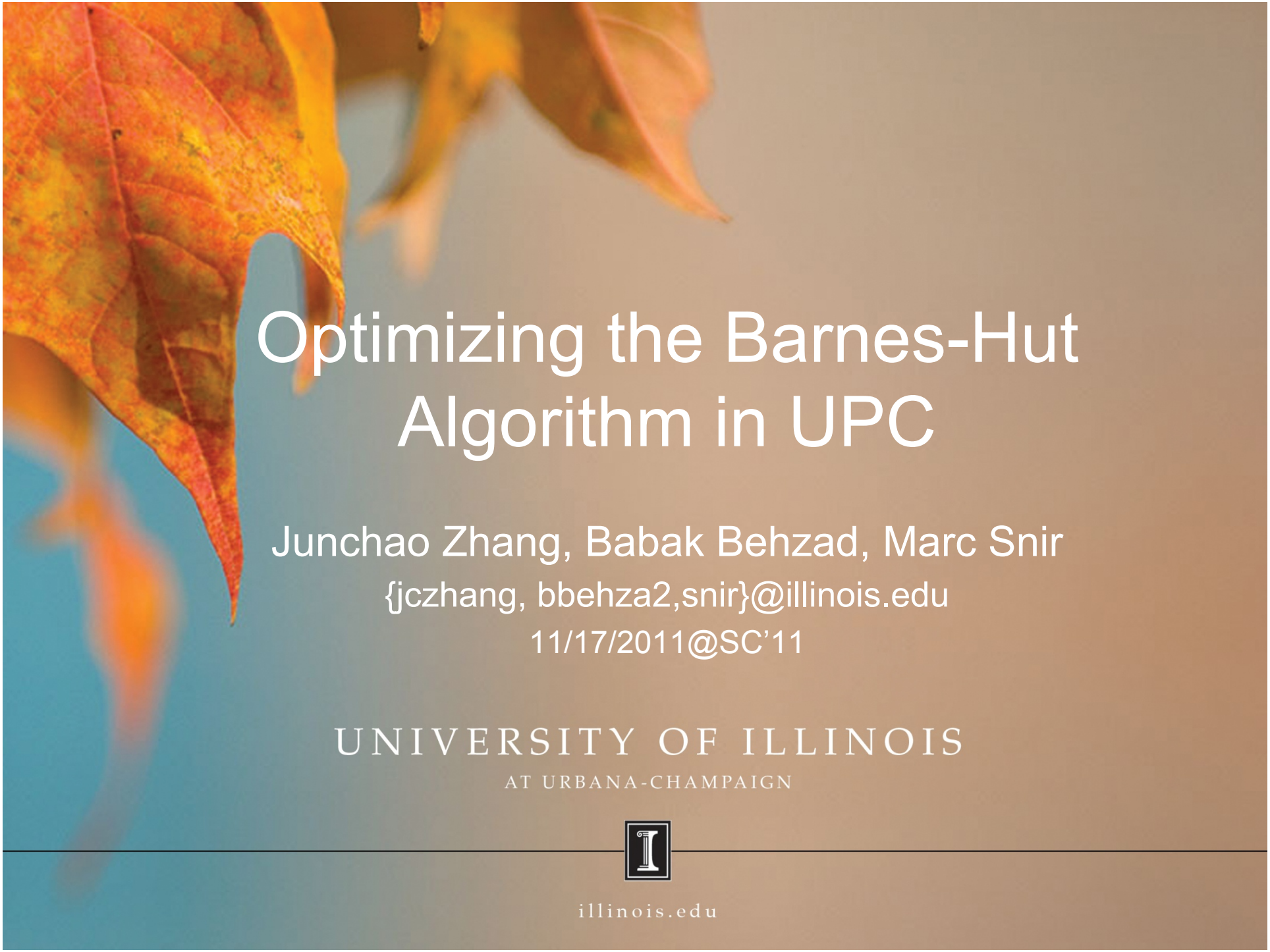


Optimizing the Barnes-Hut Algorithm in UPC

Junchao Zhang, Babak Behzad, Marc Snir
{jczhang, bbehza2,snir}@illinois.edu
11/17/2011@SC'11

UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

illinois.edu

Motivation (1)

- Parallel programming in MPI is often criticized as hard and hurting users' productivity
  - The user has to explicitly code with send/recv to effect communication
  - It is especially difficult when writing irregular applications, where communication is dynamic and data-dependant

Motivation (2)

- Partitioned global address space (PGAS) languages such as UPC, CAF, Chapel, X10 etc. promise to ease the problem by providing a global name space
  - Remote data are accessed by just referencing their names
  - Communication is implicitly done by the runtime
- Question: For irregular applications, can PGAS ease programming while providing performance?
- There are few studies of using PGAS for irregular applications

Contributions

- We thoroughly studied the coding of Barnes-Hut, a classical irregular application, in UPC
  - – A naïve shared-memory implementation has abysmal performance
  - ✔ A sequence of optimizations result in UPC code that is competitive with message-passing code
  - ✗ These optimizations are tedious and make the program MPI like, which offsets much of the benefits of PGAS
- We discussed what additions are needed in PGAS languages to achieve performance while avoiding explicit “MPI-style” coding

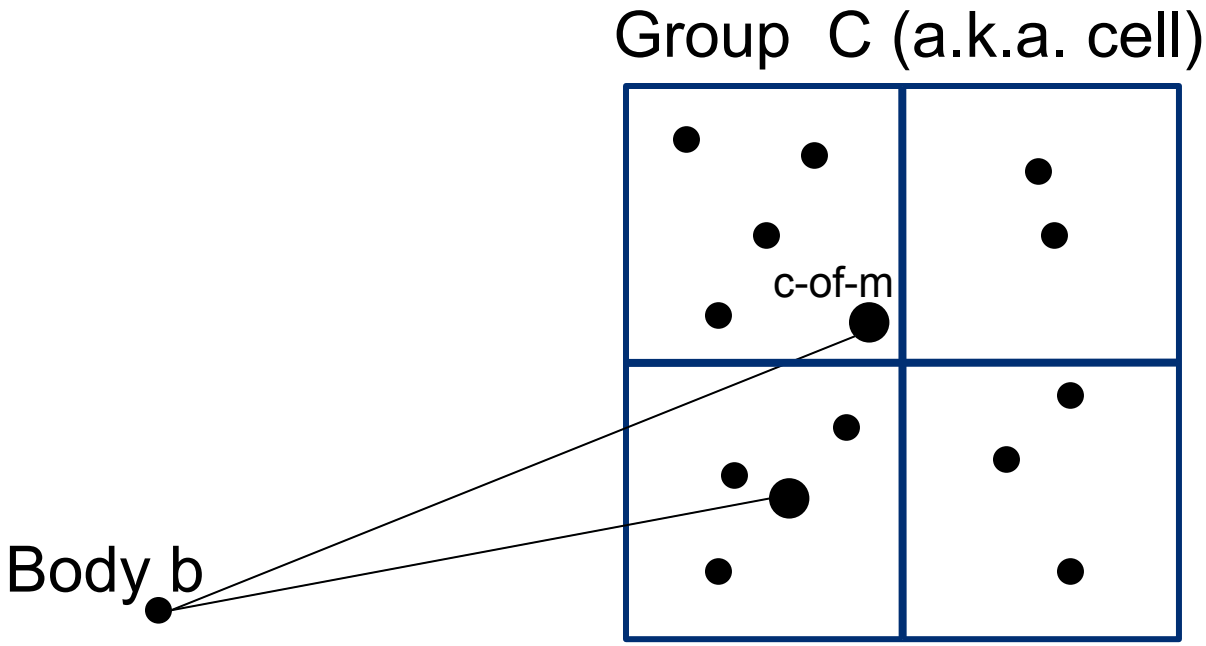
Outline

- Motivation & contributions
- Introduction to Barnes-Hut (BH) and UPC
- Optimizations
- Discussion and conclusion

