Enhancing Productivity and Performance Portability of General-Purpose Parallel Programming

Alexandros Tzannes Advisors: Rajeev Barua Uzi Vishkin
University of Maryland, College Park
tzannes@umd.edu

Abstract

This past decade has seen the shift from single core to multi-core processors in desktops and laptops, but software remains largely sequential and unable to benefit from the increasing number of cores. One difficulty is that current parallel programming approaches fall short of being *general-purpose* and in particular *performance portable* across platforms, datasets, and calling-contexts.

We propose a path towards efficient support of performance portable general-purpose parallelism and take the first steps towards that goal. Concretely, we propose *lazy scheduling*, a scheduler that adapts the granularity of parallelism to runtime load conditions, and we revisit how programmers should expose nested parallelism for lazy scheduling. We propose a methodology for measuring performance portability and show that our proposed programming approach is efficient and performance portable. This work hopefully provides enough progress to make general-purpose parallelism attractive to early adopters and to inspire more research.

1. Introduction

While multi-cores are now ubiquitous, general-purpose code remains overwhelmingly sequential and unable to benefit from the increasing number of cores. Progress of hardware and software was formerly fueled by the *software spiral*: new faster hardware would benefit existing software and encourage new software to push the hardware to its limits. Current hardware and parallel programming approaches are not recreating this productive feedback cycle based on performance portability, making general-purpose programmers reluctant to adopt parallelism.

The two main properties of general-purpose programming are *ease of programming* (henceforth *programmability*) and *performance portability*. We focus on task parallel programming (e.g., Cilk [17], Cilk++ [32], TBB [1], OpenMP [34], TPL [31]) on shared memory platforms, because of its programmability advantages (e.g., implicit synchronization, and reduced sensitivity to locality). *Modularity*, the ability to develop modules (e.g., objects, procedures) and to combine them into an application, allowing code reuse, is a key requirement of programmability. Efficient support of *nested parallelism*, the ability to expose additional parallelism from an already parallel context, is necessary to enable modularity. To accommodate arbitrary nesting of parallelism, *dynamic scheduling* is employed: the scheduler assigns tasks to worker threads (henceforth *workers*) at runtime. Due to its efficiency, Work Stealing scheduling [5, 27], enables the programmer to expose much more parallelism in their code than there are workers, which benefits programmability and performance portability.

Performance portability for parallel general-purpose code necessitates not only portability across platforms, but also across inputs and calling contexts. Different platforms with different architectures and numbers of cores affect how much parallelism is needed, and inputs can affect how much parallelism is exposed. Modularity (with nested parallelism) hides the amount of parallelism already exposed in the program, raising the issue of context portability: how much parallelism a module needs to expose depends on whether it was called from sequential or parallel code. If the programmer could expose all the parallelism in their program without incurring significant scheduling overheads, performance portability would boil down to adequate hardware support, but this is not the case; Work Stealing, like any dynamic scheduler, pays an overhead per task, which can become overwhelming if tasks are too short.

To avoid excessive scheduling overheads, programmers are required to control how much parallelism they expose by *coarsening* it if necessary. They should expose enough parallelism to feed all workers *and* have spare tasks for load-balancing, but not so much that the scheduling overheads become significant. While this may be a reasonable task in some scenarios (e.g., without nested parallelism, or without modularity [if the programmer coarsens for an entire application as opposed to per module], or without performance portability), the requirements of modularity and performance portability make coarsening very challenging. In fact, we will show that coarsening techniques commonly used today harm performance portability.

Algorithm 1 Queens pseudocode. $depth \in [1, N]$

```
1:  procedure QUEENS(depth, partSolution, N)
2:      for all i ∈ [1, N] do
3:          if OkToAdd(i, depth, partSolution) then
4:              partSolution' ← partSolution ∪ (i, depth)
5:              if depth < N then                          ▷ Recursion
6:                  if depth > CUTOFF-DEPTH then           ▷ Coarsening
7:                      QUEENS_Seq(depth + 1, partSolution', N)
8:                  else
9:                      QUEENS(depth + 1, partSolution', N)
10:                 end if
11:             else                                       ▷ Found a Solution
12:                 atomic(globalSolCount+ = 1)
13:             end if
14:         end if
15:     end for
16: end procedure
```

We consider three types of coarsening: *(1)* picking a grain-size G for a parallel construct (e.g., loop) indicating that parallel tasks should contain at least G iterations; *(2)* not parallelizing a computation, for example, a parallel loop written as a sequential loop; *(3)* explicitly serializing part of the computation (e.g., parallelism cut-off in Lines 6-10 in QUEENS, Algorithm 1). Coarsening that involves changing the algorithm is beyond our scope.

