# Optimizing the Barnes-Hut Algorithm for Multicore Clusters

Junchao Zhang, Babak Behzad, Marc Snir
Department of Computer Science
University of Illinois at Urbana-Champaign
{jczhang, bbehza2, snir}@illinois.edu

## ABSTRACT

We present a Barnes-Hut (BH) algorithm coded in the partitioned global address space (PGAS) + X programming style. Between nodes, remote data is accessed using one-sided communication. In a node, multitasking is used to hide network latency. The code consumes less memory per core, invokes less internode communication, enjoys better load balancing strategies, which makes it well suited for multicore clusters.

## Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

## General Terms

Languages, Algorithms

## Keywords

Barnes-Hut, N-body, PGAS, Cluster, Multicore

## 1. INTRODUCTION

The evolution of supercomputers exhibits several important trends: 1) The number of cores per node keeps increasing; 2) The amount of memory per core is decreasing; 3) One-sided communication (rDMA) is increasingly well supported on the interconnection networks. One-sided can have less software overhead, since code is executed only on one of the two communicating nodes; this results in lower latency. Also, it is easier to code applications with dynamic communication patterns using one-sided communication, as it avoids the need to compute required sends.

The natural idiom for irregular applications with dynamic communication patterns is to use remote reads, or get operations to access remote data. In order to achieve good performance, it is essential to hide the latency of the long round-trip of a remote memory access. Such latency hiding is most conveniently achieved by descheduling tasks that are blocked on a remote access and reusing the core to run another, ready to execute task. This requires low overhead task scheduling. Low overhead task scheduling also enables efficient load balancing, ensuring that all cores are used. It is often the case that multiple cores on a node will access the same data structures, either local or remote. Thus memory pressure can be alleviated and bandwidth can be saved through sharing data within nodes.

We demonstrate in this paper the use of these techniques in the context of the BH algorithm [1]. It is the first hybrid BH code that integrates intranode multithreading and internode one-sided communicaiton and uses multitasking to hide network latency.

## 2. BH ALGORITHM

BH is an $O(n \log n)$ algorithm for the n-body problem, which simulates a system of $n$ bodies where bodies exert forces mutually. It partitions the 3D space hierarchically into cells using an octree. To compute forces on a body, it traverses the octree recursively from the root. If a cell is sufficiently *far away* from the body, it just stops there and uses one body-cell interaction to approximate the many body-body interactions when computed directly. If not, it continues with children of the cell (i.e., *opens* the cell). The "far away" test is defined as $l/d < \theta$, where $l$ is the size of a cell, $d$ is the distance from the body to the center of mass of the cell, and $\theta$ is a user-defined constant. A time step consists of multiple phases, but we only discuss the most time-consuming phase – force computation. In a previous work, we optimized BH in UPC [6]. The octree is distributed among compute nodes. One optimization was to cache remote cells accessed during force computation. There was one UPC thread per core, with each doing its own (software) caching. Obviously, the UPC BH code took only limited advantage of the intranode hardware shared memory.

## 3. HYBRID DESIGN

To facilitate abstraction, we designed PPL, a C++ template library and re-implemented BH in it. PPL is a PGAS library, supporting a global address space and generic *global* pointers. The octree in PPL BH is distributed and linked by global pointers. We say a cell is *localized*, if its children have all been cached in local memory. Each cell has a `localized` flag to indicate whether it is localized or not; this flag is initially cleared. To *localize* a cell, we fetch its child cells, *swizzle* its global child pointers to local pointers pointing to the cached children, and then set the `localized` flag. The localization is a split-phase operation which includes making a non-blocking communication request and completing the request. With these concepts in mind, we

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357, and by the U.S. Department of Energy Sandia National Lab grant 1205852.

*SC*'13, November 17-22, 2013, Denver, Colorado, USA

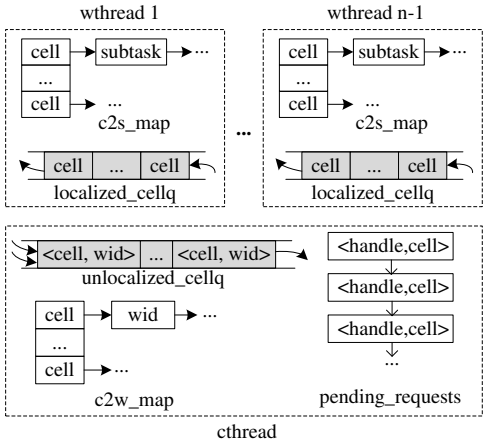


**Figure 3.1: Data structures used by threads. Shaded structures are accessed concurrently.**

now describe the intranode programming of PPL BH. Figure 3.1 illustrates the main data structures. The design has the following properties:

**Multithreading:** We spawn one process per node. Upon entering the force computation phase, each process is assigned an array of bodies, and spawns one thread per core. One thread is designated as the communication thread (cthread), which is in charge of internode communication, and the remaining are designated as worker threads (wthreads), which do force computation.

