Exploiting Vector and Multicore Parallelism for Recursive, Data- and Task-Parallel Programs

Bin Ren
College of William and Mary
bren@wm.edu

Sriram Krishnamoorthy
Pacific Northwest National Laboratories
sriram@pnnl.gov

Kunal Agrawal
Washington University at St. Louis
kunal@wustl.edu

Milind Kulkarni
Purdue University
milind@purdue.edu

Abstract

Modern hardware contains parallel execution resources that are well-suited for data-parallelism—vector units—and task parallelism—multicores. However, most work on parallel scheduling focuses on one type of hardware or the other. In this work, we present a scheduling framework that allows for a unified treatment of task- and data-parallelism. Our key insight is an abstraction, *task blocks*, that uniformly handles data-parallel iterations and task-parallel tasks, allowing them to be scheduled on vector units or executed independently as multicores. Our framework allows us to define schedulers that can dynamically select between executing task-blocks on vector units or multicores. We show that these schedulers are asymptotically optimal, and deliver the maximum amount of parallelism available in computation trees. To evaluate our schedulers, we develop program transformations that can convert mixed data- and task-parallel programs into task block–based programs. Using a prototype instantiation of our scheduling framework, we show that, on an 8-core system, we can simultaneously exploit vector and multicore parallelism to achieve 14×–108× speedup over sequential baselines.

Keywords Task Parallelism; Data Parallelism; General Scheduler

PPoPP '17 Feb. 4–8, 2017, Austin, Texas, USA.

DOI: http://dx.doi.org/10.1145/3018743.3018763

1. Introduction

The two most common types of parallelism are *data parallelism*, arising from simultaneously executing the same operation(s) over different pieces of data, and *task parallelism*, arising from executing multiple independent tasks simultaneously. Data parallelism is usually expressed through parallel loops, while task parallelism is often expressed using fork-join parallelism constructs.

These types of parallelism are each well-matched to different types of parallel hardware: data parallelism is well-suited for vector hardware such as SIMD units. The paradigm of executing the same computation across multiple pieces of data is exactly what SIMD units were designed for. While multicore hardware is more flexible (and many programming models allow mapping data-parallel execution to multiple cores), the independent execution context provided by multicores are well-matched to the task parallelism in languages such as Cilk [4].

Increasingly, however, both kinds of parallel hardware coexist. Modern machines consist of multicore chips, with *each* core containing dedicated vector hardware. As a result, it is important to be able to write programs that target both types of hardware in a unified manner.

Unfortunately, when programming data- or task-parallelism, the two paradigms are often treated separately. While programming models such as OpenMP [11] support both data- and task-parallelism, the coexistence is uneasy, requiring very different constructs, used in very different contexts. Indeed, Cilk "supports" data parallelism through constructs such as `cilk_for`, which, under the hood, translate data parallelism to task parallelism [6]. In recent work, Ren et al. developed transformations that work in the opposite direction [14]: given a task parallel program, they extract *data* parallelism for execution on vector units. None of these ap-

proaches cleanly support both multicore and vector execution *simultaneously*, especially for programs that mix data- and task-parallelism.

In this paper, we develop a system that unifies the treatment of both types of parallelism and both types of parallel hardware: we develop general schedulers that let programs mix data- and task-parallelism and run efficiently on hardware with both multicores and vector units. Importantly, our schedulers allow exploiting data- and task-parallelism at all levels of granularity: task parallel work can be decomposed into data-parallel work for execution on vector units, and data-parallel work can be forked into separate tasks for load balancing purposes through work-stealing. This grain-free treatment of parallelism allows our system to effectively and dynamically map complex programs to complex hardware, rather than "baking in" a particular execution strategy.

The key insight behind our work is a unifying abstraction for parallel work, the *task block*. Task blocks contain multiple independent tasks that can arise from both data parallel loops and from task parallel fork-join constructs. We develop a general scheduling framework that allows task blocks to be used as a unit of scheduling for both data parallelism (essentially, by executing a data-parallel for loop over the task block) or task parallelism (by allowing task blocks to be stolen and executed independently). Importantly, each of these execution modes produce new task blocks that make independent scheduling decisions, allowing for the free mixing of data- and task-parallel execution.

To evaluate our schedulers, we present a simple language that allows programmers to express task parallelism nested within data parallelism. We then extend Ren et al.'s transformations [14] to generate task block–based programs from algorithms written in our language. Programs written in our specification language (or in general-purpose languages restricted to use analogous constructs) and run using our scheduling framework can simultaneously exploit data- and task-parallelism (rather than one or the other) on both vector and multicore hardware.

In this paper, we make several contributions:

1. We develop a general scheduling framework based on task blocks for SIMD and multicore execution of computation trees. Using this scheduling framework, we make three advances: (i) we show that Ren et al.'s approach to executing task-parallel programs on vector hardware is a special case of our framework; (ii) we use our framework to develop a new class of scheduling policies, *restart* that expose more data parallelism; (iii) we provide mechanisms to support Cilk-style work-stealing between cores, unifying execution on data-parallel and task-parallel hardware.
2. We provide theoretical bounds on the parallelism exposed by our various scheduling policies. We show that our new scheduling policies can expose more parallelism than Ren et al.'s original policies, and are *asymptotically optimal* (though in practice, scheduling overheads may dominate). This represents the first theoretical analysis of vector execution of mixed task- and data-parallel programs.
3. We evaluate on eleven benchmarks, including all the benchmarks from Ren et al., as well as new benchmarks that exhibit more general structures (e.g., nesting task- and data-parallelism).

We implement a proof-of-concept instantiation of our scheduling framework. Overall, on an 8-core machine, our system is able to deliver combined SIMD/multicore speedups of 14×–108× over sequential baselines.

2. Background

We now provide some background on previous work on vectorizing task parallel programs and how task parallel programs are scheduled using work stealing.

2.1 Vectorizing Task Parallel Programs

Vectorization of programs has typically focused on vectorizing simple, dense *data parallel* loops [10]. It is not immediately apparent that SIMD, which is naturally data parallel, can be applied to recursive task-parallel programs. Such programs can be thought of as a computation tree, where a task's children are the tasks that it spawned and the leaves are the base-case tasks.[1] Any effective vectorization of these programs must satisfy two constraints:

1. *All of the tasks being executed in a vectorized manner must be (substantially) identical.* This criterion is naturally satisfied because the tasks arise from the same recursive code.
2. *Data accesses in the tasks must be well structured.* Vector load and store operations access contiguous regions of memory. If a SIMD instruction operates on data that is not contiguous in memory, it must be implemented with slow scatter and gather operations.

To understand why this second restriction is problematic, consider trying to vectorize the standard Cilk-style execution model, with each SIMD lane representing a different worker. Because the workers execute independent computation subtrees, their program stacks grow and shrink at different rates, leading to non-contiguous accesses to local variables, and consequently inefficient vectorization.

SIMD-friendly execution: Ren et al. developed an execution strategy that satisfies both constraints for efficient vectorization [14]. The key insight of their strategy was to perform a more disciplined execution of the computation, with the restriction that all tasks executed in a SIMD operation be at the same recursion depth—this allows the stacks of these tasks to be aligned and interleaved so that local variables can be accessed using vector loads and stores.

[1] This formulation does not work once we have computations with syncs. However, those programs can also be represented using a tree; albeit a more complex and dynamic one.

```
1  void foo(int x)
2    if (isBase(x))
3      baseCase()
4    else
5      l1 = inductiveWork1(x)
6      spawn foo(l1)
7      l2 = inductiveWork2(x)
8      spawn foo(l2)
```

(a) Simple recursive code. `spawn` creates new tasks.

```
1  void bfs_foo(TaskBlock tb)
2    TaskBlock next
3    foreach (Task t : tb)
4      if (isBase(t.x))
5        baseCase()
6      else
7        l1 = inductiveWork1(t.x)
8        next.add(new Task(l1))
9        l2 = inductiveWork2(t.x)
10       next.add(new Task(l2))
11   bfs_foo(next)
```

(b) Blocked Breadth-first version of code in Figure 1(a).

```
1  void blocked_foo(TaskBlock tb)
2    TaskBlock left, right
3    foreach (Task t : tb)
4      if (isBase(t.x))
5        baseCase()
6      else
7        l1 = inductiveWork1(t.x)
8        left.add(new Task(l1))
9        l2 = inductiveWork2(t.x)
10       right.add(new Task(l2))
11   blocked_foo(left)
12   blocked_foo(right)
```

(c) Blocked depth-first execution.