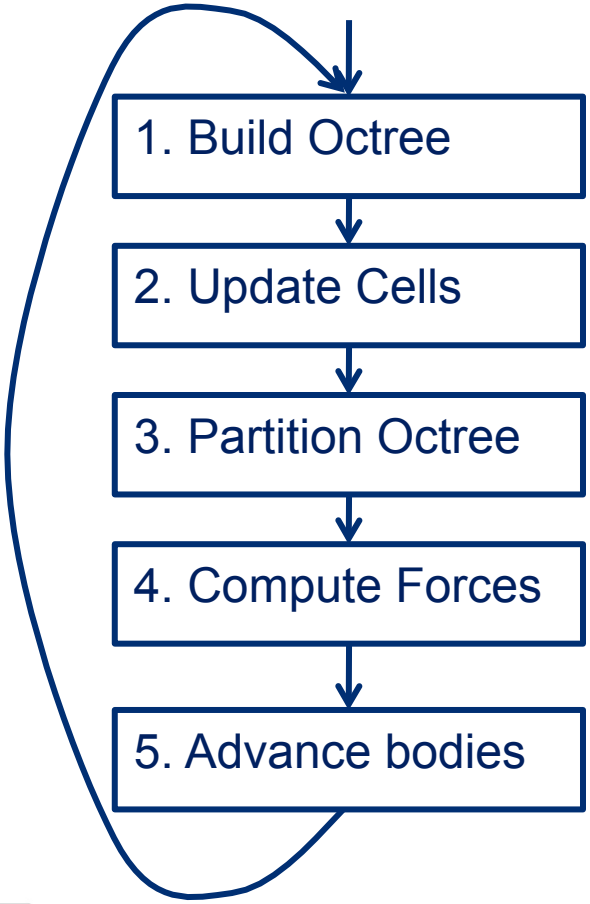
Barnes-Hut algorithm

- BH is a fast algorithm for the n-body problem, in which every body exerts forces upon each other
- A direct algorithm for n-body is of $O(n^2)$
- BH achieves *O(nlogn)* by recursively splitting the space and organizing bodies in an Octree
- BH computes force between a body and a group of bodies, instead of with individuals in the group

Far enough & cell opening

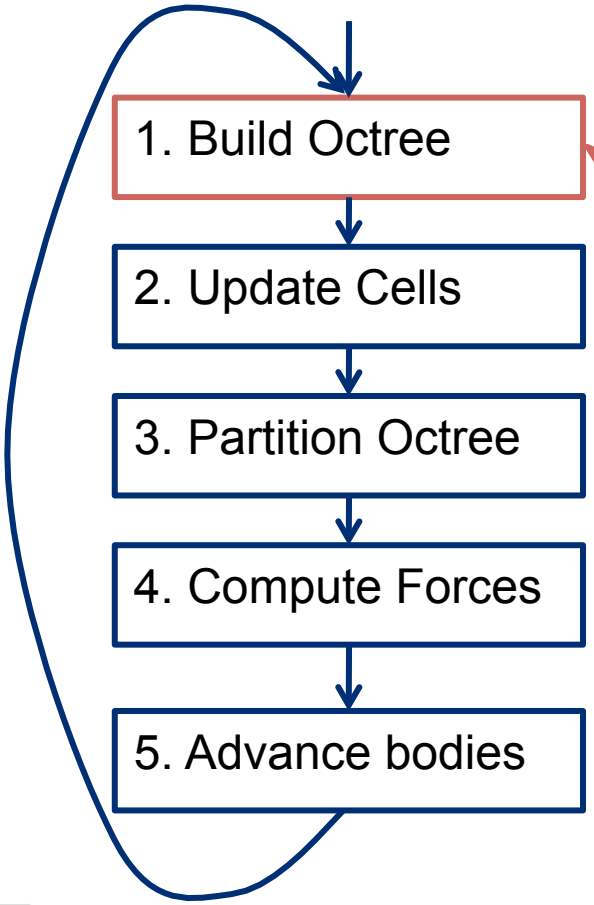


SPLASH-2 BH



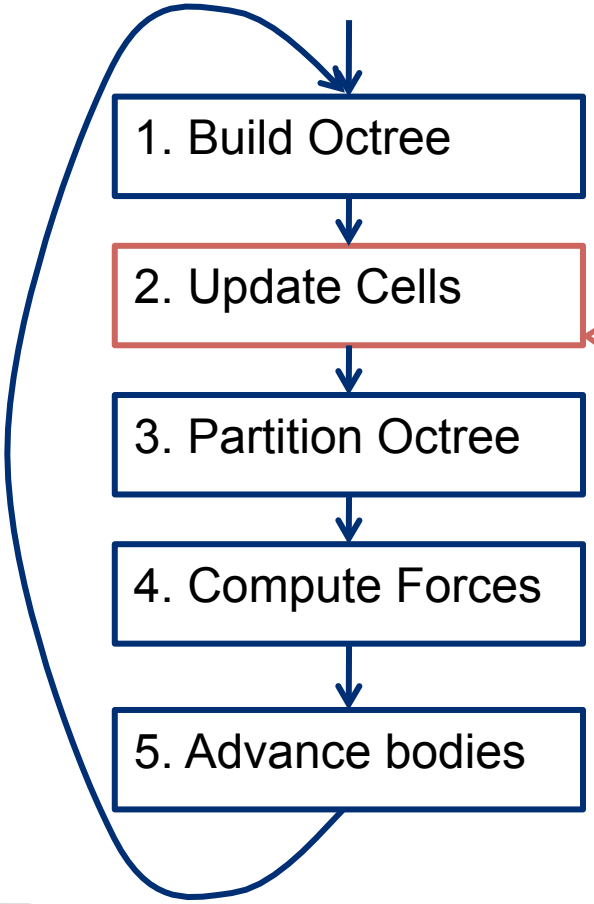
- The code is highly optimized for shared memory
- The benchmark runs multiple time-steps
- Each time-step has 5 phases; each phase is executed in parallel

SPLASH-2 BH



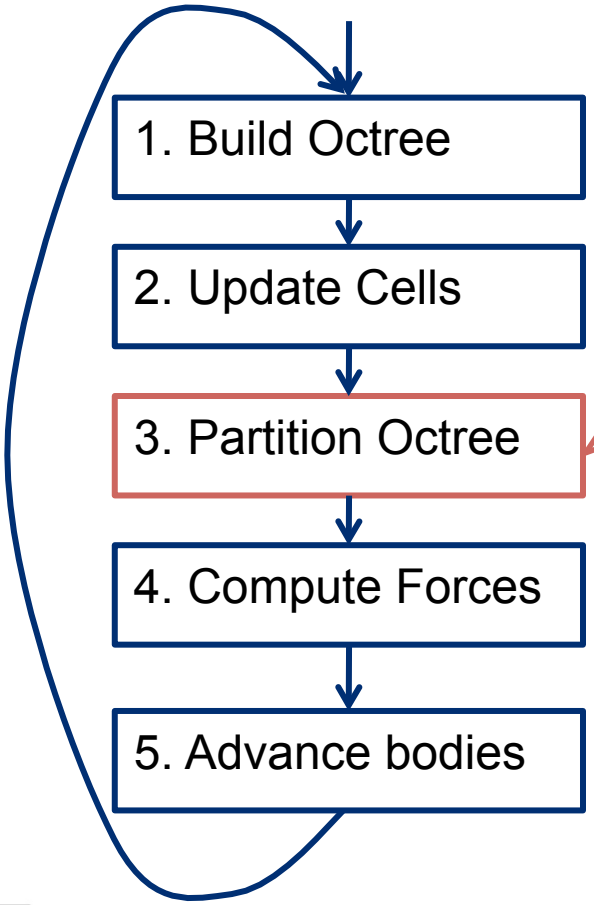
- Processes concurrently insert their bodies into the octree, using locks to avoid races

SPLASH-2 BH



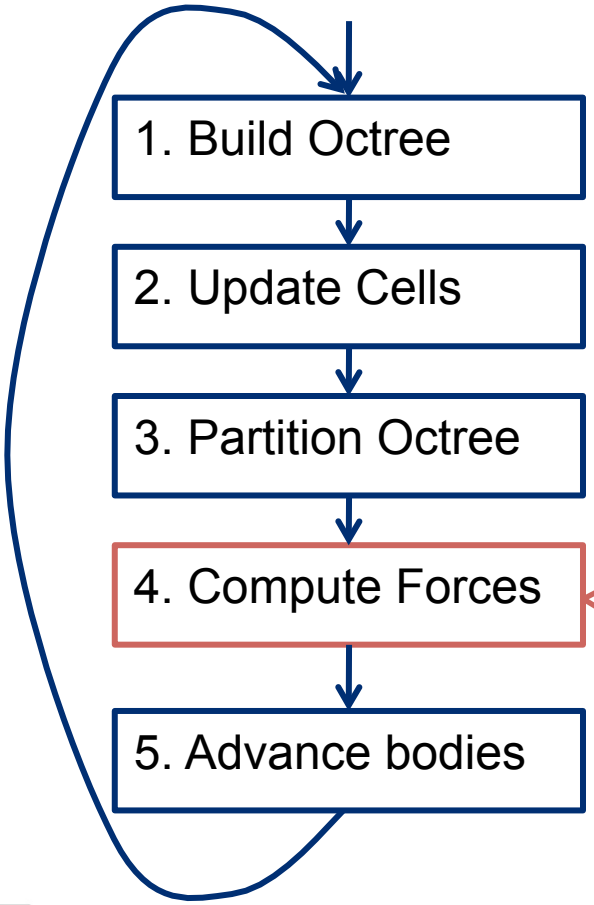
- Processes compute center of mass & cost information for their cells
  - Cost of a body = # of body-cell interactions at previous time-step
  - Cost of a cell = total cost of bodies in the cell (cost used for load-balancing)

