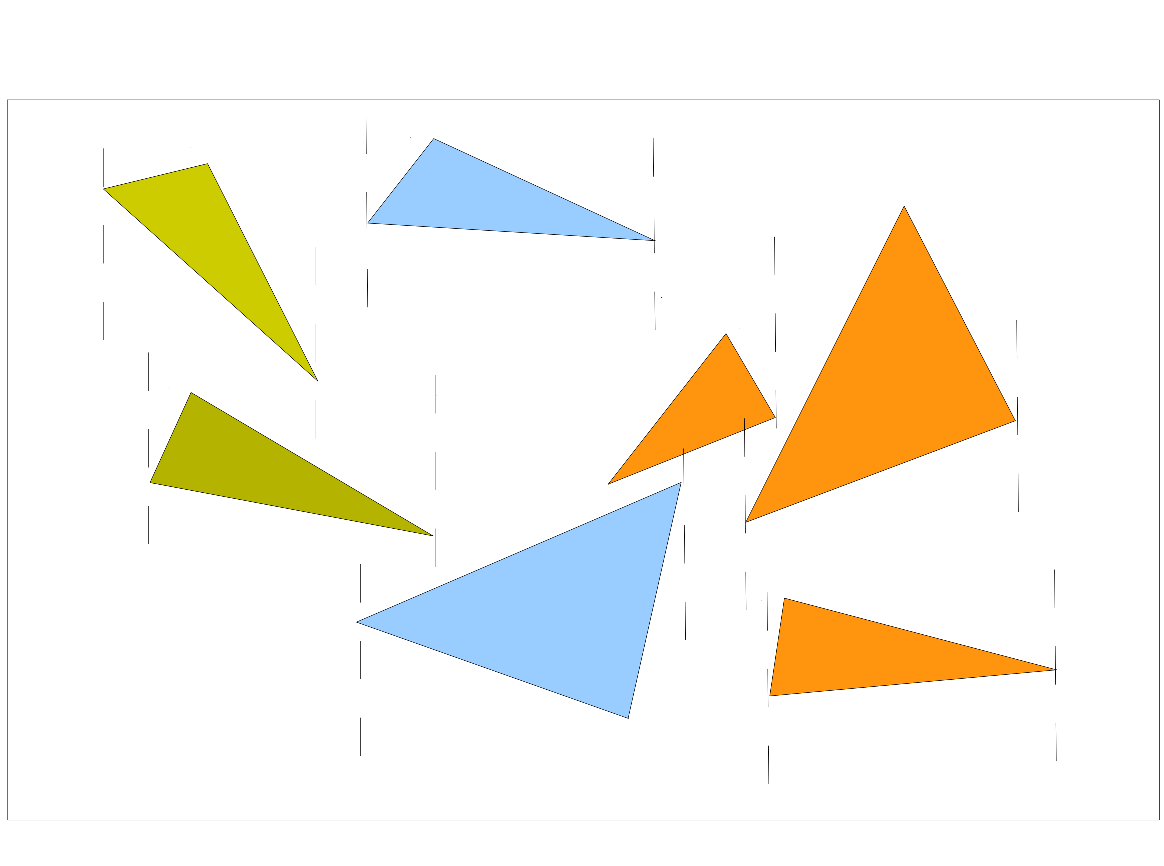
```
entry void Worker::forces(int start, int end){
 for(J in start to end){
  traverseTree(RootID,particles[J])
 }
}
```

SAH-based *k*d-trees

- Used to efficiently render complex graphical scenes
- **Task parallel** construction of independent subtrees (dynamically created *chares*)
- **Data parallel** calculation of node split point (*chare arrays*)

Binary Space Partitioning

- SAH decides position of partition based on triangle distribution and partition surface area



Charm++ pseudocode

- Use SDAG to sequence events in parallel scan

```
entry void Worker::scanTriangleCounts(ActivationRec ar,
                                      NodeTaskID N){
 dist = W >> 1;
 while(dist > 0){
  if(thisIdx < dist){
   ScanMsg m;
   m.NL = myNL; m.NR = ar.nTris-myNR;
   RefNum(m) = dist;
   workers[thisIdx+dist].recvNeighborCounts(m);
  }
  when recvNeighborCounts[dist](ScanMsg m1){
   myNL += m.NL; myNR -= m.NR;
   dist >>= 1;
  }
 }
 Plane bestPlane = calculateSAH();
 reduce(bestPlane,N,NodeTask::getBestPlanes);
}
```

