Optimizations for Message Driven Applications on Multicore Architectures

Pritish Jetley and Laxmikant V. Kale

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA

***Abstract*— With the growing amount of parallelism available on today's multicore processors, achieving good performance at scale is challenging. We approach this issue through an alternative to traditional thread-based paradigms for writing shared memory programs, namely *message driven multicore programming*. We study a number of optimizations that improve the efficiency of message driven programs on multicore architectures. In particular, we focus on the following *runtime system-enabled* optimizations: (i) grainsize control to effect a good concurrency-overhead tradeoff, (ii) dynamic balancing of processor load, (iii) low-overhead, asynchronous communication for lock-free and message-driven execution and (iv) communication-reduction through a novel *chunked* shared array abstraction. The practical impact of these optimizations is quantified through a parallel kd-tree construction program written in the message-driven paradigm. A comparison of the optimized code with a state-of-the-art parallel kd-tree construction program is also presented.**

***Index Terms*—Multicore programming, Message driven execution, Performance tuning, High performance graphics, SAH-based kd-tree construction**

I. Introduction

A large majority of supercomputers today are arranged as tightly coupled clusters of multicore processors. In addition, most user-end computers are based on multicore processors. To be able to exploit the full potential of these machines, application developers must spend a significant amount of effort tuning their applications for serial and multicore performance.

In this paper, we explore optimization techniques that are valuable in the mold of *message driven multicore programming*. Moreover, we note that all of these can be incorporated into an adaptive parallel runtime system, leading to greater modularity and reuse, and less effort on the part of the programmer. We carry out our study in the context of an application that is challenging to parallelize effectively. Surface-area heuristic (SAH) based kd-tree construction exhibits a tree-structured arrangement of parallelism, low intensity of floating point operations, and extensive data movement, making it hard to scale its performance with increasing processor counts.

We begin with a short discussion of programming on multicore architectures using the asynchronous message driven paradigm. Most of our experiments involve code written in the CHARM++ language and runtime system. Therefore, we provide a brief enumeration of the pertinent features of that system as well. This is followed by a description of the sequential SAH-based kd-tree construction algorithm due to Wald and Havran [23]. We then summarize the intuitive parallelization of this algorithm as given by Choi *et al.* [4]. Next, we identify elements of the kd-tree algorithm's implementation that offer opportunities for optimization in the asynchronous message-driven paradigm provided by CHARM++. Whereas message-driven execution (MDE) has seen notable application in the realm of distributed memory machines, we study its utility on shared-memory multicore systems. In particular, we study the following MDE-enabled opportunities for parallel performance optimization: grainsize control; dynamic load balancing and lightweight, asynchronous message based communication in place of collectives. In addition, we describe *chunked arrays*, which reduce the amount of synchronization between cores updating a shared array. Finally, the impact of these optimizations is studied by comparing the performance of the CHARM++ kd-tree construction program with the best-known multicore implementation to date [4].

II. Message Driven Programming for Multicores

The message driven execution (MDE) paradigm presents a departure from the traditional message passing and thread-based approaches to parallel programming. In a message driven program, messages hold not only the data being communicated between parallel entities, but also the intended recipient entity and the action to be performed on the data. In this sense, entities are *reactive* and different actions can be associated with the receipt of different types of message.

CHARM++ [14] is a parallel language and runtime system based on the concept of MDE. Whereas other expressions of the MDE paradigm exist (e.g. AM++ [25]), we consider the explicitly coarse-grained nature of CHARM++ objects to be well suited to the parallelization of the kd-tree algorithm presented here. In CHARM++, parallel C++ objects called *chares* encapsulate computation and exchange data with each other through asynchronous messages. Sequential pieces of computation are arranged in the form of *entry methods*. An entry method is like a regular C++ method; only, it is triggered by the receipt of a message from another chare, and once triggered, runs to completion, i.e. it is *non-preemptible*. This provision increases the level of reasoning that must be

applied to parallel programs from the ordering of individual instructions accessing shared variables (as is the case with thread-based programming) to ordering between messages.

Typically, CHARM++ applications leverage *object-based virtualization*, wherein several chares exist on each processing element (i.e. a thread, or a core). Chares may be created dynamically, and are distributed across the available processing elements by the runtime system. The runtime system has several load balancing strategies to balance dynamically generated work. The experiments in this paper use the *neighborhood* balancing technique, which is described in § V-B. The user is not exposed to threads and low-level synchronization between objects through mutex devices such as locks and fences. Synchronization between chares is realized through the causal dependencies that are embodied by entry method invocations and message receipts. Moreover, in the multicore setting that we experiment with, since chares exist in the same address space, data can be transferred efficiently via pointer exchange. This avoids the copying of data communicated between chares.

CHARM++ has been ported to most available distributed memory and shared memory platforms. Recent work has involved its tuning for multicore architectures and SMP clusters [18]. That work involved the incorporation of such techniques as CPU affinity to improve locality of memory access, fine-grained critical sections, lock-free message queues, and the use of thread-level storage to reduce false sharing between cores. We do not discuss these in detail here, but note that significant effort has already been invested in tuning CHARM++ for multicore platforms.

III. BACKGROUND

In order to better appreciate the performance issues that follow, we now touch upon the rudiments of tree-based scene rendering and kd-tree construction.

Ray tracing. The approximate, discrete representation of any physical surface may be obtained by triangulating it to the required degree of detail. This allows us to construct collections of triangles as finite approximations of arbitrarily complex graphical scenes. Although there are several methods available to map the resulting three-dimensional representation onto a two-dimensional image plane, the most extensible and flexible one is *ray tracing*. Given a three-dimensional collection of triangles, $\mathcal{C}$, a number of light sources, a two-dimensional image plane, $\mathcal{P}$, and a point of view, $\mathcal{V}$, the shade of each pixel p in $\mathcal{P}$ is determined by computing a representative subset of the rays of light that are incident on p. This is done by projecting a ray of light from $\mathcal{V}$ to $\mathcal{P}$ and tracing its path as it interacts with materials in the scene through phenomena such as reflection, refraction, scattering and diffraction. These phenomena may, in turn, cause the generation of more rays, each of which may be traced through the scene.