Figure 1: Pseudocode demonstrating vectorization transformations.

Ren et al.'s approach targets programs written using a Cilk-like language (or programs in other languages that adhere to similar restrictions). The programs they target consist of a single, k-ary recursive method:

$$f(p_1, \ldots, p_k) \textbf{ if } e_b \textbf{ then } s_b \textbf{ else } s_i$$

This method evaluates a boolean expression to decide whether to execute a base case (s_b, which can perform *reductions* to compute the eventual program result) or to execute an inductive case (s_i), which can *spawn* additional tasks by invoking the recursive method again. As in Cilk, these spawned methods are assumed to be independent of the remainder of the work in the spawning method, and hence can be executed in parallel. Ren et al.'s transformations target single recursive methods; Section 3 describes how we extend the language to handle data-parallel programs with task-parallel iterations.

Figure 1(a) shows a simple recursive program that we use to explain Ren et al.'s execution strategy[2]. The steps of execution are as follows:

1. As shown in Figure 1(b), the computation is initially executed in a breadth-first, level-by-level manner where tasks at the same level are grouped into a *task block*. Crucially, by executing block by block, the code structure is a dense loop over the block allowing one to use standard vectorization techniques [10]. If a task spawns a new task, the newly-spawned task is placed into the next task block representing the next level of tasks. Once the task block finishes, execution moves on to the next block.
2. While breadth-first execution satisfies the two constraints outlined above, it suffers from excessive space usage. A TaskBlock contains all the tasks at a level in the computation tree; therefore, its size can be $\Omega(n)$ where where n is the number of tasks. To control this space explosion, Ren et al. switch to *depth-first* execution once a TaskBlock reaches a sufficient size: each task in the task block simply executes to completion by calling the original recursive function. This limits space usage, as each of these tasks will only use $O(d)$ space to execute (where d is the depth of the tree).
   To recover vectorizability, Ren et al. apply *point blocking* [7], yielding the code in Figure 1(c). This strategy proceeds by (recursively) first executing *each task's* left subtask in a blocked manner and then each task's right subtask. Note that the code still operates over a dense block of tasks and all the tasks in any TaskBlock are at the same level, facilitating vectorization.

Because not all tasks have identical subtrees, as execution proceeds, some threads will "die out" and the TaskBlock will become empty. If there are fewer than Q tasks in a TaskBlock (Q is the number of lanes in the vector unit), it is impossible to effectively use the SIMD execution resources. To tackle this problem, Ren et al. proposed *re-expansion*: if a TaskBlock has fewer than Q tasks, they can *switch back to breadth-first execution*, generating more work and making the next-level block larger. Re-expansion is the key to effective vectorization. In Section 4, we show that re-expansion is a specific instance of a class of scheduling policies that our general scheduling framework supports.

2.2 Work-Stealing Scheduler Operation

Work-stealing schedulers are known to be efficient both theoretically and empirically for scheduling task parallel programs [1, 3]. In a work-stealing scheduler, each worker maintains a work *deque*, which contains *ready tasks*. When a task executing on worker p creates two children, the worker places its right child on its deque and executes its left child. When a worker p finishes executing its current task, it pops a task from the bottom of its own deque. If p's deque is empty, then p becomes a *thief*. It picks a *victim* uniformly at random from the other workers, and *steals* work from the top of the victim's deque. If the victim's deque is empty, then p simply performs another random steal attempt.

3. General Schedulers for Vector and Multicore Execution

This section presents our general scheduler framework for efficient vectorized execution. We first describe three mechanisms that can be combined in various ways to produce dif-

[2] Reductions, computations, etc., are hidden in the baseCase and inductiveWork methods. Here, we use Java-like pseudocode, rather than the original specification language.

ferent classes of schedulers. We next show that re-expansion is an instantiation of this framework. We then describe a new, more flexible, scheduler policy, *restart*. Finally, we explain how the framework can be extended to support multicore execution for both re-expansion and restart-based schedulers. Our schedulers are general, and apply to any type of computation tree.

Notation: We assume P cores and Q SIMD lanes per core. We assume that the recursive method has two recursive spawns; the framework readily generalizes to more spawns.

3.1 Scheduler Mechanisms

We first describe mechanisms for a "sequential" scheduler—a scheduler that supports vectorized execution on a single core. The unit of scheduling of the scheduler is *task blocks* (TaskBlock) which contain some number of tasks—nodes in the computation tree—that can be executed in a SIMD manner: if a TaskBlock has t tasks in it, the scheduler can use Q lanes to execute the t tasks in $\lceil t/Q \rceil$ SIMD steps. The scheduler has a deque, with multiple levels. Each level represents a particular level of the computation tree.

The deque starts with a single TaskBlock at the top level that contains a single task: the task generated by calling the recursive method for the first time. At any time, when *scheduling* a block b for execution, the scheduler can take one of three actions:

Breadth-first expansion (BFE): Block b is executed in a SIMD manner. Any new tasks that are generated (spawned) are placed in a block b', which is scheduled next. Note that b' can contain up to twice as many tasks as b. If b' is *empty* (has no tasks), another block is chosen for scheduling from the deque according to the scheduling policy.

Depth-first execution (DFE): Block b is executed in a SIMD manner. For each task that is executed, all new tasks generated by the first spawn call are placed in block b', and all new tasks generated by the second spawn call are placed in block b''. Note that b' and b'' contain at most as many tasks as b. If not empty, block b'' is pushed onto the deque, and block b' is scheduled next. If b' is empty, then another block is chosen for scheduling from the deque according to the scheduling policy.

Restart: The scheduler pushes block b onto the deque (without executing it). If there is already a block on the deque at the same level (e.g., if b is the sibling of a block that was pushed onto the deque), then b is merged with the block at that level. Some block is chosen from the deque for scheduling next, according to the scheduling policy.

Note the distinction between "scheduling" a block, which means choosing an action to take with the block, and "executing" a block, which means running the tasks in the block using SIMD. For brevity, we say e.g., a block is "executed using DFE" to mean that the block was scheduled, a DFE action was chosen, and then the block was executed.

Various schedulers can be created by combining these mechanisms in various ways, and changing the policies that dictate which mechanism is used at each scheduling step.

3.2 Re-expansion

Ren et al.'s re-expansion policy is, in fact, in the class of scheduling policies that can be described using our scheduling framework. Re-expansion only uses the BFE and DFE actions. In the context of the pseudocode of Figure 1, breadth first expansion is akin to executing the code in Figure 1(b)—a single new TaskBlock is generated—while depth-first execution is akin to executing the code in Figure 1(c)—two new TaskBlocks are generated.

Re-expansion schedulers have two thresholds,t_{dfe}[3] and t_{bfe}[4]. A re-expansion scheduler executes its first block in BFE mode. As long as the next block has fewer than t_{dfe} tasks, the scheduler keeps performing BFE actions. If the next block has more than t_{dfe} tasks, the scheduler switches to DFE actions to conserve space. The scheduler then remains in DFE mode until the block has fewer than t_{bfe} tasks, at which point it switches back to BFE mode. If the current block has no work to do, it grabs work from the bottom of the deque. Intuitively, the scheduler uses BFE actions to generate parallel work and DFE actions to limit space usage. However, because BFE actions can only draw from the bottom of the deque, they can only find parallelism "below" the current block, limiting the effectiveness of re-expansion, as Section 5 shows.

3.3 Restart

We now define a new class of schedulers that we call *restart* schedulers. These schedulers take advantage of the restart mechanism to be more aggressive in finding more parallel work, which can lead to better parallelism with smaller block sizes, as Section 5 shows.