SPLASH-2 BH



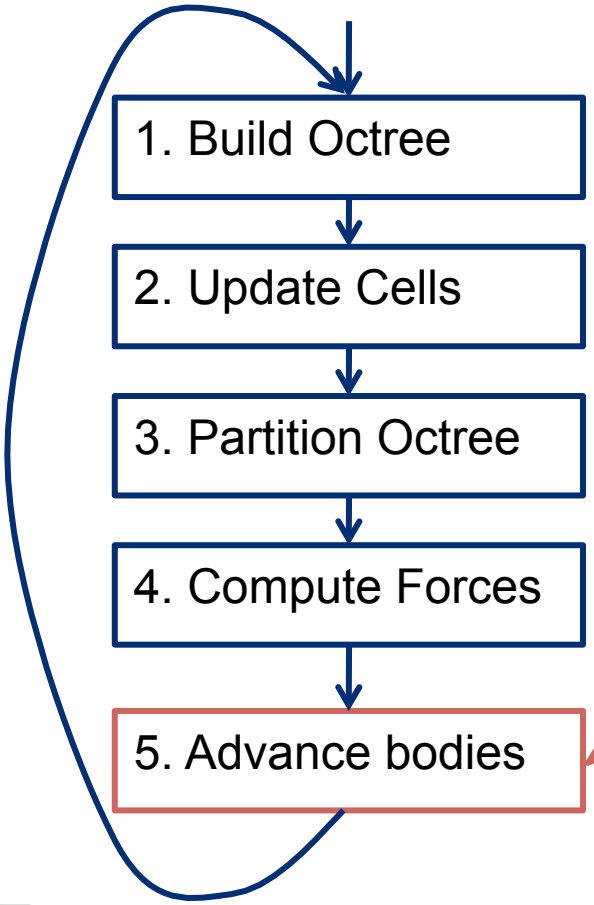
- Processes partition bodies based on their costs and positions
- "A body is owned by a process" after the partitioning

SPLASH-2 BH



- Each process computes forces for its bodies by traversing the tree

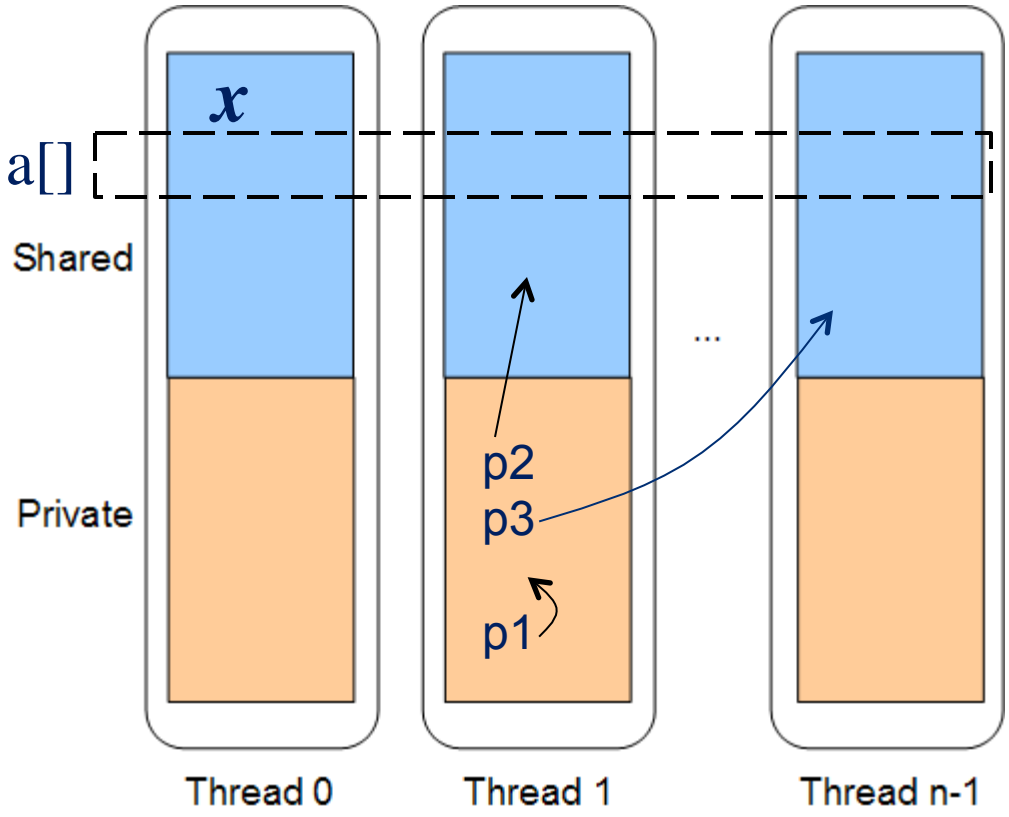
SPLASH-2 BH



- Processes update their body positions

UPC introduction

- Private and distributed shared memory
- Shared scalars (*x*)
- Distributed shared arrays (a[])
- Regular pointers to private(p1) and fat pointers to shared (p2, p3)
  - Dereference speed
    - p1 > p2 >> p3
  - Casting
    - p1 = p2; // OK and improves performance



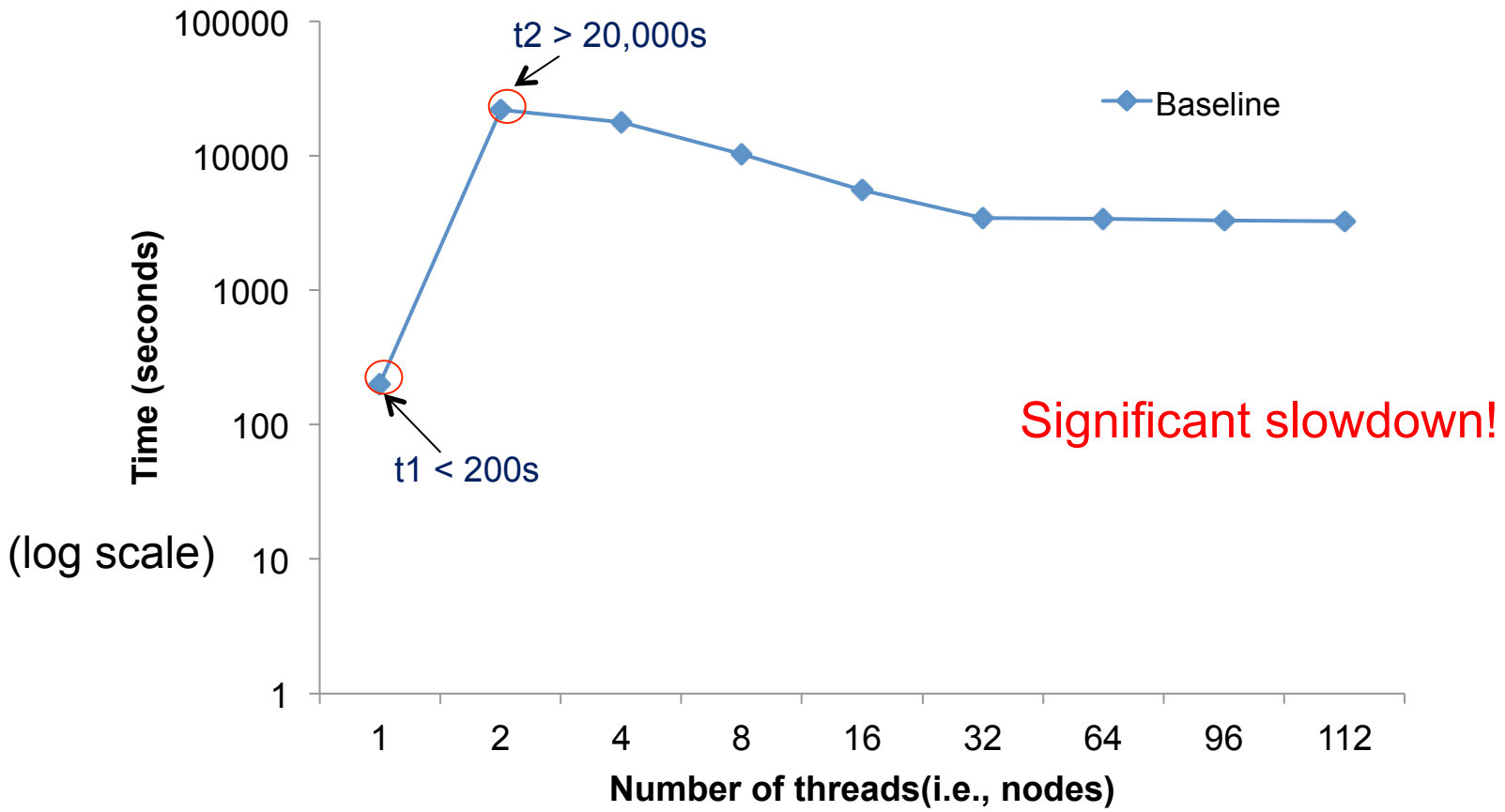
A baseline UPC BH

- Taken from Berkeley UPC 2.11
- Nearly a literal translation from the SPLASH-2 BH
  - Global parameters are declared as shared scalars
  - Parent and child cells are chained using shared pointers
  - Bodies are stored in a shared array
  - Tree nodes are allocated in shared memories
- Not coded for performance
- But is the most straightforward transition from shared-memory to PGAS