Spatial hierarchy and kd-trees. While it can yield realistic renditions of graphical scenes, the ray tracing algorithm is very demanding in terms of the amount of computation it requires. As described, the algorithm would calculate the intersection of each projected (or generated) ray with every triangle in $\mathcal{C}$. However, the spatial layout of the scene can be exploited in order to reduce the number of intersection calculations required per traced ray. This observation prompted the use of the kd-tree data structure to impose a spatial hierarchy on $\mathcal{C}$. Each node $\mathcal{N}$ of the tree has an associated three-dimensional, axis-aligned bounding box $\mathcal{B}_{\mathcal{N}}$, which encloses all the triangles in $\mathcal{N} \subset \mathcal{C}$. If a ray intersects $\mathcal{B}_{\mathcal{N}}$, its children (both of which are enclosed by $\mathcal{B}_{\mathcal{N}}$) are checked for intersection. This process continues recursively until the leaves of the tree are reached, at which point all the triangles enclosed by the leaf are tested for intersection with the ray. This scheme can greatly reduce the cost of intersction, making it proportional to leaf size and tree depth, which is expected to be logarithmic in the number of the triangles in $\mathcal{C}$.

Precise SAH-based kd-trees. As noted previously, each node of a kd-tree encloses a number of triangles. Each triangle has an extent along dimension $d \in \{X, Y, Z\}$, which is given by the minimum and maximum coordinates along d of the three vertices of the triangle. These maximum and minimum coordinates along d are referred to as the d-*events* of the triangle. Each event has an associated position, a type, which is either *START* or *END* and a reference to the triangle whose extent along d it marks. The union of the d-events of all triangles enclosed within a node n is referred to as $E_n(d)$, the set of all d-events of n. Note that each $e \in E_n(d)$ defines a plane, to which the direction vector of d is normal. In the following, we refer to an event interchangeably with the plane that it defines.

The defining characteristic of a kd-tree is the rule used to split a parent node into its two children. It has been shown that the surface area heuristic (SAH) yields kd-trees that are well-suited for rendering performance [4]. Whereas a discussion of the heuristic itself is beyond the scope of this paper (the reader will find a more thorough exploration of the subject in [9]), we summarize it in the following. In order to find the (locally) optimum partitioning plane for a node n, the SAH considers the cost of traversing the tree rooted at n for every dimension d and every prospective partitioning plane $e \in E_n(d)$. Since the tree beneath n cannot be obtained before the act of partitioning itself, the expected traversal cost is estimated by assuming that the two children obtained by splitting n at e are leaves. The plane e for which this cost is minimized, is the one selected as the partitioning plane.

Sequential kd-tree construction. The sequential algorithm for kd-tree construction is outlined in Algorithm 1. It begins by finding the best partitioning plane (*FindBestPartition*) among all possible events for the node along each dimension. As shown in the *FindBestPartition* procedure, the SAH assesses a candidate partitioning plane by considering (i) the balance of triangles between the two partitions that will be thus created and (ii) the surface area of each partition. Two appropriately set constants representing the cost of traversing a node (K_T) and the cost of performing an intersection calculation (K_I) between a triangle and a projected ray. If it is found that the estimated traversal cost of the node is greater than the cost

of performing intersection calculations for all of its enclosed triangles, the node is made a leaf. We refer to this test as the SAH opening criterion.

Algorithm 1: Sequential kd-tree construction algorithm

```
BuildKdTree(E[·,·], B)
Input: sorted events E[·,·], bounding box B
Output: tree node n
begin
    c_s ← ∞
    for d ∈ {X, Y, Z} do
        (c, p, i) ← FindBestPartition(E[d,·], B)
        if c < c_s then (c_s, p_s, i_s, d_s) ← (c, p, i, d)
    end
    if c_s > K_I × |E[d_s,·]| then
        return new TreeNode(E[d_s,·], B);
    end
    MarkTriangles(E[d_s,·], I)
    (E_L[·,·], E_R[·,·]) ← CopyEvents(E[·,·], i_s, d_s)
    (B_L, B_R) ← SplitBoundingBox(B, p_s)
    N_L ← BuildKdTree(E_L[·,·], B_L)
    N_R ← BuildKdTree(E_R[·,·], B_R)
    return new TreeNode(N_L, N_R)
end

FindBestPartition(E[·], B)
Input: sorted events E[·], bounding box B
Output: tree node n
begin
    n_L ← 0; n_R ← |E[·]|/2; A_B ← SA of B; c_s ← ∞
    for e ∈ E[·] do
        if e.type is END then n_R ← n_R − 1
        let A_L, A_R be areas of partitions of B at e.pos
        c ← K_T + K_I(n_L A_L/A_B + n_R A_R/A_B)
        if c < c_s then (c_s, p_s, i_s) ← (c, e.pos, i)
        if e.type is START then n_L ← n_L + 1
    end
    return (c_s, p_s, i_s)
end

MarkTriangles(E[·], I)
Input: sorted events E[·], split index I
begin
    N ← |E[·]|
    for e ∈ E[0..I] do
        if e.type is START then e.Δ.left ← 1
    end
    for e ∈ E[I..N − 1] do
        if e.type is END then e.Δ.right ← 1
    end
end

CopyEvents(E[·,·])
Input: sorted events E[·,·]
Output: sorted events for children, E_L[·,·], E_R[·,·]
begin
    for d ∈ {X, Y, Z} do
        E_L[d,·] ← ∅; E_R[d,·] ← ∅
        for e ∈ E[d,·] do
            if e.Δ.left then E_L[d,·] ← E_L[d,·] ∪ {e}
            if e.Δ.right then E_R[d,·] ← E_R[d,·] ∪ {e}
        end
    end
end
```

Once the best plane, p, is found and its associated dimension, d, recorded, the triangles are marked either as being to the left or to the right (respectively, above or below, on the near side or on the far side) of p along dimension d. This is done in the *MarkTriangles* phase.

Next, the events of each triangle are copied either to the left or to the right partition (or both, if the triangle straddles p), depending on how the triangle was marked in the previous phase. This phase of the partitioning is listed as *CopyEvents* in the algorithm. The marking process allows us to copy events associated with the marked triangles from a parent partition to its children, while maintaining their sorted order. This observation is crucial in achieving the algorithm's computational complexity of $\mathcal{O}(N \lg N)$. Once the triangles and events for the children partitions have been set up, the procedure is invoked recursively on each child.

