Locality-enhancing loop transformations for parallel tree traversal algorithms

Youngjoon Jo and Milind Kulkarni
{yjo,milind}@purdue.edu

School of Electrical and Computer Engineering
465 Northwestern Ave.
Purdue University
West Lafayette, IN 47907-2035

Abstract

Exploiting locality is critical to achieving good performance. For regular programs, which operate on dense arrays and matrices, techniques such as loop interchange and tiling have long been known to improve locality and deliver improved performance. However, there has been relatively little work investigating similar locality-improving transformations for irregular programs that operate on trees or graphs. Often, it is not even clear that such transformations are possible. In this paper, we discuss two transformations that can be applied to irregular programs that perform graph traversals. We show that these transformations can be seen as analogs of the popular regular transformations of loop interchange and tiling. We demonstrate the utility of these transformations on two tree traversal algorithms, the Barnes-Hut algorithm and raytracing, achieving speedups of up to 251% over the baseline implementation.

Keywords: locality transformations, irregular programs, n-body codes

1 Introduction

It has long been understood that achieving good performance in scientific applications requires attending to locality. Decades of research has led to a number of breakthroughs in automatic and semi-automatic transformation techniques for loop-based programs that operate over dense matrices and arrays (*regular* programs) [11]. This catalog of transformations, which includes entries such as *loop interchange*, *strip mining*, and *loop tiling*, provides programmers and compilers with an arsenal of tools for enhancing locality in regular programs.

While there has been significant research attention paid to automatic and semi-automatic locality-enhancing transformation techniques for loop-based programs that operate over dense arrays and matrices [11], far less attention has been paid to locality concerns in *irregular* programs, those that operate over pointer-based data structures such as trees and graphs. This relative dearth of attention is unsurprising: pointer-based data structures are highly dynamic, and are not amenable to the automated techniques for reasoning about locality that have been successfully applied to regular programs. In fact, the unstructured nature of irregular programs in many cases makes chasing after locality seem pointless.

The apparent lack of structure in irregular programs can be misleading. While the particular set of concrete memory accesses may exhibit little regularity, at an abstract level there are organizing principles governing these accesses, such as the topology of the irregular data structure, or the nature of operations on that data structure. Recent work by Pingali *et al.* has suggested that there may, indeed, be significant structure latent in irregular applications [17]. Can this structure be exploited to transform irregular applications so as to enhance locality?

In this paper, we focus on enhancing and exploiting locality in *tree-traversal* applications. Such applications are widespread; examples include scientific algorithms such as Barnes-Hut [3], graphics algorithms such as bounding volume hierarchies [21] and Lightcuts [23], and data mining algorithms such as k-nearest neighbor [8]. The goal of each of these algorithms is to compute a value (force, illumination, etc.) for each of a set of entities (bodies, rays, etc.). This computation is performed by constructing a tree-based acceleration structure and then traversing that structure for each entity to compute the desired value. In other words, these algorithms perform repeated series of tree traversals. Section 2 describes these applications in more detail.

The tree traversals performed by the aforementioned algorithms are highly irregular in nature. This is because the structure of the tree is determined primarily by the input data and because the actual layout of the tree in memory is unpredictable. Nevertheless, the trees constructed in these algorithms are traversed numerous times, leading to significant data reuse. Any time there is data reuse, there may be an opportunity to exploit temporal locality. This paper discusses a series of approaches for doing just that.

One of the motivating insights of this paper is that tree traversal algorithms exhibit similar locality properties to vector-vector outer product, a simple, regular algorithm. Moreover, loop transformations such as *loop interchange* and *tiling* have analogues that apply to tree traversals. By developing an abstract model of tree traversals based on outer products, we can reason about the locality effects of transformations on irregular tree traversals by appealing to their effects on the abstract model.

Our locality model also allows us to assess the relative impacts of various transformations proposed in the literature [20, 2, 15]. Our model implies, and experimental results bear out, that these transformations lose their effectiveness as data sets increase. However, by leveraging the correspondence between loop transformations on regular programs and transformations for tree traversal codes, we develop a new transformation, based on loop tiling, that more thoroughly exploits locality for large data sets.

We demonstrate the effectiveness of our transformation through two case studies of tree traversal algorithms, the Barnes-Hut n-body code and a raytracing benchmark based on bounding volume hierarchies. We show that our transformation can yield performance improvements of up to 251% over an optimized sequential baseline, and that this advantage persists when running in parallel. While the efficacy of these loop transformations is dependent on particular transformation parameters, we also present evidence that there may be an underlying model governing the appropriate selection of parameters.

The remainder of this paper is organized as follows. Section 2 discusses the structure of tree-traversal codes in more detail, with an emphasis on Barnes-Hut. Section 3 describes the analogy between outer products and tree-traversals, Section 4 describes how to interpret several popular loop transformations, loop interchange, loop strip-mining, and loop tiling, in the context of tree-traversal codes, and Section 5 discusses how a tree traversal code can be systematically modified to implement these transformations. Section 6 presents our case studies on Barnes-Hut and raytracing, demonstrating significant speedup over untransformed code. Related work is discussed in Section 7, and we conclude in Section 8.

2 Tree-traversal codes

As discussed in the introduction, tree-traversal codes include scientific applications such as Barnes-Hut (BH), graphics applications such as bounding volume hierarchies (BVH) and Lightcuts (LC), and data-mining appli-

```
1 Set<Entity> entities = /* entities in algorithm */
2 Set<Object> objects = /* environmental objects */
3 foreach (Entity e : entities) {
4   foreach (Object o : objects) {
5     updateContribution(e, o);
6   }
7 }
```

Figure 1: $O(mn)$ algorithm for tree-traversal codes

cations such as nearest neighbor (NN). Despite the different purposes of these applications, they all behave in roughly the same manner. Each application consists of a set of *entities*, be they bodies or particles (as in BH or NN), rays from a ray-tracer (in BVH), or points in a scene to be illuminated (in LC). For each entity, some *objective value* must be computed. For example, in BH, this value is the force on the body or particle while in BVH, this value is the color associated with a particular ray.

The objective values are based on the locations of *environmental* objects. In BH, these environmental objects are the other bodies in the system; in BVH, they are the objects in the scene. A naïve approach to computing each entity's objective value is shown in Figure 1: visit each environmental object and compute its contribution to the objective value of the current entity. In BH, we compute the force applied to the current body by the environmental body; in BVH, we determine if the ray intersects the environmental object; in LC, we calculate the contribution to the illumination of the scene point by the environmental light source; in NN, we find the nearest environmental object to a given entity. With n entities and m environmental objects, this is an $O(nm)$ algorithm.