Test environment and approaches

- Tested on Blueprint, a Power5/AIX cluster
  - Each node has 16 cores
  - Used Berkeley UPC compiler 2.12 built on LAPI
- Tested nearly with the default settings as SPLASH-2
  - The body data were generated by the Plummer model
  - Tested 4 time-steps, but didn't count the first 2 steps
- Used 1 UPC thread per node and 2M bodies for strong scaling test

Strong scaling of the baseline code



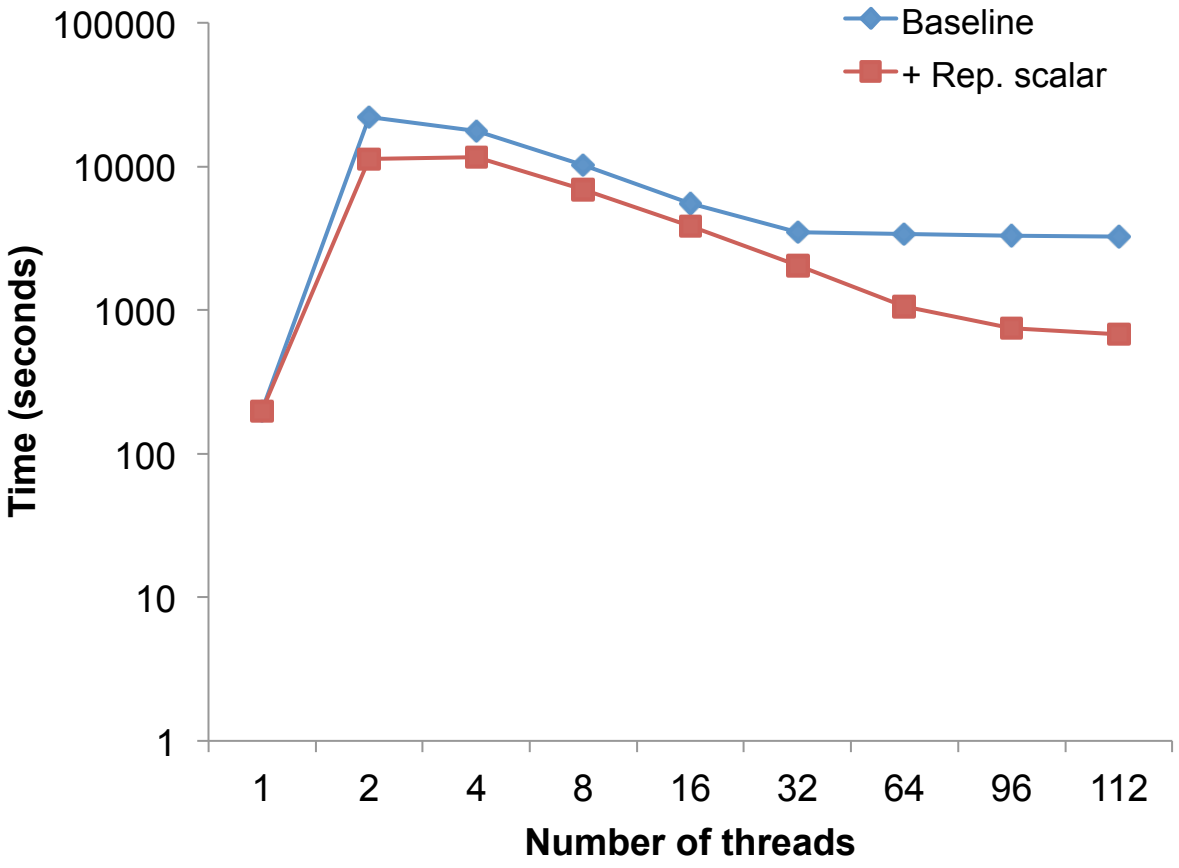
Outline

- Motivation & contributions
- Introduction to Barnes-Hut (BH) and UPC
- ➢ Optimizations
- Discussion and conclusion

Opt1: Replicate shared scalars

- Three hot shared scalars in the naïve code
- Two of them are input parameters. They're written-once and keep unchanged
  - Our approach: Privatize them; Let each thread parse the user's input
- One of them is updated by thread 0 per step
  - Our approach: Add a private copy on each thread, copy the shared scalar to local ones after thread 0's update