Parallelization. For our study, we adapted the parallelization strategy detailed by Choi *et al.* [4] to CHARM++. The partitioning of a node is coordinated by a dynamically created task (chare). As is evident from Algorithm 1, the partitioning process of a node n involves a series of data-parallel operations, namely *FindBestPartition*, *MarkTriangles* and *CopyEvents*. To accomplish each one of them, the coordinating process enlists a collection of data-parallel tasks. As we will see in § V-B this collection can be static, or dynamically created. The three event arrays, which are the inputs to the data-parallel phases, are divided evenly among these data-parallel *worker* tasks.

The *FindBestPartition* phase begins with a parallel scan operation in which each worker contributes the number of triangles to the left and to the right of the last prospective partitioning plane in its portion of the event array. Once the scan operation has finished, each worker can perform the SAH based cost calculation for each event $e \in E_n(d)$, where $d \in \{X, Y, Z\}$. The event with the optimum partitioning cost is selected as the partitioning plane for n if the SAH opening critertion is satisfied.

Upon examining the *MarkTriangles* phase, we see that the *left* and *right* flags of each triangle are set by at most one worker. Therefore, this phase is easily parallelized. The *CopyEvents* phase is similarly easy to parallelize. Note, however, that the workers perform concurrent writes into the left and right children partitions' event arrays. Therefore, prior to copying its share of events, each worker must ascertain the offsets into these shared arrays at which it can start writing. This is achieved through a scan operation on the number of events each of them will copy to the left and right partitions, respectively. Once the copying of events is complete, the left and right flags of the triangles must be reset. In the sequential version of the algorithm, this would be accomplished by simply resetting them as the copying of events proceeds. However, for this scheme to work in the parallel algorithm, we would have to perform synchronizations between otherwise independent nodes in the tree. We solve this problem by giving each node its own set of triangles to read and mark. A new phase, *CopyTriangles* is introduced as intermediate to *MarkTriangles* and *CopyEvents*. In this phase, triangles are copied from the parent partition to the left and right children as necessary. Furthermore, two indices are maintained for each triangle, recording the new position of the copied triangle in the left and right partitions, respectively. These are used during *CopyEvents* so that the events in the children partitions refer to triangles enclosed in those partitions, and not the parent. Since the workers copy triangles into shared arrays, as before,

a scan operation must precede the *CopyTriangles* phase.

IV. Experimental Setup

In the following sections, we describe the experiments conducted to identify sources of inefficiency in the CHARM++ parallel kd-tree construction program. In tuning those aspects of the performance where it is feasible to do so, we first outline a motivating experiment, and follow it with tests studying the impact of the optimization on executions with actual input datasets. We use four standard datasets of varying size and scene layout.

The first of these is also the smallest dataset, namely *bunny*, consisting of about 67k triangles. The next is *fairy*, with about 170k triangles, *angel*, with 450k triangles and finally *happy*, consisting of 1.1m triangles. Qualitatively speaking, *fairy* has the most uniform scene layout of the four inputs. In the experiments that follow, the constructed tree was restricted to 8 levels of depth. We consider performance results from deeper trees in § VI. In each case, the results of tree construction were verified for correctness by comparing the output with that of the kd-tree construction program of Choi *et al.* [4].

We conducted our experiments on a shared memory machine comprising 4 sockets with 10 Intel Xeon E7-4860 cores each. The cores were clocked at 2.27 GHz. Although the cores support SMT, we did not use them in this capacity; a study of the impact of SMT on memory-intensive algorithms is beyond the scope of this paper. The system has ample memory–a total of 132 GB of RAM shared between 40 cores, and in keeping with the Nehalem architecture, four memory channels per socket, with a QPI-based network connecting the sockets to each module of memory. We used the `gcc` compiler suite for our experiments, compiling code with the flags `-O3 -funroll-loops -fomit-frame-pointer` in all cases.

V. Performance Optimizations

The CHARM++ adaptive runtime system enables the optimization of an executing parallel program along several dimensions. The inter-related techniques we describe here have seen wide application in the realm of distributed memory programs running on thousands of SMP nodes. Here, we show how their use in the context of shared memory programs can improve parallel efficiency on multicore processors.

A. Static grain size control

The algorithm used for kd-tree construction in this paper exhibits both task- and data-parallelism. The construction of subtrees rooted at nodes that do not share an ancestor-successor relationship can be performed concurrently. This fact is exploited by dynamically spawning *node tasks*, each of which coordinates the partitioning of a node of the tree. One must consider the tradeoff inherent in this strategy of dynamic task creation: on the one hand, the generation of a large number fine-grained tasks exposes more parallelism than a small number of coarse-grained tasks; on the other, there is less runtime overhead in dynamically creating a smaller number of tasks. This tradeoff is embodied in the choice of grainsize: one must select a value for this parameter that creates ample parallel work (i.e. node tasks), while allowing the associated overheads of parallelization to be amortized over sequential tasks of reasonable duration.

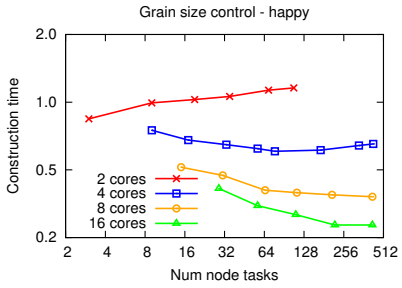

Fig. 1. Variation of tree construction time with grain size, as measured by the inverse of the number of node tasks.

Figure 1 illustrates this tradeoff for the largest of the input datasets, *happy*. The different curves in this figure represent the tree construction time on different core counts. Notice that the valleys of the curves shift progressively to the right, indicating that as more cores become available for execution, the overheads of a finer decomposition are compensated for by the reduction in total idle time across all processors. Whereas at 2 cores the best execution time is achieved with 3 coarse-grained tasks, that at 16 cores is attained with 250-500 medium-grained tasks. Recall that in our experiments the tree is no more than 8 levels deep, so that there can be a maximum of 511 node tasks in all. If we were to allow trees of greater depth, we would notice that finer dynamic decompositions of the work would lead to greater parallel overhead, and consequently, poorer performance. The red curve for 2 cores also shows that the cost of dynamic task creation can, indeed, be significant, especially at low core counts. Also notice that the yellow and green curves appear to flatten at high task counts. This observation indicates that parallel overhead is less of an issue at higher core counts.

