Unleashing Parallelism with Elastic-Barriers

AMIT TIWARI, Department of CSE, IIT Madras, India
V. KRISHNA NANDIVADA, Department of CSE, IIT Madras, India

With the rise of multi-core processors, parallel programming has become essential, and managing synchronization overheads has become crucial for efficiency. Barriers, commonly used to synchronize threads, divide the program into different phases. Existing scheduling schemes address intra-phase load imbalance to some extent but do not fully resolve the issue of thread idling, especially in the context of programs with irregular parallel for-loops. This paper proposes an innovative solution called the *elastic-barrier*, where threads that arrive early at a barrier can execute iterations of the parallel-loop from the next phase, thereby reducing the idle time, and reduce load imbalance. The approach guarantees safety by ensuring that before executing any work W from the subsequent phase, all the works in the current phase that W depends on have been executed. The paper presents a compilation scheme that integrates compile-time and runtime techniques to optimize execution. We implemented our proposed scheme in the IMOP framework. Experimental results on graph-based benchmarks show that our approach improves the performance significantly.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Parallel programming languages**.

Additional Key Words and Phrases: irregular workloads, load balancing, global barriers, code instrumentation

ACM Reference Format:
Amit Tiwari and V. Krishna Nandivada. 2025. Unleashing Parallelism with Elastic-Barriers. *ACM Trans. Arch. Code Optim.* 1, 1 (March 2025), 25 pages.

1 Introduction

With the emergence of multi-core processors, exploiting parallelism has become a necessity rather than a choice. In modern parallel programming languages like Chapel [7], Cilk [2], OpenMP [4], X10 [8], Go [13], and so on, achieving efficient execution requires managing synchronization overheads, with barriers playing a crucial role. These barriers introduce two main challenges. First, barriers lead to significant communication overhead. Second, threads perform no useful work while waiting at the barrier. While several prior barrier implementations [6, 12, 20, 32] have addressed the issues related to communication overhead, the issue of thread idling, which stems due to the load imbalance before the barrier, remains an important research domain. This issue becomes more critical in the context of irregular workloads, where work done by different threads may vary considerably.

Besides the standard scheduling schemes like `static`, `dynamic`, and `guided`, various scheduling mechanisms [5, 18, 21, 23, 26, 40, 45, 54, 57] have been proposed in the past to minimize load imbalance in irregular parallel applications. Considering the challenges in irregular parallel programs, recently there has been increasing interest [52, 55] in using combined compile-time and runtime approaches to handle the variability at runtime. The state-of-the-art in this space of loop chunking is the idea of *deep-chunking* [46] that uses a mixed compile-time and runtime technique to chunk the iterations of parallel for-loops, based on the estimated thread-workloads. While these techniques reduce the load imbalance to some extent, threads often spend significant time waiting at barriers, especially in irregular programs like graph analytics. Further, these techniques do not provide much insights on how the waiting times of these threads can be better utilized. We will illustrate these issues using an example.

Authors' addresses: Amit Tiwari, amittiwari@cse.iitm.ac.in, Department of CSE, IIT Madras, India; V. Krishna Nandivada, nvk@iitm.ac.in, Department of CSE, IIT Madras, India.

2025. XXXX-XXXX/2025/3-ART $15.00
https://doi.org/

Fig. 1 shows a simplified snippet of code derived from the KC (K-Committee) kernel [16]. It has two parallel-loops, ParLoop1 and ParLoop2, separated by a barrier, Barrier1. Each iteration in ParLoop1 processes a unique vertex in the input graph. This involves a traversal of the respective neighbors; thus, the work done by each iteration can vary significantly, and so can the workload of each thread (irrespective of the scheme used to chunk the iterations of the loop among the threads). Such imbalance can arise due to (i) high variance in the vertex-degree, (ii) limitations in the scheduling algorithm, or (iii) various architectural and runtime system constraints [22, 45]. For example, for the KC kernel running on the YouTube dataset [25], when using deep-chunking (the best known chunking scheme), we observe that threads wait for 48.8% of the time at the barrier.

To address this issue, in this paper, we present the idea of an *elastic-barrier*, where instead of simply waiting for the last thread to arrive at a barrier, a thread that arrives early performs work from the subsequent phase across the barrier, in a safe and profitable manner. This can help reduce the imbalance (and idling) significantly at the target barrier, especially in the context of irregular parallel-loops. For example, for the snippet shown in Fig. 1, we propose executing iterations from the next phase (ParLoop2) instead of idling. However, naively executing the iterations from ParLoop2 may not be safe, as it may read and write shared-memory locations (data->min_active) concurrently accessed by iterations in ParLoop1, leading to potential data races. Note: In ParLoop1, the min_active field of each vertex v is stored in the queue of each neighbor of v, and in ParLoop2, each vertex reads the values of those stored fields in its queue. Thus, before executing any iteration i2 of ParLoop2 before the barrier, we must make sure that all the iterations in ParLoop1 corresponding to the neighbors of i2 have been completed.

```
#pragma omp parallel for nowait // ParLoop1
for (int i1=0; i1<g->N; i1++) {
  node* cur = elem_at(&g->vertices, i1);
  payload* data = cur->data; ...
  for (int j1=0; j1<cur->degree; j1++) {
    node* nbr = elem_at(&cur->neighbors,j1);
    enqueue(Q, nbr->label, &data->min_active);}}
#pragma omp barrier            // Barrier1
#pragma omp parallel for       // ParLoop2
for (int i2=0; i2 < g->N;i2++) {
  node* cur = elem_at(&g->vertices, i2);
  payload* data = cur->data; ...
  data->min_active = min(data->min_active,
                         dequeue(Q, i2)); ...}
```

Fig. 1. Snippet of KC kernel from IMSuite

We address the issue of load imbalance in irregular parallel-loops by employing an approach using which, instead of idling at a barrier, threads can continue to execute work from the next phase (after the barrier), until the last thread has reached the barrier. This idea is fundamentally different from any prior work that dealt with the scheduling of iterations (within the same phase), as our proposed idea addresses the issue of load imbalance (and idling) by executing iterations from the next phase (beyond the barrier). We term such barriers as *elastic-barriers*.

We present a scheme to identify elastic-barriers in input programs, and a transformation pass to generate efficient code for the same. Our proposed pass focuses on two key dimensions: *Profitability*: Naively executing iterations from the next phase can even lead to performance degradation. Specifically, if a waiting thread continues executing such iterations beyond the arrival of the last thread, it may degrade performance. Thus, it is crucial for a waiting thread to accurately assess the waiting time, and estimate the amount of work that can be profitably executed. Further, the emitted auxiliary code should not lead to overheads masking the possible gains. *Safety*: Another important requirement is maintaining correct program semantics while executing iterations elastically. The waiting thread must ensure that it elastically executes an iteration i2 from the subsequent phase only when all iterations of the current phase that i2 depends on have been completed. Additionally, the elastically executed iterations should not be executed again in the next phase. These targets are achieved by maintaining the runtime state of all the threads, which, in turn, is achieved in two steps: (i) by statically emitting the required code, and (ii) executing this emitted code at runtime to derive the necessary information.

We present our scheme in the context of OpenMP C programs. However, we believe that the scheme can be extended to other parallel languages as well.

Contributions.

• In this paper, we present the idea of an elastic-barrier, where a waiting thread can execute iterations of the subsequent phase profitably, safely, and efficiently before the barrier, thereby utilizing the waiting time at the target barrier. Such a scheme can be beneficial for irregular parallel-loops.

• We propose a compilation scheme involving a mixed compile-time and runtime approach such that (for input codes with elastic-barriers) it leads to performant codes.

• We have implemented our proposed scheme in the IMOP [36] compiler framework. We evaluate our proposed scheme on three shared-memory graph benchmark programs from IMSuite [16]: K-committee, Bellman-Ford, and Dominating Set, and two other popular benchmarks: Pagerank and Perceptron [24, 29]. Our study of these benchmarks on three different inputs shows that our proposed scheme led to significant gains (in 479 of a total 540 configurations with varying benchmarks, number of threads, and real-world inputs) in the context of irregular workloads.

The rest of the paper is organized as follows. Section 2 gives a brief background of the workload-based deep-chunking technique, along with some minor extensions to handle OpenMP programs. Section 3 gives an outline of our technique, explaining how elastic-barriers are detected, followed by the intuition on how elasticity is realized. Section 4 provides an in-depth explanation of the proposed mixed compile-time and runtime approach for emitting code to exploit elasticity. Section 5 presents a discussion on some of the salient points of our proposed approach. Section 6 provides a detailed evaluation of the proposed techniques with performance analysis on five benchmark programs, focusing on execution time and barrier waiting time. Section 7 discusses the related work, and Section 8 concludes the paper.

2 Background

In this section, we give a brief background of two of the key ideas used from prior work [46, 52], along with some insights on adapting the same in the context of OpenMP. One main takeaway from these prior works is the use of a mixed compile-time and runtime approach to address the challenge arising out of the difficulty to statically estimating the workload of different threads and iterations of parallel-loops.

2.1 *CostExpGen*: Generating Cost-Expressions and Emitting the Workload Loop

Given a parallel for-loop, *CostExpGen* generates the cost-expression for each iteration of the loop, and emits a workload loop that computes these costs at runtime. We will use an example to illustrate how the generated cost-expressions may look like (interested reader may refer prior work [46] for details).

Fig. 2 shows the body of a sample parallel for-loop, iterating over the loop index variable `i`; the code for the parallel for-loop is omitted for brevity. Assume that `S1`, `S2`, `S3`, and `S4` represent one or more statements with input-independent execution times. Say, after profiling it is identified that the cost of `S1`, `S2`, `S3`, and `S4`, are `C1`, `C2`, `C3`, and `C4`, respectively. Further, if the predicate of the if-statement is found to be true `p%` of the time during profiling, the cost-expression for the body would be: `C1 + G[i].neighbors * (C2 + 0.01*(p * C3 + (100-p)*C4))`. It may be noted that the cost-expression has (i) constant literals derived from profiling, and (ii) expressions derived from the input program.

```
S1
for (j=0;j < G[i].neighbors; ++j) {
  S2
  if (v) S3 else S4
}
```

Fig. 2. Sample body of a parallel for-loop iterating over the nodes of a graph `G`. Assume that the `neighbors` field gives the number of neighbors of the vertex `G[i]`.

