Elastic Tasks: Unifying Task Parallelism and SPMD Parallelism with an Adaptive Runtime

Alina Sbîrlea[1] and Kunal Agrawal[2] and Vivek Sarkar[1]

[1] Rice University, USA
[2] Washington University in St Louis, USA

Abstract. In this paper, we introduce *elastic tasks*, a new high-level parallel programming primitive that can be used to unify task parallelism and SPMD parallelism in a common adaptive scheduling framework. Elastic tasks are internally parallel tasks and can run on a single worker or expand to take over multiple workers. An elastic task can be an ordinary task or an SPMD region that must be executed by one or more workers simultaneously, in a tightly coupled manner.

This paper demonstrates the following benefits of elastic tasks: (1) they offer theoretical guarantees: in a work-stealing environment computations complete in expected time $O(W/P + S + E \lg P)$, where E=# of elastic tasks, W=work, S=span, P=# cores. (2) they offer performance benefits in practice by co-scheduling tightly coupled parallel/SPMD subcomputations within a single elastic task, and (3) they can adapt at runtime to the state of the application and work-load of the machine.

We also introduce ElastiJ — a runtime system that includes work-sharing and work-stealing scheduling algorithms to support computations with regular and elastic tasks. This scheduler dynamically decides the allocation for each elastic task in a non-centralized manner, and provides close to asymptotically optimal running times for computations with elastic tasks.

1 Introduction

As multicore machines become ubiquitous, *task parallelism* has emerged as a dominant paradigm for parallel programming. Many programming languages and libraries such as Cilk [10], Cilk Plus [1], Intel TBB [21], .Net Task Parallel Library [17], and Habanero-Java [5] support this paradigm, in which the programmer expresses the logical parallelism by specifying sequential tasks and their dependences, and a runtime scheduler maps the tasks on available processors.

SPMD parallelism [8] is an alternate paradigm for exploiting multicore parallelism, in which a fixed number of worker threads execute a single SPMD region. There is general agreement that SPMD parallelism can outperform task parallelism in certain cases, but task parallelism is more general than SPMD parallelism. There has even been work on compiler optimizations to automatically transform fork-join regions of code to SPMD regions for improved performance [7,18]. However, to the best of our knowledge, there has been no prior work that combines task parallelism and SPMD parallelism in a single adaptive runtime framework.

In this paper, we propose *elastic tasks*, a new primitive that helps bridge the gap between task parallelism and SPMD parallelism. An elastic task u is

defined by: (1) $w(u)$ — the execution requirement (work) of u; and (2) $c(u)$ — the maximum number of workers (capacity) that task u can use. Elastic tasks are assumed to have linear scaling with $c(u)$ and to exhibit locality benefits from co-scheduling their internal parallelism, but they need not be data parallel. The user need only provide two additional parameters for u, both being an estimation rather than exact values: (1) the approximate length of the task on one worker as $w(u)$; and (2) the average parallelism of the task as $c(u)$. When an elastic task starts, it is assigned $a(u) \leq c(u)$ dedicated worker threads, which work on only this task until the task completes. An elastic task with $c(u) = 1$ is just like an ordinary task. An elastic task with $c(u) > 1$ is like an SPMD region which must be executed by one or more workers simultaneously in a tightly coupled manner.

We extend the work-sharing and work-stealing strategies [4] to handle computations with elastic tasks. We prove that the work-sharing scheduler completes a computation with work W and span S in $O(W/P + S + E)$ time, where E is the total number of elastic tasks. Similarly, the work-stealing scheduler completes the computation in $O(W/P + S + E \lg P)$ expected time.

Previous work, notably Wimmer and Träff [26] have considered a construct for mixed-mode parallelism, but, in their work, the number of workers assigned to a task is fixed, and user specified. In our scheduling strategy, if most workers are busy, then the elastic task is assigned fewer workers, since there is already ample parallelism in the rest of the program. If most workers are idle (or stealing), then it indicates a lack of parallelism and more workers are assigned to an elastic task. Finally, we are not aware of prior work that provides theoretical guarantees on the completion time for this form of combination of sequential and elastic tasks.

We have implemented a runtime system which implements elastic tasks, and include experimental results obtained from the work-stealing implementation.