Cores	Avg./Min.	Max./Min.
2	1.23	1.37
4	1.08	1.24
8	1.12	1.35
16	1.16	1.45

TABLE I
AVERAGE-CASE AND BEST-CASE SPEEDUPS OBTAINED BY STATIC GRAINSIZE DETERMINATION.

Table I quantifies the importance of grainsize control with an increase in core count. The table shows two figures for each core count. The first of these gives us an idea of the average improvement that can be expected by picking the appropriate grainsize for an execution. On average, the program would be 8-16% slower if static grainsize control were not employed. The second column reflects the worst case in performance slowdown–the program can be anywhere from 24-45% slower

if a bad grainsize is selected. In this context, auto-tuning techniques (e.g. [6]) are of great relevance, especially if the tree construction procedure is repeated several times.

B. Dynamic load balancing for data parallel tasks

In the previous subsection, we discussed the relevance of static grainsize control for node tasks, which embody the *task-level* parallelism inherent in the kd-tree construction algorithm. Now, we focus on the dynamic generation of *data-parallel, worker* tasks, and the dynamic load balancing that must be performed in assigning these tasks to cores.

As noted in § III, the task of splitting a parent node into its children is composed of three phases, each of which exhibits a data-parallel structure.

However, there is a great deal of imbalance in the work required to partition different nodes. We observed that the number of triangles enclosed within a node varies non-monotonically with the level (*inter-level variation*). Moreover, there is a variation of up to 10 times in the number of enclosed triangles between nodes of the same level (*intra-level variation*). These effects contribute to imbalance of work when triangles and events are distributed among workers.

In order to balance the worker tasks across processor cores, we first formulated a *static* balancing technique. In this scheme, indexed collections of worker tasks are created *a priori* and reused by different nodes in order to perform data-parallel partitioning work. Collections of several sizes are created, so that nodes with different numbers of enclosed triangles may enlist an appropriate number of data-parallel tasks. For instance, with 32 processing cores, there may be 1 collection of size 32 (call this task collection $T_{32,0}$), 2 of size 16 ($T_{16,0}$ and $T_{16,1}$), 4 of size 8 ($T_{8,0}$, $T_{8,1}$, $T_{8,2}$ and $T_{8,3}$), etc. These collections are spread across cores so that node tasks with similar amounts of data-parallel work may be partitioned in parallel. Continuing with the example, $T_{32,0}$ is spread across all 32 cores; $T_{16,0}$ and $T_{16,1}$ are distributed over cores 0-15 and 16-31, respectively; $T_{8,0}$, $T_{8,1}$, $T_{8,2}$ and $T_{8,3}$ over cores 0-7, 8-15, 16-23 and 24-31, respectively, etc. A node task on processor core p that requires a task collection of size m to perform its data-parallel work, enlists collection $T_{m,p/m}$ for this purpose. This helps to balance the data parallel work across all the cores. To further reduce contention among node tasks for collections, we replicate the collections. Node tasks then employ a randomly chosen replica from the pool of collections of a particular size. We found that these provisions coped quite well with the load imbalance engendered by inter-level variation of triangle counts. However, collections tend to *interfere* with the execution of each other, since smaller collections share their processor cores with larger ones. For instance, it may happen that the execution of one phase of $T_{16,0}$ gets delayed by the execution of another phase of $T_{8,0}$. If $T_{16,0}$ were partitioning a shallower node than $T_{8,0}$, this would delay the critical path of execution, slowing down the entire construction procedure.

To overcome this interference, we switched to a more dynamic assignment of data-parallel work to cores. Instead of using statically created task collections, node tasks dynamically create data-parallel worker tasks. The particular strategy for assignment of these tasks to cores depends on the number of data-parallel tasks created. If this number is larger than the total number of cores, the tasks are spread evenly across all cores. Otherwise, they are enqueued for execution on the local core. Periodically, processors that have little or no work poll their neighbors (as decided by a virtual processor topology) for data-parallel tasks, so that work is balanced across the entire set of processors.

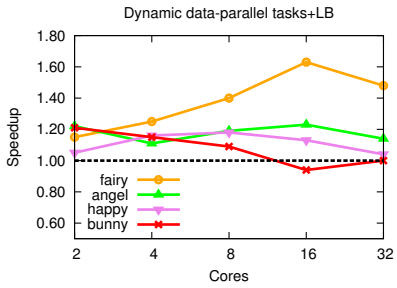

Fig. 2. Speedup obtained by using dynamic load balancing instead of a static scheme for the assignment of data-parallel tasks to cores.

Figure 2 shows the resulting improvement in performance by switching to the dynamic load balancing scheme. Whereas the *fairy* dataset shows significant benefits across the entire range of processor counts, *angel* and *happy* show an improvement in performance of less than 20%. Moreover, for the smallest of the datasets, *bunny*, the gains in performance due to improved load balance are more than offset by the overheads associated with dynamic creation of data-parallel tasks.

C. Lightweight fork-join synchronization

The parallelization of the kd-tree construction algorithm studied in this paper makes heavy use of the fork-join paradigm. A coordinating node task forks a number of worker tasks to perform a certain function, and control is eventually returned to the coordinator. This requires frequent synchronization between tasks through barriers and multicasts, which initiate the different data-parallel phases (SAH calculation, triangle and event copying) related to node partitioning. In the CHARM++ system, these modes of synchronization are provided through *section reductions* and *section multicasts*. However, in our initial implementations of the CHARM++-based kd-tree construction algorithm, we found that there were significant overheads associated with the use of these communication primitives. The reason for these overheads is as follows. As mentioned previously, the CHARM++ runtime system supports object-based virtualization, wherein several objects embodying computation are hosted on each available processor core. Having been designed for distributed memory systems where inter-processor messaging is expensive, CHARM++ reductions consist of two phases. In the first phase, all participating objects on a core make their contributions to a core-level reduction manager. This is followed by a second, spanning-tree based reduction phase over all the cores in the

system. Multicasts follow this pattern in reverse: a multicast involving all cores is performed first, followed by message delivery to all addressed objects on a core.