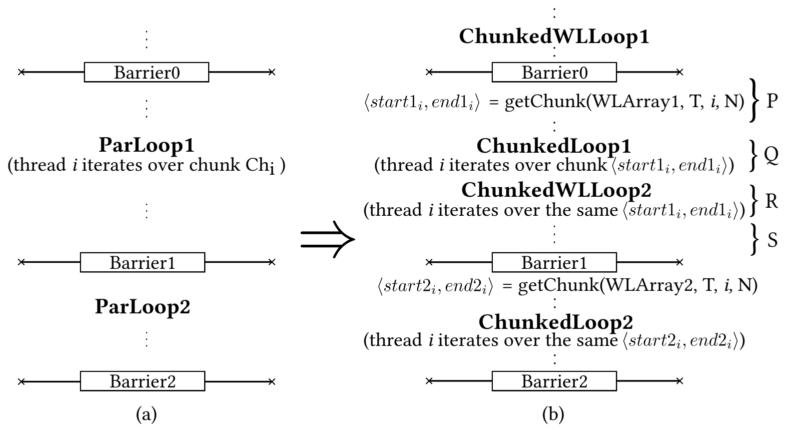

Barrier0
ParLoop1
(thread i iterates over chunk Ch$_i$)
Barrier1
ParLoop2
Barrier2
(a)
ChunkedWLLoop1
Barrier0
$\langle start1_i, end1_i \rangle$ = getChunk(WLArray1, T, i, N)
P
ChunkedLoop1
(thread i iterates over chunk $\langle start1_i, end1_i \rangle$)
Q
ChunkedWLLoop2
(thread i iterates over the same $\langle start1_i, end1_i \rangle$)
R
S
Barrier1
$\langle start2_i, end2_i \rangle$ = getChunk(WLArray2, T, i, N)
ChunkedLoop2
(thread i iterates over the same $\langle start2_i, end2_i \rangle$)
Barrier2
(b)

Fig. 4. (a) A sample input MPL code, (b) Code after invoking the loop-chunk emitter. We highlight four parts in the code to be emitted between Barrier0 and Barrier1 to perform: (P) initialization, (Q) chunk execution, (R) cost computation, (S) elastic-work execution.

Thus, during runtime, we can estimate the cost of any iteration `i` using the above cost-expression. In cases where we won't be able to 'extract' a closed form expression (like `G[i].neighbors`) as the loop bound (say, in case of a while-loop), *CostExpGen* underestimates the cost of the loop and assumes the loop bound to be 1. In Section 4.2, we show how we deal with such an underestimation.

Emitting `WLLoop`s. For each such parallel for-loop L of an `ELoop`, *CostExpGen* emits a workload loop (`WLLoop`) in the preceding phase to compute the workload of the iterations of L (stored in an array `WLArray`). We show a sample `WLLoop` in Fig. 3. Here, we assume that (i) `eN` is the expression giving the number of iterations of L, and (ii) $[\![C_L]\!]$ gives the cost-expression for any iteration `i` of L. Note that `WLLoop` is also a parallel for-loop, and its iterations are shared between the same set of threads executing the current parallel-region.

```
#pragma omp for
for (i = 0; i < eN; ++i) {
  WLArray[i] = ⟦ C_L ⟧;
}
```

Fig. 3. Sample `WLLoop`

2.2 *ChunkGen*: Loop Chunk Emitter

We first transform the input code using *ChunkGen* before invoking our optimization. For illustration, a typical program pattern is shown in Fig. 4(a), on which *ChunkGen* is applied. The transformed code is shown in Fig. 4(b). The method `getChunk` (executed by thread i) returns the chunk $\langle start1_i, end1_i \rangle$, as proposed by deep-chunking. This method takes `WLArray`, total number of threads T, current thread ID i, and the total number of iterations n of the parallel for-loop as arguments. We can see that the original parallel-loops of Fig. 4(a) have been replaced by chunked equivalents iterating over the chunk $\langle start1_i, end1_i \rangle$.

Note: For each emitted `WLLoop`, *ChunkGen* uses the chunk of the preceding parallel for-loop. However, for a parallel for-loop that has no preceding parallel for-loop, the emitted `WLLoop` will use the default static schedule of OpenMP.

3 Overview of the Proposed Solution

Informally, we say that a barrier exhibits *elasticity* if there is a subsequent parallel-loop from which some iterations can be executed before the barrier. To reduce the idle waiting of threads at such barriers, we present (i) a scheme to identify such elastic-barriers and (ii) a translation mechanism that emits efficient code to reduce the idle waiting of threads at elastic-barriers, by enabling those threads to execute a subset of the iterations of the subsequent parallel-loop. We call the threads that complete their work early as the *early-finishing* threads (EFTs, in short), and the thread that reaches the barrier at the end the *last-finishing* thread (LFT, in short).

We start the presentation of our scheme by defining a subset of a language (MPL) in Section 3.1, that we will use to explain our approach. In Section 3.2, we discuss various compatibility checks for identifying opportunities for exploiting elasticity in barriers, in programs in MPL form. In Section 3.3, we explain the overall execution flow of our proposed technique. Then, in Section 3.4, we provide an intuition of our proposed technique, by focussing on the profitability and safety of the generated code.

3.1 MPL: Multi-parallel loops

Fig. 5 shows the syntax of a subset of OpenMP C language (called MPL) that forms the basis for identifying elastic-barriers. The non-terminal ERegion represents a parallel-region that encloses multiple parallel-loops; its body is represented by the non-terminal EStmt. An EStmt may be enclosed inside sequential constructs, denoted by SEQ (EStmt); for example, it may be present inside sequential blocks (like loops, if-statements, compound statements, and so on). Importantly, an EStmt always contains two or more ELoops. Each ELoop consists of a ParLoop followed by a Barrier, with optional sequential code before and after the ParLoop. The non-terminal SeqStmt denotes any sequential statement, and forLoop denotes a sequential for-loop.

```
ERegion ::= #pragma omp parallel EStmt
EStmt ::= ELoop ELoop+  | SEQ (EStmt)
ELoop ::= SeqStmt? ParLoop SeqStmt? Barrier
ParLoop ::= #pragma omp for nowait forLoop
Barrier ::= #pragma omp barrier
```

Fig. 5. Syntax of MPL. X+ represents one or more occurrences of X, and X? represents zero or one occurrence of X.

In this section, we will assume that the input is in MPL form. Later, in Section 5.1, we describe how we preprocess any input program to expose parallel-loops in MPL form.

3.2 Opportunities For Exploiting Elasticity In Barriers

Consider a pair of consecutive ELoops, say ELoop1 and ELoop2, represented as "S1$_1$, ParLoop1, S2$_1$, Barrier1", and "S1$_2$, ParLoop2, S2$_2$, Barrier2", respectively. We assume that there is some inter-thread dependence between both the ELoops, because of which Barrier1 is not redundant. Otherwise, such a barrier can be simply elided. To mark such a non-redundant barrier (like Barrier1) as an elastic-barrier, we perform the following three checks.

Check#1: *Independence of the sequential parts.* There should be no data-dependence between ParLoop2 and any of the preceding sequential statements (S1$_2$, S2$_1$, and S1$_1$). Note that we enforce this restriction even for S1$_1$ and S2$_1$, even though they are sequential statements. This is because in the input program, they are present inside a parallel-region, and hence may be executed by multiple threads in parallel with the iterations of ParLoop2 that we plan to execute before Barrier1.

Check#2: *Dependence between the parallel-loops.* Since the independence of the sequential parts has been established via Check#1, we now assume that the dependence is arising due to one or more iterations of ParLoop2 having inter-thread dependences with one or more iterations of ParLoop1. We require that the dependence may arise due to their conflicting accesses to common elements of the shared array(s) (allocated statically or dynamically) and not shared non-arrays (e.g., scalars). If the dependence is not on array elements, we conservatively assume that each iteration of ParLoop2 depends on all the iterations of ParLoop1. Consequently, no iterations of ParLoop2 can be executed elastically until all the iterations of ParLoop1 have been completed.

Check#3: *Similarity between the parallel-loops.* For ease of exposition, we will also assume that both ParLoop1 and ParLoop2 iterate over the same iteration space (say, I). Note that the iterations of ParLoop1 and ParLoop2 may further iterate over a sub-list of I. For example, in a graph benchmark program the outer loop may traverse over the nodes, and the inner loop traverses over the neighbors.

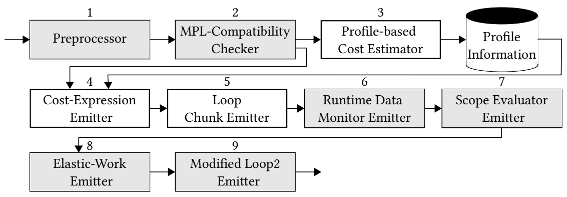


Fig. 6. Components of the proposed translation scheme. The blocks with white background are directly extended from the prior work [46], and the gray-shaded blocks indicate our contribution.

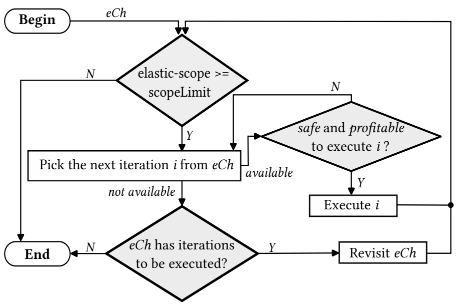

Begin
eCh
elastic-scope >= scopeLimit
N
Y
Pick the next iteration i from eCh
N
safe and profitable to execute i ?
available
Y
Execute i
not available
End
N
eCh has iterations to be executed?
Y
Revisit eCh

Fig. 7. Overview of the various components before performing the elastic-work, until the last thread has arrived.

When `ELoop1` and `ELoop2` satisfy all these three checks, we mark `Barrier1` as an elastic-barrier. For ease of exposition, we assume that there exists a barrier before each such pair of `ELoops` (as part of a preceding `ELoop` or a standalone one); if no such barrier exists, we insert one. We refer to this pattern of a barrier followed by a pair of `ELoops` as an elasticPattern. A simple depiction of such a pattern can be seen in Fig. 4(a).

3.3 Process flow

We now briefly summarize our proposed scheme to translate elastic-barriers in OpenMP programs efficiently. Our scheme has two components: (i) compile-time, and (ii) runtime.

Compile-time component (see Fig. 6). During compilation, we first invoke a preprocessing pass (discussed later in Section 5.1) that tries to transform any given OpenMP program to an equivalent MPL form (Step 1). Next, we verify the transformed program's compatibility based on the rules in Section 3.2 (Step 2).

If found compatible in Step 2, then we profile the program (Step 3) to identify the parts of the program with input-independent costs and then run the program on a small input to compute the costs; for this, we extend the techniques described by Prabhu and Nandivada [46], for OpenMP. For each MPL-compatible program, we use its profiler-generated costs to emit (parallel) code that computes the costs of the iterations of the parallel-loops (Step 4; described in Section 2.1). Next, we transform the parallel-loops such that the work-division code (via the state-of-the-art deep-chunking [46]) is emitted (Step 5; described in Section 2.2).