Strong scaling with rep. scalar

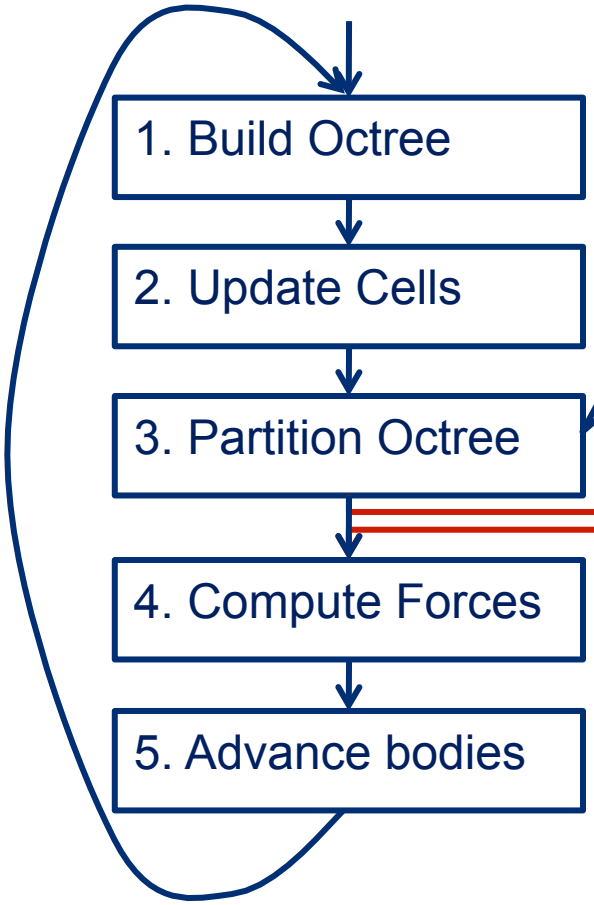


Compared with the previous version at 112 threads

Phase	Time reduced
Tree-building	86%
Force comp.	79%
Body-adv.	66%
Total	79%

$\text{Speedup}_{112} = t_1/t_{112} < 1$

Opt2: Body redistribution

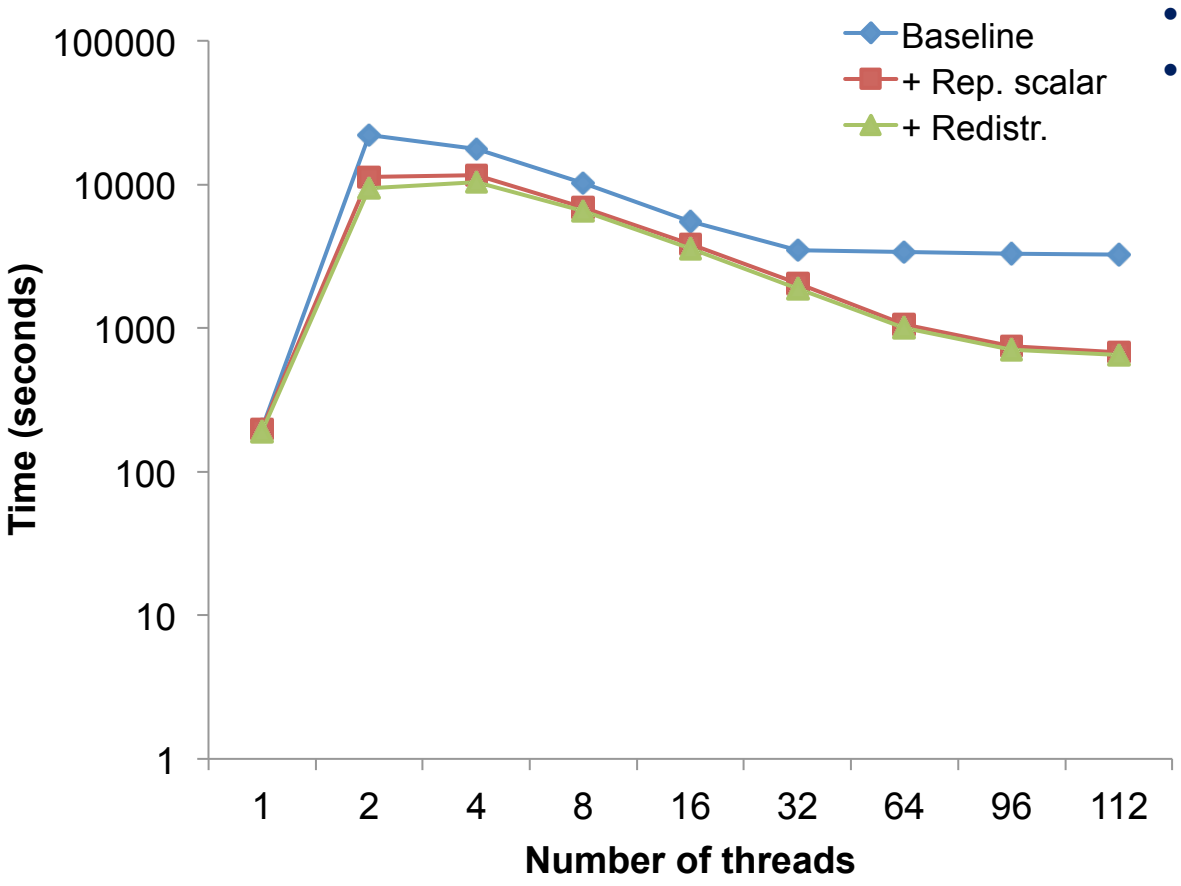


• <body, proc> ownership is dynamically established here
• However in the naïve code, bodies are statically distributed

Body Redistribution

• Copy bodies owned by a thread to its local memory
  - ➢Improve locality
  - ➢Opportunities of pointer casting: shared pointers to my bodies can be safely cast to local ones

Strong scaling with redistr.



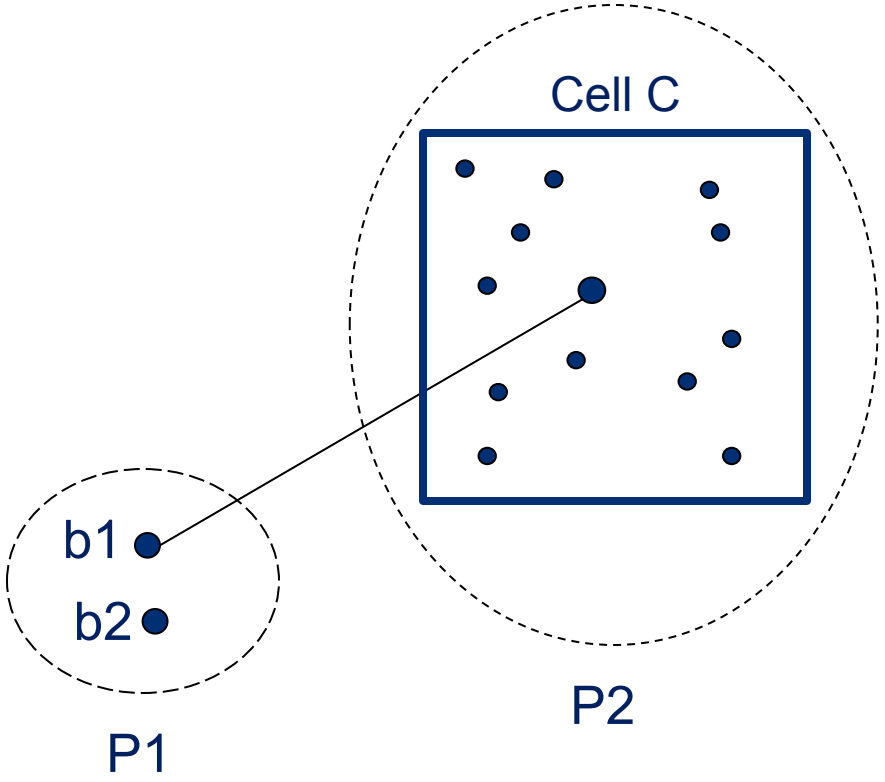
- Body redistr. cost = ~0 sec
- ~2% of bodies migrate per step

Compared with the previous version at 112 threads

Phase	Time reduced
Cell-updating	95%
Body-adv.	~100%
Total	4%

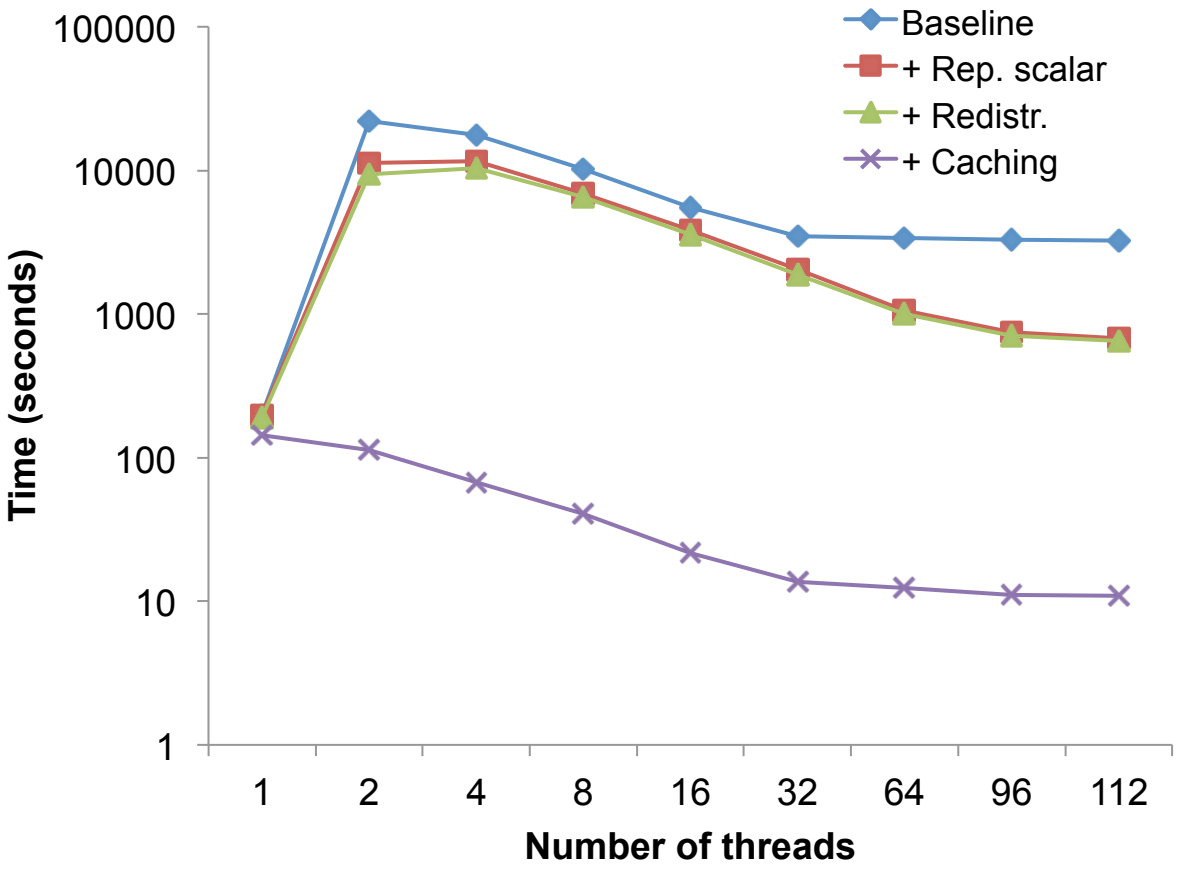
$Speedup_{112} < 1$

Opt3: Cache remote cells



- Our approach
  - Add a flag to each cell. It is initially cleared. If it is set, that means the cell's children all have been cached
  - Before opening a cell, test the flag. If it's not set, cache all child cells, redirect child pointers of the cell to local copies, and set the flag