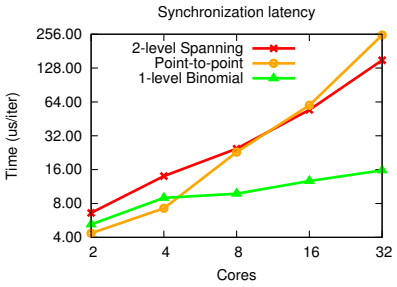

Fig. 3. A comparison of multicast-reduction performance for regular, and multicore-specific CHARM++ primitives. The y axis shows the average time taken per round of multicast-reduction synchronization.

Figure 3 quantifies the overheads associated with the use of these two-level schemes through a small benchmark program. This benchmark repeatedly performs a multicast followed by a reduction-based barrier. Since we were interested in the latency of these operations rather than the bandwidth achieved, small messages were used in the operations. The red curve in Figure 3 shows that the cost of using this two-level synchronization scheme grows rapidly with the number of cores used. As a point of comparison, we also implemented a simpler one-level scheme based on the use of point-to-point messages between the forked objects and the forking object. Its performance is depicted by the orange curve. For small core counts, the point-to-point scheme outperforms the two-level scheme significantly. However, the overheads of allocating and transmitting individual messages eventually degrade its performance.

In order to improve the communication efficiency of the runtime system for multicore systems, we wrote a lightweight synchronization module. This module provides basic section multicast and section reduction primitives through binomial tree based messaging between objects. The reduction manager of CHARM++ is not involved. This approach combines the benefits of one-level point-to-point synchronization with the efficiency of spanning tree based primitives. The performance of the binomial tree based synchronization primitives on the multicast-barrier benchmark is presented by the green curve in Figure 3. Although the point-to-point scheme is slightly more efficient than the binomial tree at small core counts, there are significant benefits in the use of the latter at larger core counts.

Figure 4 shows the increase in parallel efficiency of the kd-tree construction program through the use of this lightweight, binomial tree based synchronization module. The y axis plots the speedup obtained by using the binomial tree module for fork-join communication instead of the default CHARM++ primitives. All four input datasets show significant improvements in performance, the *fairy* dataset being the greatest beneficiary of this optimization, with a maximum speedup of over 3 on 16 cores. Even though these gains are significant, they are not as marked as the improvements in the performance of the

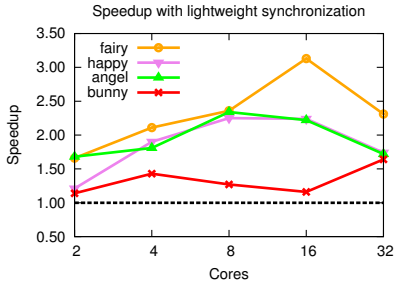

Fig. 4. Using a lightweight, variable strategy for reductions improves performance across different magnitudes of input data size and processor counts. The y-axis plots speedup obtained by choosing a combination of binomial and point-to-point message-based reductions over the default CHARM++ (2-level) implementation.

much simpler benchmark because (i) synchronization accounts for a much smaller portion of the parallel execution and (ii) the CHARM++ runtime adaptively overlaps the communication latency with useful computation.

D. Parallel scans for multicore

The parallel scan operation is another frequently used communication primitive in the kd-tree construction algorithm. Whereas a recursive doubling (RD) scheme would be more appropriate in a disributed memory setting with high communication startup overheads, as Choi *et al.* [4] note, in a multicore setting it is more efficient to use a *prescan-push* (PP) technique. Given a data array of n elements and p tasks, in the prescan phase, each task sums n/p elements in parallel, thereby computing its contribution to the scan result. This is followed by a push, in which a barrier transfers control to a single task, on which the p contributions are summed to obtain the final scan result.

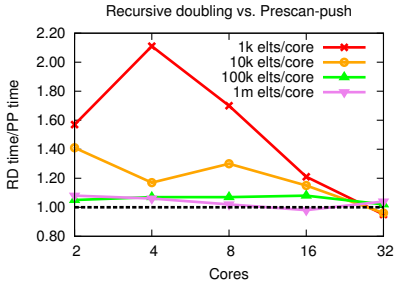

Fig. 5. A comparison of prescan-push (PP) and recursive doubling (RD) as parallel scan algorithms for multicore processors.

Figure 5 shows the relative performance of the two approaches. The y axis plots the ratio of time taken by RD to that taken by PP for a number of core counts and different amounts of sequential computation. As can be seen, the relative advantage of using the prescan-push technique is quite marked at low processor counts when there is little sequential work to be done in the prescan phase. However, as the amount of prescan work grows in comparison to the communication time, the difference between the algorithms is less evident. Furthermore, with an increase in the number of processor

cores, the difference in performance of the two algorithms diminishes. Therefore, we use the recursive doubling approach in the SAH phase of the shallower nodes of the tree, and the prescan-push algorithm for SAH computation in the deeper levels of the tree. The prescan-push approach is also used in the offset calculation scans for triangle and event copying.

However, given the limited impact of this optimization, and the fact that the number of parallel scan operations performed can be reduced significantly (§ V-E), we do not include a discussion of the improvements in application performance that it yields.

E. Chunked arrays

As mentioned in § III the parallel scan operation features in the algorithm in three contexts: (i) to obtain the number of triangles to the left and right of each event when calculating SAH values for prospective splitting planes, (ii) in order to obtain the write offsets of each worker involved in the data-parallel copying of triangles from the parent partition to its children and (iii) similarly to (ii), in order to determine write offsets for each task while copying events from parent to children partitions. Note that the scans in (ii) and (iii) are required only as a coordination mechanism between concurrent tasks, so that they may perform coherent writes to a shared array. Instead, if each task were to write to an independent buffer, and the collection of buffers thus obtained were addressable in a manner similar to a regular C++ array, we wouldn't require these coordination phases at all.

This is the primary motivation behind the *chunked array* abstraction. A chunked array is similar to a regular C++ array in that it holds a collection of values of a certain type. However, unlike an array, which is laid out as a contiguous run of bytes in memory, a chunked array consists of several *chunks* of memory, each chunk a contiguous C++ array in itself. Therefore, each element in a chunked array has two indices: a *local* one that describes its position within its chunk, and a *global* one, that identifies it uniquely among all elements (including those in other chunks) held in the chunked array.