We now list our steps to exploit elasticity in the barriers: First, we emit code that updates the progress of each thread at regular intervals (Step 6; Section 4.1). Then, we emit a function that can be used by any EFT to compute the maximum of the estimated remaining workloads of the other threads in the team (Step 7). Next, we emit code that elastically executes one or more iterations from `ParLoop2` before `Barrier1`, if it is deemed to be "safe" and "profitable" (Step 8); these two steps are discussed in Sections 4.2, 4.3, and 4.4. As we will discuss later, each thread will find a fixed chunk (called *elastic-chunk*) of such iterations from `ParLoop2` to execute elastically. Finally, we modify `ParLoop2` to ensure that elastically executed iterations are not repeated (Step 9; Section 4.5).

Runtime component. During execution, each thread computes the total workload of the parallel-loop and identifies (and starts the execution of) the chunk of iterations assigned to that thread (based on the code emitted in Steps 4, 5 and 6, Fig. 6). The code emitted in Steps 7 and 8 is executed only by the EFTs; this is done as long as it is deemed profitable. First, the EFTs estimate their waiting time based on the maximum remaining workload of other threads to perform elastic-work (Step 7). We refer to this estimated waiting time as the *elastic-scope*, indicating the opportunity it provides to the EFT to execute *elastic-work* (some iterations of `ParLoop2`). Each such EFT then executes one iteration from ParLoop2 at a time, if it is deemed safe (Step 8). All the threads, including those that performed elastic-work and otherwise, meet at Barrier1 and then proceed to execute the modified ParLoop2 that skips the iterations that have already been executed elastically (code emitted in Step 9).

To better understand the iterative process between blocks 7 and 8, we present a brief explanation of the underlying interactions in Fig. 7. The EFT starts with an elastic-chunk *eCh*, and first checks that the thread's elastic-scope is significant (greater than a predefined constant scopeLimit). If so, the EFT executes safe and profitable iterations from *eCh*, one at a time. After making a full pass over the iterations of the *eCh*, the EFT may restart the processing of *eCh* to execute the iterations of *eCh* that were not executed before (due to the safety constraints).

We now give details about Steps 6 to 9 before briefly discussing the preprocessing step.

3.4 Intuition Behind the Proposed Solution

Using the code emitted by the loop chunk emitter in Fig. 4(b), each thread first executes its chunk of iterations in ChunkedLoop1. After that, to elastically execute iterations from the next phase, it must check for profitability and safety, as explained below.

Profitability. For any EFT, a naive way to do elastic-work would be to simply execute the iterations of ChunkedLoop2 (one at a time) whenever it finds that there are other threads still working in the current phase, and it is safe to execute that iteration. However, this can be counterproductive as the naively executed iteration from ChunkedLoop2 may lead to an increased critical path in the first phase. For example, consider Fig. 8(a), which shows a possible execution scenario consisting of two parallel for-loops separated by a barrier (shown by the horizontal solid black line). Assume that at runtime there are two threads T1 and T2. The chunk assigned to each thread is shown using a box, and the numbers above the box give the time taken to execute that chunk. The dotted lines below the chunks indicate waiting times. The vertical lines inside the boxes of ParLoop2 indicate the iterations of the corresponding chunk, and the numbers above these lines indicate their respective execution times. In this example, after completing its chunk, T1 sees that T2 is yet to complete its chunk; hence, T1 can perform elastic-work. Assume that T1 can safely execute any of the iterations from its chunk in ChunkedLoop2. Now, if T1 naively picks up the first iteration from the second loop as the elastic-work (see Fig. 8(b)), then the threads take a longer time to complete the first phase, and the critical path length increases (from 220 units to 260 units). We term such a scenario where the elastic-work done is more than the elastic-scope, as "overshooting". To prevent overshooting, we identify the iterations of ChunkedLoop2 to be performed based on the current elastic-scope. Please see Section 5.6 about the overheads of the possible design choice, where in case of overshooting, the LFT also starts performing elastic-work.

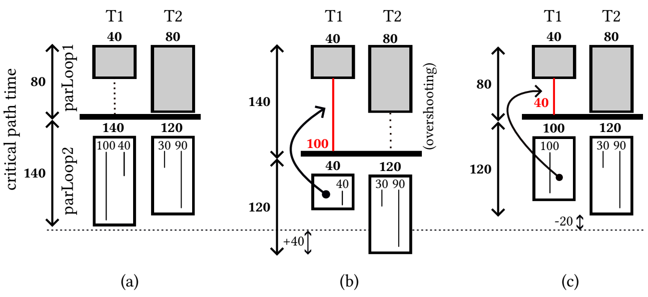


Fig. 8. Impact of overshooting: (a) A runtime scenario involving two threads, (b) naively performing elastic-work may lead to increased critical path length, (c) our proposed approach.

Safety. Say the barrier present between ParLoop1 and ParLoop2 is not redundant – otherwise, the programmer or some other compiler pass can elide it. That is, assume that one or more iterations of ParLoop2 have dependence on one or more iterations of ParLoop1. Thus, before executing any iteration j from ParLoop2, we have to satisfy the safety requirement: make sure that the list of all the iterations of ParLoop1 that iteration j depends on (called depItersList) has been executed by their respective threads. To achieve this: (i) For each iteration in ParLoop2, we need to identify the corresponding depItersList, and (ii) For each thread executing ParLoop1, we need a way to query the progress of each thread (in terms of the executed iterations), so that any EFT trying to execute an iteration from ParLoop2, as part of elastic-work, can ensure that iterations in the corresponding depItersList have been executed.

The code to populate different data structures required to efficiently compute profitability and safety is emitted in Step 6 (in Fig. 6); these data structures are used in Steps 7 and 8. We explain these steps in the following sections.

4 Transformations to Exploit Elasticity

4.1 Runtime Data Monitor Emitter

We now explain how we maintain different data structures to track the runtime state of the program, which helps compute elastic-scope and monitor thread progress. Our scheme involves emitting code to initialize and update two arrays: *threadRemWLArray* and *threadProgress*.

4.1.1 Maintaining threadRemWLArray. To compute elastic-scope, we maintain a global array called *threadRemWLArray* (size = number of threads), whose i^{th} element holds the remaining workload of thread i in ChunkedLoop1. Each EFT iterates over the elements of *threadRemWLArray* to get the remaining workload of the LFT – this gives an estimate of the elastic-scope.

After invoking getChunk to determine its chunk, each thread i initializes $threadRemWLArray[i]$ using the WLArray array that holds the iteration costs for all the iterations (see Section 2.1). This code is emitted in the "initialization (P)" part in Fig. 4(b) immediately after the call to getChunk.

A naive way to keep $threadRemWLArray[i]$ up-to-date would be to reduce the current value of $threadRemWLArray[i]$ after the execution of each iteration in ChunkedLoop1. Since the elements of the array *threadRemWLArray* may be accessed concurrently by multiple threads, these must be accessed atomically and, in turn, brings in two types of overheads: (i) contention at the atomic blocks – which may impact the execution of the LFT. (ii) false sharing among threads as *threadRemWLArray* is a global array. To avoid such overheads, each thread maintains an up-to-date version of $threadRemWLArray[i]$ in a local variable (localRemWorkLoad). Once the remaining workload of any thread becomes less than the maximum cost of the iterations of ParLoop2 (denoted by maxIterCostLoop2), thread i starts copying the value of localRemWorkLoad to $threadRemWLArray[i]$, after executing each iteration of ParLoop1. Note that the EFTs do not need the remaining workload information of the LFT when it is greater than maxIterCostLoop2. This is so because no elastically executed iteration would have a cost greater than maxIterCostLoop2, and updating progress once the remaining workload is equal/below maxIterCostLoop2 is sufficient.

4.1.2 Maintaining threadProgress. To monitor thread progress, we use a global array *threadProgress* (one element per thread) for each parallel-loop, where $threadProgress[i]$ stores the index of the last iteration executed by thread i. In the "initialization (P)" part of Fig. 4(b), we emit the code to initialize *threadProgress* after the call to getChunk, such that each thread i, sets $threadProgress[i] = start1_i - 1$ (indicating no progress). We also emit code such that after executing each iteration in the "chunk-execution (Q)" part of Fig. 4(b), each thread i updates its progress (by setting $threadProgress[i]$ to the value of the chunk index variable).

Similar to *threadRemWLArray*, accesses to the elements of *threadProgress* must be atomic. Further, we reduce the overheads due to such atomic updates by updating the shared array after processing those iterations, on which a significant number of threads ($\geq$ totalThread/4) depend; we obtain this information by processing the depItersList arrays.

```
1  while safeScope ≥ scopeLimit AND last thread has not arrived do
2      profitableIterFound = false;
3      for eIter=start1_i to end1_i do                                          // elastic-loop
4          if WLArray2[eIter] = 0.0 then                                       // already executed
5              continue;
6          if safeScope < WLArray2 [eIter] + DEPCOST then continue ;           // not profitable
7          profitableIterFound = true;
8          ... ;                                       // Dependence check for safety. See Fig. 13.
9          if isSafe == true then
10             body of ChunkedLoop2;                                          // execute elastically
11             safeScope = safeScope − WLArray2[eIter];
12             WLArray2 [eIter] = 0.0;                                         // mark eIter as done
13         safeScope = safeScope − DEPCOST;
14         if safeScope < scopeLimit OR last thread has arrived then break;
15     if profitableIterFound == false then break ;
16     safeScope =MAX(threadRemWLArray);
```

Fig. 9. Code emitted by elastic-work emitter. We can categorize the predicates in the algorithm into two groups: ones related to profitability (lines 1, 6, 14, 15) and safety (lines 4, 9).

4.2 Scope-evaluator and Elastic-work Emitter

In this section, we first describe how we emit code so that each thread can compute the available elastic-scope at any point of time during execution. We emit code that lets EFTs find the maximum value in *threadRemWLArray* and store it in a variable safeScope. This code is a serial for-loop (#iterations = #threads) placed immediately after the ChunkedWLLoop2, in the part "(R)" in Fig. 4(b).

Next, we describe the details of elastic-work-emitter (Step 8, Fig. 6). We now describe the emitted code for elastic-work execution, along with its profitability and safety-checks; all this code is emitted in the "elastic-work execution (S)" part in Fig. 4(b).

