Recursion-Driven Parallel Code Generation for Multi-Core Platforms

Rebecca L. Collins, Bharadwaj Vellore, and Luca P. Carloni
Department of Computer Science, Columbia University, New York, NY 10027
Email: {rlc2119,vrb2102,luca}@cs.columbia.edu

***Abstract*—We present Huckleberry, a tool for automatically generating parallel implementations for multi-core platforms from sequential recursive divide-and-conquer programs. The recursive programming model is a good match for parallel systems because it highlights the temporal and spatial locality of data use. Recursive algorithms are used by Huckleberry's code generator not only to automatically divide a problem up into smaller tasks, but also to derive lower-level parts of the implementation, such as data distribution and inter-core synchronization mechanisms. We apply Huckleberry to a multi-core platform based on the Cell BE processor and show how it generates parallel code for a variety of sequential benchmarks.**

I. Introduction

Recursive and hierarchical models have been used in many contexts for programming parallel systems, and are a natural fit for exposing concurrency in a program because recursion concisely captures patterns of dependencies and exposes temporal and data locality [1], [2], [3], [4]. Modern multi-core platforms feature chips with multiple processing cores connected by powerful on-chip communication networks that enable high-throughput low-latency data transfers between them. For example, IBM's Cell BE processor hosts one PowerPC core and 8 Synergistic Processing Units (SPUs) together with an Element Interconnect Bus that supports up to 205 GB/s of data transfer and latency of only 50-100 nanoseconds [5], [6], [7]. On-chip communication on these systems is an order of magnitude faster than off-chip communication, and therefore it makes sense to minimize the latter whenever possible.

We propose *Huckleberry*, a tool to enable automatic parallelization that takes advantage of the new balance of communication costs that have come with multi-core architectures. Huckleberry's *recursive parallel model* supports interaction between nested calls including data-dependencies between the calls and mutually recursive functions that alter the nested data access pattern. It does so by leveraging the powerful mechanisms for inter-core communication and synchronization that are typically provided by on-chip networks, while making their use transparent to the programmer. Huckleberry abstracts parallelism by allowing programmers to focus exclusively on data partitioning. Fig. 1 illustrates the Huckleberry design flow. The programmer provides one or more *divide-and-conquer* recursive functions that employ Huckleberry's Partition Library application programming interface (API). The code generator takes these functions together with specifications of the underlying architecture and returns a parallel implementation. The recursive task graph resulting from the combined recursive functions of an application, called an *R-Tree*, is used in the parallel implementation to: (1) break the problem up into subproblems small enough to fit onto the chip, (2) distribute data across the cores, and (3) coordinate data swaps when necessary between the cores. Huckleberry also gives the programmer the flexibility of specifying which core is responsible for which task (through its *parallel-index function* f_{pi}) and of reusing optimized local code that runs at the leaves of the *R-Tree* (i.e. on individual cores). Decentralization is a central idea to our approach, and is achieved by allowing cores to calculate for themselves which tasks they are responsible for, and when and with whom they should swap data. Our experiments show that Huckleberry is able to automatically generate a parallel implementation from sequential functions for several benchmarks for a complex multi-core platform such as the QS20 Cell Blade [8], which comprises 18 heterogeneous distributed memory cores over two Cell BE chips. By keeping as much communication local to each chip as possible, Huckleberry takes advantage of the performance edge delivered by high-speed on-chip networks.

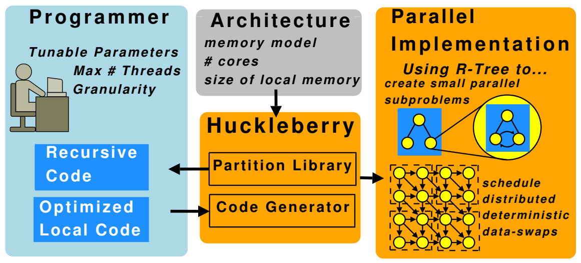

Fig. 1: Huckleberry design flow.

II. Huckleberry Programming Interface

Huckleberry is based on the C programming language, supplemented by Huckleberry's partition API. The constraints on the programmer are as follows: foremost, we support only recursive divide-and-conquer functions that have the property that the divide step can be determined before the compute steps (i.e. the partitioning of the data does not depend on the data values; however, there can be *data dependencies* between branches where several steps of the algorithm alter the same data.). If a function has this property, as it is the case for Bitonic Sort that we introduce as an example later in this section, the programmer can modify it to be accelerated by Huckleberry simply by wrapping all of the function's parameters with the API's `Partition` data structures. In return, Huckleberry abstracts away the details of implementing a parallel algorithm. The programmer does not need to separate the algorithm into independent tasks or consider architectural details like the number of cores or the size of the local memory. Huckleberry supports mutually recursive functions that invoke one another, and multidimensional data in user-defined types.