Consider the utility of chunked arrays in the kd-tree construction algorithm, specifically in the copying of triangles (*cf.* (ii) above) from parent to children partitions. In the *MarkTriangles* phase each triangle is marked as belonging to the left or right (or both) partition(s). If we were to use C++ arrays to hold triangles workers would be involved in scan operations to determine starting write offsets (§ III). However, using chunked arrays, once each task has determined its contribution to the left and right triangle partitions, it would simply allocate two chunks, fill them with the appropriate triangles and add these chunks to the left and right partitions' chunked arrays, respectively. This approach completely avoids the use of parallel scans for coordination. There is another benefit to the use of chunked arrays, that of avoiding false sharing and in generally reducing the amount of cache coherence traffic. However, we do not explore this advantage here.

Although chunked arrays save the application the communication cost of performing scan operations, there are overheads associated with their use. First, whereas with a regular C++ array only a single `malloc` is required to allocate a shared buffer, chunked arrays require as many chunks as there are tasks. However, in our experiments, good OS-level memory allocators meant that these did not account for a significant penalty to performance. The second, and more significant source of overhead, is the cost of addressing individual elements in the chunked array through a *global* index.

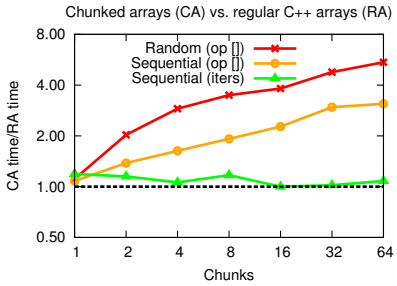

Fig. 6. Comparison of element access latency for regular C++ arrays and chunked arrays.

Figure 6 compares the cost of accessing elements stored in a chunked array (CA) relative to that of accessing elements in a regular C++ array (RA). Two modes of access, namely random and sequential access, are studied. In all experiments, the total number of elements was kept constant. Therefore, as the number of chunks increases along the x axis, the size of each chunk decreases. A chunked array exposes a square bracket operator `[]`, providing the array-access notation familiar to C++ programmers. However, as the red curve in Figure 6 shows, random access of elements through this operator can be expensive. The operator invocation costs aside, the identification of elements using a global index requires the traversal of a tree structure internal to the chunked array. Therefore, the access of a single element actually involves several metadata accesses, the cost of which grows with the number of chunks. This eventually creates a gap in performance of over 5 times between regular and chunked arrays. We note that these costs could possibly be reduced by maintaining a b-tree or a hash table, instead of a binary comparison tree as we do.

Fortunately, however, the kd-tree construction program and many other parallel applications exhibit a linear memory access pattern. The orange curve in Figure 6 shows the relative cost of using the `[]` operator to access sequential elements spread across several chunks. While less expensive than random accesses, the notational convenience to the programmer incurs significant runtime overhead–the average access time using the operator is up to 3.1 times that of a regular C++ array. However, chunked arrays allow a more efficient way to access sequential elements, namely through *iterators*. As the name suggests, these are used in a manner similar to STL iterators. Moreover, as shown by the green curve in the figure, their use incurs little overhead. In this manner, chunked arrays allow programs to avoid synchronization between tasks during shared array updates, while adding little sequential overhead.

In order to quantify the gains in performance due to this

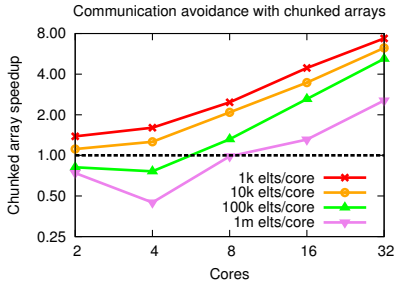

Fig. 7. Use of chunked arrays reduces synchronization costs. The y axis plots the ratio of update time of a regular array to that of a chunked array.

communication avoidance, we conducted an experiment that compared the time taken to update, in parallel, a large array that is shared between tasks, with the time taken to update a similarly sized chunked array. Figure 7 shows the benefit of avoiding the scan operation during the update. As shown by the red and yellow curves, the benefits are especially prominent when there is relatively little sequential updating to do. As expected, this performance advantage dwindles as the amount of sequential work begins to dominate. In fact, for a large enough sequential update, the overheads of chunked array use are quite significant on small numbers of cores.

Of course, it would be unreasonable to expect similarly marked improvements in the performance of the actual application itself. The reasons for this are similar to those given in § V-C. However, in order to quantify the importance of communication-avoidance, we compared the performance of two versions of the kd-tree construction code, the first of which used regular C++ arrays together with the scan-based synchronization between tasks, and the second one, chunked arrays.

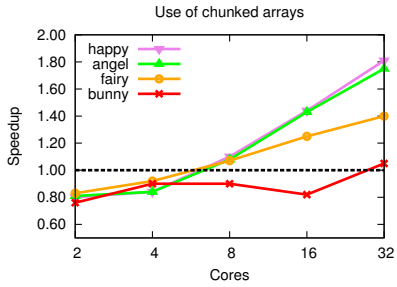

Fig. 8. Speedup obtained by using chunked arrays in the CHARM++ kd-tree construction program. On small core counts, it is not feasible to use the chunked array abstraction. However, on large core counts, its use mitigates the amount of synchronization required between cores.

The ratio of time taken by the former to that taken by the latter is shown in Figure 8. As we had observed with the performance of the benchmark, for small numbers of cores, we see a slight performance penalty associated with the use of chunked arrays. However, the abstraction is more valuable as the number of cores is scaled up. For the two largest input datasets, we see an improvement of about 80% is application performance. The results are less spectacular with the *fairy* dataset, which showed an improvement of 40% in total construction time. Performance was actually hampered by chunked array use in the smallest of the datasets, *bunny*.

VI. RESULTS

In this section we briefly examine the overall impact of the optimizations discussed in § V. As a point of comparison we use the ParKD kd-tree construction program of Choi *et al.* [4]. ParKD is based on Intel's Threaded Building Blocks [19] framework, and, to the best of our knowledge, provides the best-known speedups on multicore platforms. We used the experimental setup described in § IV for both the CHARM++ kd-tree construction program and ParKD. We only used the actual tree construction time to compare the two codes. The I/O, initialization and sorting times were not considered.