The basic pattern of a restart scheduler is as follows: the scheduler starts in BFE mode, and switches to DFE mode when the t_{dfe} threshold is hit, just as in a re-expansion scheduler. However, once in DFE mode, the scheduler behaves differently. If the current block, b, has fewer than $t_{restart}$ tasks on it, the scheduler takes a restart action. b is merged with any other blocks at the current level, and the scheduler looks for more work to be done. Unlike re-expansion schedulers, which would be constrained to begin executing the bottom block on the deque, a re-start scheduler scans up from the bottom of the deque. As the scheduler works its way up through the deque, it merges (concatenates) all blocks at a given level together (there might be more than one block at a given level due to multiple restart or DFE actions that leave blocks on the deque). After merging, if a block has at least $t_{restart}$ tasks on it, the scheduler begins execution of that block in *DFE* mode. If the restart scheduler reaches the

[3] Trigger to switch to depth-first execution

[4] Trigger to switch to breadth-first expansion

top of the deque without finding $t_{restart}$ work, it executes the top block in BFE mode. Intuitively, if a particular part of the computation tree runs out of work, the restart scheduler will look higher up in the tree and try to merge work from different portions of the tree that is at the same level to generate a larger block of work.

A restart scheduler maintains two invariants: (i) at each level *above* the currently executing block, there may be up to two blocks on the deque—one left over from a restart (that must have fewer than $t_{restart}$ tasks), and the other left over from a DFE (that may have more than $t_{restart}$ tasks, but does not have to); (ii) at each level *below* the currently executing block, there is at most one block on the deque, and that block must have fewer than $t_{restart}$ tasks.

3.4 Creating Parallel Schedulers

Both schedulers described above are sequential; though TaskBlocks can be used for vector-parallel execution, the schedulers are for single-core execution; there is just one deque of TaskBlocks. We now extend the scheduling framework to support multicore execution by adding support for multiple workers (cores), each with their own deque, and adding one more action that a scheduler can take:

Steal: The scheduler places block b onto the deque without executing it (b might be empty). The thread then steals a TaskBlock from another worker's deque. The stolen TaskBlock is added to the worker's deque at the appropriate level and, if necessary, merged with any block already on the worker's deque. The choice of which deque to steal from, which block on the deque to steal, and which block to schedule next, is a matter of scheduler policy.

Using this new action, we can extend re-expansion and restart schedulers to support multicore execution:

Re-expansion: A worker never attempts to steal until its deque is empty (i.e.. it is completely out of work). It then steals the *top* TaskBlock from random worker's deque, adds it to its deque, and executes that block as usual; that is, with DFE if it has more than t_{bfe} nodes and with BFE otherwise. Note that only right blocks generated during DFE are placed on deques and are available for stealing.

Restart: Stealing in restart schedulers is more complicated. In the sequential scheduler, a worker that is unable to create a block with at least $t_{restart}$ tasks in its deque executes the top block in its deque with a BFE action. In the parallel scheduler, that worker would instead perform a steal action. Note that at the time it performs the steal, the worker has an invariant that all of the blocks in its deque have fewer than $t_{restart}$ tasks (because it was unable to restart from its own deque). During the steal, the current worker (thief) chooses a random worker (victim) and looks at the *top* of that deque (note that the victim could be the thief itself).

The top of the victims deque contains one or two blocks. If one of them has at least $t_{restart}$ tasks, the thief steals it, adds it to its deque, and executes it with a DFE action. If the block has fewer than $t_{restart}$ tasks, the thief steals it, adds it to its deque, and executes a constant number of BFE actions starting from that block (to generate more work). If this generates a block with at least $t_{restart}$ tasks, the thief switches to DFE. Otherwise, it tries a restart action and, if that fails, steals again.

3.5 Setting the Thresholds

In producing various schedulers, there are three thresholds of importance: t_{dfe}, t_{bfe}, and $t_{restart}$. The first threshold, t_{dfe} places an upper bound on the block size: a TaskBlock has at most $2t_{dfe}$ tasks. The latter two thresholds dictate when a scheduler looks for more work: if a TaskBlock is below one of the thresholds, the scheduler performs some action to find more work— re-expansion schedulers switch to BFE, restart schedulers perform a restart. These thresholds are less constrained and should be set between Q and t_{dfe}.

4. Theoretical Analysis

This section analyzes the performance of various scheduling strategies described above, namely: a) The basic strategy that performs BFE and then switches to DFE; b) re-expansion strategy; and c) restart strategy. We prove asymptotically tight upper and lower bounds on the SIMD running time T_s for computation trees with n unit time tasks. (Utilization bounds are implied by these bounds.) We first analyze the three sequential strategies on a core with Q SIMD lanes. These results indicate that restart can provide linear speedup as long as the block size (t_{dfe}) is larger than Q. On the other hand, the basic strategy and re-expansion both require large block sizes to guarantee linear speedup; however, re-expansion requires significantly smaller block sizes. We then also analyze the parallel performance of restart where we have P cores each with Q SIMD lanes and show that restart is *asymptotically optimal* for these programs. Our theoretical results hold for any computation tree where the elements of a task block are vectorizable, and hence our restart schedulers can guarantee asymptotically optimal speedup regardless of computation tree structure.

Preliminaries: For the theoretical bounds, we model the execution of the program as a walk of a computation tree with n nodes. Each node in the computation tree is a unit time task and each edge is a dependence between tasks (tasks of longer length can be modeled as a chain). We also assume that all nodes at the same level in the tree are similar in that they can be part of a SIMD block, since, in our computations, they have the same computation and aligned stacks. We also assume that each node has at most 2 children.[5] Therefore, the tree's height h is at least $\lg n$ and at most n.

In each *step*, a core executes between 1 and Q nodes. The step is a *complete step* if the core executes Q nodes; otherwise it is *incomplete*. Recall that in our strategies, the

[5] This assumption is for simple exposition; all the results generalize for larger (but constant) out-degrees.

scheduler operates on TaskBlock of size between 1 and $kQ = t_{dfe}$. If the TaskBlock is of size larger than Q, then it may take multiple SIMD steps to execute it. We call the entire execution of a TaskBlock to encompass a *superstep*.

T_s denotes the SIMD execution time of the program on a single core with Q simd lanes; alternatively, we can think of T_s as the number of steps executed by a scheduling strategy. Some relatively obvious bounds on T_s: (1) $T_s < n$, since each step executes at least 1 node. (2) $T_s \geq n/Q$ since each step executes at most Q nodes. (3) $T_s \geq h$, since each step executes only nodes at a particular level. We will, later, also analyze the parallel execution where we have P cores each with Q SIMD lanes. We say T_{sp} is the execution time of the program on these P cores. Again, we have the obvious lower bounds of $T_{sp} \geq n/(QP)$ and $T_{sp} \geq h$.

We first state some straightforward structural lemmas.

Claim 1. *Each superstep has at most one incomplete step.*

Claim 2. *In the entire computation, the number of complete steps is at most n/Q since each complete step executes exactly Q nodes.*

Claim 3. *The breadth first expansion of a tree (or subtree) of height h has at most h supersteps, since each superstep completes an entire level of the tree.*

4.1 Sequential Strategies with and without Re-expansion

For this section, we state the theorems without proof; proofs will be provided in an extended technical report. The following theorem for the basic strategy, with no re-expansion and no restarts, implies that it guarantees linear speedup iff $k = \Omega(2^\epsilon)$; either the block size must be very large or the computation tree very shallow.

Theorem 1. *For any tree with height $h = \lg n + \epsilon$, the running time is $\Theta(\min\{2^\epsilon n/kQ + n/Q + \lg n + \epsilon, n\})$.*

Now consider the strategy with re-expansion, as described in Section 4.2. We assume $t_{dfe} = kQ$ for some k, and $t_{bfe} = k_1 Q$, where $1 \leq k_1 \leq k$.