Machine Model. A distributed memory multi-core system is made up of a set of N cores, each of which is associated with a local memory whose capacity is denoted m_i for core c_i. There may also be an *organizer core (OC)* dedicated to sequential and administrative tasks. The capacity m_i reflects the available memory for data once space for the application code and temporary buffers has been accounted for. The

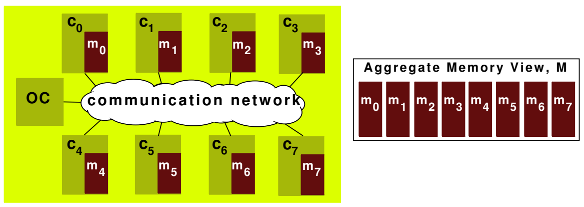

Fig. 2: Abstract machine model.

aggregate local memory, denoted $M = \bigcup_{\forall i} m_i$, is the sum total of the local memories of the cores. The value of m_i is processed by Huckleberry as an input parameter to generate the parallel code and can be varied to change the granularity of the parallel execution; m_i can also vary depending on the particular application. As in typical sequential programming models, we keep the notion of data separate from memory: The program input data set is denoted as I while D denotes the working data set that is stored in the aggregate local memory M at any given time during the execution of the program. Finally, d_i denotes the subset of D that is stored in the local memory space m_i. Fig. 2 illustrates our machine model in an abstraction of the Cell processor, with 8 SPE vector cores, and one PowerPC core which serves as the OC.

Partition Library. The `Partition` API is the centerpiece of the Huckleberry partition library. User-provided functions must use partitions for all of their parameters. Partitions support generic data structures, but annotate the actual data with meta-information about the data; for example, array-based metadata includes data type, dimensions, and where a partition's data begins and ends within each dimension. The partition API includes the following functions:

- `create_partition()` creates and fills a partition;
- `free_partition()` frees the memory of a partition;
- `left_half()` copies a partition's metadata into a new partition, altering the new partition to only include the original partition's left half;
- `right_half()` inverse of `left_half()`;
- `copy_last_element()` copies the last element of an array into a new unit partition (a small data structure that is not divided but is passed down the R-Tree intact);
- `update_int()` updates an integer unit partition;
- `mydata_intersects()` returns true if any part of two partition sets intersect and false otherwise;
- `mydata_contains()` compares two sets of partitions, a local and global set; returns true if the local set entirely contains the global set, and false otherwise;
- `partition_size()` calculates the size of a partition based on the number of dimensions in a partition and the begin and end boundaries of each partition.

The partition library provides functions that perform operations on partitions to reduce their size for the divide step of divide-and-conquer functions, including adjusting their data pointers and keeping track of the original boundaries and where the new reduced partition lies within them. Our initial implementation of Huckleberry supports data that is arranged in arrays of one or more dimensions. Data structures such as trees could also be supported using the same concepts. For example, `left_half()` and `right_half()` may take the left

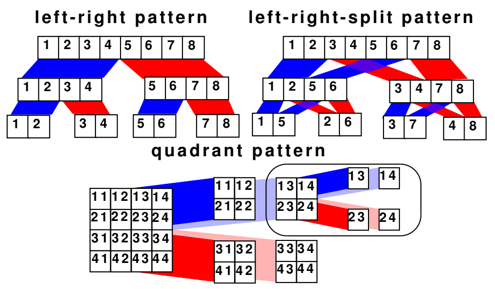

Fig. 3: Patterns of recursively applied partition methods.

```
Algorithm 1 sort(Partition $list$, Partition $dir$)
  $idir \leftarrow$ extract_int($dir$)
  $left \leftarrow$ left_half($list$)
  $right \leftarrow$ right_half($list$)
  sort($left, dir$)
  update_int($dir, idir * -1$)  // change directions for the next half
  sort($right, dir$)
  update_int($dir, idir * -1$)
  merge($left, right, dir$)
  sort2($left, dir$)
  sort2($right, dir$)
```

```
Algorithm 2 merge(Partition $left$, Partition $right$, Partition $dir$)
  $left\_of\_left \leftarrow$ left_half($left$)
  $right\_of\_left \leftarrow$ right_half($left$)
  $left\_of\_right \leftarrow$ left_half($right$)
  $right\_of\_right \leftarrow$ right_half($right$)
  merge($left\_of\_left, left\_of\_right, dir$)
  merge($right\_of\_left, right\_of\_right, dir$)
```

and right subtrees of the tree structure (assigning the root node to one of the halves).