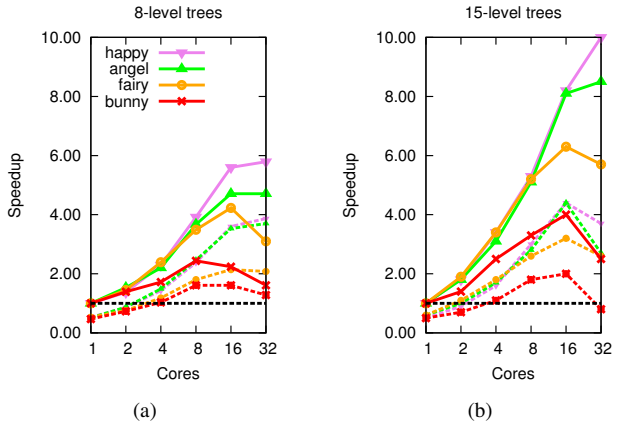

Fig. 9. A comparison of absolute speedups obtained by the CHARM++ and ParKD codes for (a) 8-level trees and (b) 15-level trees. The baseline for these speedups was the uniprocessor execution time of the CHARM++ code. Solid lines show speedups obtained with the CHARM++ code, and dashed ones, those with ParKD.

Figure 9 shows absolute speedups obtained for the two codes in constructing trees of depth 8 and 15. The speedups are calculated relative to the serial performance of the CHARM++ kd-tree code, which is faster than the sequential baseline of [4], and is therefore an appropriate comparison point for speedups. The solid lines show the scaling profile of the CHARM++ code, and the dashed ones, that of ParKD. Both codes demonstrate relatively poor overall scaling when constructing the shallower, 8-level trees (Figure 9(a)). However, the speedups become more appreciable with larger datasets–as the parallel overhead is amortized over more sequential work, we obtain better efficiency. CHARM++ code attains a maximum speedup of almost 6 on 32 cores for the *happy* dataset. By raising the number of levels in the tree to 15, we allow for a maximum of about $32k$ nodes to be constructed, thereby increasing the total amount of work significantly. As seen in Figure 9(b), this increase in work improves the performance of the CHARM++ code, allowing it to achieve a speedup of 10 times on 32 cores for the *happy* dataset. Performance analysis for this dataset identified communication overhead and limited memory bandwidth as the major impediments to better scalability. The performance of ParKD does not scale to 32 cores on any of the inputs

for the construction of 15-deep trees. Next we compare the relative performance of the two codes.

	Max. depth 8				Max. depth 15			
Cores	*bunny*	*fairy*	*angel*	*happy*	*bunny*	*fairy*	*angel*	*happy*
1	2.1	1.9	1.9	2.0	2.2	1.6	1.7	2.0
2	1.9	1.9	1.8	1.5	2.0	1.8	1.7	2.0
4	1.7	2.0	1.5	1.4	2.3	1.8	1.8	2.2
8	1.5	1.9	1.5	1.4	1.9	2.0	1.8	1.8
16	1.4	2.0	1.4	1.4	2.0	2.0	1.8	1.9
32	1.3	1.5	1.3	1.4	3.1	2.2	3.1	2.7

TABLE II
SPEED OF CHARM++ CODE RELATIVE TO PARKD. LISTED ARE THE RATIOS OF TIME TAKEN BY THE CHARM++ CODE TO THAT TAKEN BY PARKD FOR VARIOUS CORE COUNTS.

Table II shows the ratio of time taken by ParKD to construct a kd-tree to that taken by the CHARM++ code, for various input datasets and core counts. We term this ratio *relative gain*. Comparing the relative gains for trees of depth 8 first, we note that the gains of about 2 at lower core counts are significant. We believe that this difference on 1 and 2 cores is the result of better sequential performance and better grainsize management in the CHARM++ code. For *bunny*, the relative gain falls with the number of cores, since the small size of the problem makes it hard to scale either code. Even at 32 cores, however, the CHARM++ code is 1.3 times as fast as ParKD. The relative gain is greatest with the more uniform *fairy* dataset. With this dataset, the CHARM++ code is nearly 2 times as fast on upto 16 cores. Although the gain falls at 32 cores, it is still respectable, at 60%. The *angel* dataset shows a similar profile to the *bunny* dataset, even though it is much larger in size. We did not investigate the reasons for this. Finally, for the largest of the inputs, *happy*, the CHARM++ code shows a consistent relative gain of about 40% over ParKD for 2 through 32 cores.

The second set of columns in Table II shows relative gain for the construction of trees of depth 15. The results indicate that as the amount of work to be done is increased, the CHARM++ code shows more consistent gains. For this depth of tree, the CHARM++ code is 1.5-2.0 times as fast through 16 cores. At 32 cores, the performance of the ParKD code falls quite significantly, leading to a ParKD to CHARM++ ratio of nearly 3 times.

VII. RELATED WORK

We discuss related work in two parts, the first of which presents a brief literature survey of shared memory programming and performance tuning for multicore systems. Traditional languages and frameworks for programming multicore systems either provide very low-level constructs, or a performance model that is not well-matched with the underlying hardware. For instance, with *pthreads* [11] programmers must write code in terms of interleaved streams of instructions performing concurrent memory accesses. The productivity issues of this aside, the concomitant use of locks can be a hazard for performance as well. Interest in user-level threads has recently been revived with work on *QThreads* [24]. *OpenMP* [5] provides a notionally simpler way of specifying implicitly data parallel operations in the form of *for* loops whose iterations can be spread over the available set of processors. This programming model, however, obscures key elements of machine performance, such as the caching of data. More recently OpenMP has acquired primitives to specify task-parallel operations as well. Task-based fork-join parallelism is made simpler with *Cilk* [2] and Intel's *Threaded Building Blocks* [19]. However, with Cilk, communication between tasks is implicit and cannot be expressed by the programmer. Similarly, using TBB, the programmer is constrained by the range of parallel constructs provided therein. As shown in § V-C, not having explicit control over communication can hinder application performance. In the same vein as this paper, researchers have previously conducted studies of application architecture and optimization for multicore systems [3].