In summary, the contributions of this paper are:
- the elastic task primitive, its definition, properties and requirements,
- theoretical proofs for the work-sharing and work-stealing runtimes,
- the ElastiJ runtime which executes computations with both sequential and elastic tasks and automatically decides the number of workers assigned to an elastic task,
- experimental results which indicate they provide locality benefits for certain computations and provide runtime adaptability.

The rest of the paper is organized as follows: Section 2 discusses the motivation for elastic tasks, Section 3 gives the theoretical proofs, Section 4 describes the implementation details of ElastiJ, Section 5 presents the experimental results, Section 6 discusses related work and finally Section 7 concludes.

2 Motivation for Elastic Tasks

2.1 Benefits of Elasticity

Given a computation expressible as an elastic task, we have a few other alternatives. First, we could create a *sequential task*. However, this can increase the critical path length (span) of the computation, thereby decreasing its parallelism.

Second, we could create an *inelastic task* where the programmer specifies the number of workers (say $m(u)$) which must execute node u (as in [26]). In this case, if the programmer accurately specifies $m(u)$ to be large, then the scheduler must find all these workers. If most workers are busy, the scheduler must either wait for a potentially long time (idle workers for long periods) or it must interrupt workers in the middle of their execution leading to large overheads. If the programmer artificially specifies $m(u)$ to be small to avoid this, then we are potentially wasting the internal parallelism of task u and decreasing the scalability of the overall computation, as with sequential tasks. Also, it is difficult to guarantee good performance theoretically for inelastic tasks.

Third, we could convert the task to a *task parallel computation* by dividing up the computation into independent tasks that need not be co-scheduled. This may be cumbersome if the different tasks need to communicate, since we must add a control synchronization point for every communication link. This also increases overheads; barriers within independently-scheduled tasks can be very inefficient. In addition, as discussed next, this transformation means that different iterations may be executed at very different times, leading to loss in locality.

2.2   Benefits of Co-Scheduling

Compared to sequential and inelastic tasks, with elastic tasks the programmer is only responsible for providing the capacity, not the precise number of workers. The runtime then adjusts the number of workers allocated to the task based on runtime conditions. Further we compare elastic tasks to task parallel computations.

Cache Locality on Multicores: Consider a loop in which all iterations access the same data. Using an elastic task forces all the iterations to be scheduled at the same time, so the shared data will be brought into the shared cache. Instead, had we converted this loop into a task parallel program, all the iterations would have been their own task, possibly scheduled at different times. Since other tasks that access other data may execute between different iterations of the loop, the shared data may have to be brought in multiple times leading to poor cache locality. We will show experimental results that validate this intuition in Section 5.

Locality on Future Architectures: While in this paper we show the importance of elastic tasks on multicores, we expect elastic tasks to become even more valuable for future extreme scale systems where collocation based on data sharing will be critical for performance. Additionally, an elastic task with data-parallel computations can also be automatically transformed into a GPU kernel using existing tools [16,3,9]. The adaptability of elastic tasks to task granularity also implies that applications can adjust to existing and future GPUs.

3   Theoretical Guarantees

We briefly state the theoretical guarantees proven in our technical report [22].

3.1   Model of Computation

Elastic tasks and normal sequential tasks are *processor oblivious* computations — the programmer expresses the logical parallelism and the *runtime scheduler* dynamically schedules and executes the computation on P worker threads.

The computation can be abstractly expressed as a computation DAG G; nodes are computation kernels and edges are dependences between nodes. A node is *ready* to execute when all its predecessors have been executed. Without loss of generality, we assume the maximum out-degree of any node ≤ 2. There are two types of nodes: *strands* — sequential chains of instructions and *elastic nodes*.

An elastic node u has the following properties: (1) Work $w(u)$ is its execution time on one worker. (2) Capacity $c(u)$ is its maximum internal parallelism. (3) Before an elastic node u can execute, the runtime scheduler must allocate it $1 \leq a(u)$ dedicated worker threads. Once u starts executing, these $a(u)$ workers can not work on anything else until u completes, and no other workers can work on u's work. (4) We assume that each elastic node provides linear speedup up to $c(u)$ workers and no speedup thereafter. When it is allocated $c(u)$ workers, we say that the node is *saturated*; otherwise we say that it is *unsaturated*.