Strong scaling with caching



Compared with the previous version at 112 threads

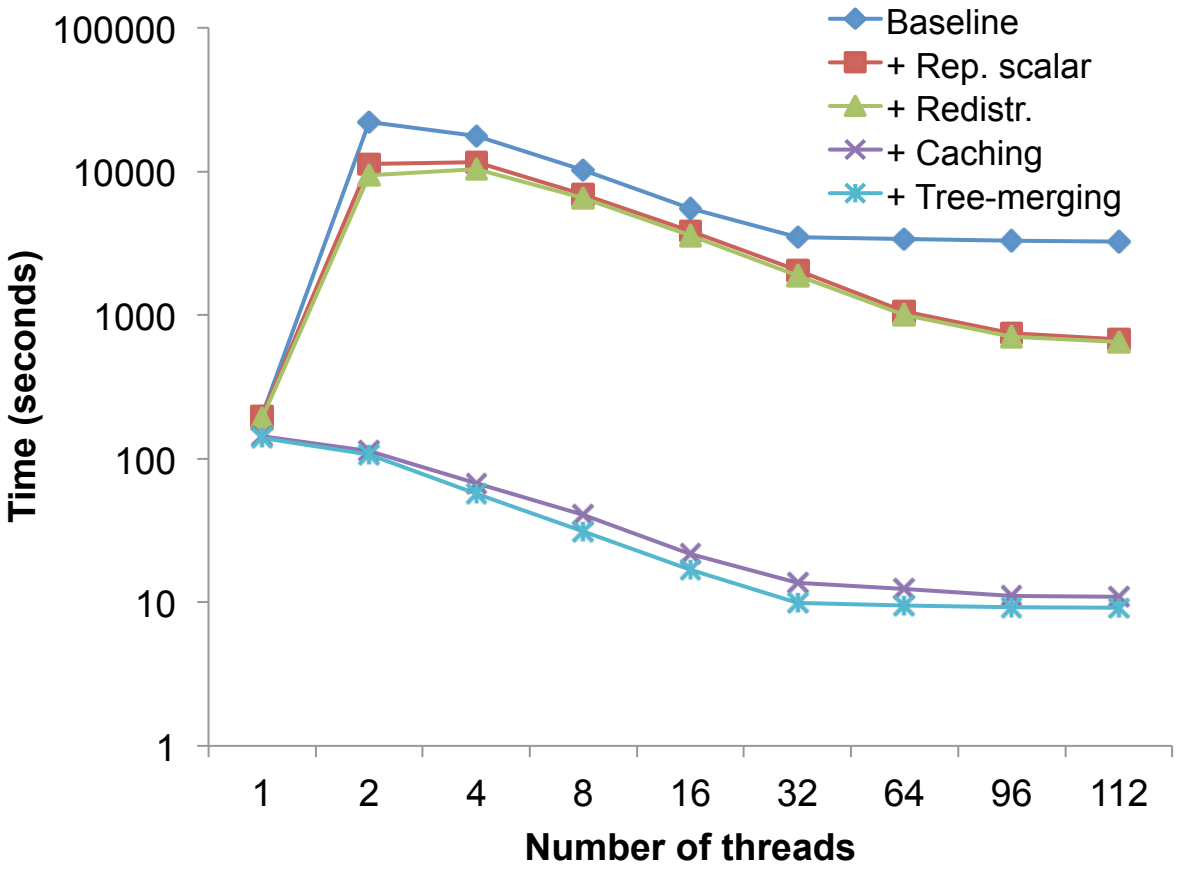
Phase	Time reduced
Force comp.	99%
Total	98%

$Speedup_{112} = 13$

Opt4: Tree-merging

- Insight lost in naïve tree-building
  - Bodies owned by a thread are close to each other. Most times, only one thread is building a subtree such that locking here is redundant
- New approach(Proposed in Singh'93, but w/ improvements)
  - Each thread first builds a local tree with its bodies without using any locks
  - Furthermore, because c-of-m computation commutes, we can also do local c-of-m computation
  - Threads then merge local trees as well as c-of-m

Strong scaling with tree-merging



Compared with the previous version at 112 threads

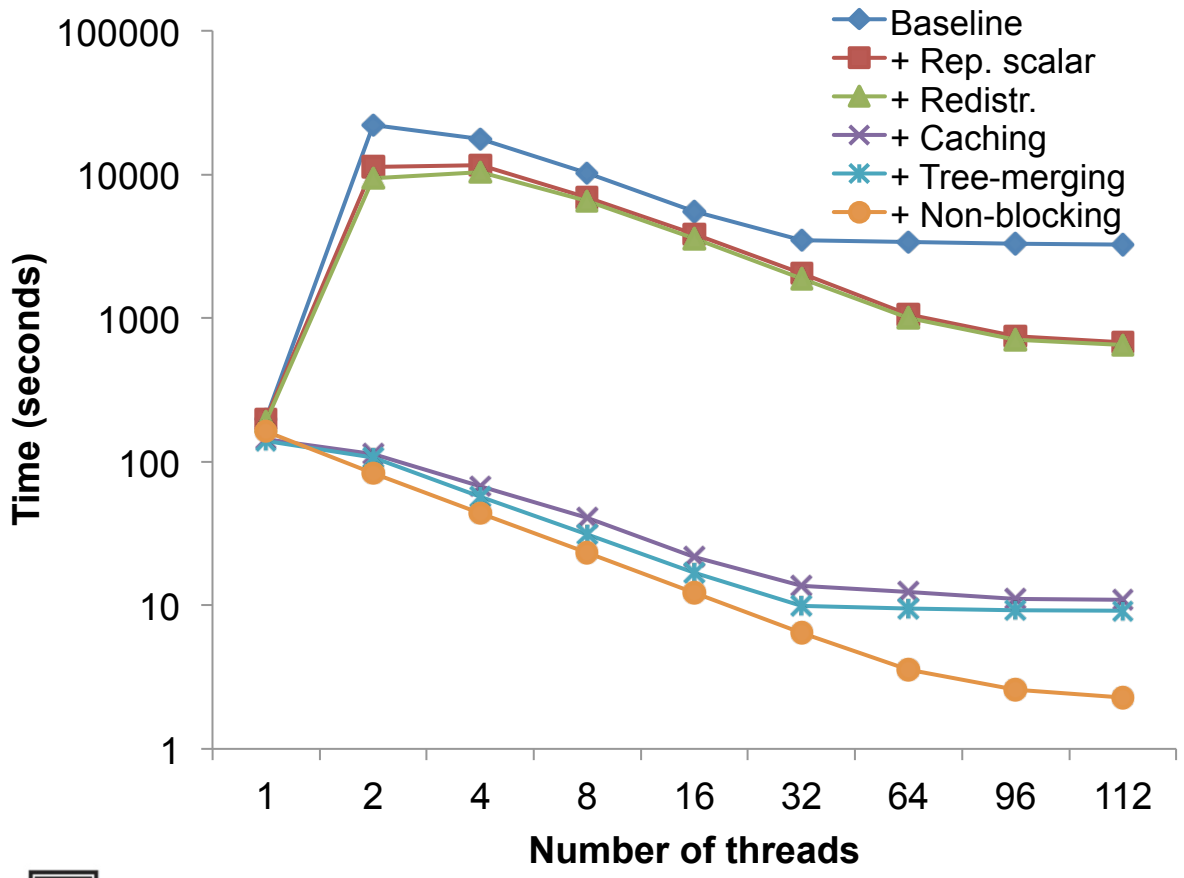
Phase	Time reduced
Tree-building	74%
Total	15%

$Speedup_{112} = 15$

Opt5: Non-blocking communication & message aggregation

- Two levels of parallelism in force computation
  - Threads can compute forces for their bodies *in any order*
  - When opening a cell, threads can proceed with child cells *in any order*, because the force summation commutes
- Our approach
  - Delay a remote cell access request, continue with other local <body, cell> computations
  - When aggregate enough requests, send them out in a non-blocking communication call