Repeatedly applying the partition library methods to data results in a *partition pattern*. Fig. 3 shows three patterns that break larger data structures down into smaller pieces in a divide-and-conquer fashion. The patterns use the same library methods, but in different combinations. For example, the *left-right* partition pattern shows how data is broken down if `left_half()` and `right_half()` are used to partition data into halves once for each branch, while the *left-right-split* pattern uses `left_half()` and `right_half()` twice per branch.

Example. *Bitonic Sort* is a divide-and-conquer algorithm where a list of elements is sorted by first sorting its two halves in opposite directions, and then merging the two halves together [9], [10]. While having a complexity of $O(n\ log^2\ n)$, which is slightly less efficient than $O(n\ log\ n)$ sorting algorithms like *Merge Sort* or average-case *Quick Sort*, Bitonic Sort is a popular parallel sorting algorithm because the order of its compare-and-swap operations is not data dependent.

Alg. 1 and 2 show a recursive implementation of Bitonic Sort written with the Huckleberry API which consists of three mutually-recursive functions. `Sort2()`, not shown, is the same as `sort()` except that it omits the first two recursive calls. The `dir` partition provides the direction that the list should be sorted in, and is a unit partition. This example demonstrates how data partitioning is expressed statically by the programmer at a high level of abstraction, while the generated parallel code adapts partitions dynamically to runtime parameters (input size, available memory, etc.). Notice that the functions lack exit conditions. Huckleberry inserts function wrappers around recursive functions which manage the exit condition based on the runtime size of the `Partition` data compared

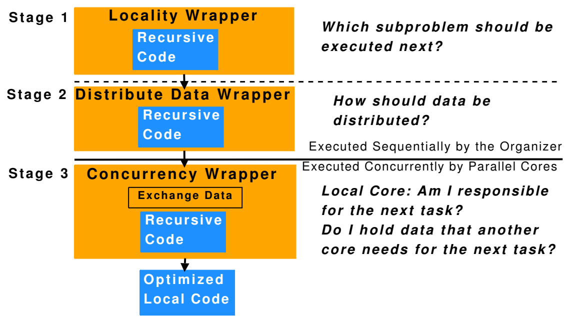
Stage 1
Locality Wrapper
Recursive Code
Which subproblem should be executed next?
Stage 2
Distribute Data Wrapper
Recursive Code
How should data be distributed?
Executed Sequentially by the Organizer
Executed Concurrently by Parallel Cores
Stage 3
Concurrency Wrapper
Exchange Data
Recursive Code
Local Core: Am I responsible for the next task?
Do I hold data that another core needs for the next task?
Optimized Local Code

Fig. 4: Stages in a Huckleberry-generated parallel application.

with the available memory in the underlying architecture. Exit conditions are necessary in the code executed at the leaves of the R-Tree, and are provided by the programmer.

Optimized Local Code. The programmer may provide an optimized implementation to be used once a partition is small enough to fit in a single core's local memory. We call this a *local* code because it is executed when all of the relevant data is in a core's local memory space. Since local code is executed sequentially, it can be optimized using standard sequential coding techniques which may be specifically designed for the hardware in use (e.g. vectorized code for Cell's SPEs). In this work, we do not address how local code is optimized, but we recognize that optimized local code is essential for good overall performance because it will be repeated many times during execution. In the Bitonic Sort example, for each instance of `sort()` that is called on a platform with 16 cores, `local_sort()` is called 1 time, `local_merge()` 10 times, and `local_sort2()` 4 times for each core (and 16, 160 and 64 times on all cores together). Thus, performance improvements in the local code can translate into significant overall improvements. Our Bitonic Sort benchmark experiences more than a 10x speedup when switching between recursive unoptimized code and (non-recursive) optimized local code.

