Data Parallel Programming for Irregular Tree Computations

Leo A. Meyerovich
Department of Computer Science
University of California, Berkeley

Todd Mytkowicz, Wolfram Schulte
Research in Software Engineering (RiSE)
Microsoft Research, Redmond

Abstract

The adoption of data parallel primitives to increase multicore utilization presents an opportunity for aggressive compiler optimization. We examine computations over the tree abstract datatype (ADT) in particular.

For better utilization than approaches like flattening, we argue that transformations should specialize for common data and computation regularities. For example, we demonstrate a novel pattern that exploits regularity in node labeling as a SIMD parallelism opportunity, which we call SIMTask parallelism. For better applicability, we argue for better linguistic support of irregularity. For example, we show how the future primitive might be used to tolerate occasional data dependencies in an otherwise associative computation.

We validate our approach on two tree computations: RepMin, popular in the functional programming community, and intrinsic widths, a stage of webpage layout in a prototype web browser. We show speedups over traditional versions of these computations of **124X** and **10X**, respectively.

1 Introduction

Many programs run well below peak performance on modern hardware: there is a utilization gap. [13] We focus on the common problem of computing over the tree abstract data type (ADT). A high utilization program might exploit data prefetching, wide cache lines, and SIMD instructions. It is difficult to combine such architectural and algorithmic knowledge with domain expertise so many tree programs have poor utilization.

In this paper, we introduce a vision for high utilization tree computations through compiler-assisted specializations. We specialize flattening, the type-directed approach of NESL and Data Parallel Haskell [4, 14, 17] that traditionally *uniformly* transforms a tree into arrays and the computation into SIMD (or multicore) instructions. Our approach is to pick a flattening strategy based on regularity in the tree datatype and the computation over it. For example, we demonstrate how to exploit whether a tree is balanced and the computation over the tree is associative. Finally, we describe how occasional irregularities should not preclude optimizations that assume regularity in the typical case; we describe futures [1] as one enabling primitive.

To validate our approach, we present two case studies in SIMD tree algorithms: a functional programming benchmark and a complicated web browser's CSS layout engine. For both studies, we discuss how to use declarative data parallel constructs in order to automate our currently manual program transformations.

Our conclusion is that with simple declarative constructs from the programmer, irregular data structures like the tree ADT, that do not interact well with features found in modern hardware, can be automatically transformed into representations that *do* interact well with features found in modern hardware.

In summary, we contribute a vision for specialized data parallel constructs that better utilize hardware and still tolerate irregularity. To do so, we also contribute the first report of implementing a SIMD layout engine and a novel SIMD tree pattern (*SIMTask* visitors).

2 An Example: RepMin

We use a simple program, RepMin, throughout this paper to discuss our approach. The RepMin problem is to replace all values held in the leaves of a tree by the minimum value of all the leaves.

RepMin is a two pass algorithm. The first pass is bottom-up: each leaf passes its value to its parent, which then compares the min from its left and right child, respectively. This is repeated until the root, which has no parent, holds the minimum value of the tree. The second pass is top-down: the minimum value is passed from the

root down to the leaves, at which point the leaf sets its value to the min.

2.1 Cilk Strawman

Cilk [10] is a popular multicore framework. The spawn keyword can be used to guide the point of recursive decomposition; spawn guides multicore utilization. The following code is a Cilk implementation of RepMin:

```
class Node {
  enum Type {Pair, Leaf};
  Node * left, * right;
  int value; Type type;
  int FindMin() {
    if (type == Leaf) return value;
    int a = spawn left->FindMin();
    int b = right->FindMin();
    return min(a, b);
  }
  void PropagateMin(int min) {
    if (type == Leaf) {
      value = min; return; }
    spawn left->PropagateMin(min);
    right->PropagateMin(min);
  }
  void Repmin() {
    int min = FindMin();
    PropagateMin(min);
  }
}
```

Unfortunately, the Cilk version of RepMin on a 4 core Intel workstation with the Intel XE 2011 compiler is 2.4x *slower* than the sequential code.