To improve on this running time, tree-traversal codes organize their environmental objects into tree structures based on spatial locality[1]. For example, in Barnes-Hut, an oct-tree is built over the particles. All of the particles are placed into the root cell, and the cell is recursively subdivided into eight equal parts until each cell contains only one particle. Each cell of the oct-tree is processed to determine the center-of-mass of the particles contained in that cell. Similar acceleration structures can be built for BVH, LC and NN. Note that building these acceleration structures is a small fraction of the overall computation time of a tree-traversal algorithm [22].

To use the oct-tree in Barnes-Hut, the objective value computation is modified. Rather than visiting every environmental object and calculating its contribution to the objective value, the oct-tree is traversed. Cells whose center-of-mass is sufficiently far from the current particle are treated as if all the points in the cell are at the center-of-mass, obviating the need to traverse deeper down that subtree. If the center-of-mass is close to the current particle, then the subtrees of the cell are traversed. In this

[1]Note that this is spatial locality in the geometric sense (*i.e.*, particles that are near each other in space), not in the memory locality sense

```
1  Set<Particle> particles = /* entities in algorithm */
2  Set<Particle> objects = particles; //environmental objects
3  OctTreeCell root = buildTreeAndComputeCofM(objects);
4  foreach (Particle p : entities) {
5    Recurse(p, root);
6  }
7
8  void Recurse(Particle p, OctTreeCell c) {
9    if (farEnough(p, c.cofm) || c.isLeaf) {
10     updateContribution(p, c.cofm);
11   } else {
12     foreach (OctTreeCell child : c.children) {
13       if (child != null)
14         Recurse(p, child);
15     }
16   }
17 }
```

Figure 2: $O(n \log m)$ algorithm for Barnes-Hut

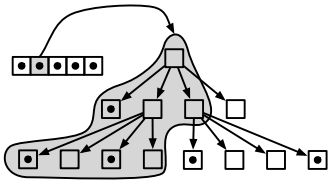


Figure 3: Portions of quadtree traversed

way, to compute the contributions of the environmental objects to the objective value, only a *partial* tree traversal need be performed, turning an $O(nm)$ algorithm into an $O(n \log m)$ algorithm [4]. Figure 2 shows the pseudocode for this accelerated algorithm. Other tree traversal codes use similar algorithmic skeletons. Figure 3 provides a graphical representation of the algorithm: to compute the objective value for the point selected from the vector of entities, only the shaded portion of the tree need be traversed.

Improving locality in Barnes-Hut Because the oct-tree is a highly dynamic data structure, exploiting locality in the traversals is difficult. (consecutively visited children may not share a cache-line, preventing the exploitation of spatial locality). However, as the same tree is traversed by each particle (the outer loop in Figure 2), it is often possible to exploit locality in the traversals. Particles that are nearby in space are likely to perform very similar traversals of the oct-tree, visiting the same set of tree cells. Thus, if these traversals are performed consecutively, the cells visited during the first traversal are likely to remain in cache during the second traversal, exploiting temporal locality.

Such a locality-exploiting order of traversals can be arranged by sorting the particles according to their geometric position, so that adjacent particles in the sorted order are nearby geometrically [2, 20]. This is a common optimization, and is implemented in the version of Barnes-Hut in the LoneStar benchmark suite [12], that we use as a baseline in this work. As we shall see in the next section,

```
1 Set<Particle> particles = /* entities in algorithm */
2 Set<Particle> objects = particles;
3 OctTreeCell root = buildTreeAndComputeCofM(objects);
4 foreach (Particle p : particles) {
5   foreach (OctTreeCell c : traverse(root, p)) {
6     if (farEnough(p, c.cofm) || c.isLeaf) {
7       updateContribution(p, c.cofm);
8     }
9   }
10 }
```

Figure 4: Abstract algorithm for tree-traversal

```
1 Particle p[n] = /* particles */
2 OctTreeCell c[m] = /* traversal */
3 for (int i = 0; i < n; i++)
4   for (int j = 0; j < m; j++)
5     Update(i, j); //A[i][j] = p[i]*c[j]
6
7 void Update(Particle p, OctTreeCell c) {
8   if (farEnough(p, c.cofm || c.isLeaf))
9     updateContribution(p, c.cofm);
10 }
```

Figure 5: Tree traversal as outer product

however, this optimization loses effectiveness as the input size (and hence the tree size) increases.

3 An abstract model

Reasoning about locality in tree-traversal codes is difficult for a number of reasons. First, unlike in regular applications, the structure of the key data structures is highly input-dependent. The oct-tree generated in BH is dependent on the particular locations of the particles in the system. Furthermore, the data structures are dynamically allocated, and hence can be scattered throughout memory. Finally, the traversals of the tree are not uniform; a traversal can be truncated (*e.g.*, due to the distance check in line 9 of Figure 2), and traversals for two different points are not necessarily similar.

However, we can still reason about locality by considering the behavior of a tree-traversal algorithm in a more abstract sense. Rather than viewing a traversal as a recursive, pre-order walk of a tree, we can instead visualize the traversal in terms of the actual tree cells touched. Fundamentally, processing a single particle in BH requires accessing some sequence of tree cells. The particular arrangement within the tree of those cells is irrelevant; all that matters is the ultimate sequence in which those cells are touched. If we imagine that there is an oracle function `traverse` that generates the sequence of cells accessed while processing a particular particle, we can rewrite the accelerated BH code as shown in Figure 4. In other words, we can view the algorithm as a simple, doubly-nested loop. Notably, *for the purposes of locality, the behavior of the original BH code is equivalent to the abstract algorithm*. All that matters to locality is the sequence of accesses; the additional computations required to determine whether to continue a traversal or not do not affect locality. Thus, the sequences of memory accesses for the code in Figure 2 and Figure 4 are identical. Moreover, locality-enhancing transformations on the abstract code will also enhance locality in the original code, if an equivalent transformation can be applied (Section 4 discusses this in more detail).

3.1 Tree traversals as outer products

This abstract algorithm provides insight into why sorting the particles (as discussed in Section 2) is useful for locality. Consider the behavior of two consecutive particles, p_1 and p_2. In the unsorted algorithm, there is little overlap between $traverse(p_1)$ and $traverse(p_2)$. Most of the inner-loop accesses for the p_2 iteration will result in cache misses. However, sorting the particles such that consecutive points have similar traversals will result in cache hits.