**Multitasking:** We define tasks and subtasks to gain fine-grain parallelisms, and then leverage them to overlap computation and communication. A *task* consists of the tree traversal and force computation for one body. During the traversal, if a cell needs to be opened but is not localized, we generate a *subtask* to handle the interactions between the body and the subtree rooted at that cell. All tasks and subtasks can be executed in parallel, and synchronization is only needed to sum forces acting on the same body. Wthreads grab bodies from the body array assigned to their owner process, construct tasks and execute them. If a wthread is blocked by an *unlocalized* cell during tree traversal, it will generate a subtask to encapsulate the context and continue the traversal along other paths if possible, otherwise it will construct new tasks and execute them. Although subtasks from the same task can be executed simultaneously by distinct wthreads, we do not do so, and execute a task and all its descendant subtasks on the same wthread. This is because the large number of bodies is sufficient to ensure that wthreads are always busy, and because it also avoids synchronization.

**Synchronization:** Each wthread has a private cell-to-subtask map (`c2s_map`) , which stores, for each cell, the list of subtasks blocked on an open request for that cell. The map acts as a hub for subtask registration and release. If a wthread wants to localize a cell, it looks up its `c2s_map` to see if the cell has already been requested *by itself*. If so, it just registers the subtask in its map; otherwise, it also pushes the cell along with a worker id (`wid`) into a concurrent queue (`unlocalized_cellq`).

The cthread pops cells from `unlocalized_cellq` and makes non-blocking communications to localize them. It has a private map (`c2w_map`) , which maps cells to wthreads who have requested them. By checking whether the map already has an entry for a cell, the cthread can filter duplicate requests for the same cell from different wthreads. The cthread periodically checks handles to pending requests. When a request is completed, it marks the cell as localized, looks up `c2w_map`, and pushes the cell back to queues of wthreads who have requested the cell. On the other side, wthreads periodically pop cells from their queue, look up their map and execute subtasks registered under the cells.

It turns out that all synchronizations can be done through either a single-producer multi-consumer queue or a multi-producer single-consumer queue, which is quite efficient.

**One-sided:** Thanks to a global address space, a cthread can fetch children of a cell by dereferencing its global child pointers, without corporation of remote sides.

**Load-balancing:** UPC BH used the *costzones* algorithm [4] for both internode / intranode load balancing. The flat model is imperfect, since it only considers computation cost. In reality, communication cost plays a role, though it is hard to predict. In PPL BH, the multithreading design provides us more flexible control. We adopted an approach like the OpenMP guided-scheduling for intranode load balancing, which is inherently adaptive to runtime fluctuation.

**Locality:** In BH, cells accessed by a body are likely to be accessed again by a nearby body. To harvest this locality, we sorted bodies in a space filling curve order. However, we found this had limited impact on performance. Instead, we could improve L1 cache reuse through tiling [2].

# 4. EVALUATION AND CONCLUSION

We tested PPL BH on a cluster. Each node has two hex-core CPUs. With $n = 1M$, $\theta = 0.5$ and 64 nodes, compared with UPC BH, PPL BH saves 59% of the memory, 57% of the off-node communication, since threads on a node share the cached cells (i.e., data sharing), and only fetch needed remote cells once (i.e., message reduction). Due to better load balancing, we only observed small ($< 2.5\%$) execution time variation on threads, far less than that in UPC BH. We found PPL BH could improve performance up to 41% over UPC BH and is more scalable. We also compared with two other state-of-the-art BH codes: PEPC [5] and Charm++ BH [3] and found PPL BH is up to 4x faster. In all, PPL BH shows a promising programming model worth further investigation in exascale computing research.

# 5. REFERENCES

[1] J. Barnes and P. Hut. A hierarchical O(nlogn) force-calculation algorithm. *nature*, 324:4, 1986.

[2] Y. Jo and M. Kulkarni. Enhancing locality for recursive traversals of recursive structures. In *ACM OOPSLA'11*.

[3] L. Kale, A. Arya, A. Bhatele, et al. Charm++ for productivity and performance: A submission to the 2011 HPC class II challenge. Technical Report 11-49, Parallel Programming Laboratory, November 2011.

[4] J.P. Singh. *Parallel hierarchical N-body methods and their implications for multiprocessors*. PhD thesis, Stanford University, 1993.

[5] M. Winkel, R. Speck, H. Hübner, et al. A massively parallel, multi-disciplinary barnes-hut tree code for extreme-scale n-body simulations. *Computer Physics Communications*, 183:880–889, 2012.

[6] J. Zhang, B. Behzad, and M. Snir. Optimizing the barnes-hut algorithm in UPC. In *ACM/IEEE SC'11*.