Summary of Contributions To circumvent the performance portability reef of coarsening, we offer the following contributions: *(1)* we distinguish two goals of coarsening, *amortizing* scheduling overheads and *pruning* parallelism, and we argue why pruning should be done by the scheduler. *(2)* We introduce *lazy scheduling*, which makes Work Stealing adaptive to load conditions by pruning parallelism as needed, without maintaining additional state and without making any irrevocable serialization decisions. *(3)* We pro-

pose a methodology for measuring performance portability, and we show that our proposed distinction of coarsening goals with lazy scheduling achieves superior performance portability, while also achieving very good efficiency. The hardware and programming models of multi-core parallel computing are in flux, making robust contributions amenable to portability timely and challenging. Our hope is that our contributions and guidelines will inspire and focus innovations on improving support for general-purpose parallelism.

2. Background and Related Work

2.1 Work Stealing

Work Stealing, dating at least back to Burton et al. [6] and Halstead [20], has become the scheduler of choice for industry and academia (e.g., TBB[1], TPL[31], Java ForkJoin[30]) for its efficient support of nested parallelism. The programmer exposes parallelism in the form of tasks, which typically greatly outnumber the workers, and Work Stealing efficiently maps tasks to workers at runtime. Some of the popularity of Work Stealing is due to the good space and performance bounds shown by Blumofe et al. [5] and to its efficient implementation in Cilk [17].

In Work Stealing, the shared work-pool is distributed: each worker owns a deque (a double ended queue). Each time a worker encounters a new task (e.g. a parallel function call), it pushes the continuation of the current task on its deque and proceeds to execute the new task. When a worker runs out of work it tries to pop tasks from its deque, but, if it is empty, it becomes a *thief*: it randomly picks the deque of a *victim* worker and tries to *steal* a task from it, until a theft succeeds.

The distributed nature and efficient implementation of the deques give Work Stealing many of its performance advantages. Each worker treats its deque as a stack, pushing and popping work on one end, and it treats all other deques as queues, stealing tasks from the opposite end. By exclusively owning one end of the deque, workers can push and pop work with practically no synchronization. Moreover, treating the local deque as a stack enforces a depth-first execution order which favors temporal locality. Finally, treating victim deques as queues returns the oldest task, which is at the shallowest nesting depth and usually contains more work, thus minimizing the number of thefts which are costly.

Blumofe et al. [5] showed that Work Stealing has good space, and time bounds for fully-strict computations in a language with parallel function calls. Let S_1 be the space needed by the sequential execution of the parallel program, then the space requirement for a Work Stealing execution with P workers is $S_1 P$. Let T_1 be the execution time on one worker (i.e., the work), and let T_∞ be the length of the critical path (i.e., depth), then the expected execution time of a Work Stealing computation will be $T_1/P + \mathrm{O}(T_\infty)$.

Adding to the language parallel constructs that introduce multiple tasks simultaneously (such as loops, reducers, and scans [1, 18]) raises some issues. First, each construct creates a vector of tasks which is summarized by a *task descriptor*. Then, there are different ways to schedule a task descriptor [19, 37]. A widely used one is *binary splitting*: iteratively splitting and pushing half of the task descriptor on the deque (Figure 1). The stop-splitting-threshold (*sst*) typically defaults to 1, unless it is defined by the programmer. The space and time (critical path) overheads are then $\mathrm{O}(\log N)$.

One issue with Work Stealing is its eagerness to expose all potential parallelism on its deques. While this helps to prove the above time bound, in practice, it can lead to excessive overheads if tasks are too short. In the common case, tasks pushed onto a deque are executed by the same worker that pushed them, so the deque management overhead is wasted. Moreover, because the size of deques is potentially unbounded, their implementation becomes more complex and expensive [12, 21]. Our lazy scheduling approach uses the size of the deque as a heuristic for inferring at runtime if other workers might need work and for avoiding to push tasks on the deque if they are likely to be executed locally. As an added benefit, the implementation of deques becomes simpler and more efficient because their maximum size is known statically.

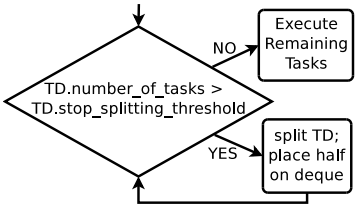

Figure 1. Binary Splitting of a Task Descriptor (TD)

2.2 Adaptive Runtimes

Kranz [27] and Certner [11] use runtime conditions to make serializing decisions, but maintain additional global state (e.g., a counter) to gauge the load, which creates a hot-spot and does not scale. Moreover, their serializing decisions are irrevocable, which can cause deadlock [27]. Conversely, lazy scheduling uses existing local information to infer load and only postpones pushing tasks on the deque, allowing to revisit that decision and to avoid deadlock.

Duran et al.[14] propose a scheduler-agnostic technique to limit the creation of excessive parallelism. They inject code that collects statistics about the amount of work of different procedures as a function of the depth (of the call-stack) at which they are called, and, when enough statistics are collected, they turn off this profiling and use the information to decide which procedures to serialize and at what depth. Given a recursive parallel procedure such as quicksort, their approach will decide at which depth of the recursion to start calling a serial version of quicksort. This is an interesting way to automate coarsening, but the need to turn-off profiling because of overheads raises questions about its applicability to long-running applications, where a function can be used in different contexts by different stages of the computation.