As with traditional task parallel processor oblivious computations, we define two parameters for G. *Work W* is the sum of the computational requirements of all nodes or the execution time on one processor. *Span* is the weighted length of the longest path in the DAG where each node's weight is equal to its span. Span can also be seen as the execution time of the computation on an infinite number of processors. The parallelism of the program is defined as W/S and describes the maximum number of workers the computation can use effectively. Note that the execution time of the computation on P workers is at least $\max\{W/P, S\}$.

3.2 Theoretical Guarantees in a Work-Sharing Runtime

Theorem 1. *Given a computation graph with work W, span S, and E elastic nodes, the execution time of this computation on P workers using the work-sharing scheduler is $O(W/P + S + E)$.*

The above theorem states that the work-sharing scheduler provides linear speedup, and we prove this in the extended version of the paper [22].

3.3 Theoretical Guarantees in a Work-Stealing Runtime

Extending the Work-Stealing Scheduler with Elastic Nodes In a regular work-stealing scheduler, a program is executed by P workers each having its own private deque of ready nodes. At any time, a worker p may have a node u **assigned** to it. When a worker finishes u, if p's deque is empty, then p becomes a thief, selects another worker p_1 as a **victim** at random and tries to steal from the top of p_1's deque. If p_1's deque is not empty and p gets a node, then the steal attempt is successful, otherwise p tries to steal again. Blumofe and Leiserson [4] prove that this randomized work-stealing scheduler finishes a computation with work W and span S in expected time $O(W/P + S)$ time on P worker threads.

For work-stealing schedulers with elastic nodes, a worker's assigned node may be a strand or an elastic node. The changes due to the elastic nodes affect what happens on steals and when a worker is assigned an elastic node:

1. If p picks up an elastic node u, p starts waiting on u, instead of starting execution.

2. When p is a thief, it randomly chooses a victim q. If q is waiting on an elastic node u, then u is also assigned to p and p also starts waiting on it. At this time $a(u)$ is incremented by 1. Otherwise, p steals the node at the top of q's deque; if the deque is empty then p tries again.
3. While u is waiting, its total waiting time $wait(u)$ is incremented by $a(u)$ in every time step.
4. An elastic node starts executing when either $a(u) = c(u)$ — the node is saturated; or its total wait time $wait(u) \geq w(u)$.
5. When an elastic node finishes executing, the worker that first enabled the elastic node enables its children. All other workers assigned to the elastic node start work stealing, as all their deques are empty at this time.

Analysis of Work-Stealing Scheduler

Theorem 2. *Given a computation graph with E elastic nodes, work W and span S, the expected execution time of this computation on P workers using the work-stealing scheduler is $O(W/P + S + E \lg P)$.*

If we compare this result to the result for computations without elastic nodes, we notice that the additional term is only $E \lg P$. This term is negligible for any computation where the number of elastic nodes is $O(T_1/P \lg P)$ — which implies that most elastic nodes have parallelism $\Omega(P)$ and at most $1/\lg P$ fraction of the work of the computation is contained in elastic nodes.

We mention that the constant factors hidden within the asymptotic bounds are not much larger than those hidden within the standard work-stealing bounds. An additional terms similar to $O(E \lg P)$ also appears in standard work-stealing if we consider the contention on the child counter (generally ignored).

In this section, without loss of generality, we assume that each strand is a unit time computation. A longer strand is simply expressed as a chain of unit time strands. We separately bound the types of steps that a worker can take at any time step. A worker could be working, waiting on an elastic node or stealing. The total number of work-steps is at most W; and the total number of waiting steps is at most $W + PE$. Therefore, we need only bound the steal steps.

We classify steal attempts in three categories: (1) regular steal attempts occur when no elastic node is waiting and no unsaturated elastic node is executing. (2) waiting steal attempts are those that occur when some elastic node is waiting. (3) irregular steal attempts occur when some unsaturated elastic node is executing and no elastic node is waiting. We will bound the number of steal attempts in these three categories separately.