Fig. 9 shows a sample code inserted by the elastic-work-emitter. The code consists of a serial for-loop that goes over all the iterations of the elastic-chunk (at Line 3); we call this loop as *elastic-loop*. To avoid overshooting (see Section 3.4), an EFT performs a check at Line 6. Here, it checks if executing the current iteration is profitable by comparing safeScope against the cost of the iteration that is being planned to be executed elastically, and the cost (DEPCOST) to perform a safety-check to be executed later at Line 8. Also, the value of safeScope is reduced by DEPCOST (at Line 13), irrespective of whether the iteration is executed elastically or not - as we do pay the cost of dependence check. The value of DEPCOST is obtained by separately profiling the code used to perform the dependence check.

For example, for the scenario illustrated using Fig. 8(a) (ignoring DEPCOST for simplicity), the thread T1, which computes the safeScope to be 40 units, will skip the iteration with higher cost (100 units) and instead elastically execute the next iteration with lower cost (40 units), as the latter will not lead to increased critical path length; this is illustrated in Fig. 8(c).

Next, we emit code such that for each iteration eIter of ChunkedLoop2 that is deemed safe, (a) eIter is executed, (b) safeScope is updated, and (c) WLArray2[eIter] is reset to zero (Lines 9-12). We note two salient points about these four lines (Lines 8-12): (i) an iteration that may not be safe to execute at one point of time (as its dependences may not have been satisfied) may become safe later. Hence, the for-loop to execute the iterations elastically is emitted within a while-loop (Line 1), (ii) in the emitted body of ChunkedLoop2 for elastic execution, all the free occurrences of its loop induction variable have been replaced with eIter. Note that before executing an iteration elastically, the EFTs use the code in Fig 13 (discussed in Section 4.4) to ensure safety.

Before entering the body of the while-loop (starting at Line 1) and after executing an iteration elastically (Line 14), we emit an important profitability-related condition to ensure that safeScope >

```
Loop1: for (i=0;i<N;i++) {
       ... access vertex V[i] }
-------------------------------
Loop2: for (j=0;j<N;j++){
       ... access vertex V[j] }

             (a)
```

```
Loop1: for (i=0;i<N;i++){
          ... access vertex A[i] }
-------------------------------
Loop2: for(j=0;j<N;j++){
         for(k=0;k<Nbrs(V[j]);k++){
         ... access k-th neighbor }}

             (b)
```

```
Loop1: for (i=0;i<N;i++){
         for (k=0;k<Nbrs(V[i]);k++){
          ... access k-th neighbor }}
-------------------------------
Loop2: for (j=0;j<N;j++){
          ... access vertex V[j] }

             (c)
```

Fig. 10. Sample codes to depict the challenges in computing dependences. Say, at least one of the accesses in each of the figures is a write operation.

scopeLimit and the LFT has not yet completed its assigned chunk; the thread will stop executing the elastic-work if the condition fails.

We use a shared variable that is atomically incremented (in part Q, Fig. 4(b)) by each thread after executing ChunkedLoop1. When the value of this variable equals the total number of threads, it indicates that all the threads have arrived at the barrier.

In Fig. 9, we emit a check (Line 15) that exits the while-loop, if the EFT encountered no profitable iterations in the elastic-loop. This is based on the premise that after executing the elastic-loop, it is beneficial to continue the iterative process only if there remain profitable iterations of the ChunkedLoop2 that are yet to be executed (and not executed because of some dependences).

Besides handling iterations that may become safe to execute in the future, the outer loop at Line 1 is also useful to handle imprecision in computing the cost-expressions (and consequently the values stored in WLArray). For instance, consider a ChunkedLoop1, where the body of the loop contains a while-loop. As discussed in Section 2.1, the cost of a while-loop is set to the cost of its body, conservatively. Consequently, in such a scenario, safeScope is an underestimation of the actual available scope. Thus, it may so happen that despite reducing safeScope at Lines 11 and 13, safeScope may not match the actual current scope, as the LFT may take more time than estimated. In such situations, the recomputation of safeScope (at Line 16) and the outer loop at Line 1 can help perform elastic-work despite the possible underestimation of costs.

Furthermore, to reduce the overheads, instead of recomputing safeScope by iterating over *thread-RemWLArray* after every iteration of the elastic-loop, we recompute safeScope after executing each instance of the elastic-loop at Line 16. Note that, inside the elastic-loop, if in any iteration, it is found that safeScope is less than scopeLimit, then we exit the loop (Line 14) and recompute safeScope again (Line 16). We set scopeLimit to be the cost of the lowest-cost iteration of ChunkedLoop2.

4.3 Dependence Check

As discussed in Section 3.4, before elastically executing an iteration j of ParLoop2, we must ensure that all the iterations of ParLoop1 that it depends on have been executed. In this section, for ease of exposition, we will assume that the dependences are arising out of both the chunked-loops accessing the elements of some shared list sL, and at least one of the access is a write.

As discussed in Section 3.2, we consider only those cases where both ParLoop1 and ParLoop2 iterate over the same iteration-space I. Say, in any iteration j of ChunkedLoop2, the program accesses $sL[e_2]$, where e_2 is any arbitrary expression involving j. Thus, before elastically executing iteration j of ChunkedLoop2, all the iterations of ChunkedLoop1 that access $sL[e_2]$ must be completed. Let (i) iteration i of ChunkedLoop1 accesses $sL[e_1]$, where e_1 is any arbitrary expression involving i, and (ii) there exists an inverse function $e_{1_{inv}}$, such that $e_{1_{inv}}(a) = \{x | x \in I \wedge e_1[i/x] = a\}$. We use the notation $e_1[i/x]$ to denote that every free occurrence of i in e_1 has been replaced by x. Thus, the iteration j of ChunkedLoop2 can be elastically executed if the set of iterations (depItersList) present in $e_{1_{inv}}(e_2)$ have already been executed in ChunkedLoop1. We illustrate this with an example.

Consider the example code shown in Fig. 10(a), which shows two parallel-loops, separated by a barrier (OpenMP directives skipped). The two loops access the vertices of a graph, stored in a shared list V. Based on the above discussion, e_2 = j, e_1 = i, and depItersList = $e_{1_{inv}}(e_2)$ = [j]. Thus, before elastically executing any iteration j from ChunkedLoop2, the iteration j of ChunkedLoop1 must have been completed. As another example, say Loop2 accesses V[j+1]; let us ignore the corner case where j=N-1, for simplicity. Here, e_2 = j+1, e_1 = i, and $e_{1_{inv}}(e_2)$ = j+1. Thus, before elastically executing any iteration j in ChunkedLoop2, the iteration j+1 of ChunkedLoop1 must have been completed. Note that the list of iterations in depItersList for an iteration j in ParLoop2 may have been assigned to a thread that is different from the thread intending to elastically execute j.

While it is easy to compute the inverses for such simple accesses, in general, this can be a very challenging problem. For example, see the code shown in Fig. 10(b). This is a typical graph analytics code where each iteration of Loop2 iterates over the neighbors of the vertex V[j]. Fig. 10(c) shows another variation of the same challenge. We have seen that load imbalance in graph analytics codes is typically very high, and hence, in the next section, we discuss ways to handle such graph analytics code so that we can efficiently execute elastic-work.

4.4 Specializing the Dependence Checks for Graph Analytics

Typically, parallel graph analytics codes with load imbalance have the following properties: (i) they may have many loops that iterate over the nodes of graphs, (ii) while iterating over the nodes of the graph, the codes may iterate/access over their neighbors, (iii) the list of neighbors of a node is a subset of the nodes of the graph, (iv) each node has a unique id, and (v) a parallel-loop iterating over the nodes of the graph, in iteration i processes the node with id i.

We now discuss how we can efficiently compute the inverse functions (discussed in the previous section) for such graph analytics codes in MPL form. We first define two types of common access patterns based on which we present our intuition about efficient dependence checks.

Self-Access. We define a parallel-loop to be performing self-access if it iterates over the nodes of the graph, and iteration i of any loop accesses at most one node, and that node has id i.

Neighbor-Access. We define a parallel-loop to be performing neighbor-access if it performs self-access, and the iteration i of the loop may also access the neighbors of the $\texttt{i}^{th}$ node.

Based on these access patterns, depending on the number of data-dependences between the iterations of ChunkedLoop1 and ChunkedLoop2, we highlight three key types of dependences (between the ChunkedLoops), for which we have identified efficient schemes to compute the inverse functions. In all the three cases, the accesses involve one or more common fields of the graph nodes, such that at least one of the accesses is a write.

(i) *One-One Dependence.* Both the chunked-loops perform only self-access (see Fig. 10(a), for example). (ii) *One-Many Dependence.* ChunkedLoop1 performs only self-access, and ChunkedLoop2 performs neighbor-access (see Fig. 10(b), for example). (iii) *Many-One Dependence.* ChunkedLoop1 performs neighbor-access, and ChunkedLoop2 only performs self-access (see Fig. 10(c), and Fig. 1, for example). Note that irrespective of the dependence between the parallel-loops, they may additionally include loop(s) to access neighbors (without leading to any dependence between the parallel-loops). For ease of explanation, if two ChunkedLoops have one of the above dependences, then we say that the enclosing elasticPattern (defined in Section 3.2) *exhibits* the same dependence. Please see Section 5 for a discussion on handling many-many dependence.

4.4.1 Emitting Dependence Checking Code For Key Dependences

Handling One-One Dependence. If ChunkedLoop1 and ChunkedLoop2 have one-one dependence, then we can avoid emitting safety-checking code by ensuring that the elastic-chunk matches the chunk in ChunkedLoop1. This is because, there will be a guarantee that any EFT, before executing an iteration j elastically, would have completed the execution of the iteration j of ChunkedLoop1.

Handling One-Many and Many-One Dependences. In case of one-many dependence, to elastically execute a single iteration `j` of `ChunkedLoop2`, an EFT must ensure that all iterations in `ChunkedLoop1` corresponding to the neighbors of the node `j` have completed their execution. In the case of many-one dependence, a naive way to resolve dependences before elastically executing such an iteration `j` would be to iterate over all the iterations of `ChunkedLoop1`, and then ensure that every such iteration accessing node `j` has been completed. But this can be expensive. Instead, we can leverage the property of undirected graphs, and safely use the same check as that used for one-many dependence. Thus, in both these cases, the depItersList for any iteration `j` is given by the iterations in `ChunkedLoop1`, corresponding to the neighbors of node `j`.

For directed graphs, we treat the dependences differently. (a) one-many dependence: if the neighbor-access is being done on the outgoing (or incoming) neighbors of node `j`, then EFT must wait for the iterations of the outgoing (or incoming) neighbors of node `j`. (b) many-one dependence: if the neighbor-access is being done on the outgoing neighbors of node `j`, then EFT must wait for the iterations corresponding to the incoming neighbors of node `j`, and *vice-versa*.