Robison et al. [37] present Auto-Partitioner (AP), an adaptive scheduler that starts by splitting task descriptors into $K \cdot P$ *chunks*, where K is a constant parameter that defaults to 4. This prevents excessive splitting overheads for flat parallelism but not for nested. AP's adaptivity is triggered by thefts, whereby a stolen task descriptor is split into at least V chunks, possibly beyond the original $K \cdot P$ limit. V is also a constant parameter that defaults to 4.

Pan et al. [36] address the issue of core over-subscription by Work Stealing codes when, for example, TBB code calls `libprocess` code; each of these two libraries assumes they should create as many workers as there are cores, which results in twice as many workers as cores. A similar problem arises on multiprogrammed machines when the operating system needs to dynamically allocate or reclaim cores from processes (also addressed by [2]). These techniques complement our proposed approach and are needed for supporting general-purpose parallel programming.

3. Approach and Uniqueness

3.1 Characterizing Coarsening

The two goals of manual coarsening of task-parallel code are *(1)* to ***amortize the scheduling overhead per-task*** (OPT) by increasing the granularity of extremely fine-grained tasks, and *(2)* to ***prune*** the exposed parallelism to avoid deploying too much parallelism. We argue that static coarsening is adequate for amortizing the OPT but inadequate for pruning parallelism, and we propose a novel dynamic coarsening technique for pruning, *lazy scheduling*.

Understandably, the two goals of coarsening are not decoupled. Amortizing the OPT also prunes parallelism, but it may still leave an excessive amount of parallelism exposed. Symmetrically, pruning parallelism also increases granularity, but, if there is too lit-

tle work, it may fail to amortize the OPT. To make things harder, the same technique can sometimes be used to achieve either of the goals. In Algorithm 1 for example, we should cut-off the last α recursive levels (for some α) to amortize the OPT (i.e., when $depth > N - \alpha$), whereas, to prune the amount of exposed parallelism, we should cut-off after the first R recursive levels (when $depth > R$). Fig. 2 illustrates these two cases, where tasks are nodes, and the distance of a node from the root is its recursive depth. To amortize the OPT, one groups nodes starting from the leaves, but to to prune parallelism, one only allows the first few levels of the tree to unfold.

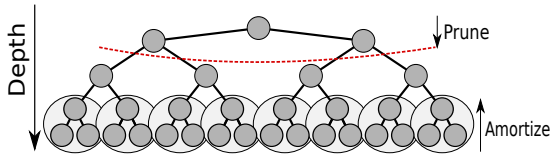

Figure 2. Amortizing vs. Pruning Parallelism

But why should pruning parallelism be necessary if we properly amortize the OPT by coarsening very fine-grained tasks? In QUEENS, grouping subtrees near the leaves amortizes the OPT, but, to reach those coarsened tasks, the computation has to traverse the whole tree, paying the scheduling overhead multiple times, once for each edge traversal. In effect, we have not amortized tasks mid-tree which creates the need for pruning parallelism.

The reason for not coarsening mid-tree is that it changes its shape (depth and branching factor) and therefore the algorithm. Since the goal is to reduce scheduling overheads, the challenge is to make all of the outgoing edges (i.e., subtasks) of a coarsened mid-tree node available to the scheduler before proceeding with the execution of one of them. Failing to achieve that may result in significant parts of the tree being unavailable for execution, potentially causing scaling problems. Moreover, the aggregate scheduling overheads of all the outgoing edges should be reduced, because just saving the overheads of the internal edges of a coarsened mid-tree nodes may not be sufficient. For example, for a K-ary tree given that the last C last levels have been serialized, further coarsening mid-tree can only save up to $1/K$ of the remaining scheduling overhead (by serializing from the root to level $D - C$, where D is the depth of the tree). *Given that coarsening mid-tree is difficult, not very profitable in many cases, and prone to changing the computation and causing load-imbalance, we maintain that amortizing the OPT and pruning parallelism should be performed separately and that coarsening mid-tree should generally be avoided.*

Deciding how much to *amortize* the OPT is relatively easy, because the OPT of modern schedulers is very low, which makes coarsening necessary only for extremely fine-grained tasks, and because such tasks are generally short and simple. Moreover, the amount of coarsening only depends on the OPT, which can be approximated by a constant upper-bound (e.g. 100 cycles), and the work per fine-grained task, which can often be estimated statically.

Amortizing the OPT must happen before the runtime tries to schedule the tasks because the goal is to prevent calling the scheduler for tasks that are too short. Eventually, a mature compiler may be able to completely relieve the programmer from this coarsening task. This is desirable because the OPT may depend on the platform, affecting the amount of coarsening needed. However, we expect the cross-platform variation of the OPT to be small enough [39] to permit manual coarsening without harming performance-portability, so it is reasonable for programmers to amortize the OPT, while compilers make progress towards automating this task.