III. Huckleberry Parallel Code Generator

The code generated by Huckleberry creates a flow of execution that passes through three major stages (Fig. 4). All stages are generated by refactoring the original recursive program with wrapper functions that make different scheduling decisions, which are implemented as follows: (1) the user provides a recursive function called `myprog()`; (2) the code generator inserts a wrapper by replacing calls for `myprog()` with `wrapper_myprog()`, including within the body of `myprog()` itself; (3) `wrapper_myprog()` performs bookkeeping steps and then calls `myprog()`. Interleaving calls to `myprog()` and `wrapper_myprog()` in this way has the effect of executing some extra code around each of the instances of `myprog()`. For each of the three stages, there is a wrapper and a separate copy of the original recursive function. The current implementation of Huckleberry assumes that the input recursive functions correctly partition data so that each branch covers a proper subset of the data of its parent. Partition set size is used in the exit conditions of the wrappers.

Locality Wrapper. The *locality* wrapper stage decides what part of the problem should be executed next when the initial input problem size is too large for the aggregate local memory

Algorithm 3 `loc_wrapper_myprog`($data$). In the Bitonic Sort example, $data$ for `loc_wrapper_sort()` includes `list` and `dir`.

```
if $|data| \leq M$ then
    call dd_wrapper_myprog()
else
    loc_myprog($data$)
end if
```

Algorithm 4 `loc_merge`(Partition $left$, Partition $right$, Partition dir)

```
$left\_of\_left \leftarrow$ left_half($left$)
$//(more\ initializations...)$
loc_wrapper_merge($left\_of\_left, left\_of\_right, dir$)
loc_wrapper_merge($right\_of\_left, right\_of\_right, dir$)
```

Algorithm 5 `dd_wrapper_myprog` ($data$, $depth$). Let m be the size of data assigned leaf nodes.

```
if $|data| \leq m$ then
    $i \leftarrow f_{pi}(id\_seq)$
    if $data$ has not been already sent then
        send $data$ to core $i$
    end if
else
    dd_myprog($data$, $depth$)
end if
$id\_seq[depth] \leftarrow id\_seq[depth] + 1$
```

M of the cores. This step is executed only by the OC, and it is executed sequentially to preserve the causal dependencies in the recursive program while maximizing data locality. The steps of the locality wrapper are shown in Alg. 3. Note that the wrapper is application-independent, and will look the same for any program `myprog`. Alg. 4 illustrates how the locality wrapper is wrapped into the `merge()` function from Alg. 2, with `merge()` renamed to `loc_merge()` in order to distinguish it from its counterparts which are called by the concurrency and distribute data wrappers. The locality wrapper checks that the problem size is small enough for M by iterating through a list of the input partition parameters and calculating their size based on the `partition_size()` subroutine from the partition library. When the exit condition is met (i.e. the size *is* small enough), the locality wrapper calls the next stage, the *distribute data* wrapper. The divide-and-conquer `myprog()` function ensures the problem size is reduced with each step.

Distribute Data Wrapper. The *distribute data* wrapper distinguishes between individual cores and their neighbors. The *distribute data* stage starts with a problem that will fit in the aggregate local memory of the N cores, and breaks the problem up into N pieces based on the application's R-Tree. For example, a divide-and-conquer function that divides its input two ways can be represented as a binary tree, whose leaves correspond to instances of the function that reach the exit case. Each node in the tree is uniquely assigned to a specific core that is determined by calling the *parallel-index function* (f_{pi}) on the node's position in the tree. F_{pi} operates on two parameters: depth and sibling order id (e.g. left child 0, right child 1) of a node and its parents in the R-Tree.

The *distribute data* wrapper (shown in Alg. 5) does several things. First, it keeps track of the current sibling id at each level of the tree with an array $id\_seq[]$ and the current depth. $Id\_seq[depth]$ is incremented every time `dd_wrapper_myprog()` is called. `Dd_wrapper_myprog()` includes depth as an input parameter; for example, `dd_wrapper_merge`($left\_of\_left$, $left\_of\_right$, dir) becomes

Algorithm 6 `con_wrapper_myprog` ($data$,$depth$).

```
context switch if necessary
if $|data| \leq m$ then
  $i \leftarrow f_{pi}(id\_seq)$
  if $i$ is $rank$ then
    if mydata_contains($data$) then
      wait for $data$
    end if
    local_myprog($data$) $//not\ to\ be\ confused\ with\ loc\_myprog()$
  else if mydata_intersects($data$) then
    send $data$ to core $i$
  end if
else
  con_myprog($data$, $depth$)
end if
$id\_seq[depth] \leftarrow id\_seq[depth] + 1$
```

`dd_wrapper_merge`($left\_of\_left$, $left\_of\_right$, dir, $depth$ + 1). The minimum possible depth is determined by the size of each core's local available memory. Second, the wrapper applies f_{pi} to identify which core is responsible for the next data. Last, it keeps track of which data it has already sent to the cores. Data may be revisited several times in the R-Tree, but it only needs to be transferred to the chip once.