Strong scaling with non-blocking & message aggregation

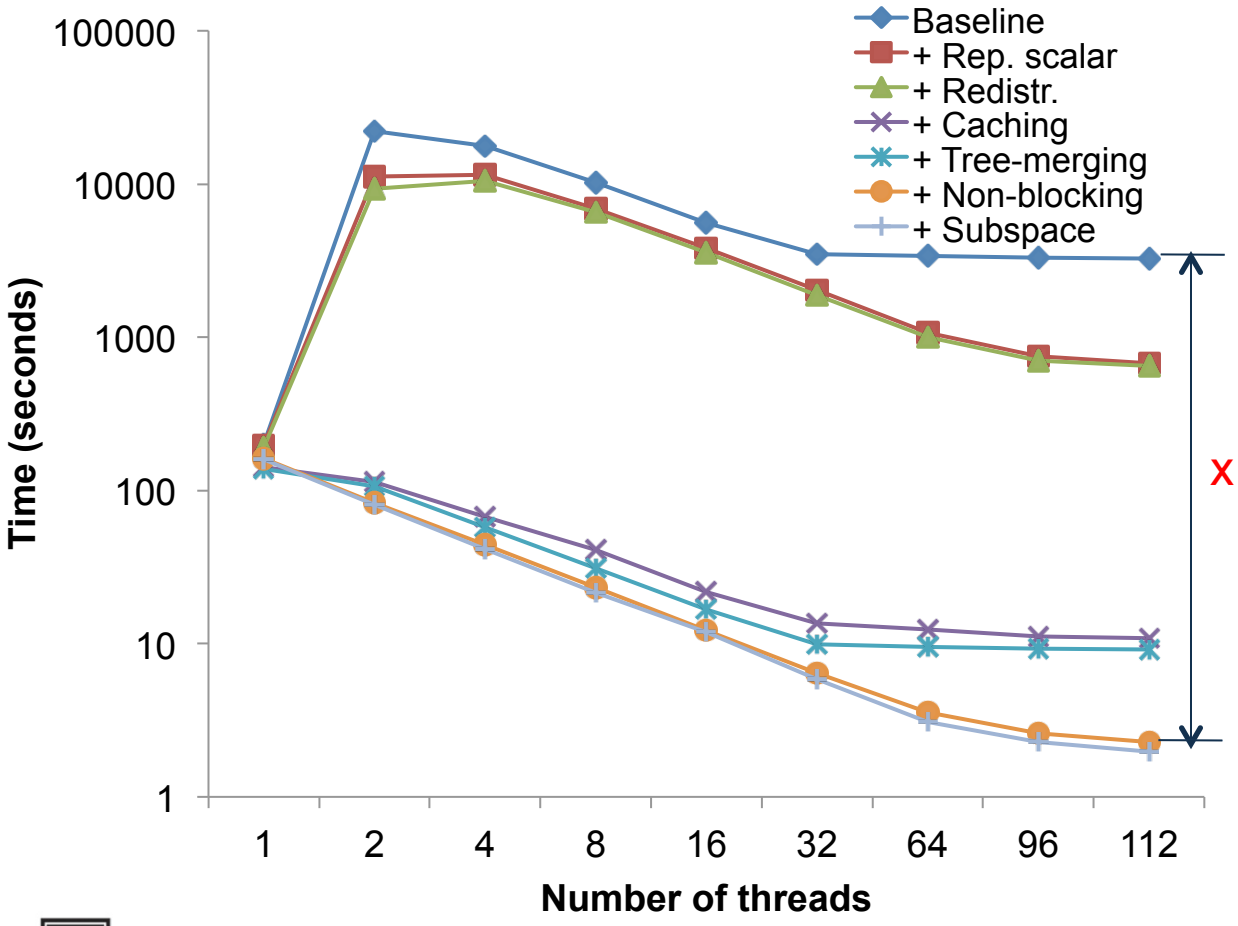


Compared with the previous version at 112 threads

Phase	Time reduced
Force comp.	81%
Total	75%

$Speedup_{112} = 71$

Strong scaling with another opt
(i.e., subspace tree-building, talked later)