Performance Issues

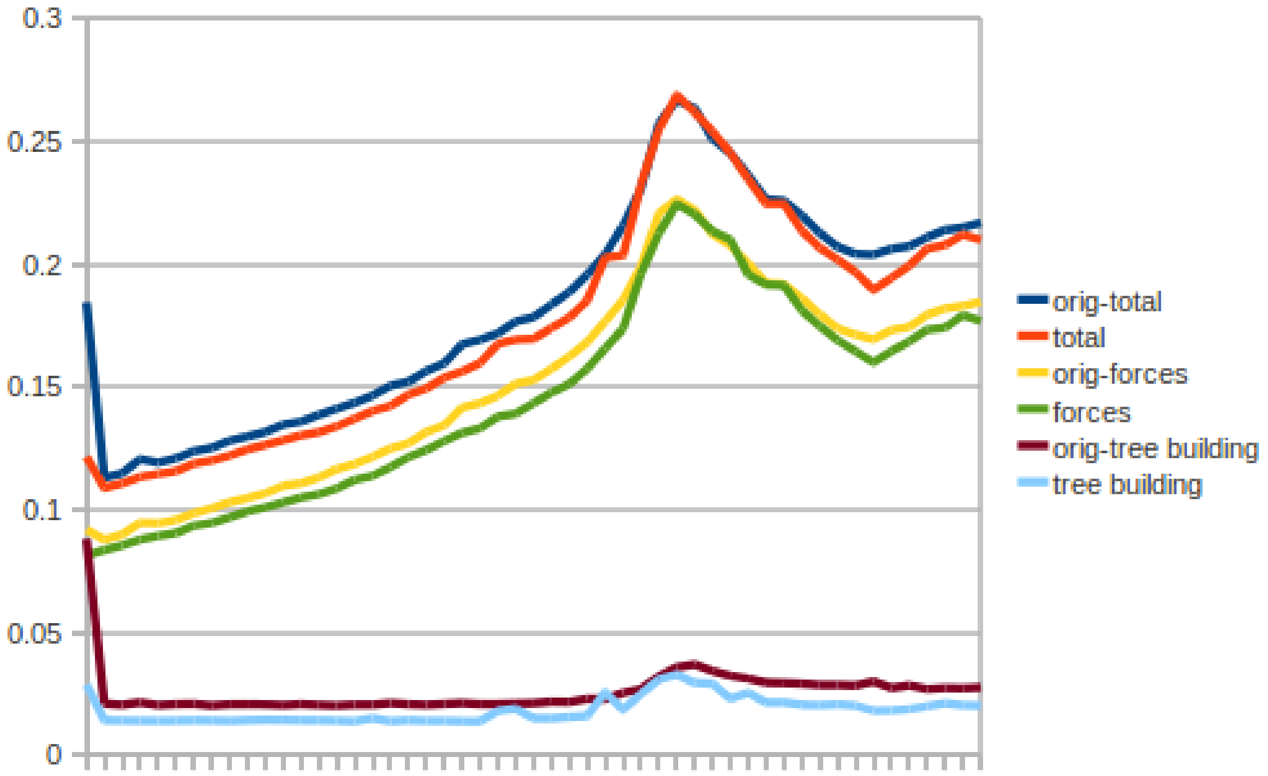
- Optimizations from distributed memory don't necessarily work well in shared memory
  - Spanning-tree multicasts, reductions
  - Recursive doubling scan (with little local work)