Concurrency Wrapper. The *concurrency* wrapper is similar to the *data distribute* wrapper because it starts with the same data, uses the same f_{pi} function, and handles $depth$ and $id\_seq[]$ in a similar fashion. However, while the *data distribute* wrapper is executed once on the OC, the *concurrency* wrapper is executed in parallel on each core over the global data set that is shared among the cores. Each core is aware of its own $rank$ in the group. In addition to identifying which core is responsible for each task, the *concurrency* wrapper also organizes synchronization among the cores and data swapping. Data swapping is needed when one core must read or modify data that has already been modified by another core, i.e. there is a data dependency between tasks assigned to different cores. In the Bitonic Sort example shown in Alg. 1 and 2, the `sort()` function calls recursive `merge()` and `sort2()`. Because different recursive functions may use different partition patterns, the divide-and-conquer pattern may be disrupted when switching between different recursive functions as is the case when switching between `sort()` and `merge()`. The *concurrency* wrapper handles context switches by recalculating the correct depth depending on the data size and resetting the values of $id\_seq[]$ to 0 for elements beyond the new depth. The steps of the *concurrency* wrapper are shown in Alg. 6. Data is swapped with a *push* handshake protocol: a core that needs data simply waits to proceed until another core sends data, and the sending core will not send data until it has reached the same node in the R-Tree as the first core. The downside of this protocol is that some concurrency may be lost because the sending core does not send data as soon as it is available.

Example: Traversing the R-Tree. Fig. 5 shows an example of the recursive call tree, or R-Tree, constructed for Bitonic Sort from its three functions `sort()`, `merge()`, and `sort2()`. The OC initially traverses the locality stage tree until it reaches a point where the data size is less than M. Next, the OC continues in the distribute data stage. For example, with $N = 2$ cores and an overall input problem size $|I| = 2M$, the data size in a sub-sort tree is reduced to $|I/2|$ after branching. To distribute data, it is not necessary to traverse the entire tree; it is possible to stop the traversal when all of the data has been distributed once. Once data has been moved into M, the cores exchange data if necessary as they traverse the concurrency stage tree. Execution returns to the locality stage tree once the complete subtree has been executed in the concurrency stage tree. The distribute data and concurrency stage trees use f_{pi} to determine which core is responsible for which data and tasks. In the distribute data stage in Fig. 5, the first leaf task is assigned to core $f_{pi}(depth = 1, sibling - id = 0) = c_0$. A different f_{pi} function might assign the first leaf to another core, for example, if $N > 2$. Note that it is also possible to tune the granularity smaller so that the OC traverses deeper and distributes multiple smaller data partitions to the cores. Since `merge()` uses a different partition pattern than `sort()` and `sort2()`, it is necessary to schedule data exchanges throughout the concurrency tree traversal. Calls to `merge()` also change the depth context since it is invoked on the entire data set.

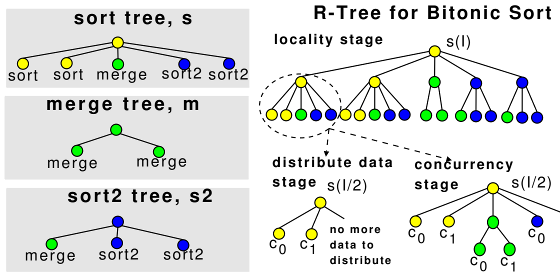

Fig. 5: Traversing the R-Tree.

IV. Experiments

To evaluate our initial implementation of Huckleberry we use the QS20 Cell Blade [8] because of its flexibility and computational power and because it is representative of the class of distributed-memory multi-core platforms. Each QS20 features two Cell BE processors together with 1GB of XDRAM. Originally designed for the PlayStation 3 game console, Cell processors currently make up two thirds of the processors in IBM Roadrunner, the fastest supercomputer in the Top500's list [11]. We use Huckleberry to derive parallel implementations targeting the QS20 for four benchmarks:

1. Smith-Waterman Sequence Alignment is a dynamic programming algorithm which computes a similarity score between two sequences such as DNA sequences [12], [13]. The algorithm involves filling in a matrix m starting from the top-left corner with $m[i, j]$ values that are function of $m[i-1, j]$, $m[i, j-1]$, and $m[i-1, j-1]$. Using Huckleberry we implement it with a combination of the *quadrant* pattern on its 2D data and the *left-right* pattern on its 1D data.