Using static coarsening to decide how much to *prune* the exposed parallelism is unnatural, however, because the amount of parallelism is often unknown at compile-time (or at programming-time): *(1)* the parallelism exposed by a module (e.g., a function) may depend on the size or shape of the input D; *(2)* the amount of the available hardware parallelism depends on the context C from which the module was called, and *(3)* the number of workers P available to a process depends on the platform and on other processes running concurrently. Static coarsening cannot adapt to the dynamic nature of these parameters (D, C, P) and typically results in overfitting to a small range of these parameters. Outside that range, the performance of the code can be far from optimal and, hence, not performance-portable. *Instead of the inadequate static coarsening currently used, we propose that the scheduler, which has access to dynamic load information, should adaptively deploy the exposed parallelism as needed, effectively pruning parallelism at runtime*. Next, we propose such an adaptive scheduler.

3.2 Lazy Scheduling

Lazy Scheduling effectively prunes parallelism by adaptively pushing tasks onto the deques only when they are likely to be stolen and, thus, help keeping workers from idling.

The first insight of lazy scheduling is that pushing tasks onto the local deque is likely a wasted overhead if other workers are busy with other work. It is better for a worker to execute some tasks and then check the system load again to possibly push work onto its deque. That way, unnecessary splitting and deque transactions are avoided, but workers are kept busy.

Directly implementing lazy scheduling to follow the above insight is not obvious because checking if other workers are busy can be expensive. For example, maintaining global state, such as a count of idle workers, does not scale without hardware support, and, on the other hand, querying workers to see if they are idling requires expensive remote accesses. Furthermore, the mechanism for checking if other workers need work must be fast to prevent them from idling too long.

The second insight of lazy scheduling is to use the size of the local deque as a light-weight heuristic for inferring the system load at runtime. If the deque size is below a threshold (empty in our implementation), it is a strong indication that other workers were hungry and stole work from it; in that case the worker pushes work onto its deque. Conversely, if the local deque is above the threshold, pushing tasks onto it is postponed, and the worker executes locally the next task. This prevents tasks from always having to become available on the deque before being executed, and effectively results in *load-based pruning of parallelism*.

To reduce synchronization and benefit performance, reading the size of a deque can be done in a racy way relative to a concurrent theft, provided that the value returned is that of the size before or after the theft. This is acceptable because a slightly stale value does not perturb the accuracy of the heuristic in practice.

Lazy Scheduling creates a new logical state in which tasks may be in, the *postponed* state. Postponed tasks are those that have become available for parallel execution, but the scheduler has not yet pushed them on a deque because the system is under load; those tasks reside in thread-private memory (e.g., the stack). A worker first executes its postponed tasks, then its deque tasks, then steals.

Figure 3 shows how lazy scheduling processes a task descriptor, including a *profitable parallelism threshold* (*ppt*) that is responsible for amortizing the OPT and is picked by the compiler. First, notice there are two new boxes in this decision diagram: the deque check, and executing some tasks while postponing to push the rest of the descriptor on the deque. Second, an empty deque results in pushing (possibly after splitting) the *oldest* postponed task descriptor (TD). To enable that, postponed TDs are kept on a private deque. This is more efficient than putting them on the (shared) local deque by default because *(1)* the private deque is accessed without synchronization, *(2)* postponed TDs are not split $\log N$ times but placed unsplit onto the private deque, and *(3)* the private deque can be allocated on the stack because its growth follows the depth-first exe-

cution and because postponed TDs are pushed unsplit onto it. More details can be found in my dissertation [39].

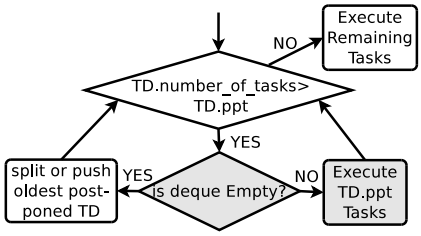

Figure 3. Processing a Task Descriptor with Lazy Scheduling

In our original paper on lazy scheduling [40], we did not push the oldest posponed TD because the added overhead of maintaining the private deque was not beneficial for our benchmarks on our experimental platform (XMT [43]). Instead, we pushed half of the newest TD. Bergstrom et al. [4] extended our implementation for a functional language whose arrays are implemented as trees, and they also pushed the newest TD without encountering scaling issues on commercial multicores. However, we show in [39] that failing to push the oldest postponed TD can harm the scalability of certain codes.

3.3 Measuring Performance Portability

We define two performance portability metrics, the *worst-case efficiency* and the *average efficiency*. The first one captures the minimum efficiency achievable, given a set of variables over which the programmer does not have control, such as the input and the target platform. The second metric captures the average efficiency in the same setup. To formally define those metrics, we start by defining *two groups of variables*: the variables the programmer controls and the variables *"nature"* controls. The first group includes:

$\mathbb{B}$: the set of implementations (including different algorithms) that we care to evaluate for the given problem.