Intuition for the Analysis We adopt a *potential function* argument similar to Arora et al.'s work-stealing analysis [2], henceforth referred to as ABP. In the ABP analysis, each ready node is assigned a potential that decreases geometrically with its distance from the start of the dag. For traditional work stealing, one can prove that most of the potential is in the ready nodes at the top of the deques. Therefore, $\Theta(P)$ random steal attempts suffice to process all the nodes on top of the deques. Therefore, one can prove that $O(PS)$ steal attempts are sufficient

to reduce the potential to 0 in expectation. The ABP analysis does not directly apply to bounding the number of steal attempts for computations with elastic nodes because a steal may turn into a wait if the victim p has an assigned node u. Since u may contain most of the potential (particularly if p's deque is empty), and u cannot be stolen, steals are not longer effective in reducing the potential until u completes. Therefore, we must use a different argument to bound the steal attempts that occur while u is assigned to p.

Regular Steal Attempts: These occur when either a worker is assigned an elastic node (the normal ABP argument applies) or any elastic node that is assigned is saturated and is executing. We use a potential function argument very similar to the ABP argument, but on an augmented DAG in order to account for steal attempts that occur while a saturated elastic node is executing.

Waiting Steal Attempts: These occur when some elastic node (say u) is waiting — at this time, u is assigned to some worker(s), say p and p'. If any worker q tries to steal from p or p' during this time, then q also starts waiting on u and $a(u)$ increases by 1. Therefore, only a small number of steal attempts (in expectation) can occur before $a(u) = c(u)$ and u becomes saturated and stops waiting. We use this fact to bound the number of waiting steal attempts.

Irregular Steal Attempts: These occur when no elastic node is waiting and some unsaturated elastic node is executing. The analysis here is similar to the one we used to account for idle steps in the work-sharing scheduler (see [22]). Since this elastic node started executing without being saturated, it must have waited for at least $w(u)$ time — and during this time, all the workers not assigned to this elastic node were busy doing either work or waiting steal attempts. Therefore, any steal attempts by these workers can be amortized against these other steps.

The formal analysis can be found in the full technical report [22]. Below we provide the essential components that lead to the work-stealing bound.

Lemma 1. *Total number of regular steal attempts is $O(PS + P\lg(1/\epsilon))$ in expectation.*

Lemma 2. *There are $O(P\min\{\lg c(u), \lg P\})$ steal attempts in expectation while a particular elastic node u is waiting. Therefore, the expected number of waiting steal attempts over the entire computation is $O(PE\lg P)$.*

Lemma 3. *The total number of irregular steal attempts is at most $O(W + PE\lg P)$.*

Proof of Theorem 2: Combining Lemmas 1, 2 and 3 tells us that the total number of steal attempts over the entire computation is $O(W + PS + PE\lg P)$. In addition, we know that the total number of work steps and waiting steps is at most $O(W + PE)$. Therefore, if we add all types of steps and divide by P (since we have P worker threads and each take 1 step per unit time), we get the overall expected running time of $O(W/P + S + E\lg P)$. □

4 Implementation details

We created ElastiJ, a system that supports the creation and execution of elastic tasks. ElastiJ is build on top of the Habanero-Java library (HJlib) [14].

Elastic tasks are created using *asyncElastic*, a regular spawn call (*async* in HJlib) with two additional parameters: *work* - $w(u)$ and *capacity* - $c(u)$. The capacity of a task is the approximation of its average parallelism — for any task we can simply assume that it is the task's work ($w(u)$) divided by its critical path length (or span). The work $w(u)$ is the total running time of the task on 1 processor. Many tools exist to measure parallelism; e.g., for CilkPlus programs, Cilkview can be used to get $c(u)$. Two additional optional parameters: $(b(u), e(u))$ can be used to describe the computation much like describing the iteration space for a loop. The runtime divides this range and assigns non-overlapping ranges to the workers executing the elastic node, similar to OpenMP's loop static scheduling [19], except that the number of workers is dynamic. For non-data-parallel computations these values can have a different meaning or simply not be used; e.g., in Section 5.2 we use Quicksort, where the partition phase - a computation that is not data parallel - is implemented as an elastic task.