2. Black-Scholes is an algorithm for stock-option pricing. We implemented using the *left-right* pattern to distribute data.

3. One-Dimensional FFT is implemented based on the 'four-step' method [14] with the bit-reversal algorithm [15]. The input array is regarded as a matrix, and the FFT is computed by performing smaller FFTs on the matrix rows and columns. For the FFT we used the *left-right* pattern.

4. Bitonic Sort is implemented as described in Section II.

All these benchmarks are amenable to a divide-and-conquer specification, but they are different in nature and stress our

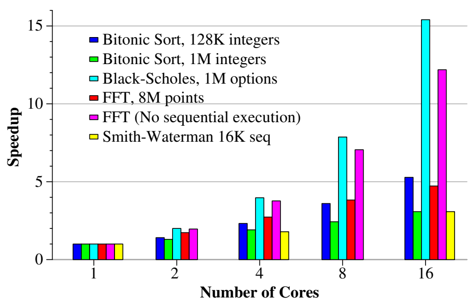

Fig. 6: Scaling cores: speedup when D and m_i are fixed and the number N of available cores scales up.

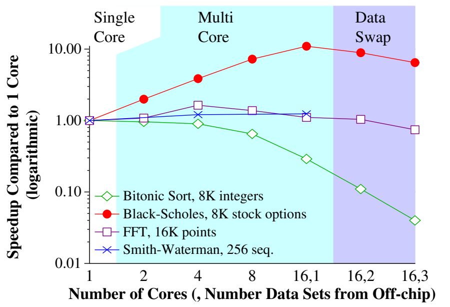

Fig. 7: Scaling task granularity: speedup when I is constant, but m_i is scaled down, forcing more cores to work on the problem.

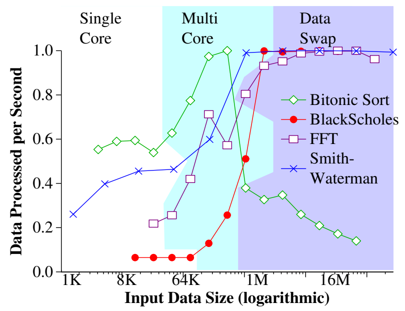

Fig. 8: Scaling data size: m_i remains fixed, while I scales up, normalized w.r.t. the highest throughput instance in that benchmark.

approach in different ways. Notice that we focus on evaluating the overhead and trade-offs of communication rather than on optimizing local single-core code to get the best performance.

Scalability. Fig. 6 shows the performance speedup for large problem sizes as the number of cores is increased. These problem sizes require multiple stages in the *locality wrapper*. The Black-Scholes benchmark performs almost ideally, which is expected, since the benchmark is an example of an "embarrassingly parallel" program. This demonstrates that the overhead of partitioning and distributing the problem in the absence of inter-core communication is very low. The overhead of inter-core data passing and synchronization is more difficult to quantify with respect to alternative implementations; however, using double-buffering to hide the overhead is a potential solution in both cases, though we do not implement it here. Two curves for Bitonic Sort are shown; the Bitonic Sort benchmark achieves slightly more than a 5x speedup with 16 cores for the smaller problem size (128K integers), but the speedup degrades as the problem size increases. We believe that the size of local memory plays a role in the parallel speedup in this case because a hand-coded recursive implementation was able to compute larger problems on a single core than the Huckleberry-generated implementation, and achieved a speedup closer to 7x with larger problem sizes. Reducing local memory usage is one of our future goals. For the Smith-Waterman benchmark, speedup is limited by data dependencies of the algorithm. Namely, imposing the dependencies of high levels of the hierarchy onto lower levels causes some cores to wait for data exchanges longer than is necessary. This behavior may be improved by changing the data swap protocol. The FFT benchmark achieves a speed-up of 5x, though notably, increasing 4 to 8 and 8 to 16 cores does not significantly improve performance; as per Amdahl's law, matrix transpose and multiplication operations are performed sequentially on the OC in our implementation, even though performance is near ideal when the sequential operations are excluded. We expect that algorithmic optimizations and the use of huge page sizes will reduce the time consumed by these operations, as suggested by Chow *et al.* [16].

Problem Granularity. Fig. 7 shows how performance changes as the problem granularity becomes finer. The problem is small enough to fit in the local memory of a single core, but more cores are recruited for their additional computational power. For example, the Black-Scholes curve corresponds to pricing 8K stock option values. With one core, all options are calculated by this core; with four cores, each core calculates $\frac{8K}{4} = 2K$ options. Cases *16,2* and *16,3* correspond to each core calculating 256 and 128 options, respectively.