Bentley [1] and Friedman *et al.* [7] did some of the earliest work on kd-trees for distance-based searches and linear-time matching algorithms. Median-based construction, which involves the partitioning of prospective nodes along their medians, is a well studied problem. Zhou *et al.* [26] have provided a fast algorithm for the construction of such trees. The surface area heuristic for partitioning kd-tree nodes is known to improve scene rendering time, and has been studied by several researchers [8], [17]. The sequential algorithm that forms the basis of the CHARM++ kd-tree construction program and ParKD, is due to Wald and Havran [23]. The complexities of ray tracing and kd-tree construction are also discussed in great depth in Havran's doctoral thesis [9]. Work has been done to speed up the evaluation of node partitioning heuristics for kd-trees, while maintaining the quality of these partitions. Shevtsov *et al.* [20] discuss an approximation to the SAH in their work. Several parallel implementations of SAH-based kd-tree construction programs also exist. In particular, the work of Choi *et al.* [4] has been key in the exploration of new parallelizations of the kd-tree construction algorithm. In addition to providing the basis for the kd-tree construction algorithm used here, they also provide a so-called *in-place* algorithm that reduces the amount of data movement. Researchers in the graphics community have studied alternatives to the kd-tree for hierarchical representations of scenes. Lauterbach *et al.* [16] and Wald [21] provide fast parallel implementations for BVH construction. Wald [22] also gives efficient implementations of BVH construction on the recently released MIC architecture. Hou *et al.* [10] discuss the generation of BVHs and kd-trees on GPUs, and Kalojanov *et al.* [15] and Ize *et al.* [13] use grids to accelerate ray tracing. In addition, researchers have looked at using the hardware parallelism available in multicore systems in a different way–Ize *et al.* [12] provide an "asynchronous" algorithm that conducts tree construction and ray tracing in parallel for deformable scenes.

VIII. CONCLUSION AND FUTURE WORK

In this paper we explored several techniques for the optimization of *multicore MDE* programs. We studied the ben-

efits of these techniques in the context of a parallel kd-tree construction program written in CHARM++. The parallelism inherent in the kd-tree algorithm has a hierarchical layout, so that each node can be constructed by an independent task. Moreover, the partitioning of a node involves medium-grained data-parallel work that is memory intensive, with few floating point operations to keep CPUs occupied. These are characteristics that kd-tree construction shares with many other parallel applications, so that the results presented herein have more general appeal. We showed that multicore MDE performance can be improved through static grainsize control, load balancing of dynamically created data-parallel tasks, the provision of efficient communication primitives suited to multicore systems, and the use of abstractions such as the *chunked array* to reduce synchronization between cores. Note that these optimizations apply equally well in the distributed memory world.

Finally, in order to see the cumulative impact of these optimizations, we compared the performance of the CHARM++ code to that of ParKD, which is a parallel kd-tree construction program based on Intel's TBB framework. We demonstrated significant improvements in performance over a range of input datasets and processor cores. We observed that as the depth of the constructed trees was increased, thereby leading to more parallel work, these improvements gained in magnitude and became more consistent. The CHARM++ version was 1.5-2.5 times as fast as ParKD for such trees.

Future work will explore the multicore MDE paradigm in the context of other applications. While the performance benefits of using MDE have been studied, we would also like to examine the improvements in programmer productivity that this paradigm engenders. More generally, we would like to see what abstractions would enable the portability of multicore MDE programs across shared- and distributed-memory machines.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the financial support of the NSF (grant ITR-HECURA-0833188) and the provision of hardware through the UPCRC collaboration between the University of Illinois, Intel and Microsoft.

REFERENCES

[1] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509–517, September 1975.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[3] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. W. Vuduc. Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In *IPDPS'10*, pages 1–12, 2010.

[4] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart. Parallel sah k-d tree construction. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 77–86, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[5] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1), January-March 1998.

[6] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[7] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematics Software*, 3(3):209–226, September 1977.

[8] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *Comp. Graph. and Appl., IEEE*, 7(5):14 –20, may 1987.

[9] V. Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[10] Q. Hou, X. Sun, K. Zhou, C. Lauterbach, and D. Manocha. Memory-scalable gpu spatial hierarchy construction. *IEEE Trans. Vis. Comput. Graph.*, 17(4):466–474, 2011.

[11] Draft Standard for Information Technology—Portable Operating Systems Interface (Posix), September 1994.

[12] T. Ize, I. Wald, and S. G. Parker. Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures. In *Proc. 2007 Eurographics Symp. on Parallel Graphics and Visualization*.

[13] T. Ize, I. Wald, C. Robertson, and S. G. Parker. An evaluation of parallel grid construction for ray tracing dynamic scenes. In *In Proc. 2006 IEEE Symp. on Interactive Ray Tracing*, pages 47–55, 2006.

[14] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, 1993.

[15] J. Kalojanov and P. Slusallek. A parallel algorithm for construction of uniform grids. In *HPG '09: Proceedings of the 1st ACM conference on High Performance Graphics*, pages 23–28, New York, NY, USA, 2009. ACM.

[16] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. *Comput. Graph. Forum*, pages 375–384, 2009.

[17] D. J. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6:153–166, May 1990.

[18] C. Mei, G. Zheng, F. Gioachin, and L. V. Kalé. Optimizing a Parallel Runtime System for Multicore Clusters: A Case Study. In *TeraGrid'10*, number 10-13, Pittsburgh, PA, USA, August 2010.

[19] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[20] M. Shevtsov, A. Soupikov, and A. Kapustin. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Comput. Graph. Forum*, 26(3):395–404, 2007.

[21] I. Wald. On fast Construction of SAH based Bounding Volume Hierarchies. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing*.

[22] I. Wald. Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture. *IEEE Transactions on Visualization and Computer Graphics*, 2010. (to appear).

[23] I. Wald and V. Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–70, 2006.

[24] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *International Parallel and Distributed Processing Symposium/International Parallel Processing Symposium*, pages 1–8, 2008.

[25] J. Willcock, T. Hoefler, N. Edmonds, and A. Lumsdaine. AM++: A generalized active message framework. *The International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 09/2010 2010.

[26] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27:126:1–126:11, December 2008.