$\mathbb{C}$: let C_b be the set of all possible (or reasonable) coarsenings for an implementation $b \in \mathbb{B}$, then $\mathbb{C} = \bigcup_{b\in\mathbb{B}} C_b$.

$\mathbb{S}$: the set of all possible system configurations, including different user-level schedulers, dynamic memory allocators, garbage collectors, and generally all the user-level "system code".

The variables nature controls and their corresponding sets are:

$\mathbb{I}$: the set of all inputs we care to evaluate.

$\mathbb{P}$: the set of all platforms we care to evaluate.

$\mathbb{W}$: let W_p be the set of all possible subsets of a platform $p \in \mathbb{P}$ that may be allocated for the execution, then $\mathbb{W} = \bigcup_{p\in\mathbb{P}} W_p$.

Nature-controlled variables are intended to capture performance portability: $\mathbb{I}$ across datasets D, $\mathbb{P}$ across platforms P, and $\mathbb{W}$ across contexts C and with multiprogramming. One can add or subtract variables from the above framework to adapt it to their needs. The important thing is to distinguish the variables the programmer controls from the ones they do not. We briefly discuss how to navigate the potentially infinite state space defined by the variables $(b, c, s; i, p, w)$ in [39].

$$\mathit{Eff}(b,c,s;i,p,w) = \frac{\min\limits_{b\in\mathbb{B}\ c\in C_b\ s\in\mathbb{S}} Time(b,c,s;i,p,w)}{Time(b,c,s;i,p,w)}$$

Efficiency Given a code $b \in \mathbb{B}$ using a coarsening $c \in C_b$, with system code $s \in \mathbb{S}$, an input $i \in \mathbb{I}$, on a platform $p \in \mathbb{P}$, using a subset $w \in W_p$, we define the ***efficiency*** of the tuple $(b, c, s; i, p, w)$ as the ratio of the best time achievable by varying the programmer controlled variables over the time achieved with (b, c, s). *For clarity in the notation, we use a semicolon to separate the variables the programmer controls from the ones they do not.* The variables controlled by nature (i, p, w) are unknown to the programmer and affect the best achievable performance, so they are left out of the minimizing clause. Intuitively, for each set-up nature throws at us, efficiency is the ratio of the achieved performance with a given programmer-controlled set-up, over the best achievable performance with any programmer set-up. Note that our definition of efficiency differs from habitual ones.

$$\mathit{Eff}_{WC}(b,c,s) = \min_{i\in\mathbb{I}\ p\in\mathbb{P}\ w\in W_p} \mathit{Eff}(b,c,s;i,p,w)$$

Worst Case Efficiency The worst-case efficiency for an implementation b with coarsening c using a system configuration s is defined as the minimum efficiency over the sub-space covered by the variables controlled by nature (i, p, w). In other words, the worst-case efficiency is the global minimum of the efficiency function for constant values of the programmer controlled variables (b, c, s).

$$\mathit{Eff}_{Mean}(b,c,s) = \sqrt[\gamma]{\prod_{i\in\mathbb{I}\ p\in\mathbb{P}\ w\in W_p} \mathit{Eff}(b,c,s;i,p,w)}$$

$$\gamma = |I| \cdot \sum_{p\in\mathbb{P}} |W_p|$$

Average Efficiency The average efficiency is defined similarly by replacing the minimum function by the geometric mean. We use the geometric mean because efficiency is a ratio. This definition assumes a uniform probability distribution over all nature-controlled variables, otherwise weights must be added. Conversely, worst-case efficiency never needs weights as it gives the minimum.

Now, given these two metrics the programmer can try to find a triplet (b, c, s) that maximizes one or possibly both of them. E.g.,

$$(b,c,s)_{WC} = \arg\max_{b\in\mathbb{B}\ c\in C_b\ s\in\mathbb{S}} \mathit{Eff}_{WC}(b,c,s)$$

$$(b,c,s)_{Mean} = \arg\max_{b\in\mathbb{B}\ c\in C_b\ s\in\mathbb{S}} \mathit{Eff}_{Mean}(b,c,s)$$

4. Results and Contributions

In this section, we show *(1)* that performance portability is sensitive to coarsening, *(2)* that using lazy scheduling to prune parallelism helps achieve good worst case efficiency (a performance portability metric), and *(3)* that lazy scheduling delivers superior performance on fine-grained codes when compared to TBB's Work Stealers while remaining competitive on coarse codes. We present results on an UltraSPARC-T2 platform with 8 cores and 8 hardware threads per core, running at 1.2GHz, with 4MB L3 cache and 32GB DDR2 RAM. The installed kernel was Solaris 5.10 and we compiled the TBB[1] C++ codes with the installed g++ 3.4.3 and with full -O3 optimizations. We implemented Lazy Scheduling within TBB, and we compare to TBB's default partitioner, AP (auto partitioner), because AP outperformed TBB's other options (simple and affinity partitioner) on our benchmarks. In [39], we show results on three more platforms, including XMT, designed for general-purpose portable parallel programming [33, 41]. XMT is as a scalable architecture [3, 8, 23–25, 43], that is programmable [22, 35, 38, 42] and achieves good performance even on irregular applications [7, 9, 10, 13, 15, 16]. The XMT cycle accurate simulator and compiler are publicly available [26].