ElastiJ uses a from-scratch work-sharing or work-stealing runtime implemented in Java. We use the work-stealing runtime in our results due to its better performance as shown in [10,12]. The runtime executes as described in Section 3.3, but the mechanism threads join an elastic task differs. The simplest approach is as follows: the first thread starts to wait $w(u)$, the second thread wakes up the first and both of them continue to wait for half of the remaining time, and so on. This approach causes a lot of overhead due to the several sleep and notify calls, in particular when many threads want to join at once. A better approach is for the first thread to wait $w(u)$ and store his wait start time. The second thread uses this wait time to compute how much the first thread already waited, and waits half of the remaining time, also storing the time he starts the wait. The process goes on until the last thread either saturates the task or has waited the remaining fraction of time and it wakes up all threads. This second approach has the advantage that each thread only goes to sleep once and is woken up once when it can start the work. However it also experiences a lot of overhead due to context-switching when the method wait is called. Since the waiting threads do not perform any work, a more efficient approach observed in practice is a bounded busy-wait approach. This third approach is the one we used in our experiments. In addition, the runtime uses as the total wait time a fraction of the full estimated work given to the task, in order to account for the constant factor in the theoretical proof, and thus offer competitive performance. The *asyncElastic* construct we propose also includes an implicit phaser [23] for all workers that join the task. The use of phasers instead of barriers can provide additional performance benefits [18].

5 Experimental results

In this section, we use a series of benchmarks to demonstrate how elastic tasks perform in different scenarios. We first assess if elastic tasks have the potential to provide better locality using a synthetic micro-benchmark. Next, we use Quicksort and FFT algorithms to demonstrate that elastic tasks are easily integrated into task-parallel recursive programs and provide easy adaptability to task granularity. Finally, we evaluate the performance of ElastiJ using a set of single-level fork-join benchmarks from the IMSuite [13] set; we show little or no overhead from using

elastic tasks and analyze their sensitivity to applications and parameters. The performance results were obtained on two platforms: (1) IBM POWER7: node with four eight-core POWER7 chips running at 3.86GHz, with 4 MB L3 cache per chip, and (2) Intel Westmere: node with 12 processor cores per node Intel Xeon X5660 running at 2.83 GHz.

5.1 Benefit from locality

In Section 2, we described scenarios when elastic tasks give locality benefits. In this section, we evaluate this hypothesis by creating the following synthetic fork-join style application: the benchmark spawns n tasks of type ua in a single finish scope, where n is a multiple of the number of cores P. Each ua task accesses the same M elements of a vector A. Each ua task spawn P ub tasks, and each ub task accesses the same M elements of a different vector B. The capacity $c(u)$ of each elastic task is P, the maximum number of machine cores. All experiments in this section were obtained on the POWER7 (P=32). The program accepts a parameter to set the fraction α of ua tasks that are elastic tasks, and creates $n \times \alpha$ elastic tasks and $(1 - \alpha) \times n$ regular tasks (strands). The program spawns α elastic tasks using $asyncElastic$, and $(1 - \alpha)$ simulated elastic tasks using $async$. This simulation essentially means creating P regular tasks for each elastic task.

We expect a locality benefit due to the co-scheduling of the elastic tasks since all the workers executing the elastic task access the same array A. Therefore, it is likely that A will remain in the shared cache while this elastic task is executing. On the other hand, when we convert this elastic task into a normal task-parallel for-loop with P strands, these P strands may execute far apart in time — therefore, some of the ub tasks may pollute the cache in the meantime. The results below show this makes a difference when M is large enough to cause the ub tasks to evict part of the A data from L3 cache.

We run experiments by varying the fraction of elastic tasks: $0\% \leq \alpha \leq 100\%$ (0% means all tasks are regular tasks, while 100% means all tasks are elastic). All experiments report the average of the 20 best iterations over 3 runs, each with 30 iterations, to remove the JVM warm-up and variability between runs [11].

We set the size of the arrays to $M = 1,000,000$; with A being an array of integers this adds to 4MB of data. Note that we ignore the L1 and L2 caches, under the claim that the size chosen is large enough to ensure at least some evictions from L3 cache. The results of the experiments are shown in Figure 1a. We notice that for elastic tasks the execution time remains essentially constant, while for regular tasks the performance is degrading as we increase their number. We ran an experiment with identical task size but with $M = 64.000$, and noticed constant performance when varying from 0-100% so the action of splitting the task cannot be the cause. We therefore go back to what we inferred and assume the data accessed by tasks ua is being invalidated by tasks ub. We use the $perf$ tool on the POWER7 to confirm this. Figure 1b plots the cache misses obtained by the $perf$ tool. We see that the number of cache misses increases up to $1.7\times$ when using regular tasks as opposed to elastic tasks.