The curves are highlighted in three groups. In the first, data is small enough to fit on a single core; in the second, data is small enough to fit entirely in the aggregate local memory space; in the third, data is swapped on and off the chip. Breakpoints between groups occur at different places for the benchmarks. For example, Bitonic Sort requires many inter-core data exchanges. During data exchanges, temporary buffers take up some of the local memory space, and limit the size of the input data that can be assigned to a single core.

The benchmarks perform strikingly differently. The Black-Scholes benchmark improves almost linearly as more cores are utilized (note the logarithmic y-axis). However, data swapping eventually becomes a bottleneck as the problem granularity is reduced. The Smith-Waterman benchmark's speedup improves slightly as N is increased to 16, but is relatively flat. Performance of the FFT benchmark first improves and then degrades as granularity is increased, while the Bitonic Sort benchmark performs best when the entire problem is handled on one core, For all benchmarks, the cost of additional off-chip data swapping outweighs the benefits of increased concurrency.

Data Processing Throughput and the Role of Local Memory. Fig. 8 plots the performance of the benchmarks as the input data size scales up, but granularity is fixed. The results in Fig. 8 show why Bitonic Sort performs better with smaller data sizes in Fig. 6. For input sizes that do not require data swapping, the benchmark throughput increases with the input size, but once data is large enough to require data swapping, the throughput drastically decreases. For the other three benchmarks throughput stays steady as the input size increases beyond what will fit on a chip. We infer that since data swapping is not the bottleneck for these benchmarks, a good balance of communication and computation has been achieved.

V. Related Works

Many programming models and tools for the Cell processor, which is a representative distributed memory multi-core platform, have been proposed in recent years [6], [17], [18], [19]. Huckleberry provides a C-based approach to accelerating recursive divide-and-conquer functions. Our vision is that functions compiled in Huckleberry can be combined with stream or SIMD functions compiled with other tools that are more appropriate for those models. Programming distributed memory multi-core platforms has distinct challenges: (1) hardware may not provide cache-coherency, (2) the on-chip communication network is an important system resource which must be used effectively, and (3) the number of cores is likely to scale much higher in future generations, thus requiring new algorithms that avoid single-point bottlenecks. Huckleberry addresses these challenges for the programmer by abstracting parallelism through data partitioning and creating an efficient parallel implementation that leverages the capabilities of on-chip networks to distribute application code and data at runtime in a scalable and transparent way.

Huckleberry follows a number of works in recursive parallel programming. The Sequoia programming language uses hierarchical program design to leverage data locality in the memory hierarchy of parallel system, and also supports the Cell architecture [1], [2]. In Sequoia, different layers of the hierarchical tree are associated with different levels of memory. Concurrent tasks are isolated and do not synchronize, but communicate through their parent task (which may be mapped to the same core). The Sequoia compiler plays a role in optimizing the parallel implementation. The Huckleberry compiler, in contrast, performs no optimizations, but is paired with a distributed application-independent runtime library that has access to the aggregate memory view and partition metadata on local cores. Compilers can parallelize divide-and-conquer programs by analyzing memory references to detect dependencies [20], [21]. Cilk is an expressive general purpose C-based parallel programming language that includes support for recursion [22]. Cilk does not abstract parallelism from the programmer to the same extent that Huckleberry does; the programmer must expose parallelism in applications through the use of thread keywords such as *spawn* and *sync*.

Other works, including NESL and Algorithmic Skeletons which are described below, optimized the recursive model for vector processors of the mid-1990s which presented relatively high inter-node communication costs (throughput around 1 Gbps and application-level latency at 40-100 microseconds). Our work is novel with respect to NESL, a nested parallel programming language [4], because in Huckleberry data passing and inter-core synchronization are determined at runtime via a distributed decision making process which is fully integrated with the distributed tasks. Algorithmic Skeletons capture abstract communication patterns of parallel programs, and are intended to be developed separately from the algorithmic specification of an application by systems and application experts, respectively [23]. The divide-and-conquer skeleton, which supports the parallelization of recursive programs, is implemented with SPMD parallelization based on the *powerlist* data structure [3]. Huckleberry's implementation does not separate the recursive algorithm from the application's communication pattern, but instead models the communication after *partition patterns*.