4.1 Sensitivity to Coarsening

In this section, we show that a code as trivial as QUEENS (Alg. 1) can be hard to coarsen if the input and the platform subset are beyond programmer control. We use platform subsets $W_{T2} = \{1, 4, 8, 16, 24, 32, 40, 48, 56, 64\}$, inputs $i \in \{4, 6, 8, 10, 12\}$, TBB's AP, and several constant cut-off depths.

Fig. 4 shows the sensitivity of efficiency to the used platform subset for an 8 by 8 input. Even for the two best cut-offs (2 and

3), there is a platform subset that gets less than 80% efficiency, an indication that a constant cut-off is not performance portable.

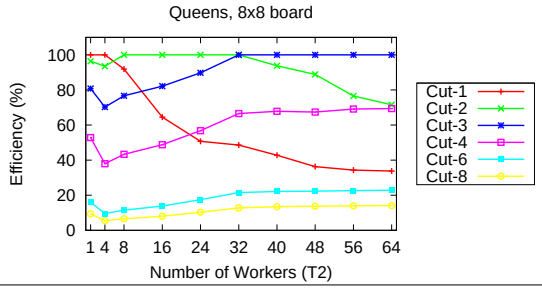

Figure 4. Sensitivity of Efficiency to Contexts (platform subsets)

Fig. 5 plots the efficiency as a function of the cut-off for different input sizes with 16 workers. Each input has a different optimal cut-off, and for all cut-offs there is an input that performs badly with it. Note that, for each N, cut-offs $c > N$ are not plotted, as they are equivalent to $c = N$.

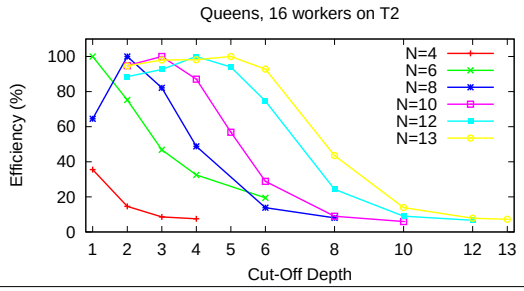

Figure 5. Sensitivity of Efficiency to Inputs

It should be clear by now that constant depth cut-offs result in poor performance portability. This is not surprising as it only prunes parallelism (poorly), but it does not amortize the OPT. In Table 1, we evaluate amortizing cut-offs of the form $N - k$, for some values of the constant k, as well as pruning cut-offs of the form N/k. The table also lists the partial worst case efficiency as a function of the input to show that using only pruning or amortizing cut-offs is not performance portable, but using a combination (e.g., min(N/3, N-5)) works well, reaching 71.6% worst-case efficiency. Furthermore, $N = 6$ is the smallest input that profits from parallelism which justifies $c = N - 5$ as the optimal amortizing cut-off.

$Eff_{WC}(QUEENS, c, AP; i)$

	Size of N (Side of the Board)					
Depth	4	6	8	10	12	$Eff_{WC}(c)$
N-4	100.0	64.6	38.0	28.4	24.3	24.3
N-5	100.0	96.9	70.3	55.5	45.9	45.9
N-6	100.0	50.2	71.6	84.8	74.5	50.2
N/2	8.5	29.6	38.9	55.5	74.5	8.5
N/3	17.4	64.6	71.6	92.2	98.9	17.4
min($^N/_3$,N-5)	100.0	96.9	71.6	92.2	98.9	71.6

$\max_{c \in \mathbb{C}} Eff_{WC}(QUEENS, c, AP) = 71.6\%$ with c=min($^N/_3$,N-5).

Table 1. Worst-Case Efficiency per input and overall.

4.2 Performance Portability of Lazy Scheduling

In this section, we will show that, by delegating pruning of parallelism to lazy scheduling, we achieve comparable worst-case efficiency to what we did after careful manual tuning in the previous section. Therefore, Lazy Scheduling improves programmability by relieving the programmer from pruning parallelism, and it improves portability by adaptively pruning parallelism based on runtime load conditions.

Our compiler can currently amortize the OPT only for simple parallel loops, but not for recursive codes like QUEENS. We manually find the amortizing cut-off to be $c = N - 5$, as we explained in the previous section (and elaborate in [39]). We envision this step to be automated by the compiler or auto-tuners in the future, as it is simpler than tackling the combined coarsening problem of amortizing and pruning. The worst case efficiency with the amortizing cut-off and Lazy Scheduling for pruning parallelism is 77.53%, which is comparable to what we got with careful manual tuning without Lazy Scheduling.