We conclude that the use of elastic tasks should be used in a setting where their granularity amortizes the overhead of setting them up and that they can offer performance benefits due to improved locality.
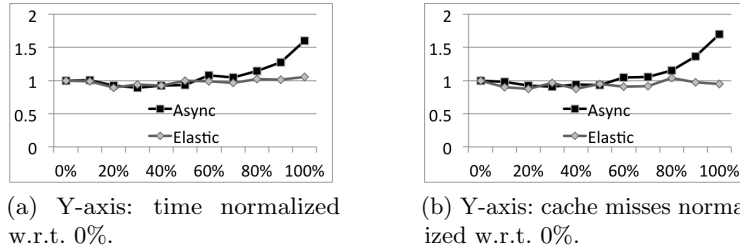
(a) Y-axis: time normalized w.r.t. 0%.

(b) Y-axis: cache misses normalized w.r.t. 0%.

Fig. 1. Microbenchmark comparing Elastic vs. Async task scheduling. X axis:% of elastic tasks.

5.2 Adaptability

We argue that elastic tasks provide the benefit that they integrate seamlessly with normal task parallelism which is suited to recursive algorithms. The quicksort algorithm is a recursive divide-and-conquer algorithm and can be easily expressed as a task-parallel program using async-finish or spawn-sync constructs. However, the divide step consists of a partition step. While partition can be expressed as a divide-and-conquer algorithm, that is a cumbersome implementation and many parallel implementations of quicksort simply use a sequential partition algorithm.

In this section, we evaluate the benefit of implementing the partition algorithm as an elastic task [25]. Note that the partition step is called at every level of recursion. The advantage of using elastic tasks for expressing it is that the number of workers assigned to this partition step is decided at runtime. At the shallow levels of the recursion tree where there is not much other work being done, the partition step will be automatically assigned more workers. At deeper levels of the tree, when there is already enough other parallel work available, the partition will execute mostly sequentially. Therefore, elastic tasks provide automatic adaptability without any intervention needed by the programmer.

In Figure 2a, we compare elastic and async parallel implementations for quicksort with a parallel partition phase implemented with N asyncs, with N=# of machine cores, or as an elastic task. We present the results normalized w.r.t. the aync runs. In Figure 2b, we compare two implementations of FFT, where the recombine phase is implemented either as a binary tree spawn using asyncs or as an elastic task. The data sets we use are 10^7 and 10^8 for quicksort, and 2^{22} and 2^{23} for FFT. For both quicksort and FFT we used a rough estimate for the work based on the array length; additional opportunities exist for auto-tuning the work based on the input, recursion depth, etc. We present results for both benchmarks on the POWER7 and the Westmere, by using the reporting the average of the best iterations out of 90 iterations, using 3 JVM invocations [11]. We get up to 70% gain for quicksort and up to 16% gain for FFT from using elastic tasks. We also note that the gains are larger when the datasizes increase a trend that we believe will lead to increased benefits in larger scale applications.

5.3 Sensitivity Analysis

We use 8 benchmarks from the IMSuite Benchmark Suite [13] to compare regular task spawning with the creation of elastic tasks. These benchmarks have a single-
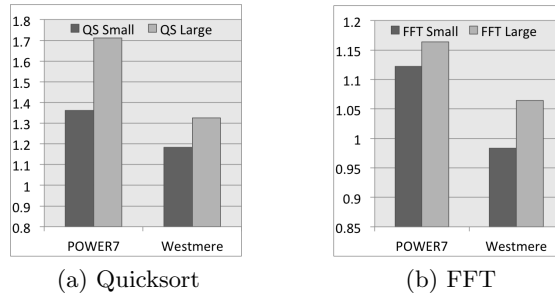
(a) Quicksort   (b) FFT

Fig. 2. Elastic task runs normalized over Async runs. > 1 means elastic runs are better.

level fork-join structure, i.e., there is at most a single forAll loop active at a time, which means they do not offer any opportunity for benefits from elasticity. Our goal is to demonstrate on-par performance with regular work stealing (Async Finish primitives). We discover that elastic tasks can offer benefits even in this scenario due to their resilience to the theoretical assumptions and to the elastic task parameters. The benchmarks are: Bellman Ford, Byzantine Agreement, Dijkstra, General Leader Election, Maximum Independent Set, Vertex Coloring, k-Committee and Minimum Spanning Tree [13].