We illustrate how the dependence check works using a two-threaded system by considering a runtime snapshot of the program as shown in Fig. 11. Here, threads T_x and T_y iterate over their respective chunks in `ChunkedLoop1` denoted by chunk$_{T_x}$ and chunk$_{T_y}$. Here, T_x acts as the EFT, and T_y as the LFT, with T_x completing its chunk while T_y has finished its 3rd iteration. The snapshot shows the *threadProgress* array storing "x-1" for T_x and "2" for T_y. Further, we demonstrate two cases when T_x attempts to execute `eiter1` and `eiter2` (represented using a black and a dotted circle, respectively), from elastic-chunk$_{T_x}$. For each such iteration, say *j*, T_x identifies depItersList in `ChunkedLoop1` and the corresponding thread(s) that is/are expected to execute them. For simplicity, we omit the details on depItersList and assume that `eiter1` and `eiter2` depend on T_y for the completion of chunk$_{T_y}$'s 2nd and 4th iteration in `ChunkedLoop1`, respectively. In such a case, T_x can look up the *threadProgress* array to identify `eiter1` can be elastically executed, but not `eiter2` as T_y has not yet completed the 4th iteration in `ChunkedLoop1`.

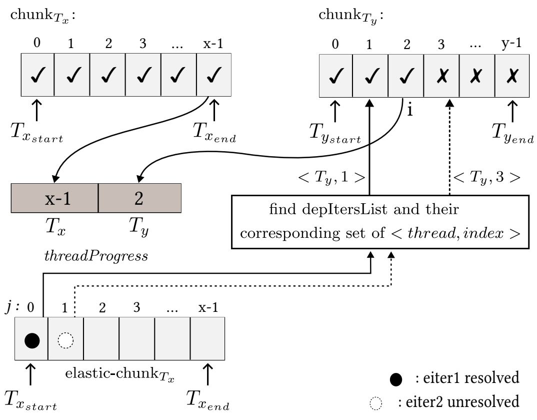
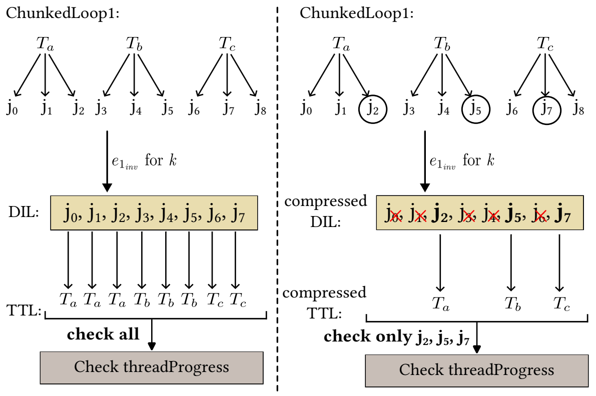


Fig. 11. Safety-checks for performing elastic-work

4.4.2 *Efficient Checking of Dependences*

As discussed in Section 4.4.1, in order to ensure safety for one-many, and many-one dependences, elastic execution of an iteration may involve checking the completion of many iterations of `ChunkedLoop1`, which may lead to prohibitively high costs. See Fig. 12 (a) for an illustration. Say, an EFT attempts to elastically execute iteration *k* from `ChunkedLoop2`. The EFT can first compute the associated depItersList by applying the inverse function, $e_{1_{inv}}$, on *k*. For each element in depItersList, the EFT then identifies the target thread by inspecting the starting and ending indices of the chunks of all the threads in `ChunkedLoop1`. Finally, as discussed in Section 4.4.1, the EFT ensures the safe elastic execution of iteration *k* by reading from *threadProgress* to determine whether each dependent iteration in depItersList has been executed in `ChunkedLoop1` by its corresponding target thread.

While straightforward, this naive dependence check can be inefficient, especially in graph analytics programs. For example, if the depItersList corresponds to the set of neighbors for node k, the EFT may need to perform a large number of checks before executing iteration k, which, in the worst case, can be equal to the number of nodes in the input graph. The overheads of such dependence checks may outweigh the expected benefits of elasticity, particularly in real-world graphs with millions of nodes. To address this issue, we propose an efficient approach to perform the dependence checks, discussed next.



(a) Straight-forward approach (b) Ordered-nodes approach

Fig. 12. Straight-forward approach Vs. Ordered-nodes approach; time complexity: O(#Vertices) Vs. O(#Threads). Abbreviations used: DIL=depItersList. TTL=targetThreadsList.

Efficient Dependence Check. Our proposed dependence check is illustrated in Fig 12 (b). The first step in our proposed scheme is a pre-pass that sorts the list of neighbors for each node as per their node-id's (which matches the order in which they are executed in ChunkedLoop). The key intuition behind our proposed dependence check is that instead of inspecting the completion of execution of all the dependent iterations (or neighbors) by their target threads in ChunkedLoop1, an EFT performs a range-based check. Here, it inspects the completion status of only the last neighbor per target thread – that is, the neighbor with highest node-id which is present in the chunk of the target thread in ChunkedLoop1. For example, in the shown figure, checks are done by the EFT for only iterations j_2 for target thread T_a, j_5 for T_b, and j_7 for T_c. This scheme ensures that the number of dependence checks per iteration is bounded by the total number of target threads, unlike the total number of neighbors in the naive approach.

Our efficient approach is facilitated by emitting code at compile-time, in two steps for each elasticPattern: (i) initialization-step, which is executed during the first runtime instance of the elasticPattern, and is used to populate the data structures needed to perform the dependence checks, and (ii) inspection-step, which performs the actual dependence check for each iteration to be executed elastically. In the initialization-step, we populate two compressed lists per iteration (that is, per graph node): compressed depItersList and compressed targetThreadsList, as shown in Fig 12 (b). For any iteration k, the compressed depItersList contains only a subset of the indices that need to be stored in the corresponding depItersList: at most one index per thread, indicating the neighbor with the highest node-id present in the chunk of that thread in ChunkedLoop1. Naturally, the compressed targetThreadsList will contain at most one entry per thread. Thus, the maximum size of these lists is bounded by the number of target threads.

In Fig 13, we show the pseudocode emitted in the inspection-step to perform the dependence check before elastically executing any iteration from ChunkedLoop2. In order to execute iteration k, an EFT first traverses over the two lists, compressed depItersList and compressed targetThreadsList, corresponding to node k. The elements of the two lists can be viewed as pairs of the form $\langle T, i \rangle$ such that before elastically executing iteration k, the EFT has to ensure that thread T has completed executing iteration i of ChunkedLoop1. This is done by comparing the last neighbor's ID against the value read from the *threadProgress* array for the target thread, which represents the iteration number that has been most-recently executed by the target thread in ChunkedLoop1 (Line 11).

If the read value is greater than the last neighbor's id, it implies that all the neighbors corresponding to the target thread have already been executed in ChunkedLoop1. Note, in case the target thread and the EFT are same, we bypass the corresponding check as the dependence is resolved implicitly (Line 10).

```
1  Function dependenceCheckAnalysis(threadId, $k$)
2      depCheckSucc := true;
3      compressedDepListSize := getCompDepItersSize($k$);
4      compressedDepItersList := getCompDepItersList($k$);
5      compressedTargetThreadsList :=
         getCompTargetThList($k$);
6      if compressedDepListSize ! = 0 then
7          for $i \leftarrow 1$ to compressedDepListSize do
8              targetThId := compressedTargetThreadsList [$i$];
9              lastVertexIndex := compressedDepItersList [$i$];
10             if targetThId == threadId then continue ;
11             if lastVertexIndex > threadProgress [targetThId]
                 then
12                 depCheckSucc := false;
13                 break;
```

Fig. 13. Dependence-check emitted code to resolve dependence for an iteration; time complexity: O(#Threads).

4.5 Modified Loop2 Emitter

In the earlier sections, we explained how iterations are executed elastically before reaching Barrier1. We have to make sure that those iterations are not executed again after Barrier1 in ChunkedLoop2. This is done by emitting a simple code as the first statement of ChunkedLoop2; if the loop index variable of ChunkedLoop2 is j, we emit `'if (WLArray2[j] == 0.0) continue;'`.

5 Discussion

In this section, we describe some of the salient points related to our proposed scheme.

5.1 Preprocessing: Transforming OpenMP programs to MPL form.

We have presented the transformation techniques for programs in MPL form. We now briefly describe how we convert any general OpenMP programs to MPL form. (i) If a parallel-region is present inside a function called from the main function, try to inline the method. (ii) If a parallel-region is present inside a loop or an if-statement, try to interchange the loop/if-predicate and the parallel-region (similar to the rules of Shirako et al. [51]) by adding an additional barrier at the end if required. (iii) For sequential statements in the main function not inside a parallel-region, enclose them inside an *omp-master* construct and then surround the *omp-master* construct inside a parallel-region. (iv) If multiple parallel-region blocks are present next to each other, fuse them. (v) Make the implicit barriers in ParLoops explicit (similar to the work of Tseng [56]).

5.2 Avoiding High Overheads

Avoiding Statically Determinable Overheads. We take cognizance of possible overheads in our proposed scheme and add two heuristics to address them. (a) In Section 4.4, we discussed three key types (one-one, one-many, and many-one) of dependences for which we provided specialized safety-checking code. A natural extension of the three key types is the many-many dependence. We have found that using the default scheme for many-many dependence can be very costly and hence, we do not apply our proposed transformation in the presence of such a dependence. (b) If we can statically prove that there is not much imbalance between the iterations of ParLoop1, then again, we skip applying our technique.

Avoiding Too Many Redundant Dependence Checks. Consider a scenario where an EFT is waiting for an iteration j of ParLoop2 to be ready to be executed elastically by continuously performing the dependence checks. If iterations of ParLoop1, on which the execution of j depends, take a long time to complete, then the EFTs may waste a lot of time in performing the dependence checks. To avoid such a scenario, we let the EFT sleep for a fixed interval of time (=1ms, deduced experimentally), if the EFT finds that less than $K\%$ (a constant) of iterations are ready to be executed.

We experimented with various values of K (1, 2, 4, 8, 10, 15, 20, 25, ... 75) and found that setting $K - 40$ was most beneficial.

Avoiding Possible Overheads Due To Graphs With Uniform Degree Distribution. Our proposed scheme targets loops that lead to irregular distribution of workloads among threads. To avoid the overheads when the input does not lead to irregular workload, for graph analytics programs, we propose a simple scheme based on multi-versioning: emit code such that the optimized code is executed only if the input is a power-law graph; otherwise, the original code is executed. We approximate the power-law graph property with a simple heuristic predicate, checked at runtime: $\Delta - \delta > m\lambda/n$, where Δ is the maximum degree, δ the minimum degree, n the vertex count, and m the edge count. λ is a tunable constant controlling acceptable variation within a graph class. Experimentally, we set $\lambda = 20$.