Theorem 2. *For any tree with height $h = \lg n + \epsilon$, the running time is $\Theta(\min\{(\frac{(\epsilon - \lg k)}{k_1} + 1)n/Q) + \lg n + \epsilon, n\})$.*

Comparing Theorems 1 and 2, we see that dependence on ϵ is linear instead of exponential in the strategy with re-expansion. Therefore, re-expansion provides speedup for larger values of ϵ and smaller values of k than the strategy without re-expansion. In addition, we should make $k_1 \approx k$; that is we switch to BFE as soon as we have enough space in the block to do so since that decreases the execution time.

4.2 Sequential Execution with Restart

We now prove that the restart strategy, as described in Section 4.3, is asymptotically optimal. We set $t_{restart} = k_2 Q$, where $1 \leq k_2 \leq k$. At this point, we start depth first expansion again. We say that a superstep that executes a block of size smaller than $k_2 Q$ is a partial superstep and a block of size greater than $k_2 Q$ is a full block.

The following lemma is obvious from Claim 3.

Lemma 1. *There are at most h partial supersteps during the computation.*

Theorem 3. *The running time is $\Theta(n/Q + h)$ for a computation tree with n nodes and depth h, which is asymptotically optimal for any scheduler.*

Proof. There are at most $n/(k_2 Q)$ full supersteps and at most h partial supersteps; thus, we have at most $n/(k_2 Q) + h$ incomplete steps by Claim 1. Adding the bound of n/Q for complete steps (by Claim 2), gives us the total bound. This is asymptotically optimal due to lower bounds mentioned in the preliminaries section. □

Note that, unlike the previous strategies, the execution time of the restart strategy does not depend on k or k_2; therefore, we can make the block size as small as Q and still get linear speedup.

4.3 Work-Stealing with Restart

We now bound the running time of the parallel work-stealing strategy with restart, as described in Section 4.4.

We can first bound the number of partial supersteps by observing that a partial block is only executed as part of a breadth first execution, which only occurs either at the beginning of the computation or after a steal attempt.

Lemma 2. *The number of partial supersteps is at most $h + O(S)$.*

For the purposes of this proof, we will assume that a steal attempt takes 1 unit of time. The proof can be generalized so that a steal attempt takes c time for any constant c, but for expository purposes, we stick to the simple assumption. We can then bound the total completion time in terms of the number of steal attempts:

Lemma 3. *The completion time of the computation is at most $O(n/QP + S/P)$.*

Proof. Each step is either an incomplete step, a complete step, or a steal attempt. There are a total of $O(n/Q)$ complete steps by Claim 2. Each full superstep has at least k_2 complete steps. Combining this fact with Lemma 2, we can conclude that there are at most $O(n/k_2 Q + S)$ supersteps. By Claim 1, we can bound the number of incomplete steps by the same number. Since there are P total processors, we can add the number of complete steps, the number of incomplete steps, and the number of steal attempts and divide by P to get a bound on the completion time. □

From the above lemma, we can see that the crux of this proof will depend on bounding the total number of steal attempts S. We will bound the number of steal attempts using an argument very similar to the one used by Arora et. al [1] (henceforth referred to as ABP). Similar to that

paper, we will define a potential function on blocks which decreases geometrically with the depth of the block (in ABP, the potential function is on nodes).

For any block b, we define $d(b)$ as the depth of the block, which is equal to the depth of all nodes in the block. We define $w(b) = h - d(b)$. We can now define various potentials:

- The potential of a full ready block is $4^{3w(b)}$, the potential of a partial ready block is $4^{3w(b)-1}$. The potential of a block that is executing is $4^{3w(b)-1}$.
- The potential of any processor is the total potential of all the blocks on its deque added to the potential of the block it is executing (if any).
- The potential of the computation is the sum of the potential of all processors.

Like the ABP proof, we show that the potential never increases and that a certain number of steal attempts decrease the potential by a constant factor, allowing us to bound the number of steal attempts. However, the proof also differs due to the following reasons: 1) Unlike standard work-stealing, where there is one node at each depth on the deque, we can have up to two blocks at each depth, necessitating a change in the potential function. 2) In standard work-stealing, a worker only steals when its deque is empty; here it may steal even if its deque contains partial blocks. 3) In work-stealing, all the nodes on a processor's deque have a larger potential than the one that the worker is executing; this may not be the case for our scheduler. 4) Each node takes unit time to execute in the ABP model, while our blocks can take up to k time steps to finish executing; this changes the bound on the number of steal attempts.

The following three lemmas prove that the potential never increases, and that steal attempts reduce potential by a significant amount.

Lemma 4. *The initial potential is 4^{3h} and the final potential is 0. The potential never increases during the computation.*

Proof. Potential changes due to the following reasons:

1. A block starts executing: The potential decreases since an executing block has a lower potential than a ready block.
2. An executing block with weight w finishes executing: This block generates at most 2 ready blocks with lower weight. The potential decreases the least when both are full blocks. In this case, the change in potential is $4^{3w-2} - 2 \times 4^{3(w-1)} \geq 0$.
3. Two blocks with weight w merge: If the result of the merge is a partial block, the potential obviously decreases. If the result is a full block, it immediately starts executing. Therefore, the potential also decreases. □

Lemma 5. ***(Top Heavy Deques Lemma)*** *The heaviest block on any processor is either at the top of the processor's deque or it is executing. In addition, this heaviest block contains at least $1/3$ of the total potential of the processor.*

Proof. By construction, the deque is ordered by depth and the weight decreases as depth increases. Therefore, the heaviest node has to be on the top of the deque or executing.

Let x be the heaviest block (if there are two, we pick one arbitrarily). The potential of this block is at least $\Phi(x) = 4^{3w-1}$ (if it is executing or partial); otherwise it is higher. If there is another node at the same depth, its potential is also 4^{3w-1} since it must be a partial block. Other than this, there may be 2 blocks each with all weights lower than w; one full and one partial. Therefore, the total potential of the processor is$\Phi(p) \leq 2 \times 4^{3w-1} + \sum_{y=1}^{w-1}(4^{3(y-1)} + 4^{3(y-1)-1}) \leq 3 \times 4^{3w-1} = 3\Phi(x)$ Thus, the heaviest block has at least a third of the potential. □

Lemma 6. *Between time t and time t', if there are at least kP steal attempts, then $\Pr\{\Phi_t - \Phi_{t'} \geq 1/24\Phi_t\} \geq 1/4$*

Proof. We divide the potential of the computation into two parts. $\Phi(D_t)$ is the potential of the workers where the heaviest block is on the deque and $\Phi(A_t)$ is the potential of the workers where the heaviest block is executing at time t.

First, let us consider a processor p whose potential is in $\Phi(A_t)$ and let us say that the currently executing block is x. From Lemma 5, we know that $\Phi(x) = 4^{3w(x)-1} \geq 1/3\Phi(p)$. For kP steal attempts to occur, at least k time steps must have passed (since each steal attempt takes 1 time step). Therefore, by time step t', x has finished executing. In the worst case, it enables two full blocks with a cumulative potential of $2 \times 4^{3(w(x)-1)} = 1/8\Phi(x)$. Therefore, the potential of p reduces by at least $7/8 \times 1/3 = 7/24$.

Now consider the processors whose potential is in $\Phi(D_t)$, and say p is one of these processors. If p is a victim of a steal attempt this time interval, then the heaviest block x will be stolen and assigned. From Lemma 5, we know that $\Phi(x) \geq 1/3\Phi(p)$. Once x is assigned, its potential drops by a factor of $1/4$. Therefore, the potential that used to be on p has reduced by a factor of $1/12$.

We use Lemma 7 (Balls into Bins) from ABP to conclude that with probability $1/4$, after P steal attempts, the $\Phi(D_t)$ reduces by at least a factor of $1/24$. □

Lemma 7. *The expected number of steal attempts is $O(kPh)$. In addition, the number of steal attempts is $O(kPh + P\lg(1/\epsilon))$ with probability at least $(1-\epsilon)$.*

Proof. The initial potential is 4^h and the final potential is 0. From Lemma 6, we conclude that after $O(kP)$ steal attempts, the potential decreases by a constant factor in expectation. Therefore, the expected number of steal attempts is $O(kPh)$. The high-probability bound can be derived by applying Chernoff Bounds. □

The final theorem is obvious from Lemmas 3 and 7.