VI. Conclusions

As multi-core systems of the future scale up to large numbers of cores, there is a need for tools that can abstract away the process of separating a program into parallel tasks. Our goal with Huckleberry is to create such a tool for recursive divide-and-conquer programs. Using Huckleberry, we generate parallel implementations of four different benchmarks for the Cell architecture. In our experiments, the speedup available from parallelization is affected by the interaction of data dependencies and workload requirements with the amount of local memory. Some benchmarks that parallelize effortlessly, like Black-Scholes option pricing, scale well regardless of the size of local memory. Other benchmarks, like Bitonic Sort, clearly perform better with larger local memory.

Acknowledgements

This research is partially supported by DARPA MTO under grant ARL-W911NF-08-1-0127 and the NSF (Award #: 0811012). The authors thank Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the NSF, for access to their Cell Broadband Engine resources.

References

[1] K. Fatahalian *et al.*, "Sequoia: Programming the memory hierarchy," in *Proc. of ACM/IEEE Conf. on Supercomputing*, Nov. 2006.

[2] T. J. Knight *et al.*, "Compilation for explicitly managed memory hierarchies," in *Symposium on Principles and Practice of Parallel Programming*. ACM, Mar. 2007, pp. 226–236.

[3] J. Misra, "Powerlist: A structure for parallel recursion," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1737–1767, Nov. 1994.

[4] G. E. Blelloch, "Programming parallel algorithms," *Communications of the ACM*, vol. 39, no. 3, pp. 85–97, Mar. 1996.

[5] M. Gschwind *et al.*, "Synergistic processing in Cell's multicore architecture," *IEEE Micro*, vol. 26, no. 2, pp. 10–24, 2006.

[6] J. Kahle *et al.*, "Introduction to the CELL multiprocessor," *IBM J. Res. Develop.*, vol. 49, no. 4-5, pp. 589–604, Sep. 2005.

[7] M. Kistler, M. Perrone, and F. Petrini, "Cell multiprocessor communication network: Built for speed," *IEEE Micro*, vol. 26, no. 3, pp. 10–23, 2006.

[8] A. K. Nanda *et al.*, "Cell/B.E. Blades: Building blocks for scalable, real-time, interactive, and digital media servers," *IBM J. Res. Dev.*, vol. 51, no. 5, pp. 573–582, 2007.

[9] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill, 2001.

[10] B.Gedik, R.Bordawekar, and P.Yu, "CellSort: High performance sorting on the Cell processor," in *Very Large Data Bases Conf.*, Sep. 2007.

[11] K. J. Barker *et al.*, "Entering the petaflop era: the architecture and performance of Roadrunner," in *Proc. of ACM/IEEE Conf. on Supercomputing*, Nov. 2008, pp. 1–11.

[12] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J. Mol. Biol.*, vol. 147, no. 1, pp. 195–197, Mar. 1981.

[13] O. Gotoh, "An improved algorithm for matching biological sequences," *J. Mol. Biol.*, vol. 162, no. 3, pp. 705–708, Dec. 1982.

[14] D. H. Bailey, "FFTs in external of hierarchical memory," in *Proc. of ACM/IEEE Conf. on Supercomputing*, Nov. 1989, pp. 234–242.

[15] D. Jones, "Decimation-in-time (DIT) Radix-2 FFT," Sep. 2006, connexions Web site http://cnx.org/content/m12016/1.7/.

[16] A. Chow, G. Fossum, and D. Brokenshire, "A programming example: Large FFT on the Cell Broadband Engine," IBM, Tech. Rep., May 2005.

[17] M. Ohara *et al.*, "MPI microtask for programming the Cell Broadband Engine processor," *IBM Syst. J.*, vol. 45, no. 1, pp. 85–102, Jan. 2006.

[18] M. D. McCool, "Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform," in *GSPx Multicore Applications Conf.*, Oct. 2006.

[19] Gedae, http://www.gedae.com/.

[20] M. Gupta *et al.*, "Automatic parallelization of recursive procedures," *Intl. J. of Parallel Programming*, vol. 28, no. 6, pp. 537–562, 2000.

[21] R. Rugina and M. Rinard, "Automatic parallelization of divide and conquer algorithms," in *Symposium on Principles and Practice of Parallel Programming*, May 1999, pp. 72–83.

[22] R.Blumofe *et al.*, "Cilk: An efficient multithreaded runtime system," *J. of Parallel and Distrib. Comput.*, vol. 37, no. 1, pp. 55–69, Aug. 1996.

[23] S. Gorlatch, "Programming with divide-and-conquer skeletons: A case study of FFT," *J. Supercomput.*, vol. 12, no. 1-2, pp. 85–97, Jan./Feb. 1998.