2.2 Cilk Deconstructed

In the Cilk model, significant irregularity in the task tree is supported through work stealing, and, to prevent interference (e.g., false sharing), a multicore memory allocator. [18] For better utilization, the Cilk programmer might manually optimize data layout (e.g., squeeze two tree nodes into a single cache line) and control patterns (e.g., manage the number of calls to the spawn keyword). Overall, Cilk uses the decomposition hint to address the multicore utilization problem *and little else*.

In contrast, we advocate using data and control declarations to automatically optimize layout and access patterns for caches, SIMD instructions, etc. as well.

3 Compiler Target: Tree Optimizations

We demonstrate the 2 orders of magnitude single-core speedup for RepMin one can achieve *if* the toolchain were able to (i) change the data layout of a tree and (ii) change the order of tree traversal. Such high-utilization optimizations represent what we believe multicore frameworks should support; Section 4 examines how we believe we can balance productivity with tractability to do so.

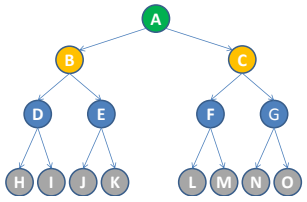

Figure 1: A balanced tree colored by level.

3.1 Optimizing Data Layout

Commodity hardware optimizes regular memory access patterns better than randomly accessed pointers. Thus, instead of representing a tree as a randomly-access pointer-based structure, we use a *flat* representation of indexed arrays [4]. The regularity of the tree can be further exploited in picking which flat representation: we compare an encoding that supports an irregular tree against one specialized for balanced trees.

We stress that our interest is not in the (likely lack of novelty) in these representations. The importance is the demonstrably strong performance advantage of increasingly regular encodings (Section 3.3), and, later, how programmers might automatically use them (Section 4).

Fixed Encoding: With this encoding, we know that the tree is balanced and has a fixed branching (i.e. 2 nodes per child). We use a simple breadth-first encoding: if i is the index of the current node, the left child is at $2i+1$ and the right child is at $2i+2$. For example, here is the encoding for the first 3 levels of the tree in **Figure 1**:

index	0	1	2	3	4	5	6
node	A	B	C	D	E	F	G

This encoding is desirable because (i) it lays each level out contiguously in memory which aids SIMD execution over the array and (ii) if the leaves are the only place where values are stored, we have to store and often only iterate over the last "level" of the array.

Variable Encoding: Not all trees have a fixed branching, so we have a variable encoding that allows for arbitrary children at each node. This encoding contains two arrays, one that contains the *length*—or number of children—for that node, and another that holds the value of the node. The following encoding is for the first 3 levels of the tree in Figure 1:

index	0	1	2	3	4	5	6
length	0	0	2	0	0	2	2
node	D	E	B	F	G	C	A

The *node* array assigns indexes into the array via a postorder traversal of the tree. For example, the first node in a postorder traversal of the tree is node *D*: *D* has index 0 into our arrays. The *length* of *D* is 0—*D* has no children. Conversely, node *A* has two children (*B* and *C*, respectively). *C* is *A*'s index - 1, while *B* is the index of *C*'s leftmost subchild - 1.

This encoding is desirable because it allows us to deal with irregular tree organization while at the same time gives us an efficient representation on which to do a bottom-up traversal (from left to right over the *length* array) or a top-down traversal (right to left).

3.2 Optimizing Traversal Order

Like data layout, traversal order optimizations benefit from regularity in trees. In particular, if we can change the traversal order over the tree, we might exploit SIMD parallelism. We examine vectorization of large functions in Section 5; as a basis, we first consider two ways of just using intrinsic SIMD MAX instructions.

First, we exploit SIMD reduction instructions for the bottom-up FindMin pass. The minimum value in node D can be expanded and then vectorized as one instruction:

```
D = MIN_REDUCTION([H, I, MAXI, MAXI])
```

where MAXI is the maximum integer for that platform. Keller et al [14] automate this transformation.