Theorem 4. *The running time is $O(n/QP + kh)$ in expectation and $O(n/QP + kh + \lg(1/\epsilon))$ with probability at least $(1-\epsilon)$.*

Corollary 1. *Since n/QP and h are lower bounds on the completion time, restart mechanism provides an asymptotically optimal completion time.*

4.4 Space Bounds and Discussion

Since each worker can have at most 2 blocks at each depth level in its deque, we get the following bound for space for both restart and re-expansion.

Lemma 8. *The total space used is $hkQP$.*

We can now compare the different strategies in terms of the performance they provide. First, let us consider the strategies on a single processor. The above theorems hold for all values of $h = \lg n + \epsilon$. However, different strategies require different block sizes in order to provide speedup. Here we look at the implications of various block size values on speedup provided by the various strategies. Since the restart strategy always provides optimal speedup, we only compare the strategies with and without re-expansion. In addition, as we mentioned earlier, for the restart strategy, it is not advantageous to make k_1 large; so for the following discussion, we make $k_1 \approx k$.

1. If $k > 2^{h-\lg n}$, all schemes give an asymptotically optimal parallelization.
2. If $k \leq 2^{h-\lg n}$, but $k > 2^{h-\lg n}/Q$ then we see very little speedup without re-expansion. With re-expansion, we get optimal speedup if $k \geq \lg Q$ (a reasonable assumption since k is already so large).
3. For smaller k, where $k > h - \lg n$, we get no speedup without re-expansion, but continue to get speedup with re-expansion as long as $k > \lg Q$.
4. For smaller k, re-expansion provides some speedup until k reaches $(h-\lg n)/Q$. After that, it provides no speedup.

Therefore, while the basic strategy requires very large block sizes to provide speedup, re-expansion can provide speedup with moderate block sizes, which nevertheless increase with the height of the tree.

We also implement parallelism using both restart and re-expansion. However, we only prove the bounds for the restart strategy in this section, since that is the more complicated bound. We can use a similar technique to prove the following bound on re-expansion: $O((\frac{(h-\lg n-\lg k)}{k}+1)n/PQ)+kh)$ (assuming we make k_1 as large as we can). However, while with restart, we can set $k = 1$ to get asymptotically optimal speedup, this is not possible for re-expansion. As we can see from the bounds, there is a trade-off between multicore parallelism and SIMD parallelism. Larger k reduces the first term, and increases the second term — thereby increasing SIMD parallelism at the cost of reducing multicore parallelism. Smaller k does the opposite. As for space, for a given k, both strategies have the same bound. However, since restart can provide linear speedup at smaller block sizes, it may use less space for the same performance.

```
1  void c_f(Node p, Node r, float d)
2    x = root.c_x − p.c_x
3    /∗ similarly y and z∗/
4    dr = x ∗ x + y ∗ y + z ∗ z
5    if dr >= d || (r.isLeaf && p!=r)
6      //update p using reduction
7    else
8      for (Node c: r.children)
9        if c != null //task ||
10         spawn c_f(p, c, d/4)

12 void comp_f_all(ps, root)
13   foreach (Node p: ps) //data ||
14     c_f(p, root, d)
```

Figure 2: Barnes-Hut, with data and task parallelism

5. Extended language specification

To evaluate our scheduling framework, we can simply target the same type of recursive, task-parallel programs that Ren et al.'s approach targets. However, we also want to validate our schedulers for more general computation trees. To do so, we extend Ren et al.'s specification language to target a broader class of applications: those that combine data- and task-parallelism by nesting task-parallel function calls inside data-parallel loops. We then extend Ren et al.'s transformation so that programs written in this extended specification language generate the TaskBlocks required by our scheduling framework.

5.1 Mixed data- and task-parallel programs

Ren et al.'s model captures classical programs such as those for solving n-queens or min-max problems. However, many interesting task-parallel programs have a different pattern. In particular, in many domains, the application performs task-parallel work for each item in a set of data; this is often expressed as a *data*-parallel loop with each iteration consisting of task-parallel work.

For instance, consider the Barnes-Hut simulation algorithm (Figure 2) [2]. In it, a series of bodies in space (typically, stars in a galaxy) are placed into an octree. Each body then traverses the octree to compute the gravitational forces exerted by each of the other bodies. At its heart, Barnes-Hut consists of a data-parallel loop: "for each body in a set of bodies, compute the force." However, the force computation can be parallelized using task parallelism by visiting subtrees of the octree in parallel [15]. Many other benchmarks follow this general pattern [8, 13].

Prior work on vectorizing such applications, including Barnes-Hut [8] and decision-tree traversal [13], have focused on the data-parallel portion of the computation and do not exploit the task parallelism. Moreover, these attempts have relied on special-purpose transformations targeted specifically at data-parallel traversal applications, rather than general data-parallel applications with embedded task parallelism.

5.2 Extending the language

Ren et al.'s work, and hence their language specification, does not admit task-parallelism nested within data-parallelism: it targets single recursive calls. We can extend the specification language to admit two types of programs: a single recursive method, as before, or a data-parallel loop enclosing a recursive method; this simple extension to the language allows us to express programs such as Barnes-Hut.

foreach $(d : data) f(d, p_1, \ldots, p_k)$ **if** e_b **then** s_b **else** s_i

Note that the non-recursive functions from Ren et al.'s language specification—such as `baseCase` and `isBase` from Figure 1—can include data-parallel loops. This means that our modified language supports programs that nest data-parallelism inside task-parallelism inside data-parallelism.

5.3 Extending the transformation

In these programs, Ren et al.'s transformation framework treats each iteration of the outer data parallel loop as a single task-parallel program and hence execute the data-parallel outer loop sequentially. Therefore, *only* task-parallelism is exploited, not data-parallelism (the opposite problem of that faced by prior work [8, 13]).

To exploit both data- and task-parallelism, we can readily extend the transformation described in Section 2.1. Note that the transformed program in Figure 1(b) operates on a block of tasks. The initial call to this code consists of a TaskBlock with a single task in it, representing the initial call to the recursive method. When transforming data parallel enclosing task parallel code, we can iterate over the data parallel work to construct *multiple* tasks, one for each iteration of the data-parallel loop. These tasks can then be placed into a TaskBlock, which is passed to the initial breadth-first, task-parallel code. If the initial TaskBlock is too big (i.e., if the data-parallel loop has too many iterations), strip mining can be used to create smaller TaskBlocks that are then sequentially passed to the vectorized, task-parallel code.

If there are data-parallel loops within the non-recursive functions of the task-parallel body (i.e., if a single task has a nested data-parallel loop), the iterations of the data-parallel loops are placed into a TaskBlock generated from the enclosing task. Because these data-parallel loops do not have any further nesting, no further transformations are necessary to handle this additional level of parallelism.

6. Implementation

In this section, we present our Cilk implementation—using *spawn* and *sync*—of the re-expansion and restart strategies to schedule recursive task-parallel programs.

Vectorization: We do not discuss the specifics SIMDization in this paper. At a high level, the data parallel loops over task blocks can be vectorized using the same techniques as in Ren et al.'s work [14], such as *AoS to SoA transformation*, together auto-vectorization and SIMD intrinsics when the auto-vectorizer fails, and the process of adding new tasks to blocks can be vectorized using *Streaming Compaction*.

Re-expansion: Figure 3(a) shows the parallel, blocked re-expansion version of the example in Figure 1(a). If re-expansion is triggered, the left and right TaskBlock are merged together. Otherwise, they are executed independently. This code demonstrates that re-expansion can be naturally expressed as a Cilk program by marking recursive function calls (`blocked_foo_reexp`) in lines 13, 15, and 16 as concurrent using the `spawn` keyword. Other aspects of TaskBlock management, such as the stack of task blocks, are handled by the default Cilk runtime.