When the points are sorted, the variability between consecutive traversals will be a fairly small second-order effect, so we can simply consider consecutive traversals in the sorted case to be the same. This approximation lets us further simplify the abstract algorithm. The outer loop iterates over a vector (of particles) and, for each particle, the inner loop iterates over a vector (containing the cells of the traversal). If there are n particles, and the average traversal is m cells, then this is an $O(mn)$ algorithm with an access pattern equivalent to an $m \times n$ outer product. Figure 5 demonstrates this correspondence, showing how a tree traversal is analogous to the outer product of a vector p and a vector c.

One insight we can glean from this model is that the efficacy of sorting points to improve locality diminishes as the tree, and hence traversals, get larger. Consider the behavior of outer product when the c vector is small enough to fit in cache. For an element from the p vector, every element from the c vector is touched. When the next element is drawn from the p vector, all of c will be in cache, and there will be no further cache misses. Unfortunately, once c is too large to fit in cache, an LRU replacement policy will lead to disastrous results: when the second iteration of the outer loop begins, although some of c is in cache, the first element(s) will not be. Bringing those elements into cache will kick the next elements out of cache, and no accesses to c will ever hit.

Analogously, we would expect that for small input sizes, when the average tree traversal fits in cache, we would see a relatively small number of cache misses. As the average tree traversal grows, we would expect far more misses. To test this hypothesis, we investigated the cache behavior of the Barnes-Hut benchmark from the Lonestar benchmark suite [12] on three different in-

# particles	Traversal size (bytes)	L2 miss rate (%)	% improvement over unsorted
10000	63,944	21.61	67.3
100000	108,656	44.97	45.9
1000000	139,616	55.30	26.4

Table 1: L2 miss rates for various traversal sizes

put sizes when run on a Pentium machine with 1M of L2 cache. Table 1 shows, for each input, the input size (in number of particles), the average traversal size (in bytes of the oct-tree touched per particle), the L2 miss rate and the percentage improvement of the sorted version over the unsorted version. As predicted, the miss rate increases as the traversal size increases (note that traversal sizes increase as the log of the number of points). As the final column of the table shows, we see that this increased miss rate hinders the efficacy of sorting. For small inputs, which result in small traversals, sorting the points can lead to large (67%) improvements in performance. Unfortunately, as the traversal sizes get larger, sorting the points is insufficient to exploit locality, and we see correspondingly less improvement from the sorting optimization.

If sorting the points is of limited utility for larger inputs, what can be done to improve locality in such situations? As we shall see, loop transformations developed for regular algorithms hold the key.

4 Loop transformations for tree-traversal codes

By viewing the locality behavior of tree-traversal codes as an analog of vector-vector outer product, we can predict the effects that various computation re-orderings might have on the tree-traversal algorithm by considering the effects that an analogous transformation would have on the corresponding outer product algorithm. Since outer product is a regular algorithm, there are many well-understood loop transformations that can be applied to improve its locality behavior. These transformations will inform the development of similar transformations to enhance locality in tree-traversal codes.

In this section, we consider three well-known loop transformations: *loop interchange* (also called loop permutation), *loop stripmining* and *loop tiling* (also called loop blocking) [11]. For each of these transformations, we will discuss what the transformation means in the context of tree traversals, and elucidate their locality effects through analogy with the outer-product model.

4.1 Loop interchange

Loop interchange is one of the simplest loop transformations that can be applied to doubly-nested loops such as

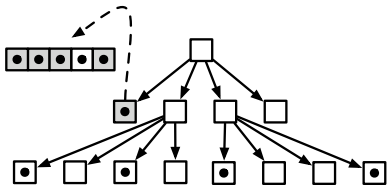


Figure 6: Portions of particle list accessed while processing shaded tree node

outer product. It merely consists of swapping the order of the loops:

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < m; j++)
    A[i][j] = p[i]*c[j]
```

becomes

```
for (int j = 0; j < m; j++)
  for (int i = 0; i < n; i++)
    A[i][j] = p[i]*c[j]
```

In the original version of outer product, an element of p is "pushed through" the entire vector of c before moving to the next element of p. In the new version, every element of p interacts with the first element of c before moving to the next element of c.

What does loop interchange mean in the context of tree traversals? Recall the abstract algorithm of Figure 4, where the traversal is provided by an oracle. This is a simple doubly-nested loop that can be interchanged just as in the outer product analog. This places the traversal loop on the outside. However, as there is no longer a specific point associated with the traversal, iterating over a particular traversal is nonsensical. Instead, we consider the outer loop as iterating over each oct-tree cell, while the oracle tells us which particles should interact with this cell:

```
foreach (OctTreeCell c : octtree)
  foreach (Particle p : interact(c, particles))
    //do work
```

In other words, we take all the points and push them through the root of the oct-tree. We then move to a child node and process all points whose traversals would take them down to this child. We continue for every cell in the tree until every point has interacted with every oct-tree cell that it would have in the original traversal[2].

Rather than performing a partial tree traversal for each particle, as the original Barnes-Hut algorithm does, this transformed version instead surveys a subset of the particles for a particular tree node. (as shown in Figure 6) The transformed code is quite complex, and assessing its locality behavior is difficult. We instead analyze the effects

[2]This is broadly similar to a transformation proposed by Makino, though he did not consider its locality effects [15].

of interchange on the outer product model, and thereby infer its effects on the traversal code.

In the original outer product code, we note that regardless of how large p is, we incur only cold misses on its elements, while we incur only cold misses on c until it exceeds cache, at which point we continually suffer capacity misses. Interchanging the loops reverses this effect: we will only see cold misses for c, and the misses in p depend on how large the p is. It is thus useful to place the smaller vector in the inner loop, as it is more likely to fit in cache.

Note that loop interchange is usually performed to expose *spatial* locality in one of the loops. However, because tree traversals are irregular, there is no guarantee that traversing the nodes in order, or the tree cells in order, will actually exploit spatial locality. The adjacency of nodes in the tree, and the order of particles to be processed, has no relation to their actual location in memory.

The outer product analysis suggests the following behavior for Barnes-Hut: if the outer loop is the point loop, then we will suffer no misses (other than cold misses) on the points, while we will suffer misses on every tree cell access. Conversely, if the outer loop is the tree loop, we will suffer only cold misses in the tree, but miss on every point access. Because the average traversal size is $O(\log n)$, while there are n points, this analysis argues for keeping the particle loop on the outside, a hypothesis confirmed by the experimental results of Section 6.

4.2 Loop stripmining

Loop stripmining is perhaps the simplest loop transformation, as it merely tiles a single loop:

```
for (int i = 0; i < n; i++)
```

becomes

```
for (int ii = 0; ii < n; ii += B)
  for (int i = ii; i < ii + B; i++)
```

In other words, we split the loop into blocks of size B, and iterate over the blocks first, and then within the blocks. Note that stripmining alone does not change the traversal pattern of an algorithm—it has no effect on locality.

In Barnes-Hut, stripmining the particle loop is intuitive: the worklist of particles is instead a worklist of chunks of particles. After removing a chunk from the worklist, the points within the chunk can be operated on. The interpretation of stripmining the traversal loop is somewhat more abstract. Effectively, the tree is split into "tiles," and each tile is considered a block of the stripmined loop. Again, however, stripmining does not change the traversal pattern. It is only when combined with loop interchange that stripmining becomes interesting, producing the transformation known as loop tiling.

4.3 Loop tiling

One of the most complex, and most effective, loop transformations is *tiling*: This can be viewed as stripmining followed by interchange:

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < m; j++)
    A[i][j] = p[i]*c[j];
```

becomes

```
for (int ii = 0; ii < n; ii += B)
  for (int j = 0; j < m; j++)
    for (int i = 0; i < ii + B; i++)
      A[i][j] = p[i]*c[j];
```

To see what loop tiling means for locality, consider applying the transformation to the outer product algorithm when both vectors are too large to fit in cache. The outer loop chooses a block p_B (of size B) from p. It then iterates over c, processing every element from p_B for each element of c. If B is chosen appropriately, then the elements of p_B are never evicted from cache while the block is processed, and elements of c only incur misses the first time they are touched per block. This results in radically fewer cache misses than the non-tiled version of the code.

What does tiling Barnes-Hut entail? While tiling the traversal loop is complex, tiling the particle loop is straightforward:

```
foreach (ParticleBlock b : particles)
  foreach (OctTreeCell c : traversal(b))
    foreach (Particle p : b)
      //do work
```

Note that because the particle vector is sorted, the traversals of all of the particles in a given block b are likely to overlap significantly, and hence the middle loop will perform $O(\log n)$ iterations. To gauge the behavior of this loop, we note that the inner two loops are equivalent to the interchanged algorithm presented earlier, except that the number of particles in b is far fewer than the total number of particles. Hence b is likely to remain in cache as we visit each tree cell in the traversal. Thus, we incur only cold misses on the particles, and we suffer a miss on a tree cell once per block.

Realizing this transformation in tree-traversal codes is non-trivial. Section 5 discusses the specific changes that are necessary to efficiently implement tiling in tree-traversal codes.

Because blocking the particles loop can result in good locality regardless of the size of the particle list or the oct-tree, we apply this transformation to Barnes-Hut and ray-tracing. Section 5 discusses the specific changes that are necessary to efficiently implement tiling for tree-traversal codes.

Note, the right choice of tile size is crucial. Too large, and the particle block will not remain in cache, defeat-

ing the purpose of tiling. Too small, and the program will suffer more misses (since a tree cell suffers one miss per particle block). In the world of dense linear algebra, there has been significant research into choosing the right tile size, from empirical search [24] to analytical models [25]; ultimately, the appropriate tile size for dense linear algebra kernels is dependent on cache parameters. It is unclear whether similar approaches are feasible for irregular applications such as tree traversal codes. In Section 6, we discuss the sensitivity of the blocked version's performance to tile size, and present evidence that the optimal tile size is correlated with underlying architectural features.

5 Implementation

To investigate the utility of our loop transformations for tree traversal codes, we performed a case study on Barnes-Hut (BH) and bounding volume hierarchy-based raytracing (BVH).

5.1 Barnes-Hut

Here we discuss several implementations of Barnes-Hut: the baseline implementation; a sequential blocked implementation, which exploits the tiling transformation discussed in the previous section; and a parallel version of the blocked implementation.

5.1.1 Baseline

We used the Barnes-Hut implementation from the Lonestar benchmark suite [12] as our baseline. The baseline performs its force computation using the algorithm of Figure 2; each particle is pushed through its entire traversal of the tree before moving on to the next particle. As discussed earlier, the particles are processed according to their locations in space, enhancing locality through the techniques described in [2, 20].

The baseline is parallelized by processing multiple particles at the same time. No synchronization is necessary on the tree or the particles: the tree is only read during the force computation phase, and the state of each particle is only updated by the thread it is assigned to. To maintain the locality provided by sorting the particles, particles are distributed equally to threads at the beginning of execution. Work stealing is performed for load balance, implemented via lock-free double-ended queues as in Cilk [6]. Cilk style work stealing is used for both the baseline and blocked version for both Barnes-Hut and raytracing.

```
1  Set<Particle> particles = /* entities in algorithm */
2  Set<Particle> objects = particles;
3  Set<Block> blocks = makeBlocks(particles);
4  OctTreeCell root = buildTreeAndComputeCofM(objects);
5  foreach (Block b : blocks) {
6    stack[0] = b;
7    recurseForce(root, 0);
8  }
9
10 void recurseForce(OctTreeCell c, int level) {
11   Block b = stack[level].block;
12   Block nextB = stack[level + 1].block;
13   nextB.recycle();
14   foreach (Particle p : b.particles) {
15     if (farEnough(p, c.cofm) || c.isLeaf) {
16       updateContribution(p, c.cofm);
17     } else {
18       nextB.add(p);
19     }
20   }
21   if (!nextB.isEmpty) {
22     foreach (OctTreeCell child : c.children) {
23       if (child != null) {
24         RecurseForce(child, level + 1);
25       }
26     }
27   }
28 }
```

Figure 7: Pseudocode for blocked implementation

5.1.2 Blocked

The sequential blocked algorithm is a transformed version of Barnes-Hut that uses the loop tiling optimization discussed in Section 4. Pseudocode for the algorithm is given in Figure 7. It begins with a list of sorted particles, and splits them into blocks, each containing b particles, where b is the block size. Each block is then considered in turn (line 5), and is recursively "pushed" through every tree cell (the call to `recurseForce` in line 7).