Figures 3 gives the results on the Westmere platforms (see the technical report [22] for full experimental analysis). Figure 3a shows better performance using
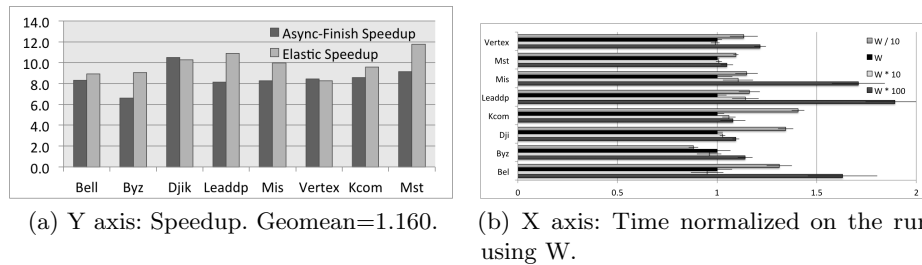

(a) Y axis: Speedup. Geomean=1.160.   (b) X axis: Time normalized on the run using W.

Fig. 3. IMSuite results on Westmere. (a) The geomean is for elastic times normalized over Async runs (>1 if elastic runs are better).

elastic tasks for most benchmarks. The reason is that these benchmarks fail to scale linearly past 8 cores and our work estimation offers better performance by selecting fewer cores than the maximum. This means that elastic tasks can be used to tune applications even when the theoretical guarantee of linear scaling does not hold. So we looked into how sensitive the applications are to the API parameters, in particular the estimated work (restricting the capacity only limits parallelism so it makes no sense to restrict it). Figure 3b shows that the times are large for the W/10 case, which is expected, since a small estimation means not enough wait time for threads to join the task, thus wasting parallelism. Conversely, with W large we delay the computation, which leads to two combined causes

for performance degradation: a longer wait time and a larger running time on 12 cores. Overall, the running time variation when W is varied is small and in many cases the added overhead is at most 10%. In the few cases where the percentage-wise variation is larger, the absolute time is very small([22]).

We conclude our overall results with the following observations: (a) elastic tasks provide a common API for expressing task and SPMD parallelism and are straightforward and easy to use; (b) they can benefit from locality due to their coscheduling property; (c) they can be used to adapt the degree of parallelism for recursive benchmarks; (d) they can offer comparable performance with forall constructs, and are fairly resilient to the theoretical assumptions of elastic tasks and to the user-provided parameters.

6   Related work

The closest related work is described as team-building [26] where programmers express inelastic tasks and the runtime allocates all workers. As mentioned in Section 1, our approach is quite distinct and we give theoretical proofs that our enhanced scheduler gives the same asymptotic bounds as regular work-stealing.

The problem of thread scheduling for locality improvements has been investigated in depth for work-stealing schedulers [12,20,6], both shared memory [12] and distributed [20,6]. ADAPT [15] proposes a framework implemented at OS level, which adapts the number of threads in a program, on machines where multiple programs concurrently. These works are orthogonal to elastic tasks.

A previous study [24] looked at a series of works which attempt to do coscheduling of tasks in general; this requires a coordinated effort which adds a lot of overhead in the absence of additional information. The authors formulate a mathematical model to analyze current approaches. Our work eases the challenge of coscheduling of tasks, when computations are expressible as an elastic task, and it can also be coupled with existing strategies. A more restrictive form of co-scheduling is gang scheduling, generally used for inelastic tasks.