Restart: Figure 3(b) shows the parallel blocked re-start execution for the same example according to the strategy explained in Section 4.4. This figure illustrates an ideal implementation. Each thread maintains a deque of TaskBlock and RestartBlock (`tbs` and `rbs` in the figure), with at most one task block and restart block per level in the computation tree. Until termination, each thread steals a TaskBlock or RestartBlock (lines2–4), places it in a TaskBlock at the corresponding level in its local deque, and processes it recursively until no level has a non-empty TaskBlock. At the end of this working phase, a worker's local deque only consists of restart blocks. For conciseness, we do not show the BFE step on a steal as explained in Section 4.3. While efficient, this strategy does not naturally map to Cilk-like programming models. A particular challenge of the strategy in Figure 3(b) is that both the continuation of the TaskBlock being executed and the RestartBlock are pushed to the deque and made available for stealing. Rather than modifying the implementation of the spawn-sync constructs in Cilk, we use a simpler strategy that does not guarantee the same space or time bounds, but nevertheless provides good performance.

Simplified restart: The implementation of our simplified restart strategy is illustrated in Figure 3(c). Rather than maintain the restart blocks in the deque, they are maintained in an explicitly managed RestartBlock stack. The restart stack is implemented as a RestartBlock linked-list consisting of an array of tasks at the current level and a pointer pointing to RestartBlock at the next level. The blocked recursive function (`blocked_foo_restart`) takes a TaskBlock stack and a RestartBlock stack as input and returns a RestartBlock stack. If the total number of tasks in the TaskBlock and RestartBlock is less than the restart threshold, the tasks in the TaskBlock are moved into the RestartBlock, which is returned. Otherwise, we fill up the TaskBlock with tasks from the RestartBlock and spawn the TaskBlock for the next level. This spawn only exposes the task block's continuation—not the restart stack—for stealing by other worker threads. The returned RestartBlock stacks from all spawned instances of next level are merged into one RestartBlock stack (`merge(rleft, rright)`), which is then returned. The merge function is also implemented as a blocked func-

```
1  void blocked_foo_reexp(TaskBlock tb)
2    TaskBlock left, right
3    foreach (Task t : tb)
4      if (isBase(t.x))
5        baseCase()
6      else
7        l1 = inductiveWork1(t.x)
8        left.add(new Task(l1))
9        l2 = inductiveWork2(t.x)
10       right.add(new Task(l2))
11   if (/*do re−expansion*/)
12     left.merge(right)
13     spawn blocked_foo_reexp(left)
14   else /*depth−first execution*/
15     spawn blocked_foo_reexp(left)
16     spawn blocked_foo_reexp(right)
```

(a) Re-expansion

```
1  /*tbs/rbs: task/restart block stack */
2  while (not done)
3    tb = steal(); lvl = tb.lvl; tbs[lvl] = tb
4    blocked_foo_restart(lvl)
5  void blocked_foo_restart(int lvl)
6    if (tbs[lvl].size + rbs[lvl].size < t_restart)
7      /* empty tasks in tbs[lvl] into rbs[lvl]*/
8      return
9    /*fill tbs[lvl] with tasks from rbs[lvl]*/
10   TaskBlock left, right
11   foreach (Task t : tb)
12     if (isBase(t.x)) baseCase()
13     else left.add(new Task(inductiveWork1(t.x)))
14   tbs[lvl+1] = left; blocked_foo_restart(lvl+1)
15   foreach (Task t : tb)
16     if (not isBase(t.x))
17       right.add(new Task(inductiveWork2(t.x)))
18   tbs[lvl+1] = right; blocked_foo_restart(lvl+1)
```

(b) Ideal restart

```
1  RestartBlock blocked_foo_restart(TaskBlock tb,
       RestartBlock rb)
2    if (tb.size + rb.size < t_restart)
3      /*move tasks from tb into rb*/
4      return rb
5    /*fill tb with tasks from rb*/
6    TaskBlock left, right
7    foreach (Thread t : tb)
8      /*Figure 5(a) lines 4−10*/
9    RestartBlock rleft, rright
10   rleft= spawn blocked_foo_restart(left, rb.next)
11   rright = spawn blocked_foo_restart(right, NIL)
12   sync
13   rb.next = spawn merge(rleft, rright)
14   sync
15   return rb
```

(c) Simplified restart

Figure 3: Pseudocode demonstrating blocked, parallel, vectorized execution with different scheduling strategies

tion that recursively merges the input restart stacks, and invokes `blocked_foo_restart` if the number of tasks at a level exceeds the restart threshold.

Frequent restart stack merges may be expensive. To minimize this cost, we introduce an optimization to directly pass the restart stack from one `spawn` to next if *no steal* happened. As shown in Figure 3(c) lines 9–13, in normal case, each `spawn` takes as input a distinct input restart stack and returns a distinct restart stack. These restart stacks are merged together after the `sync`. To reduce the cost, we test whether the a steal immediately preceded the given `spawn` statement. If not, the previous `spawn`'s output restart stack is provided as the input restart stack to this `spawn`, and the merge operation between them is eliminated. In terms of the pseudo-code, the *spawn* in the `blocked_foo_restart` function (line 11) is replaced by the following:

```
1  if NO_INTERVENING_STEAL:
2    rright = spawn blocked_foo_restart(right, rb.next)
3  else:
4    rright = spawn blocked_foo_restart(right, NULL)
```

Limitations of simplified restart: As shown in Figure 3(c), the simplified restart strategy can be directly translated into a Cilk program. While simpler to implement than the restart strategy formulated in Section 5.3, this simplified strategy suffers from significant limitations. These arise from the fact that the RestartBlock stack returned by the function is not available for further processing until all work preceding the immediately enclosing sync scope complete. We briefly discuss these limitations, but do not present concrete proofs due to space limitations. First, passing restart stacks through the return path can lead to an important source of concurrency being stashed away, unavailable for execution by idle workers. In the worst case, this can lead to a fully serialized execution. Second, consider a execution of a TaskBlock at depth h. At each level 0 through $h-1$, a restart stack can be suspended, requiring a total space of $h^2 t_{restart}$ space, which is worse than the space overhead for the ideal restart strategy. Despite these limitations, we observe in Section 7 that this restart strategy achieves competitive performance for the benchmarks considered.

7. Experimental Evaluation

We conduct our experiments on an 8-core, 2.6 GHz Intel E5-2670 CPU with 32 KB L1 cache per core, 20 MB last-level cache, and 128-bit SSE 4.2 instruction set. The input recursive program, re-expansion (`reexp`), and restart (`restart`) variants are parallelized as Cilk programs and compiled using MIT Cilk [5][6]. All program variants are compiled using the Intel icc-13.3.163 compiler with `-O3` optimization. We report the mean of 10 runs. We do not report the negligible standard deviation observed due to space constraints. We observed little difference in performance between enabling and disabling compiler vectorization for the input program, despite some loops getting vectorized. We use execution times of the sequential runs with compiler auto-vectorization as the baseline.

Table 1 list the basic features of the benchmarks: problem input, number of levels in the computation tree, and total number of tasks. The sequential version of each benchmark is obtained by removing the Cilk keywords (spawn and sync). The execution time for this sequential execution time (T_s) serves as the baseline for all evaluation in this section. The benchmarks have varying computation tree structures: `knapsack` is a perfectly balanced tree with all base case tasks at the last level. `fib`, `binomial`, `parentheses`, `Point correlation`, and `knn` (k-nearest neighbor) are unbalanced binary trees with varying numbers of base case tasks in the middle levels. All other benchmarks are more unbalanced with larger fan-outs. `uts` is a binomial tree and much deeper than `nqueens`, `graphcol`, `minmax`, and `Barnes-Hut`[7].

The benchmarks show various mixing of *task parallelism* and *data parallelism*: `knapsack`, `fib`, `parentheses`,

[6] The non-parallelized version of `reexp` is, effectively, Ren et al.'s vectorization strategy [14].