For each tree cell, we process each particle in the block. If the cell is a leaf (it contains a single particle) or far enough away (the many particles it represents can be approximated as a single particle), we compute its contributions to the particle (line 16). If the particle does not interact with the cell, it needs to traverse the children of the cell, and the particle is added to the next block (line 18), which contains all particles that must interact with children of the current cell. Finally, we recursively call `recurseForce` on the children of the cell (line 24).

Note that the high level algorithmic structure of this implementation, despite its recursive nature, is "for each block of particles, for each tree cell, for each (valid) particle in the block, do work." This matches the tiled algorithm described in Section 4.

For efficiency reasons, the blocks of valid particles are held in preallocated arrays drawn from a global stack. This avoids the need to reallocate the blocks at each recursive steps. We only need to maintain one set of particle arrays per level, since the recursion is done in depth first order. The `nextB` array is the preallocated block for the next level (*e.g.* level + 1) and is cleared at the start

of `recurseForce` (line 13) so that it contains only particles added for the cell being processed. Managing the particle blocks adds instruction overhead to the blocked implementation compared to the baseline. As we will see in Section 6, this additional overhead is more than compensated for by improved locality.

One of the nice features of the blocked implementation is that it lends itself to parallelization. We simply parallelize the outer loop, assigning particle blocks to threads and running `recurseForce` on blocks in parallel. In other words, we can treat the loop in line 5 as a data parallel loop. As in the baseline, no synchronization is necessary.

5.2 Raytracing

As our baseline raytracing implementation, we used the BVH-based raytracer from [22]. The baseline uses a binary tree to represent the bounding volume hierarchy representing the scene to be rendered. The baseline renders the image by traversing the BVH for each ray to determine which object it intersects, before moving on to the next ray. The rays are naturally sorted by being processed in an order corresponding to the pixels of the screen. The baseline is parallelized by processing multiple rays at the same time. As in Barnes-Hut, the tree is only read during the rendering phase, and no synchronization is necessary.

Raytracing is different from Barnes-Hut in that each original entity generates additional entities, each incident ray generates a shadow ray for each light source. We found that shadow rays from different incident rays but from the same light source had more similarity in their traversals compared to shadow rays from the same incident ray but different light sources. Hence we processed a block of incident rays first, and gathered shadow rays for each light source into separate blocks of shadow rays, and then processed the shadow ray blocks. The algorithm for processing a single block (be they incident rays or shadow rays) is similar to blocked Barnes-Hut. Rays which need to proceed to the next level are passed via the `nextB` array as described in Section 5.1.2. Parallelization is accomplished by processing blocks at the same time.

6 Evaluation

We evaluated our blocked algorithm, as well as the loop-interchanged version (outer traversal loop, inner particle loop) and the baseline algorithm on three systems with different cache configurations.

- The **Niagara** system runs SunOS 5.10 and contains two 8-core UltraSPARC T2 chips in SMP configuration. Each chip has 8K L1 data cache per core and 4M shared L2 cache. We present results up to 32 threads, at which point our system is employing 2-way multithreading.
- The **Opteron** system runs Linux 2.6.24 and contains four dual-core AMD Opteron 2222 chips in SMP configuration. Each chip has 128K L1 data cache per core and 1M L2 cache per core. We present results up to 8 threads.
- The **Pentium** system runs Windows Vista SP2 and contains a dual-core Intel Pentium T4200 chip. The chip has 32K L1 data cache per core and 1M shared L2 cache. We present results up to 2 threads.

The baseline and our new blocked algorithm were implemented in the Galois system [13]. The applications were written in Java 6 and executed on the Java HotSpot VM version 1.6 for all systems. The **Niagara** and **Opteron** systems used a 12GB heap, and the **Pentium** system used a 1.5GB heap. To account for the effects of JIT compilation, each configuration was run 10 times, and the average of the latter 7 runs was recorded.

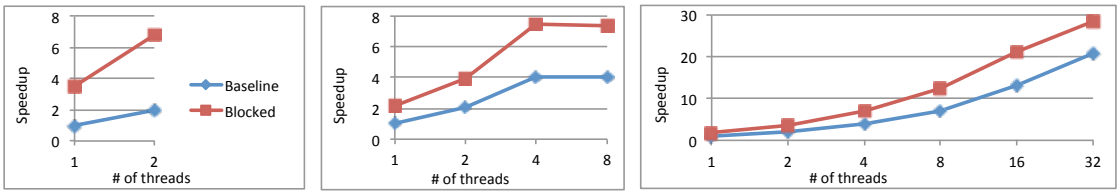
6.1 Barnes-Hut

For Barnes-Hut, we used the one million particle input from the Lonestar benchmarks [12]. We first present results that assume we know the optimal block size for a particular architecture (recall that block size affects tiling performance) and show that our blocking transformation both attains significant performance gains, and scales well with many parallel threads. Then we will discuss the sensitivity of our transformation's performance to different block sizes, and investigate the relationship between the optimal block size and cache configurations.

6.1.1 Speedups of optimal block

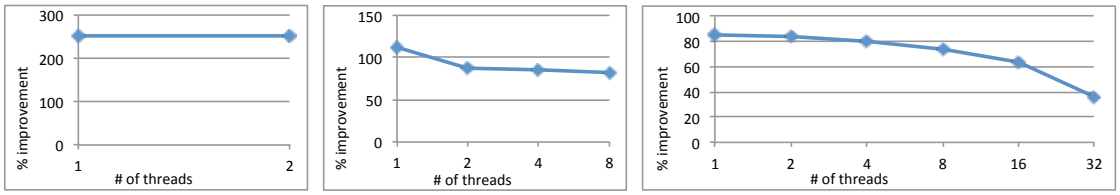
Figure 8 shows speedups of the blocked implementation with an empirically determined optimal block size compared to the serial baseline. Figure 9 shows % improvement of the blocked implementation compared to the parallel baseline. We attain improvements of up to 251%, 113% and 85% over the parallel baseline respectively for the **Pentium**, **Opteron** and **Niagara** systems. These speedups are sustained as we increase the number of threads for the **Pentium** and **Opteron** systems.