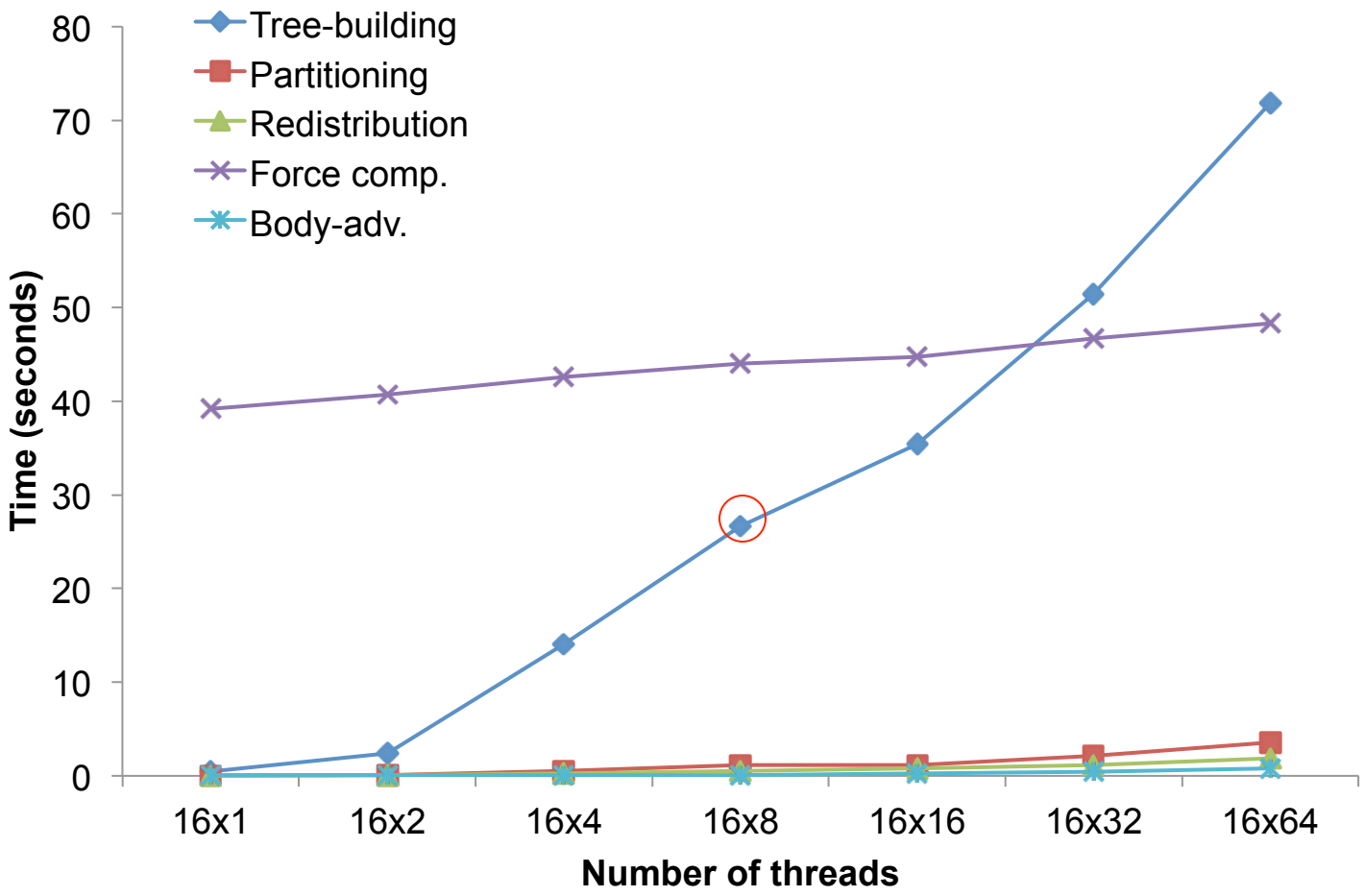
x1644

$Speedup_{112} = 81$

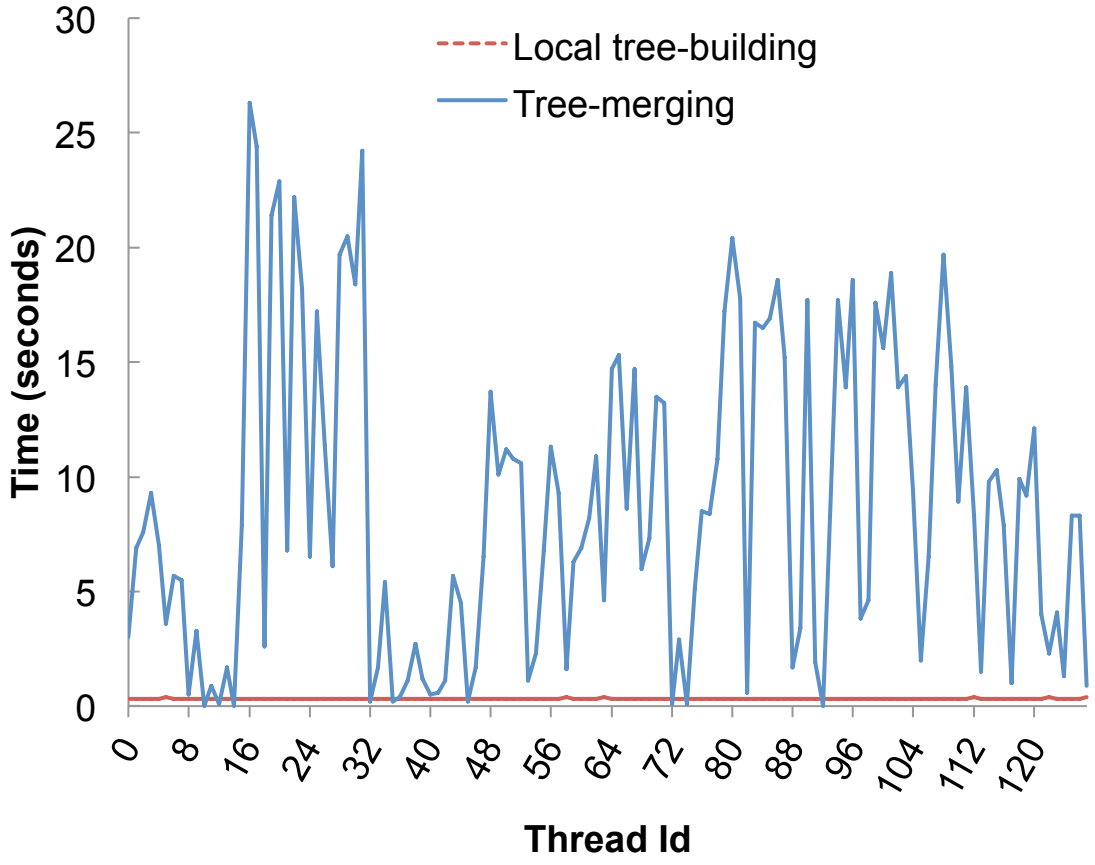
Weak scaling test approach

- Use 16 UPC threads per node
- Use BUPC –*pthreads* support to take advantage of intra-node HW shared-memory
- 250K bodies per thread

Weak scaling of each phase with all opts



Tree-building@128 threads

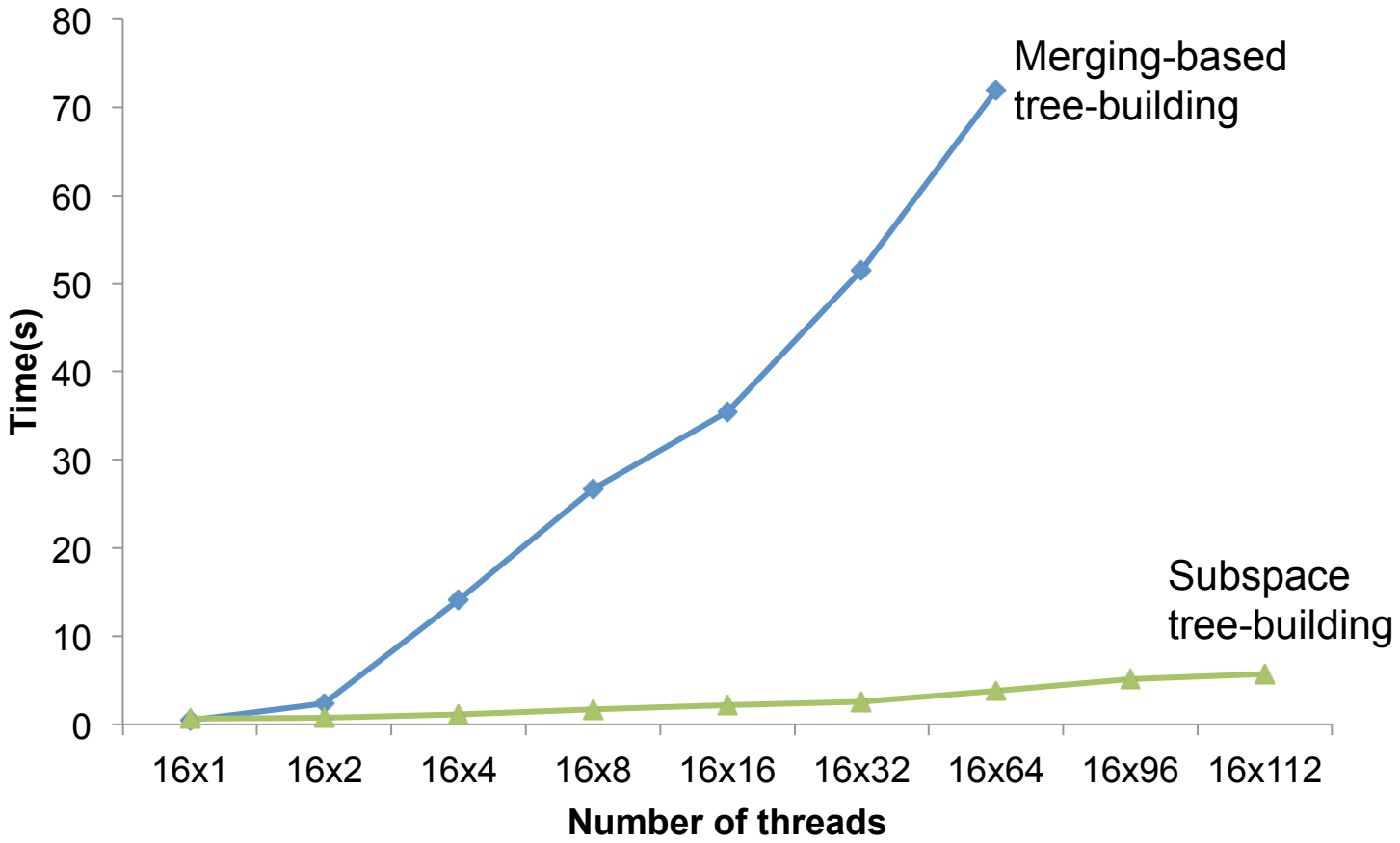


- Local tree-building time is nearly identical at all threads
- Tree-merging time has huge variance
- Probably due to lock contention: late-comers are repeatedly delayed and do more work

Opt6: Subspace tree-building

- Intuition
  - If we can build local trees *without* overlap, then tree-merging can be lock-free and very fast (Proposed in Shan'98, w/ key improvements)
- How
  - Split the space, assign *non-overlapping* subspaces to threads
  - Threads only build subtrees in their subspaces
  - Details are in the paper

Weak scaling of two tree-building algs



Comparison with a Message-Passing Implementation

- ChaNGa is a state of the art Cosmology simulation code written in Charm++
- Tested on Ranger@TACC, an X86/Linux cluster
  - 16 cores/node, Infiniband, with BUPC –pshm
- Tree inputs from ChaNGa benchmarks, measured average time per step in sec.

Input(# of bodies)	dwf1 (5M)		dwf1.6144(50M)		Lambb(80M)	
# of threads	16	128	168	1344	432	1360
ChaNGa	23.8	3.3	29.4	10.9	19.6	9.1
UPC BH	19.9	2.9	28.2	7.2	16.7	8.2

UPC BH 10% better, on average

Outline

- Motivation & contributions
- Introduction to Barnes-Hut (BH) and UPC
- Optimizations
- ➢ Discussion and conclusion

Discussion (1)

- A naïve shared-memory style UPC BH had abysmal performance
- Need multiple optimizations to reduce communication and synchronizations costs in UPC
- Resulting code makes heavy use of explicit (message-passing style) communication
- Can we get the desired performance without “coding MPI code in UPC”?
- We believe that
  - Pure compiler/run-time approach is not sufficient
  - Compiler/run-time + modest UPC extensions can do most of the job

Discussion (2)

- Many of our optimizations are different forms of software caching. Several factors facilitate it
  - Objects (e.g., bodies or cells) are “logical cache lines”. This avoids false sharing
  - Objects are accessed through pointers. Associative lookup is avoided by pointer swizzling
  - The coherence state of an object (shared, exclusive), and the object owner (for exclusive) do not change during a computation phase. All objects of the same type are in the same state. This bulk change is easier to handle than individual, asynchronous changes tracked by normal coherence protocols
- *Hypothesis*: Simple directives and good compiler/run-time support can provide efficient software caching

Optimizations (1)

Shared scalar replication

- Standard compiler optimization (privatization)
- Can be helped with directives

Optimization (2)

Body redistribution

- Each thread has list of bodies it will own in next phase and the bodies are redistributed accordingly
  - Redistribution (probably) has to be specified by user
  - Compiler + run-time can handle the actual data movement and pointer swizzling to redirect accesses to local copy

Optimization (3)

Cache remote cells in force computation

- User can mark objects of type cell as read-only for the next phase
- Run-time can handle caching

Optimization (4)

Non-blocking communication and message aggregation in force computation

- Two levels of parallelism provide parallel slackness for latency hiding
  - User specifies parallelism
  - Compiler & run-time hide latency with concurrent multitasking

Conclusion

- A shared-memory programming model must have two essential features
  - Shared variables can be accessed by any thread
  - Shared variables can be cached without changing names to access them
- We believe that PGAS languages can provide such a model – *if suitably extended*
- **Future work**: validate this hypothesis and test it on other applications

Thanks & Questions