4.3 Performance of Lazy Scheduling

Below, we compare Lazy Scheduling (LS) with TBB's best scheduler for our set of benchmarks, Auto-Partitioner (AP). Table 2 shows the benchmark datasets categorized into declarative, where all parallelism is exposed, and corasened. The last column of the table shows the profitable parallelism threshold picked by our compiler to amortize the OPT with Lazy Scheduling. TSP is the traveling salesman problem, SpMV is sparse matrix vector multiplication, BFS is breadth first search, FW is an all-pairs shortest path algorithm, MM is matrix multiplication, and CONV is convolution.

Declarative	**Dataset**	**ppt**
QUEENS(decl)	N = 14 Nodes (no cut-off)	1
TSP(decl)	N = 12 Nodes (no cut-off)	1
SpMV(decl)	80Kx5K, 40M non-zero (40M tasks)	77
BFS	10K Nodes, 200K Edges	53
FW	N = 512 Nodes	91
Coarse(ned)	**Dataset**	**ppt**
QUEENS(cut)	N = 14 Nodes ($c = N/2$)	1
TSP(cut)	N = 12 Nodes ($c = N/2$)	1
SpMV(coarse)	80Kx5K, 40M non-zero (80K tasks)	1
MM	1024x1024 (N^2)	1
CONV	4Kx4K (N^2) image, 16x16 (M^2) filter	1

Table 2. Benchmark Summary

Fig. 6 shows the execution time of benchmarks using Lazy Scheduling as a percentage of the time when using AP. For declarative code, the automatic pruning of lazy scheduling gives a clear advantage over AP, with the exception of FW which does not have nested parallelism, so AP is competitive. Just as importantly, even with coarsened code, Lazy Scheduling manages to beat AP on average by a small margin.

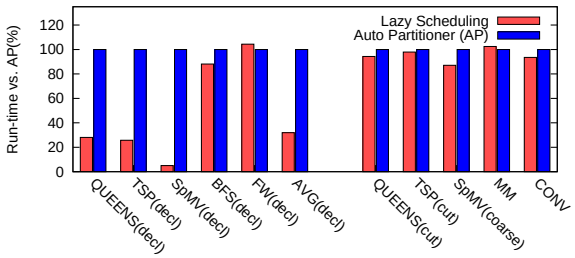

Figure 6. Lazy Scheduling vs. Auto Partitioner on T2

5. Conclusion

In this work, we distinguished two goals of coarsening, amortizing and pruning, and by tackling the second automatically with our lazy scheduling technique, we have improved programmability, since the remaining coarsening task is simpler for the programmer, and performance portability by adaptively deploying parallelism at runtime as needed. Furthermore, we proposed a framework and two metrics for quantifying performance portability, an important quality of code. As far as we know, these are the first concrete metrics for performance portability. Finally, while more work is needed, notably adequate hardware support (e.g., [28, 29, 43]), we believe this work makes a significant contribution towards efficiently supporting general-purpose performance portable parallel programs.

References

[1] Intel Threading Building Blocks. http://threadingbuildingblocks.org/.

[2] K. Agrawal, C. E. Leiserson, Y. He, and W. J. Hsu. Adaptive work-stealing with parallelism feedback. *ACM Transactions on Computer Systems*, 26:7:1–7:32, September 2008.

[3] A. O. Balkan, G. Qu, and U. Vishkin. Mesh-of-trees and alternative interconnection networks for single-chip parallelism. *IEEE Trans. Very Large Scale Integr. Syst.*, 17(10):1419–1432, 2009.

[4] L. Bergstrom, M. Rainey, J. Reppy, A. Shaw, and M. Fluet. Lazy tree splitting. In *Proc. of the Intl. Conf. on Functional Programming*, 2010.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999.

[6] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *Proc. of the Conf. on Functional Programming languages and Computer Architecture*, 1981.

[7] G. Caragea, F. Keceli, A. Tzannes, and U. Vishkin. General-purpose vs. GPU: Comparison of many-cores on irregular workloads. In *Proc. USENIX Workshop on Hot Topics in Parallelsim*, 2010.

[8] G. Caragea, A. Tzannes, F. Keceli, R. Barua, and U. Vishkin. Resource-aware compiler prefetching for many-cores. In *Proc. Intl. Symposium on Parallel and Distributed Computing*, 2010.

[9] G. C. Caragea and U. Vishkin. Brief announcement: better speedups for parallel max-flow. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, 2011.

[10] G. C. Caragea, B. Saybasili, X. Wen, and U. Vishkin. Performance potential of an easy-to-program pram-on-chip prototype versus state-of-the-art processor. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures*, 2009.

[11] O. Certner, Z. Li, P. Palatin, O. Temam, F. Arzel, and N. Drach. A practical approach for reconciling high and predictable performance in non-regular parallel programs. In *Proc. of the Conf. on Design, Automation and Test in Europe (DATE)*, 2008.