For the **Niagara** system, we note that our optimized implementation's advantage over the baseline tapers off as the number of threads increases. We believe this is because the L2 cache on the Niagara is shared among cores, and hence the effective L2 cache size decreases as the number of threads increases (note that our implementation is more sensitive to caching effects than the baseline, which demonstrates poor locality regardless). There



(a) **Pentium** (b) **Opteron** (c) **Niagara**

Figure 8: Speedup vs serial baseline for Barnes-Hut



(a) **Pentium** (b) **Opteron** (c) **Niagara**

Figure 9: % Improvement vs parallel baseline for Barnes-Hut

is no similar degradation of performance as the number of threads increase on the **Pentium** and **Opteron** because each core has its own individual L2 cache.

More interesting than the slight decline in advantage from 2–16 threads is the behavior of our blocked implementation on 32 threads, where its advantage over the baseline is notably smaller. This is because on 32 threads, the Niagara uses 2-way multithreading. The Niagara's implementation of multithreading is meant to hide latency: when one thread stalls due to a cache miss, the second thread can execute. As the Niagara is already hiding latency through multithreading, it obviates the need for our transformations, which hide latency through restructuring.

6.1.2 Sensitivity of performance to block size

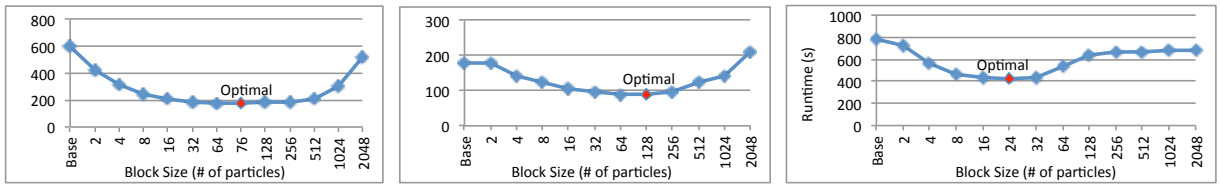
To investigate the sensitivity of our algorithm's performance to block size, we evaluated the algorithm with many different block sizes. Figure 10 shows the serial runtimes in seconds with varying block sizes for the three systems.

In general, we would expect that a block size that is too small would perform poorly due both to the additional instruction overhead and to the fact that misses in the tree are incurred for every block (as discussed in Section 4)—hence, fewer blocks will result in fewer misses in the traversal. However, if the block becomes too large to fit in cache, then we will begin to incur misses on the particles instead. We thus expect there to be a "sweet spot," where the blocks are large enough to avoid most misses in the tree, but small enough to fit in cache, an expectation borne out by the results. In each figure, the best block size is highlighted, and is surrounded by block sizes that perform worse.

The leftmost point on the x axis is the baseline which corresponds to a block size of 1. The baseline is generally faster than block sizes of 2 or 4 because it executes fewer instructions than the more complex blocked algorithm. The opposite extreme would be a block size of 1000000 (all the particles in the input), which corresponds to the loop interchange described in Section 4.1. We conjectured that this would result in much worse performance as the average traversal size is $O(\log\ n)$, while there are n points, and having the larger set of n points as the inner loop leaves less room to exploit locality. With 1000000 particles cycling through cache and up to 6 doubles accessed per particle, it is apparent that they would not fit in even the relatively large 4M L2 cache of the Niagara system. Indeed, performing the interchange results in 58%, 460% and 204% *increases* in sequential runtime on the Pentium, Opteron and Niagara systems respectively.

The optimal block sizes were found to be 24, 76 and 128 for the Niagara, Pentium and Opteron systems respectively (these are the highlighted points in Figure 10). We note that the optimal block size is correlated with the L1 data cache sizes of the systems. The relationship is not linear due to the irregularity of the application, and other locality effects (such as those from the L2).

To examine these tradeoffs in more detail, we used Intel VTune to access the performance counters on the Pentium system, and recorded the L1 and L2 miss rates for different block sizes, as plotted in Figure 11. Recall that the locality we are trying to exploit is temporal locality between particles within a block, which we can get as long as all the particles in a block stay in cache. Increasing the block size decreases L1 miss rates up to a block size of 64. Blocks larger than 64 particles exceed L1, and miss rates



(a) **Pentium** (b) **Opteron** (c) **Niagara**

Figure 10: Runtime with varying block sizes for Barnes-Hut

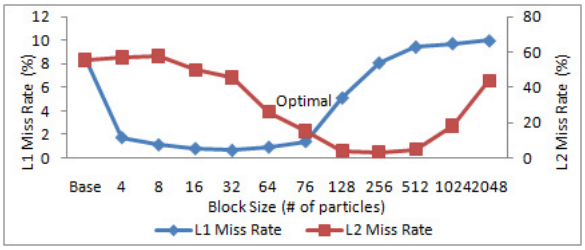


Figure 11: Miss rate vs. block size on **Pentium**

increase significantly. However the increase in L1 misses is covered by L2 which is 32 times larger than L1 on the system. L2 is able to handle a block size of up to 512 before the block footprint starts to outstep L2. The optimal block size comes at a point where the combined effects of L1 and L2 provide the best balance, and this block size is 76.

6.2 Raytracing (BVH)

For BVH we used a randomly generated scene with 1 million triangles. We shot 4 million rays for the Niagara and Opteron systems, and 1 million rays for the Pentium system.

6.2.1 Speedups of optimal block

Figure 12 shows speedups of the blocked implementation of BVH using an empirically determined optimal block size compared to the serial baseline. Figure13 shows % improvement of the blocked implementation compared to the parallel baseline.

The blocked algorithm attains improvements of up to 87%, 44% and 16% over the parallel baseline respectively for the **Pentium**, **Opteron** and **Niagara** systems. As in Barnes-Hut these speedups are sustained as we increase the number of threads for the **Pentium** and **Opteron** systems. For the **Niagara** system, speedups decline for many threads because the L2 is shared among cores, and multi-threading hides cache latency.

To understand why the speedups for BVH are less than those for BH, we compared the average traversal size for a single entity (*e.g.* particle in BH and ray in BVH), shown

Benchmark	# objects	Traversal size (bytes)	L2 miss rate (%)
Barnes-Hut	1000000	139,616	55.30
Raytracing	1000000	35,834	8.49

Table 2: Average traversal sizes for BH and RT

in Table 2. Even with the same number of objects, BVH has a much smaller traversal and L2 miss rate than BH. This is because particles in BH traverse many paths of the oct-tree to multiple leaves to calculate the force contribution by other nearby particles, whereas rays in raytracing typically follow a single path toward the leaf containing the object in the direction the ray is shot.