7   Conclusions

In this paper we introduced the concept of elastic tasks, a construct that enables programmers to express adaptive parallel computations and rely on an elastic runtime to offer good performance from locality and load balancing. We proved that the work-stealing scheduler completes the computation in $O(W/P + S + E \lg P)$ expected time, where E is the total number of elastic tasks. We also showed practical results, that elastic tasks have the potential of improving the locality of computations, can yield comparable performance with regular tasks and that they are able to adapt at runtime, based on the load of the application.

We are interested in extending our approach to take more of the machine topology into account: the locality benefits of elastic tasks should be more pronounced when workers assigned to an elastic task share a certain level of proximity. As discussed in Section 2, elastic tasks are potentially useful for writing portable applications for heterogeneous machines that contain CPUs and GPUs and on distributed systems. Finally, elastic tasks have a potential to be useful for multiprogramming environments.

References

1. Agrawal, K., et al.: Executing task graphs using work-stealing. In: IPDPS ('10)
2. Arora, N.S., et al.: Thread scheduling for multiprogrammed multiprocessors. In: SPAA ('98)
3. Auerbach, J.S., Bacon, D.F., Cheng, P., Rabbah, R.M.: Lime: a java-compatible and synthesizable language for heterogeneous architectures. In: OOPSLA ('10)
4. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM 1999 (1999)
5. Cavé, V., et al.: Habanero-Java: the New Adventures of Old X10. In: PPPJ'11 ('11)
6. Chen, Q., Guo, M., Guan, H.: Laws: Locality-aware work-stealing for multi-socket multi-core architectures. In: ICS ('14)
7. Cytron, R., et al.: A Compiler-Assisted Approach to SPMD Execution. SC ('90)
8. Darema, F., George, D., Norton, V., Pfister, G.: A Single-Program-Multiple-Data Computational model for EPEX/FORTRAN. Parallel Computing 7(1), 11–24 (1988)
9. ExascaleHabanero: Habanero C, https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C
10. Frigo, M., et al.: The implem. of the Cilk-5 multithreaded lang. In: PLDI ('98)
11. Georges, A., et al.: Statistically Rigorous Java Perf. Evaluation. In: OOPSLA ('07)
12. Guo, Y., et al.: SLAW:Scalable Locality-aware Adaptive Work-stealing Scheduler. In: IPDPS ('10)
13. Gupta, S., Nandivada, V.K.: IMSuite: A Benchmark Suite for Simulating Distributed Algorithms. ArXiv e-prints (Oct 2013)
14. Imam, S., Sarkar, V.: Habanero-Java Library: a Java 8 Framework for Multicore Programming. In: PPPJ ('14)
15. Kumar Pusukuri, K., et al.: Adapt: A framework for coscheduling multithreaded programs. ACM Trans. Archit. Code Optim. (Jan 2013)
16. Luk, C.K., Hong, S., Kim, H.: Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: MICRO ('09)
17. Microsoft: "MSDN Magazine: Task Parallel Library", Retrieved 2014-09-11, http://msdn.microsoft.com/en-us/magazine/cc163340.aspx
18. Nandivada, V.K., et al.: A transformation framework for optimizing task-parallel programs. ACM Trans. Program. Lang. Syst. 2013 (2013)
19. OpenMP Architecture Review Board: The OpenMP API specification for parallel programming, version 4.0 (July 2013)
20. Paudel, J., Tardieu, O., Amaral, J.N.: On the merits of distributed work-stealing on selective locality-aware tasks. In: ICPP ('13)
21. Reinders, J.: Intel threading building blocks. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edn. (2007)
22. Sbirlea, A., et al.: Elastic Tasks: Unifying Task Parallelism and SPMD Parallelism with an Adaptive Runtime. Research Report TR15-02, Rice University (2015), http://engr.rice.edu/uploadedFiles/Tech_Reports/TR15-02_Elastic_Tasks.pdf
23. Shirako, J., Peixotto, D.M., Sarkar, V., Scherer, W.N.: Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In: ICS ('08)
24. Squillante, M.S., et al.: Modeling and analysis of dynamic coscheduling in parallel and distributed environments. In: SIGMETRICS ('02)
25. Tsigas, P., Zhang, Y.: A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000. In: PDP ('03)
26. Wimmer, M., Träff, J.L.: Work-stealing for mixed-mode parallelism by deterministic team-building. In: SPAA ('11)