[12] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proc. of the Symp. on Parallelism in Algorithms and Architectures*, 2005.

[13] T. M. DuBois, B. Lee, Y. Wang, M. Olano, and U. Vishkin. XMT-GPU: A PRAM architecture for graphics computation. In *Proc. International Conference on Parallel Processing*, 2008.

[14] A. Duran, J. Corbalán, and E. Ayguadé. An adaptive cut-off for task parallelism. In *Proc. of the 2008 Conf. on Supercomputing*, 2008.

[15] J. A. Edwards and U. Vishkin. Better speedups using simpler parallel programming for graph connectivity and biconnectivity. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 103–114.

[16] J. A. Edwards and U. Vishkin. Brief announcement: Speedups for parallel graph triconnectivity. In *Proc. 24th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2012.

[17] M. Frigo, C. E. Leiserson, and R. K. H. The implementation of the Cilk-5 multithreaded language. In *Proc. of the Conf. on Programming Language Design and Implementation*, 1998.

[18] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proc. of the Symposium on Parallelism in Algorithms and Architectures*, 2009.

[19] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. Slaw: A scalable locality-aware adaptive work-stealing scheduler. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010.

[20] R. H. Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. In *Proc. of the Symp. on LISP and functional programming*, 1984.

[21] D. Hendler, Y. Lev, M. Moir, and N. Shavit. A dynamic-sized non-blocking work stealing deque. *Distributed Computing*, 18:189–207, February 2006.

[22] L. Hochstein, V. R. Basili, U. Vishkin, and J. Gilbert. A pilot study to compare programming effort for two parallel programming models. *Journal of Systems and Software*, 81:1920–1930, November 2008.

[23] M. N. Horak, S. M. Nowick, M. Carlberg, and U. Vishkin. A low-overhead asynchronous interconnection network for GALS chip multiprocessors. In *Proc. Intl. Symposium on Networks-on-Chip*, 2010.

[24] F. Keceli, T. Moreshet, and U. Vishkin. Thermal management of a many-core processor under fine-grained parallelism. In *Proc. 5th Workshop on Highly Parallel processing on Chip(HPPC 2011)*, Bordeaux, France, August 2011. In conjunction with Euro-Par.

[25] F. Keceli, T. Moreshet, and U. Vishkin. Power-performance comparison of single-task driven many-cores. In *17th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2011.

[26] F. Keceli, A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin. Toolchain for Programming, Simulating and Studying the XMT Many-Core Architecture. In *16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (in conjunction with IEEE IPDPS)*, Anchorage, Alaska, USA, May 2011.

[27] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. Mul-T: a high-performance parallel lisp. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, 1989.

[28] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *Proc. of the 34th annual International Symposium on Computer Architecture*, 2007.

[29] E. Ladan-Mozes, I.-T. A. Lee, and D. Vyukov. Location-based memory fences. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, 2011.

[30] D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, 2000.

[31] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Proc. of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, 2009.

[32] C. E. Leiserson. The Cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, 2009.

[33] D. Naishlos, J. Nuzman, C.-W. Tseng, and U. Vishkin. Towards a first vertical prototyping of an extremely fine-grained parallel programming approach. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, 2001.

[34] OpenMP Architecture Review Board. OpenMP Application Program Interface, Ver. 3.1 July 2011.

[35] D. Padua, U. Vishkin, and J. C. Carver. Joint UIUC/UMD parallel algorithms/programming course. In *Proc. NSF/TCPP Workshop on Parallel and Distributed Computing Education, in conjunction with IPDPS*, 2011.

[36] H. Pan, B. Hindman, and K. Asanović. Composing parallel software efficiently with Lithe. In *Proc. of the 2010 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2010.

[37] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in TBB. In *Proc. Intl. Parallel and Distributed Processing Symp. (IPDPS)*, 2008.

[38] S. Torbert, U. Vishkin, R. Tzur, and D. J. Ellison. Is teaching parallel algorithmic thinking to high school students possible?: one teacher's experience. In *Proceedings of the 41st ACM technical symposium on Computer science education*, SIGCSE '10, pages 290–294, 2010.

[39] A. Tzannes. *Enhancing Productivity and Performance Portability of General-Purpose Parallel Programming*. PhD thesis, University of Maryland at College Park, 2012.

[40] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *Proc. of the 15th Symp. on Principles and Practice of Parallel Programming*, 2010.

[41] U. Vishkin. Using simple abstraction to reinvent computing for parallelism. *Commun. ACM*, 54:75–85, Jan 2011.

[42] U. Vishkin, G. C. Caragea, and B. C. Lee. *Handbook of Parallel Computing: Models, Algorithms and Applications*, chapter Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform. CRC Press, 2007.

[43] X. Wen and U. Vishkin. FPGA-based prototype of a PRAM-on-chip processor. In *Proc. of the conference on Computing Frontiers*, 2008.