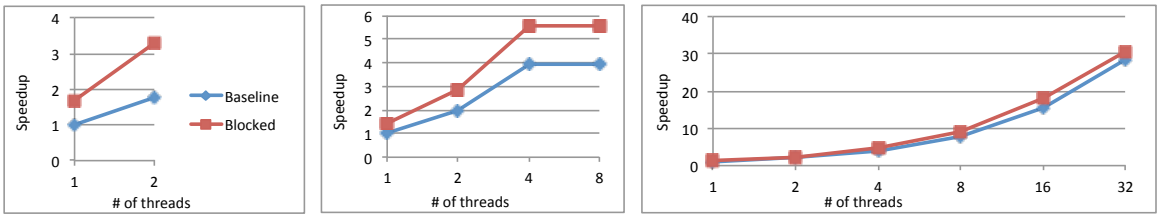
6.2.2 Sensitivity of performance to block size

Figure 14 shows the serial runtimes in seconds with varying block sizes for raytracing. As in Barnes-Hut, we can see that there is a "sweet spot," where the blocks are large enough to avoid most misses in the tree, but small enough to fit in cache. In each figure, the best block size is highlighted, and is surrounded by block sizes that perform worse.

7 Related Work

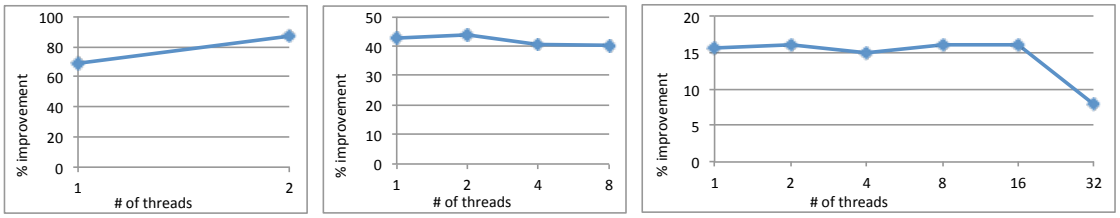
7.1 Locality Transformations

Salmon used Orthogonal Recursive Bisection [5] to directly partition the particle space to provide physical locality [19]. Singh *et al.* recognized that N-body problems already have a representation of the spatial distribution encoded in the tree data structure and partitioned the tree instead of partitioning the particle space directly [20]. Amor *et al.* exploited locality among particles by linearizing them using space filling curves [2]. Both these approaches improve locality up to a point, as discussed in Section 2, and both our baseline and transformed code exploit this particle ordering. These approaches have the limitation that as the traversal sizes get larger, simply sorting the particles is insufficient to exploit locality, whereas our work exploits localitly even in this case by tiling the traversals with a group of particles which will fit in cache.



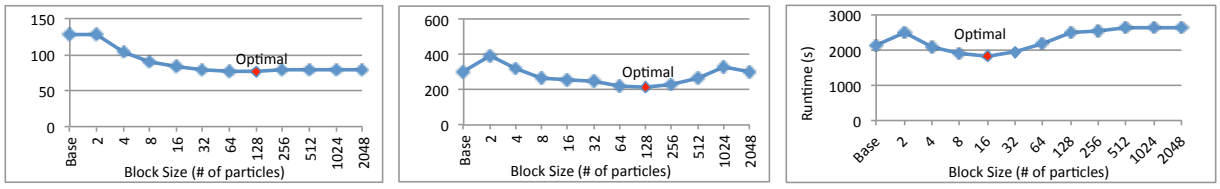
(a) **Pentium** (b) **Opteron** (c) **Niagara**

Figure 12: Speedup vs serial baseline for Raytracing



(a) **Pentium** (b) **Opteron** (c) **Niagara**

Figure 13: % Improvement vs parallel baseline for Raytracing



(a) **Pentium** (b) **Opteron** (c) **Niagara**

Figure 14: Runtime with varying block sizes for Raytracing

Singh *et al.* also proposed *costzones* to improve load balance across multiple Barnes-Hut timesteps; we expect their effects are largely orthogonal to the transformations presented here. Amor *et al.* proposed communication optimizations for distributed memory systems. While we evaluate our techniques on shared memory systems, we expect similar improvements if applied to an optimized distributed memory implementation.

Mellor-Crummey *et al.* proposed a combination of data reordering and computation reordering to improve memory hierarchy performance for n^2 interaction algorithms [16]. They use space filling curves to linearize the set of objects, and reallocate them into contiguous memory to exploit spatial locality. This is infeasible for hierarchical $O(n \log m)$ algorithms, as objects in the hierarchy encompass other objects, and it is not possible to map the hierarchy to a single dimensional space. There is no notion of the root being closer to one object than another; it encompasses all the objects. They also propose loop blocking, but this is after the entire interaction list has been computed, and is equivalent to loop tiling in regular applications.

7.2 Vectorization Transformations

Hernquist vectorized Barnes-Hut across nodes of the tree, so that each particle traverses all nodes at the same level simultaneously [10]. This approach effectively changes the order of the tree traversal from depth-first to breadth-first. This has two drawbacks. First, it changes the traversal order of the tree, affecting the result in the presence of non-commutative operations (such as floating-point addition). Second, there typically are not many nodes per tree level, leading to short vectors (and less parallelism).

Makino vectorized the tree traversal across particles, instead, leading to a per-particle parallelization similar to our baseline [15]. An interesting aspect of Makino's approach is that to enable vectorization, the code is transformed in a manner similar to the loop interchanged implementation described in Section 4.1. However, there are a few key points to note. First, as we demonstrated in Section 6, a simple loop interchange does not suffice to exploit locality. Second, Makino's transformation relies on a pre-computed traversal of the tree, and changes the order in which particular tree nodes are visited by different points, reducing the generality of his transformation.

Work on vectorizing Barnes-Hut have naturally extended to GPU implementations of n-body algorithms [14, 9]. These implementations generally group many particles in the leaves of the tree, so that the particles within a single leaf are ensured to have identical interaction lists and can be divided among processing units. The interaction lists are computed on the CPU and sent to the GPU for mass parallel force computation. The GPU's natural execution model results in traversals of the interaction list similar to tiling. However, computing the interaction lists still requires traversing the tree, and the locality penalties of a naïve traversal remain.

7.3 Other Tree Traversal Transformations

Aluru *et al.* discussed changing the tree structure of Barnes-Hut to improve performance [1]. We note that our transformations are independent of the type of tree used (indeed, the tree in raytracing is different from that in Barnes-Hut), and hence our approach can apply to their algorithm as well.