5.3 Extending Scope of Elasticity

The text discussed in Section 3 mainly focuses on ELoops organized as elasticPatterns. However, The grammar for MPL supports ELoops organized in many other forms. We now discuss two such forms that we encountered in our studies.

Cascaded Elasticity. Consider a sample snippet of a sequence of three ELoops (L1, L2, and L3), which could be part of a larger program. In such a scenario, L2 can be identified as part of two elasticPatterns: "L1; L2" and "L2; L3". Using the techniques discussed in the Sections 3 and 4 L2 may be identified as part of only one of these two elasticPatterns. However, we note that the parallel-loop in L2 can be translated as ParLoop2 for the first elasticPattern and as ParLoop1 for the second elasticPattern. We refer to this scenario as cascaded elasticity and the resulting patterns as cascaded elasticPatterns.

Circular Elasticity. Consider a sample serial-loop, which encloses an ELoop L1 as its last statement, and there is syntactically no ELoop after L1 in the serial loop. We found that we can apply the idea of elastic-barriers if the serial-loop body starts with an ELoop (say, L2). In such a scenario, L1 from the i^{th} iteration can be paired with L2 from the $(i + 1)^{th}$ iteration to form a new elasticPattern. This is an *inter-iteration* pattern and is referred to as a circular elasticPattern. At runtime, if the outer serial-loop executes n number of times, then $n - 1$ instances of L1 can perform elastic-work from L2. Further, if "L2; L1" forms an elasticPattern, then we can also take advantage of cascaded elasticPattern and emit code such that n instances of L2 will perform elastic-work from L1. Note that using the idea of circular elasticity, we can detect and optimize elasticPatterns, even in cases where there is only a single ELoop inside an outer serial-loop (where, L2 is same as L1).

5.4 `ParLoop2` without Load imbalance

In scenarios where the input `ParLoop2` did not have any load imbalance at the barrier, then load imbalance may get introduced due to the elastic-work execution in `ParLoop1`, thereby possibly nullifying the gains. Incidentally, in our evaluation, even though we encountered instances of such `ParLoops` (say L2), it did not impact the performance negatively. This is because L2 was part of a cascaded or circular elasticPattern: in both cases, L2 was identified as `ParLoop1` for the cascaded/circular elasticPattern. When L2 was part of the cascaded elasticPattern, it executed iterations from the subsequent `ParLoop`, while in the case of circular elasticPattern, it executed iterations from `ParLoop` of the next iteration of the outer serial-loop.

5.5 Elastic-Barriers in Other Parallel Languages and Real World Programs

The idea of elastic-barriers discussed in this paper, in the context of OpenMP C, can also be extended to other similar languages that support parallel-loops, whose iterations are shared among the runtime workers. Such languages include, Cilk Plus [2], Threading Building Blocks (TBB) [44], X10 [8], Chapel [7], OpenACC [11], Julia [1] and so on. The main difference would be the specific issues related to dependence analysis in the respective languages. We leave these topics as future work.

In real-world, it is common [43] to find many instances of parallel-loops executing one after the other; typically, these instances are derived from one or more parallel-loops being executed inside a loop covering multiple rounds of computation. We argue that elastic-barriers provide an opportunity to realize performant codes in many of these programs.

5.6 Elastic-Barrier Vs. No Barriers.

Using the approach discussed in Section 4, an EFT before executing an iteration (elastic-work) *I* from `ParLoop2` first confirms that all the iterations from `ParLoop1` that *I* depend on have completed execution. The EFT halts at the barrier when the elastic-scope is exhausted, ensuring an accurate execution point where load imbalance is minimized. However, one may argue that we can as well eliminate the barrier and let all the threads (including the LFT) execute the iterations from `ParLoop2` (by ensuring that the dependences have been resolved). Naturally, such a scheme leads to unacceptable overheads, as all the iterations of `ParLoop2` pay the additional overheads of dependence checks, even after the LFT has reached the barrier and there is no further need to check the dependences – leads to an increase in the length of the critical path. In our case, only the EFTs pay the overhead of checking, but that does not impact the length of the critical path. Note that the alternative for the EFTs is to simply wait at the barrier.

Another similar argument stems from the fact that due to some conservativeness in the estimations, while an EFT is performing elastic-work, the LFT may finish and end up waiting at the barrier. Hence, one may wonder if the LFT should also start executing elastic-work. However, this argument leads to the previous point about “no barriers”, as every time a thread completes executing some elastic-work and finds that some other threads are busy, it can pick up additional elastic-work, and this will continue. Hence, we let each EFT pick one iteration at a time from `ParLoop2` (and check the status of LFT after each iteration), and we let the LFT wait at the barrier.

5.7 Limitations of elastic-barriers and the Proposed Translation Scheme

We have proposed the idea of elastic-barriers, and provided an implementation and evaluation of our technique on real-world kernels. In Section 6, we also show the performance gains on real-world systems. We now discuss some limitations of our proposals, and some future directions:

- The proposed idea of elastic-barriers helps us reduce the critical path by replacing the idle waiting with work from the following phase. However, the problem of the load balancing within the same phase still remains an important problem to solve.
- Our proposed scheme targets programs/languages that work on shared memory systems. Many real-world applications, especially in large-scale data processing, run on distributed platforms. Realizing elasticity in barriers of such systems would require rethinking of the used shared data-structures, and the inter-thread/process communication mechanism.
- We present specialized schemes for efficiently performing dependence analysis for graph-analytics codes. Designing such schemes for arbitrary codes remains another interesting future work.

6 Implementation and Evaluation

We have implemented our proposed techniques (in short, `elastic`) in IMOP [36, 37], a source-to-source compiler framework for C OpenMP programs. We use the self-stabilizing feature [38] of IMOP to keep the analysis results up-to-date, during the multiple passes of analysis and transformation. Further, we have also implemented the state-of-the-art workload-based deep-chunking scheduling

Bench.	#LOC	#Barr	#oFor	1-1	1-M	M-1	e-EP
1. KC	355	6	6	0	2	2	4
2. BF	258	3	3	0	0	1	1
3. DS	525	12	12	2	4	2	2
4. PR	117	5	5	0	1	1	2
5. PC	112	4	4	0	1	1	2

Fig. 14. Benchmark programs characteristics. Abbreviations: LOC = lines of code, Barr = barriers, oFor = omp-for-loops, 1-1: one-one dependence, 1-M: one-many dependence, M-1: many-one dependence, e-EP: effective elastic-patterns.

Dataset	#V	#E	minD	maxD
1. YouTube	1.13M	2.99M	1	28.7k
2. WikiTalk	2.39M	5.02M	1	95.4k
3. Skitter	1.70M	11.10M	1	35.4k
4. Google	0.88M	5.11M	1	6.4k
5. DBLP	0.31M	1.05M	1	321
6. Slashdot	82.17k	0.95M	1	2.4k

Fig. 15. Input datasets collected from SNAP. Abbreviations: V: Vertices, E: Edges, minD: minimum degree, maxD: maximum degree.

technique [46] (in short, deep-chunk), tailoring it to OpenMP C programs. Overall, our codebase spans 19k lines of code in Java language. We present the empirical evaluation of our proposed technique by comparing it against our baseline deep-chunk. When performing the evaluation for overall execution time, for reference, we also consider other standard OpenMP scheduling strategies: `static`, `dynamic`, and `guided`. We compile the translated programs using `GCC`.

We have performed our evaluations on five popular shared-memory benchmark programs (see Fig. 14). These include K-committee (KC), BFS Bellman-Ford (BF), Dominating Set (DS), Pagerank (PR), and Perceptron (PC). The last two are written based on prior well-known algorithms [24, 29], and the first three are taken from the IMSuite benchmark suite [16]. The remaining kernels from IMSuite are skipped for the following reasons: (i) [kernels LCR, BY]: no elastic-barriers were found (see Section 3.2), or (ii) [kernels: BFS, HS, DP, MIS, MST, VC]: they were statically deemed non-profitable (see Section 5).

In Fig. 14, along with the number of lines of code, number of barriers, and number of parallel for-loops, we also enumerate the number of elasticPatterns with different types of dependences found in each program (Section 4.4). Out of these listed elasticPatterns, the two compile-time heuristics in Section 5.2 marked a few of them non-profitable and hence were not optimized. We label the remaining elasticPatterns as effective elasticPatterns.

Our proposed elastic-barrier approach is suitable for input workloads where the imbalance is significantly high and sufficient elastic-work is available. The experiments were conducted on three large real-world datasets (YouTube, WikiTalk, Skitter) from SNAP [25]. For DS, where the standalone program execution takes a large amount of time (>15 minutes) for these inputs, we used three medium-sized graphs: Google, DBLP, and Slashdot from the SNAP dataset. For the purpose of profiling, we ran the benchmark programs using small graph inputs. The details for the datasets are shown in Fig. 15.

Our experimentation is performed on two Ubuntu servers: (i) KM, a 64-thread Intel(R) Xeon(R) Gold 5218 CPU @2.30GHz system (dual-sockets, 16 cores per socket, two threads per core) with 64 GB of memory, GCC compiler version 11.4.0, and OpenMP version 4.5; and (ii) DB, a 128-thread Intel(R) Xeon(R) Gold 6338 CPU @ 2.00GHz (dual-sockets, 32 cores per socket, two threads per core), with 128 GB of memory, GCC compiler version 11.1.0, and OpenMP version 4.5. On both the systems, we use the default thread to core affinity mapping. The experiments are performed on a varying number of threads (2, 4, 8, 16, 32, 64, and 128) in DB and (2, 4, 8, 16, 32, 48, and 64) in KM; this is achieved by setting the environment variable `OMP_NUM_THREADS` appropriately. For each configuration, for reporting the execution time, we take an average over seven runs.

In our comparative study, we mainly focus on the following two important dimensions of evaluation: (i) percentage improvement in the overall execution time of the computation kernel, and (ii) reduction in the barrier waiting time due to our proposed transformation.