If the tree is sparse, as in our ongoing example, the encoding underutilizes vector lanes (i.e., the MAXI padding). If MAXI is segmented, to fill the lanes, we can exploit the associativity and commutativity of the minimum operation to combine reductions across a level. For example, we would instead assign:

```
DE = MIN_REDUCTION([H, I, J, K])
```

Second, we can exploit more efficient SIMD piecewise-parallel instructions in a novel way. Observe that the task-parallel decomposition has identical instruction traces across tasks. Instead of computing the values of nodes D, E, F and G on four different cores simultaneously, we can compute their values on four different lanes simultaneously, doubling our theoretical utilization:

```
DEFG = MIN_PCWISE([H, J, L, N], [I, K, M, O])
```

We describe this novel parallelization strategy as an instance of Same Instruction, Multiple Task parallelism (*SIMTask*). We use the term SIMTask as the relationship of SIMTasks to SIMThreads [16] is analogous to that of tasks to threads. Unlike static vectorization (e.g., implicit vectorization or NESL's flattening), it highlights more dynamic scheduling and allocation approaches (e.g., semi-static in Section 5).

3.3 Evaluation

We obtained significant speedups for RepMin by optimizing data layout and traversal order (we examine the more complicated CSS example in Section 5). Our baseline is the *sequential* C++ code shown Section 2.1 (so without spawn). We built 30 trees each with a height of 23 levels. We set the values at the leaves to random values.

The following table illustrates two cases: the speedup due to flattening (fixed and variable encodings) and the speedup of using SIMD operations on the flattened tree.

Speedup	Sequential	SIMD
Fixed	52x	124x
Variable	43x	112x

We see large speedups over the sequential code using flattening. This is due to the hardware being able to extract regularity from the tree traversal over an array, as opposed to a traversal with child pointers (better prefetching, branch prediction, etc.). We see about a 124x speedup by using SIMD registers to calculate the *min* operation 4 values at a time. Our SIMD code does not achieve the idealized 4x speedup over the flattened code due to branching effects and other irregularities we have to contend with in the SSE 4.2 SIMD operations.

4 Programming Models

In the previous section, we demonstrated *backend* optimizations on data layout and traversal order we would like to automate. In this section, we discuss frontend programming models that expose enough information such that a compiler do so. Throughout the following examples, we see a pattern of exposing exploitable regularity through declarative constructs and supporting productivity by automatically handling irregularities.

4.1 Data Parallel Tree Map and Reduce

Dean et al.'s MapReduce [8] popular data parallel model highlights the adoptibility of map and fold functions. Applying the generalization noted by Fokkinga et al. [9] generalizes such functions to catamorphisms, we now explore map and fold functions over trees.

We consider tree reductions first. Tree reduce takes two arguments: an operator to apply to the values from left and right child nodes, and a default value which is used as the start of the reduction. For example, the first, bottom-up pass in RepMin could be written in C# as so:

```
int min = tree.Reduce(Math.Min, int.MaxValue);
```

A toolchain should know Math.Min is associative and commutative (e.g. the programmer annotates it as such), so the toolchain can infer that it can traverse the tree in

any order. Because the programmer did not specify the physical representation of the tree, the runtime system can flatten it tree into an array and put the nodes in a level of tree next to each other, allowing SIMD parallelism.

The second pass in RepMin is a top-down pass and could be written using a destructive version of Map:

```
tree.Map(Leaf leaf => { leaf.SetValue(min); });
```

Again, the programmer does not specify the physical representation nor the traversal order. If the runtime system can infer that the delegate is associative and commutative (e.g., through another programmer-provided annotations), this too can be a SIMD operation.

In summary, MapReduce-style programmers are capable of describing computation regularity (e.g., associativity) relevant to SIMD utilization while the runtime can automatically handle data irregularity (e.g., tree layout).