Rinard and Diniz used a commutativity analysis to parallelize an N-body code in a unique manner [18]. Rather than distributing the particles among threads, they are able to prove through compiler analysis that updates to the particles commute, and hence multiple threads can update particles simultaneously. This is akin to parallelizing the traversal loop in our abstract model, rather than the particle loop.

Ghiya *et al.* proposed an algorithm to detect parallelism in C programs with recursive data structures [7]. These tests rely on shape analysis to provide information on whether the data structure is a tree, DAG or general graph, and apply different dependence tests depending on data structure shape. Their analyses focus on parallelization and do not consider locality, but we believe their approaches might inform an automatic transformation framework that implements our techniques.

8 Conclusions and future work

In this work, we demonstrated that, despite their seeming irregularity, many tree-traversal codes possess a common algorithmic structure. Furthermore, this algorithmic structure has an interesting analog to vector-vector outer product, a simple regular algorithm. By exploiting this analogy, we were able to show that classical loop transformations such as loop tiling have a corresponding instantiation for tree traversal algorithms, and crucially, both transformations have similar effects on locality.

To demonstrate the effectiveness of our tiling transformation, we applied tiling to two tree-traversal algorithms, Barnes-Hut and raytracing. We showed that our tiled implementation of tree traversal exhibits far better locality than the baseline implementation and that our locality gains persist even as we scale up the parallelism.

Future work There is ample opportunity for further investigation in this area. As suggested by our results, there appears to be an underlying model, based on cache parameters, that determines the optimal tile size for tiled tree traversals. There may be an analytical approach to deriving this model, or it may be necessary to apply machine learning techniques to infer the model.

At present, the transformations to tree traversal codes are performed by hand. However, the tiling transformation is a fairly systematic restructuring of the application, and automation may be possible. It is an open question whether a compiler could perform the transformation by inferring that a tree traversal is being performed, or whether a small amount of programmer intervention will be required to guide the transformation process.

References

[1] Srinivas Aluru, John Gustafson, G. M. Prabhu, and Fatih E. Sevilgen. Distribution-independent hierarchical algorithms for the n-body problem. *J. Supercomput.*, 12:303–323, October 1998.

[2] Margarita Amor, Francisco Argüello, Juan López, Oscar G. Plata, and Emilio L. Zapata. A data parallel formulation of the barnes-hut method for n -body simulations. In *Proceedings of the 5th International Workshop on Applied Parallel Computing, New Paradigms for HPC in Industry and Academia*, PARA '00, pages 342–349, London, UK, 2001. Springer-Verlag.

[3] Josh Barnes and Piet Hut. A hierarchical $o(nlogn)$ force-calculation algorithm. *Nature*, 324(4):446–449, December 1986.

[4] Guy Blelloch and Girija Narlikar. A practical comparison of n-body algorithms. In *In Parallel Algorithms, Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1997.

[5] Geoffrey C. Fox. A graphical approach to load balancing and sparse matrix vector multiplication on the hypercube. *Institute for Mathematics and Its Applications*, 13:37–+, 1988.

[6] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998.

[7] Rakesh Ghiya, Laurie Hendren, and Yingchun Zhu. Detecting parallelism in c programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1:35–47, 1998.

[8] Alexander G. Gray and Andrew W. Moore. N-Body Problems in Statistical Learning. In Todd K. Leen, Thomas G. Dietterich, and Volker Tresp, editors, *Advances in Neural Information Processing Systems (NIPS) 13 (Dec 2000)*. MIT Press, 2001.

[9] Tsuyoshi Hamada, Tetsu Narumi, Rio Yokota, Kenji Yasuoka, Keigo Nitadori, and Makoto Taiji. 42 tflops hierarchical n-body simulations on gpus with applications in both astrophysics and turbulence. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[10] Lars Hernquist. Vectorization of tree traversals. *J. Comput. Phys.*, 87:137–147, March 1990.

[11] Ken Kennedy and John Allen, editors. *Optimizing compilers for modren architectures:a dependence-based approach*. Morgan Kaufmann, 2001.

[12] Milind Kulkarni, Martin Burtscher, Keshav Pingali, and Calin Cascaval. Lonestar: A suite of parallel irregular programs. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 65–76, April 2009.

[13] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. *SIGPLAN Not. (Proceedings of PLDI 2007)*, 42(6):211–222, 2007.

[14] M. Harris L. Nyland and J. Prins. Fast n-body simulation with cuda. *GPU Gems*, (3):677–695, 2007.

[15] Junichiro Makino. Vectorization of a treecode. *J. Comput. Phys.*, 87:148–160, March 1990.

[16] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *Int. J. Parallel Program.*, 29(3):217–247, 2001.

[17] Keshav Pingali, Milind Kulkarni, Donald Nguyen, Martin Burtscher, Mario Mendez-Lojo, Dimitrios Prountzos, Xin Sui, and Zifei Zhong. Amorphous data-parallelism in irregular algorithms. Technical Report TR-09-05, Department of Computer Science, The University of Texas at Austin, February 2009.

[18] Martin Rinard and Pedro C. Diniz. Commutativity analysis: a new analysis technique for parallelizing compilers. *ACM Trans. Program. Lang. Syst.*, 19(6):942–991, 1997.

[19] John K. Salmon. *Parallel hierarchical N-body methods*. PhD thesis, Pasadena, CA, USA, 1991.

[20] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. *J. Parallel Distrib. Comput.*, 27(2):118–141, 1995.

[21] Ingo Wald. On fast construction of sah-based bounding volume hierarchies. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 33–40, Washington, DC, USA, 2007. IEEE Computer Society.

[22] Bruce Walter, Kavita Bala, Milind Kulkarni, and Keshav Pingali. Fast agglomerative clustering for rendering. In *IEEE Symposium on Interactive Ray Tracing (RT)*, pages 81–86, August 2008.

[23] Bruce Walter, Sebastian Fernandez, Adam Arbree, Kavita Bala, Michael Donikian, and Donald Greenberg. Lightcuts: a scalable approach to illumination. *ACM Transactions on Graphics (SIGGRAPH)*, 24(3):1098–1107, July 2005.

[24] R. Clint Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimization of software and the atlas project. *Parallel Computing*, 27(1–2):3–35, 2001.

[25] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 63–76. ACM Press, 2003.