[7] A more detailed characterization can be found in [14]

Benchmark	Problem	#Levels	#Tasks	T_s(s)	T_1(s)	T_{16}	Block size	RB size	T_s/T_1	1-worker		T_s/T_{16}	16-worker	
										T_s/T_{1x}	T_s/T_{1r}		Ts/T_{16x}	Ts/T_{16r}
knapsack	long	31	2.15B	9	61	3.9	2^{12}	any	0.14	1.68	1.69	2.2	24.8	24.6
fib	45	45	3.67B	9	99	6.7	2^{14}	4096	0.09	1.57	1.58	1.3	22.9	22.7
parentheses	19	37	4.85B	10	134	8.8	2^{13}	4607	0.08	1.42	1.40	1.2	20.3	19.4
nqueens	15	16	168M	233	267	16.7	2^{12}	2040	0.89	4.00	4.08	14.2	61.6	60.9
graphcol	3(38-64)	39	42.4M	31	37	2.4	2^{10}	473	0.85	8.74	8.78	13.0	108	107
uts	30	228	19.9M	146	152	9.6	2^{11}	2047	0.97	1.58	1.59	15.4	23.0	23.8
binomial	C(36,13)	36	4.62B	8	126	8.1	2^{13}	4096	0.07	1.00	0.97	1.0	14.9	14.1
minmax	4×4	13	2.42B	19	58	3.7	2^{10}	32767	0.31	1.65	1.57	4.9	20.6	17.1
Barnes-Hut	1M	18	3.00B	81	227	53.8	2^9	511	0.35	1.34	1.48	1.5	16.0	17.5
Point corr.	300K	18	1.77B	132	197	13.1	2^{10}	256	0.74	1.89	1.74	11.1	29.9	27.9
knn	100K	15	1.36B	71	105	7.6	2^9	128	0.65	1.24	1.20	8.9	19.3	18.4
Geo. mean									0.31	1.89	1.87	4.2	26.7	26.0

Table 1: Benchmark characteristics and performance. T_s: sequential execution time; T_1: single-threaded execution time of the input Cilk version; T_{1x}, T_{1r}: single-threaded SIMD execution time of re-expansion and restart versions, respectively; T_{16}, T_{16x}, T_{16r}: execution time of input, re-expansion, or restart versions on 16 workers, respectively. All re-expansion and restart implementations use 16-wide vector operations, except 8-wide for knapsack,and 4-wide vector operations for uts, Barnes-Hut, Point correlation, and knn. Block size: Best block size for re-expansion and restart, except 16-worker nqueens (2^{11}), graphcol (2^9), and minmax (2^{15} for re-expansion and 2^9 for restart). RB size: Restart block size used by the restart versions.

	scalar	reexp			restart		
		Block	SOA	SIMD	Block	SOA	SIMD
1-worker	0.3	0.5	0.6	1.9	0.5	0.6	1.9
16-worker	4.2	6.4	9.5	26.7	8.2	9.3	26.0
Scalability	13.6	11.7	15.3	14.2	16.2	15.2	13.9

Table 2: Geometric mean of speedup with respect to T_s for implementation variants on 1 and 16 workers. scalar: input Cilk program; Block: Blocked using re-expansion or restart strategy; SOA: Blocked version transformed to struct-of-arrays layout to enable vectorization; SIMD: SIMD vectorized version of the benchmark in SOA form. Scalability shows the relative performance of the same implementation running on 16 workers as compared to 1 worker.

`uts`, `binomial`, and `minmax` only show *task parallelism* in the form of recursions; `nqueens` and `graphcol` have nested *data parallelism* within their outer *task parallel* recursions; `Barnes-Hut` has a for-loop outside recursive function calls, i.e., *task parallelism* is nested in *data parallelism*; and `Point corr.` and `knn` are more complex, with three levels of parallelism, *data parallel* base cases nested within *task parallel* recursion, which is, in turn, nested within *data parallel* outer loops.

To improve vectorization potential, we use the smallest possible data type, without affecting generality, for each benchmark: `knapsack` and `uts` use `short` and `int`, respectively. `Barnes-Hut`, `Point correlation`, and `knn` use `float`. Other benchmarks use `char`.

7.1 Speedup from Blocked SIMD Execution

Table 1 shows the effectiveness of various strategies to exploit task and data parallelism on both single core (1-worker) and multi-cores (16-worker). Even on one thread, re-expansion and restart strategies, coupled with SIMD vectorization, improve (1.8× average) over sequential execution times. Combining compiler vectorization with Cilk parallelization achieves a speedup of 4.2 (average) over sequential execution. Re-expansion and restart significantly improve upon this default parallelization, achieving 26× (average) speedup over T_s. This shows the overall efficacy of the re-expansion and restart strategies. Re-expansion and restart strategies exhibit similar performance across all the benchmarks considered. For `restart`, we select the optimum $t_{restart}$ threshold, shown as *RB Size* in the table.

Our speedup primarily benefits from three sources: first, the reduction of recursive function calls from pure task parallelism to the combination of both task parallelism and data parallelism; second, the vectorization optimization, and associated efficient data layout transformations such as *AoS to SoA*; and third, good scalability from one worker to multiple workers. The first effect is important for kernels with little sequential work. In these kernels, the Cilk recursive scheduling incurs relatively large overheads. Our blocked execution can efficiently reduce such overhead, because our task blocks coarsen the granularity of spawned tasks. This is also why we may get more than `SIMD-width` times speedup for some of these benchmarks' 1-worker version. Table 2 demonstrates the relative speedup of each transformation on the input benchmark from the input version to the blocked, layout transformed to struct-of-arrays form so as to enable SIMD vectorization (`SOA`), and the final vectorized version.

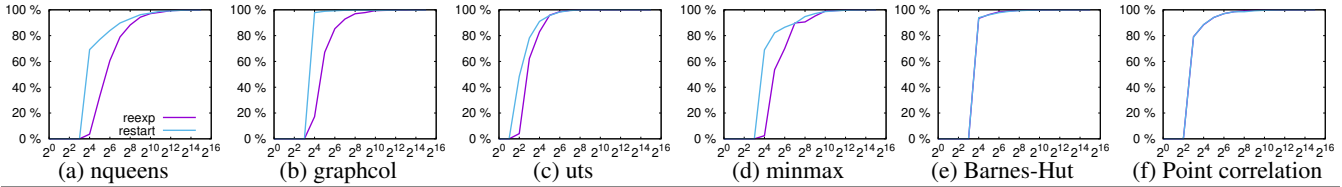

Figure 4: SIMD utilization. x-axis: block size; y-axis: %age of tasks that can be vectorized. knn is identical to Point correlation.

For both re-expansion and restart strategies, we see that the `SOA` transformation to enable vectorization improve performance on top of blocking. However, we observe even greater improvements from vectorizing this layout-transformed version. In general, comparing to the first effect, the second and third effects are more important. Therefore, in the following sections, we evaluate the impact the `reexp` and `restart` strategies have on vectorization efficiency and scalability.

7.2 Impact of Block Size on SIMD Utilization

Increasing the block size can expose more work to be executed by the vector unit, increasing *SIMD utilization* (the ratio of complete SIMD steps to total steps during execution). However, larger block sizes consume more space and increase cache pressure. A good scheduling strategy should achieve good SIMD utilization with small block sizes.

Figure 4 shows the SIMD utilization of `reexp` and `restart` at various block sizes (for space reasons, we present results only for the larger benchmarks). SIMD utilization grows with increasing block size, and, consistent with the theoretical analysis, at each block size `restart` matches or exceeds the SIMD utilization achieved by `reexp`. For smaller block sizes, `restart` performs better than `reexp` for all benchmarks except for `knapsack`, `Barnes-Hut`, `Point correlation`, and `knn`, where both strategies achieve similar performance improvements. For benchmarks like `graphcol` and `uts`, `restart` can achieve $> 90\%$ SIMD utilization at block size of 2^4, while `reexp` requires 2^7 and 2^5. For benchmarks such as `nqueens` and `minmax`, SIMD utilization improves more slowly for both `reexp` and `restart`. However, SIMD utilization for `restart` continues to exceed that for `reexp` for smaller block sizes. In the case of `Barnes-Hut` and `Point correlation`, both `restart` and `reexp` achieve identical SIMD utilization for all block sizes considered.