4.2 SIMTask Visitors

In Section 3.2, we saw a piecewise-SIMD computation of the minimum values of RepMin using a task-parallel (*SIMTask*) decomposition. Unlike typical task parallelism, we require the instructions of any tasks that are to be run in SIMD fashion together to have mostly convergent instruction sequences. I.e., be the same type of task. We now demonstrate basic programming constructs to help scale SIMTasks from a simple RepMin-style reduction to larger and more irregular programs typically thought of being, at best, task parallel.

First, we consider supporting instruction divergence in a slightly less regular computation. The visitor pattern is common in programs like compilers, and, as we will see, in web browsers (Section 5). Instead of performing the same task for every node, there are different types of nodes and a different task for every type of node. Below, we rewrite RepMin as a visitor with a fixed bottom-up traversal pattern specialized to binary trees that communicate integers between levels:

```
class FindMin : ParallelBottomUp<int> {
  public override int visit(Node n) {
    return Math.Min(resLeft, resRight);
  }
  public override int visit(Leaf n) {
      return n.Value;
...
```

To handle the instruction divergence of different tasks, visit(Node n) SIMTasks should run together with other visit(Node n) SIMTasks, and the same for visit(Leaf n). Just as a developer relies upon type dispatch to call the appropriate FindMin function on a node, so should she rely upon a SIMTask Visitor framework to batch SIMTasks together and run them using SIMD instructions.

SIMTasks demonstrate another case of exposing and exploiting data and computation regularity.

4.3 SIMTask Futures

Finally, we consider occasional (computational) irregularities in the task dependency graph. We might change Node.FindMin() to very rarely inspect the value of a sibling, not just its children:

```
return random() == 100 ?
  parent.right.value :
  Math.Min(resLeft, resRight);
```

Relying strictly upon an upwards traversal is incorrect: the right value may not be available at time of evaluation. To efficiently but cleanly support such irregularity, programmers use futures [1] to reference a value before it is available. Evaluation speculatively occurs in an upwards fashion, as is most efficient and as suggested by the programmer. However, if the value is not available at the time of reference, the runtime will compute or at least wait for the required value and only then continue. Our interest in such a problem is motivated by our examination of SIMD parallelism in a browser (Section 5).

Futures for SIMTasks represent exposing and automatically handling computation irregularity.

5 A Real Example: CSS

In order to demonstrate our approach is applicable to real-world problems more complicated than RepMin, we applied data parallel programming to the CSS layout engine of an HTML5 platform.

The layout engine employs a visitor pattern over the input tree – a webpage. Due to the low-level nature of modern SIMD tools, we considered one pass that calculates the widths of elements on the page. We manually rewrote each visit function for that pass as a symbolic (SIMD-unfriendly) operation followed by a SIMTask (SIMD-friendly) operation. After parsing, we flatten the tree into an array-backed structure, using a variant of the variable indexing scheme discussed in Section 3.1 for prefix trees. We then use the Intel® vectorizing compiler to extract SIMD parallelism from running a loop over simultaneous SIMTask visits.

We investigate the speedup from (i) flattening the pointer-based tree into an array and (ii) applying SIMD operations to the flattened representation. For all benchmark results, we use a baseline that has a pointer-based tree with manually specified visitors. All experiments are of an Intel Core i7 (620M, MacBook Pro) on Craigslist, MSDN, Wikipedia, and Twitter content pages. Speedup results are for calls that run multiple nodes together.

For tree nodes matching nearby node, we found that flattening the tree provided a 3.8x speedup over the explicit pointer based representation. Note that this speedup also includes memoization of calls to the symbolic visit operations by exploiting the associativity of

visits and peculiarities of the operation. When we used the vectorizing compiler, we saw a 4.9x speedup over the normal pointer-based formulation.