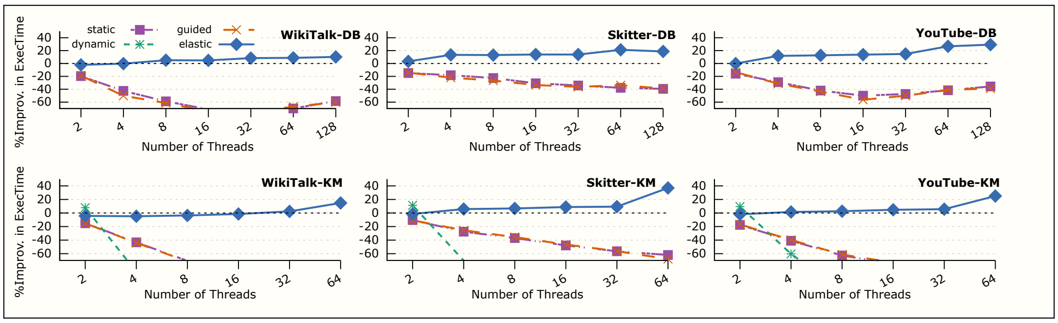


Fig. 16. Execution time analysis; KC

6.1 Performance Analysis: Execution time

To demonstrate the improvement in the overall execution time, for any schedule S (one of deep-chunk, elastic, static, dynamic, or guided), we compute the percentage improvement as: 100 × (execution time using deep-chunk − execution time using S) / (execution time using deep-chunk).

KC. In Fig. 16, we analyze the behavior of the KC kernel in terms of overall execution time. Across different configurations, elastic more or less outperforms all other scheduling mechanisms by a significant margin. The performance improvements over deep-chunk varied between -2.14% to 29.27% (0.98× to 1.41×) in DB and -4.77% to 36.98% (0.95× to 1.59×) in KM. We found that these high gains were arising as KC had two *impactful* serial-loops each having two ELoops (say, E1 and E2). In both the serial-loops we could detect and optimize an elasticPattern considering E1 and E2, and a cascaded-circular elasticPattern (see Section 5.3) considering E2 and E1. Consequently, the idling at the barriers of both the ELoops in the program reduced, which in turn reduced the overall critical path considerably. Prabhu and Nandivada [46] have shown that deep-chunk scales well with an increasing number of hardware threads. Our evaluation shows that elastic also scales well with an increasing number of hardware threads, and compared to deep-chunk, the increase is similar (or sometimes even better, for example, on KM with inputs Skitter and YouTube) as we increase the number of hardware threads.

Note that the overall benefit of our proposed approach depends on balancing two contrasting factors: the minor increase in the execution time of the LFT due to the additionally emitted code above the elastic-barrier, and the reduction in execution time in the modified ParLoop2 (because of the execution of fewer number of iterations). The improvements are visible when the second factor outweighs the first, leading to reduction in critical path length. The exact gains depended on the specific input (and the benchmark program under consideration).

An interesting observation from Fig.16 is that dynamic with two threads leads to better execution time than others. It is well known that dynamic can yield good performance [30] if an appropriate block size can be estimated (an extremely challenging task). In our case, the default block size appears to have hit an optimal point for KC on KM with two threads. However, this improved performance of dynamic is not consistently seen in other cases.

BF. In the evaluation of BF (as shown in Fig. 17), elastic again outperforms deep-chunk (as well as, other scheduling schemes) most of the time; the performance improvements over deep-chunk varied between -7.58% to 11.93% (0.93× to 1.14×) in DB while -13.27% to 10.55% (0.88× to 1.12×) in KM.

For elastic, the best results were observed when utilizing all available cores of a processor (16 threads on KM and 32 threads on DB) without interprocessor communication. We believe that the gains are getting slightly impacted as we use multiple processors and all the hardware threads. Despite large thread counts, the results still remained favorable for elastic. We observe a slight

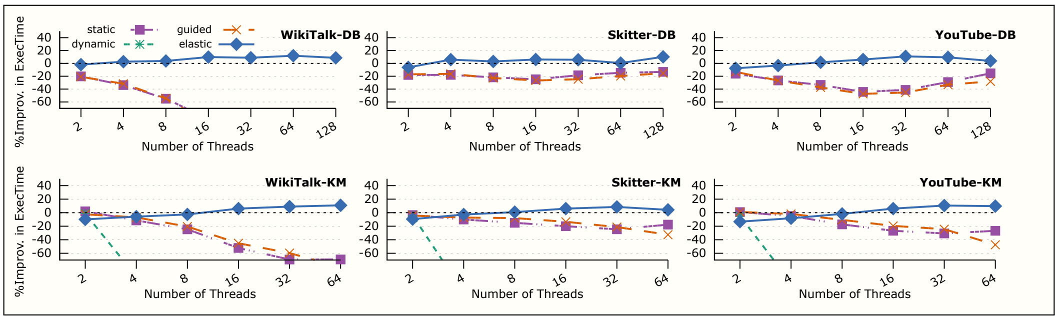


Fig. 17. Execution time analysis; BF

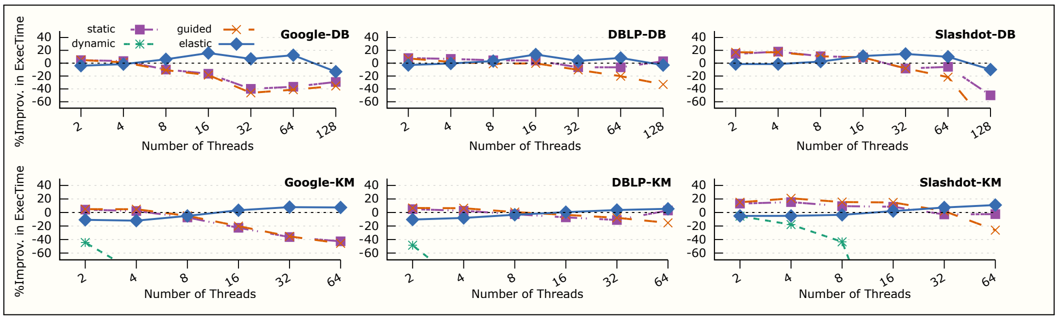


Fig. 18. Execution time analysis; DS

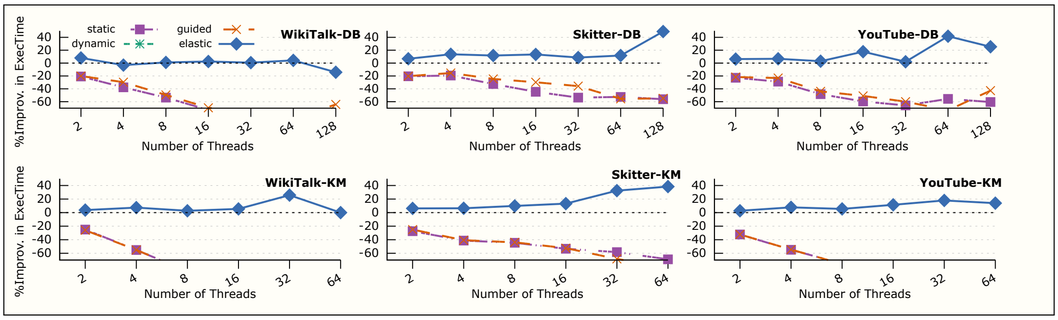


Fig. 19. Execution time analysis; PR

degradation in some cases with fewer threads (2, 4, or 8), but this quickly changes as the number of threads increases, leading to more EFTs and greater opportunities to perform elastic-work.

DS. We now analyze the results using the DS kernel (see Fig. 18). The figure shows that `elastic` generally performed better than `deep-chunk`, except for small number of threads (2/4/8) in some cases (for the same reasons as described for BF). As the number of threads exceeds half the maximum available (64 in the case of DB, and 32 in the case of KM), we see slight unpredictability in performance. We believe that it is due to the fact that the system starts using the SMTs here: SMTs can introduce non-deterministic factors like cache contention and memory overheads, leading to slight unpredictability in performance that became visible as DS did not have much scope for gains due to elastic-barriers, in the first place. The performance improvements varied between -13.23% to 15.78% (0.88× to 1.19×) in DB and -12.09% to 11.02% (0.89× to 1.12×) in KM.

PR. We now discuss the evaluation of the PR kernel as illustrated in Fig. 19. The results show that `elastic` performs better on both the systems for the Skitter and YouTube datasets, across all configurations. For WikiTalk, however, `deep-chunk` outperforms `elastic` in DB when using

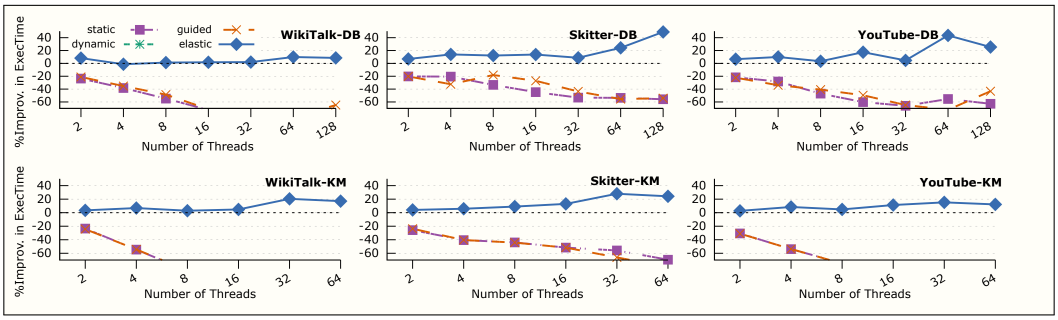
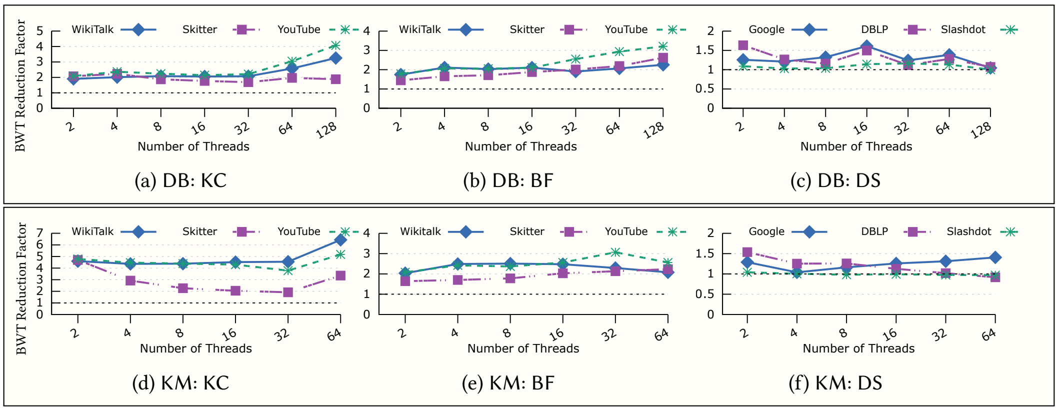


Fig. 20. Execution time analysis; PC



(a) DB: KC (b) DB: BF (c) DB: DS

(d) KM: KC (e) KM: BF (f) KM: DS

Fig. 21. Reduction in Barrier Waiting Time (BWT) using `elastic` compared to `deep-chunk` for KC, BF, DS.

larger number of threads with SMTs, as seen in Fig. 19. Overall, we see that the gain for `elastic` varied between -11.62% to 49.01% (0.90× to 1.96×) and 0.00% to 41.71% (1.00× to 1.72×) on DB and KM, respectively. The primary reason for improvements in `elastic` is the presence of an impactful serial-loop like the ones present in KC.