7.3 Scalability of Blocked SIMD Execution

The scalability for the best block size is shown in Table 1. For the best block size, `reexp` and `restart` achieve comparable speedups for most benchmarks, and the performance difference is within 5%.

Given the difference in SIMD utilization between re-expansion and restart at smaller block sizes, we show scalability of of `reexp` and `restart` with various numbers of Cilk workers for a small block size (2^5) in Figure 5. For this small block size, we find that `restart` performs better if it can improve the SIMD utilization (such as `nqueens`, `graphcol`, `uts` and `minmax`). Otherwise, it may result in some slowdown due to the implementation complexity and the cost of manipulating the restart stacks. For `nqueens` and `minmax`, `restart` can provide a reasonable speedup whereas `reexp` still results in some slowdown relative to the non-SIMD version. For benchmarks such as `Barnes-Hut`, `Point correlation`, and `knn`, the relatively poor scalability from 8-worker to 16-worker is due to the interference between hyper-threaded workers executed on the same core in the 8-core system.

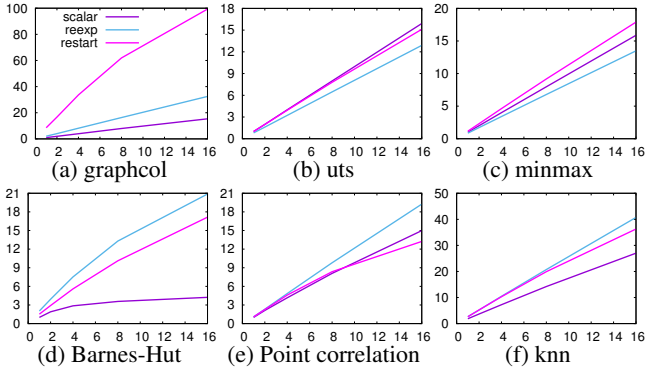

Figure 5: Xeon E5 scalability for block size 2^5. x-axis: number of workers; y-axis: speedup relative to 1-worker Cilk baseline.

In summary, both scheduling strategies have their advantages: `reexp` is easier to implement and has very low scheduling overhead, while `restart` has good theoretical bounds and can potentially improve the SIMD utilization for smaller block sizes. With either scheduling method, we can transform our candidate codes to achieve good speedup.

8. Related Work

In many programming systems, parallelizing recursive programs involves user identification of concurrent function invocations coupled with a runtime system that dynamically maps concurrent function invocations to the processing cores. Variants of the spawn-sync construct used in this paper constitute the most common subset of many such programming systems (Cilk [3, 4], Thread Building Blocks [12], OpenMP [11], X10 [17], etc.). These programming systems also allow programmers to write programs that mix task- and data-parallelism, and are not necessarily tied to SIMD architectures, so they are more general. However, they focus on mapping that parallelism to multicores, rather than tackling SIMD; thus, they typically decouple vectorization considerations from multicore parallelism and

attempt to use SIMD instructions primarily in the base case. These approaches hence exploit SIMD less thoroughly than ours, which vectorizes the recursive steps in addition to base cases, taking advantage of SIMD at all steps of the computation.

Intel's ispc [**?**] is another existing effort closely related to our work. It targets programs that have task- and data-parallelism, and also attempts to exploit SIMD units in addition to multicores. However, its handling of nesting of task- and data-parallelism differs substantially from ours. In particular, given a data-parallel for loop, different iterations are mapped to individual lanes of a SIMD unit. If that for loop contains nested task-parallelism, that task parallelism is not exploited: the entire iteration is executed within a single SIMD lane, underexploiting parallelism, causing divergent execution, and incurring unnecessary *gather* and *scatter* memory operations among all SIMD lanes.

Optimizations to improve recursion overhead include partial recursive call inlining [16] and transforming the program to a loop program [9]. Partial inlining indirectly helps vectorization by adding more instructions to the inductive work basic block, but often not enough to keep wide SIMD units busy.

Similar to our approach, Jo et al. [7] optimized tree traversal algorithms by vectorizing across multiple root-to-leaf traversals. The nodes of the tree, rather than the call stack, are used to track execution progress. Also, their work did not consider multi-core parallelization. Ren et al. [14] presented the re-expansion strategy to vectorize recursive programs. However, as discussed in the introduction, they provide no theoretical analysis of the benefits of their strategy, do not deal with multicore execution, and do not consider the restart strategy.

9. Conclusions

We presented a unified approach for executing programs that exhibit both task- and data-parallelism on systems that contain both vector units and multicores. Rather than "baking in" execution strategies based on programming models (e.g., converting data parallelism to task parallelism in Cilk [6], or vice versa [14], or choosing to only exploit one type of parallelism in programs that contain both [8, 13]), our unified scheduling framework allows programs to execute in the manner best suited to the current context and hardware: if there is enough data parallelism at a particular point in an execution, vector units can be exploited, while if there is too much load imbalance at another point, data parallelism can be traded off for task parallelism to allow for work stealing. This *grain free* execution strategy allows for the free mixing and arbitrary exploitation of both types of parallelism and both types of parallel hardware. By using our framework, we are able to achieve 14×–108× speedup (23×–259× over 1-worker Cilk) across ten benchmarks by exploiting both SIMD units and multicores on an 8-core system.

10. Acknowledgments

The authors would like to thank our shepherd, Mary Hall, as well as the anonymous reviewers for their helpful suggestions and comments. The authors would also like to thank Youngjoon Jo for contributing to several benchmarks. This work was supported in part by the U.S. Department of Energy's (DOE) Office of Science, Office of Advanced Scientific Computing Research, under DOE Early Career awards 63823 and DE-SC0010295. This work was also supported in part by NSF awards CCF-1150013 (CAREER), CCF-1439126, CCF-1218017, and CCF-1439062. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830.

References

[1] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. In *SPAA*, pages 119–129, 1998.

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

[3] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, 46(5):720–748, 1999.

[4] 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, August 25, 1996.

[5] Cilk. Cilk 5.4.6. http://supertech.csail.mit.edu/cilk/.

[6] I. Corp. Intel Cilk Plus Language Extension Specification, 2011.

[7] Y. Jo and M. Kulkarni. Enhancing Locality for Recursive Traversals of Recursive Structures. In *OOPSLA*, pages 463–482, 2011.

[8] Y. Jo, M. Goldfarb, and M. Kulkarni. Automatic Vectorization of Tree Traversals. In *PACT*, pages 363–374, 2013.

[9] Y. A. Liu and S. D. Stoller. From Recursion to Iteration: What are the Optimizations? In *PEPM*, pages 73–82, 2000.

[10] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua. An Evaluation of Vectorizing Compilers. In *PACT*, pages 372–382, 2011.

[11] OpenMP Architecture Review Board. OpenMP Specification and Features. http://openmp.org/wp/, May 2008.

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

[13] B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte. SIMD Parallelization of Applications that Traverse Irregular Data Structures. In *2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10. IEEE, 2013.

[14] B. Ren, Y. Jo, S. Krishnamoorthy, K. Agrawal, and M. Kulkarni. Efficient Execution of Recursive Programs on Commodity Vector Hardware. In *PLDI*, pages 509–520, 2015.

[15] M. Rinard and P. C. Diniz. Commutativity analysis: a new analysis technique for parallelizing compilers. *ACM Trans. Program. Lang. Syst.*, 19(6):942–991, 1997. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/267959.269969.

[16] R. Rugina and M. C. Rinard. Recursion Unrolling for Divide and Conquer Programs. In *LCPC*, pages 34–48, 2000.

[17] X10. The X10 Programming Language. www.research.ibm.com/x10/, Mar. 2006.