We should note two further outcomes: first, we did see a slowdown when we could only used a single lane of the SIMD registers (e.g., the above 4.9x speedup is when we were able to use two lanes of the SIMD registers). Second, less than half of the SIMTask code was vectorizable via the Intel compiler due to data dependencies it was not able analyze. These issues require further investigation.

Our experience reveals that despite our significant speedup numbers, there are many challenges to getting data parallel models for large irregular computations to exploit modern hardware. At the same time, it shows that they are worth investigating.

6 Related Work

We build upon and attempt to integrate a variety of work:

Manual optimization: Indicative of the challenge of manual approaches is an early paper manually vectorizing a Barnes-Hut n-body simulation [2], titled "Don't laugh, it runs!". As seen with FAST [15], tree computations are still a challenge. There are a variety of individual algorithms, e.g., jump pointers [19]. We want to automate such optimizations.

Parallelism through Control Abstraction. Skeletons are frameworks parametrized by user-supplied functions. Skeletons typically ignore knowledge about the functions, wasting algorithmic opportunities. Stencils, in contrast, are applied at design time when a designer believes the regularity of the problem matches the stencil. Stencils do not tolerate irregularity. We want to match the generic nature of skeletons with the performance of stencils. Task parallel systems such as Cilk [10] and MapReduce [8] tolerate irregularity, but limit the scope of their optimizations: we want the best of both worlds. Gibbon [12] demonstrates linguistic progress in abstracting over both data and computation, which might be applied to parallel patterns.

Datatype-Directed Parallelism. Blelloch [4]'s NESL language demonstrates one transformation for lifting arbitrary computations over bounded nested vectors (e.g., matrices) to use vector instructions. As extensions, Keller et al. [14, 17] support recursive types (e.g., trees) and target multiple cores and GPUs [3, 7]. Catanzaro et al. [5] presents an alternate transformation in Copperhead of computations over various forms of bounded nested vectors that targets GPUs: the transformation should specialize for data structure and hardware. We further advocate specializing for computation structure.

Optimizing DSLs. Systems like Matlab® specialize for particular data structures and computations over them (e.g., sparse matrices). In contrast, Data Parallel Haskell [17] employs a generic flattening transformation to uniformly vectorize a variety of abstract data types. We desire a mixture: declarative constructs in the style of DPH, but, instead of just one transformation, support many (e.g., our algorithms for sparse trees).

SIMThreads and GPU DSLs. The relationship of SIMTasks to SIMThreads [16] is similar to that of tasks to threads. CUDA and the related OpenCL language that target GPUs are still best thought of as SIMThread languages with optimized parallel looping constructs. Ct and Intel's ArrBB [18] are making noteworthy progress, slowly raising the abstraction level to approach SIMTasks.

Managed language optimization. Gal et al's trace-based just-in-time compilation (TJIT) [11] not only eliminate useless instructions but might also expose regularity useful for hardware. Fox et al's approach of selective just-in-time specialization (SEIJITS) [6], which has been used to compile stencils at runtime, is a strong first step.

7 Conclusions

The computing industry is facing a utilization gap for which we believe the adoption of data parallel multicore frameworks presents an opportunity. By specializing for regularities in data structures, data parallel languages can support high utilization. For example, we show that flattening a tree can speedup RepMin by 40 to 50x depending on regularity and SIMD vectorization achieves a total 124x speedup.

Scaling to larger software is difficult; a modern vectorizing compiler supports about half our rewritten layout engine visitors. However, we do see potential: flattening achieves up to a 3.8x speedup and we achieve up to a 4.9x total speedup (2.3x average SIMD speedup). To increase applicability, we propose using constructs such as futures to tolerate occasional irregularities.

Overall, our experience provides motivation for our vision: a declarative approach to programming irregular computations allows (i) a programmer to focus on solving her problem and (ii) significant opportunities for a toolchain to extract regularity from her code and exploit it.