Performance Issues

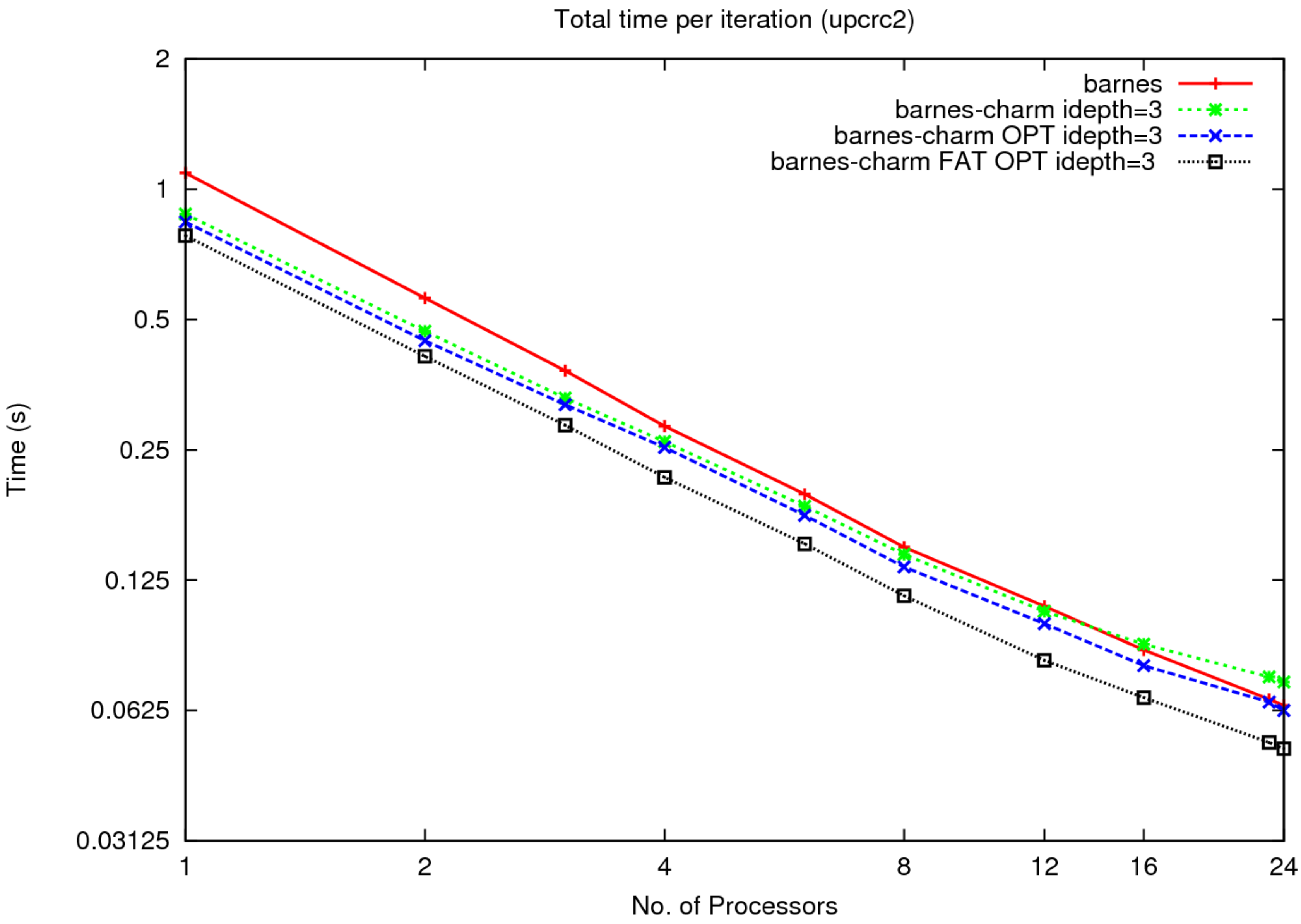
- However, some themes do carry over well
  - Dynamic balancing of tasks (chares)
  - Mapping of data-parallel objects to cores
    - Pipelining
    - NUMA-awareness
  - Prioritization of tasks
    - Do not delay tasks on critical path

Barnes-Hut

- 8-core Xeon E5405 2.0 GHz
- Slight gain in performance; 20% reduction in code



Barnes-Hut results



More abstractions

- *Chunked-array* abstraction to share data without needless cache traffic
- *Boomerang arrays* to make seamless transition from single-node to multiple SMP nodes

Conclusions

- Data and control dependencies are explicit in a message-driven system: no locks, barriers, fences
- Medium-grained tasks lead to good cache performance
- Adaptive RTS does much of the dynamic optimization of a program
  - Load balancing
  - Prioritization of tasks to speed up critical path
  - Automatic overlap of computation and communication