8 Acknowledgments

A thank you to Herman Venter, Rastislav Bodik, Dan Grossman, Nikolai Tillmann, Bryan Catanzaro, Krste Asanavic, and Ariel Rabkin, as well as reviewers, for guidance in this work. Research supported by Microsoft and Intel funding (Award # 20080469). This material is based upon work supported under a National Science Foundation Graduate Research Fellowship.

References

[1] BAKER, JR., H. C., AND HEWITT, C. The incremental garbage collection of processes. *SIGPLAN Not. 12* (August 1977), 55–59.

[2] BARNES, J. E. A modified tree code: don't laugh; it runs. *J. Comput. Phys. 87* (March 1990), 161–170.

[3] BLELLOCH, G. E. Scans as primitive parallel operations. *IEEE Trans. Comput. 38* (November 1989), 1526–1538.

[4] BLELLOCH, G. E., CHATTERJEE, S., HARDWICK, J. C., SIPELSTEIN, J., AND ZAGHA, M. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing 21*, 1 (Apr. 1994), 4–14.

[5] CATANZARO, B., GARLAND, M., AND KEUTZER, K. Copperhead: Compiling an embedded data parallel language. In *Principles and Practices of Parallel Programming (PPoPP)* (2011), pp. 47–56.

[6] CATANZARO, B., KAMIL, S. A., LEE, Y., ASANOVIÄĞ, K., DEMMEL, J., KEUTZER, K., SHALF, J., YELICK, K. A., AND FOX, A. Sejits: Getting productivity and performance with selective embedded jit specialization. Tech. Rep. UCB/EECS-2010-23, EECS Department, University of California, Berkeley, Mar 2010.

[7] CHAKRAVARTY, M. M., KELLER, G., LEE, S., MCDONELL, T. L., AND GROVER, V. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming* (New York, NY, USA, 2011), DAMP '11, ACM, pp. 3–14.

[8] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM 51* (January 2008), 107–113.

[9] FOKKINGA, M., JEURING, J., MEERTENS, L., AND MEIJER, E. A translation from attribute grammars to catamorphisms, 1994.

[10] FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada, June 1998), pp. 212–223. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[11] GAL, A., EICH, B., SHAVER, M., ANDERSON, D., MANDELIN, D., HAGHIGHAT, M. R., KAPLAN, B., HOARE, G., ZBARSKY, B., ORENDORFF, J., RUDERMAN, J., SMITH, E. W., REITMAIER, R., BEBENITA, M., CHANG, M., AND FRANZ, M. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2009), PLDI '09, ACM, pp. 465–478.

[12] GIBBONS, J. Design patterns as higher-order datatype-generic programs. In *Workshop on Generic Programming* (Sept. 2006), R. Hinze, Ed.

[13] HAUSWIRTH, M., SWEENEY, P. F., DIWAN, A., AND HIND, M. Vertical profiling: understanding the behavior of object-priented applications. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2004), OOPSLA '04, ACM, pp. 251–269.

[14] KELLER, G., AND CHAKRAVARTY, M. M. T. Flattening trees. In *Euro-Par* (1998), pp. 709–719.

[15] KIM, C., CHHUGANI, J., SATISH, N., SEDLAR, E., NGUYEN, A. D., KALDEWEY, T., LEE, V. W., BRANDT, S. A., AND DUBEY, P. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the 2010 international conference on Management of data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 339–350.

[16] LINDHOLM, E., NICKOLLS, J., OBERMAN, S., AND MONTRYM, J. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro 28* (March 2008), 39–55.

[17] PEYTON JONES, S. Harnessing the multicores: Nested data parallelism in haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems* (Berlin, Heidelberg, 2008), APLAS '08, Springer-Verlag, pp. 138–138.

[18] PHEATT, C. Intel threading building blocks. *J. Comput. Small Coll. 23* (April 2008), 298–298.

[19] ROTH, A., AND SOHI, G. S. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th annual international symposium on Computer architecture* (Washington, DC, USA, 1999), ISCA '99, IEEE Computer Society, pp. 111–121.