PC. We now present the evaluation over the PC kernel (see Fig. 20). We observe that `elastic` outperformed `deep-chunk` for the most part. Overall, we see that the gain for `elastic` varied between -1.31% and 48.90% (0.99× to 1.96×), and 2.70% to 27.99% (1.03× to 1.39×), on DB and KM, respectively. The primary reason for improvements in `elastic` is the presence of an impactful serial-loop like PR.

Summary across all the benchmarks and inputs. While the state-of-the-art `deep-chunk` outperforms all standard OpenMP schemes most of the time, `elastic` shows improvement over `deep-chunk`, with the highest percentage improvement being 49.01% (1.96×). We have tested that over the total 540 configurations (on varying benchmarks, number of threads, and inputs), `elastic` outperforms `deep-chunk` in 479 configurations.

6.2 Performance Analysis: Barrier Waiting Time

In Section 4, we discussed how we reduce the execution time by ensuring that the EFTs do elastic-work instead of simply waiting at barriers. We now present a study on how much reduction in waiting time (compared to `deep-chunk`) we achieved. We measured the waiting time for an execution by computing the sum of waiting times of every thread, across all the barriers (of the transformed parallel for-loops). We compute the reduction factor for a kernel K using the formula: (barrier waiting time for K using `deep-chunk`) / (barrier waiting time for K using `elastic`).

Fig. 21 showcases the waiting-time reduction factor in KC, BF, and DS kernels. Across all three inputs, `elastic` reduced the waiting time between 1.69× to 4.07× on DB and 1.91× to 6.43× on KM for the KC kernel. Similarly, for BF, the reduction factor varies between 1.45× to 3.21× and 1.64× to 3.07×, on DB and KM, respectively. For the DS kernel, the reduction factor is slightly lower, and ranges between 1.00x to 1.63x and 0.97x to 1.54x on DB and KM, respectively.

Fig. 22 shows the reduction in waiting time in the context of PR and PC. For PR, the reduction factor varied between 2.1× to 17.3× and 2.6× to 18.5×, on DB and KM, respectively. Similarly, for PC, the reduction factor ranged between 2.1× to 19.2× and 2.4× to 19.1×, on DB and KM, respectively.

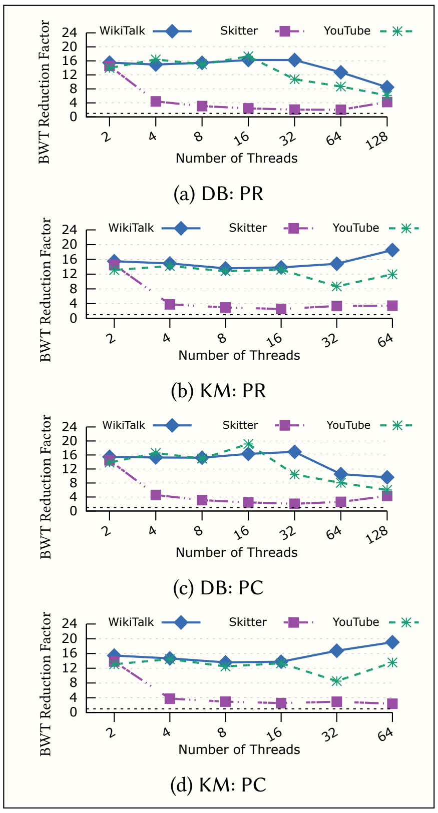


Fig. 22. Reduction in Barrier Waiting Time (BWT) using `elastic` compared to `deep-chunk` for PR and PC.

Overall, it can be seen that across all the benchmarks, for irregular inputs, our proposed scheme reduces the waiting time significantly. However, we see that it is difficult to compare the reduction in waiting times across the different benchmark kernels. The actual waiting times depend on multiple factors, such as the actual load imbalance (in `ParLoop1`s), available elastic-work (in `ParLoop2`s), and so on, all of which depend on the specific kernel and the input graph.

Note that lower waiting time itself does not necessarily guarantee lower execution time, as there are other contributing factors, such as amount of elastic-work performed, the actual reduction in the critical path, and other latent factors (based on memory, cache, and so on.). Further, the percentage reduction in execution time (shown in the previous section) depends not only on the reduction in the critical path, but also on the difference between the reduction in the critical path and the original running time. This difference, in turn, depends on multiple factors like: benchmark characteristics, specific input, the consequent available elastic-work (in `ParLoop2`s), precision of the computed workloads, effect of the transformed code on cache, and so on. Considering the complexity and interdependence among these factors, it becomes difficult to quantify the impact on the individual factors.

7 Related work

Reducing the Overheads of Barriers. Many prior researchers have identified the challenges due to the overheads of barriers and have proposed many techniques to reduce the same. These include lifting and placing the barriers appropriately [9, 39, 48, 53]; eliding redundant barriers [17, 27, 28, 31]; replacing barriers with a less expensive alternative like send-receive/post-wait signaling mechanism [33–35, 59, 60], or counters based approach [56]; and so on. In contrast, our proposed scheme can be applied to barriers that cannot be moved or elided (using the above techniques) to reduce the waiting time at the barriers. Thus, our technique can be used along with the above techniques, or techniques for efficiently implementing the barriers [15, 50, 58].

Scheduling. Many prior techniques have been proposed to handle the complex issue of scheduling iterations of parallel for-loops among the threads. These include the default *static*, *dynamic*, *guided* schemes present in OpenMP [4], besides many others [3, 5, 18, 21, 26, 30, 40, 45–47, 52, 55, 57] that have been proposed to minimize load imbalance in irregular parallel applications. Some of the recent works among these [30, 46, 52] make their scheduling decision based on the estimated workload. All of these schemes try to reduce the idling time of threads at any barrier by focusing on parallel-loops independently and changing the default assignment of iterations to threads. In contrast, for programs with multiple (irregular) parallel-loops, in our proposed scheme of elastic-barriers, the EFTs avoid waiting at the barriers and instead try to execute the iterations from the subsequent parallel for-loop. Thus, our proposed techniques can be applied along with the existing schemes of scheduling.

Tackling Idleness at the Barrier in Shared Memory Systems. There have been prior works that try to reduce the idling of threads at barriers. For example, similar to the idea of delay-slots [19], Gupta [14] introduced the concept of a fuzzy-barrier, which allows the programmer to specify additional instructions that an EFT can execute instead of simply idling at a barrier. Further, the idea of fuzzy-barriers is implemented in hardware. Another interesting approach was proposed by Pedrero et al. [41, 42], who proposed speculative execution of instructions present after the barrier, without waiting at synchronization points. This technique uses the underlying hardware support and utilizes Transactional Memory (TM) that allows synchronization among the concurrent threads to support the rollback, if needed. In contrast to these techniques that depend on hardware support, our scheme can run on any general-purpose system, and it also checks for dependences at runtime and can execute iterations of the next phase once the dependences are satisfied. One can imagine applying the ideas of thread-level speculation (TLS) [10] to elasticPatterns, wherein a thread completing its assigned iterations in the first parallel-loop, may execute iterations of the second parallel-loop speculatively. In contrast to TLS, our proposed scheme is non-speculative in nature and does not have to detect dependence-violations or perform rollback. It remains an interesting future work to extend the ideas of speculative execution (via TLS, or TM) for elasticPatterns, and compare them with our implementation of elastic-barriers. Rinard [49] proposed the idea of early termination of parallel computation (when very few parallel tasks are pending) to prevent idling processors, but this can distort results. Unlike our approach, which handles a variable range of remaining tasks, their method applies only when few tasks are left before termination. Their technique also addresses load imbalance by naively halting computation, whereas we focus on completing tasks safely, efficiently, and with minimal imbalance.

8 Conclusion

In this paper, we introduced a novel synchronization primitive called elastic-barrier, which allows waiting threads to (elastically) execute iterations from the next phase, instead of simply idling. We proposed a translation scheme that significantly minimizes load imbalance and improves overall execution time, for irregular parallel programs containing elastic-barriers. Additionally, we developed an efficient scheme to perform dependence check that is required to ensure safety of execution of the iterations that are being elastically executed. We implemented our proposed scheme in the IMOP framework, and evaluated it on on five popular kernels, with six input datasets across two platforms, and varying thread counts (leading to a total of 540 configurations). The results show that `elastic` outperforms `deep-chunk` in 479 out of 540 configurations, in terms of the overall execution time. Our approach also clearly surpasses statically scheduled chunk allocation schemes like `static`, runtime-managed schemes like `dynamic` and `guided`.

We have presented an optimized translation (and evaluation) for graph analytics based applications with elastic-barriers. However, we believe that our technique can be extended to other domains as well; we leave a detailed study of the same as a future work. Another very interesting future work relates to the study of the impact of such load balancing techniques on the cache behavior, and optimizing it for high performance.

Acknowledgments

We express our gratitude to the ACM TACO editor David Kaeli, associate editor Lixin Zhang, and the anonymous reviewers for their insightful suggestions, which contributed to enhancing the quality of this paper. We are especially grateful to Aman Nougrahiya for his constant help during the implementation in the IMOP compiler. This work is partially supported by the SERB CRG grant (sanction number CRG/2022/006971).

References

[1] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: an efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '95)*. Association for Computing Machinery, New York, NY, USA, 207–216. https://doi.org/10.1145/209936.209958

[3] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sep 1999), 720–748. https://doi.org/10.1145/324133.324234

[4] OpenMP Architecture Review Board. 2018. *OpenMP Application Programming Interface Version 5.0*. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[5] Joshua Dennis Booth and Phillip K. Lane. 2020. An adaptive self-scheduling loop scheduler. *Concurrency and Computation: Practice and Experience* 34 (2020). https://api.semanticscholar.org/CorpusID:226967766

[6] Eugene D. Brooks. 1986. The butterfly barrier. *International Journal of Parallel Programming* 15 (1986), 295–307. https://api.semanticscholar.org/CorpusID:31364842

[7] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.* 21, 3 (Aug. 2007), 291–312. https://doi.org/10.1177/1094342007078442

[8] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 519–538. https://doi.org/10.1145/1094811.1094852

[9] Alain Darte and Robert Schreiber. 2005. A linear-time algorithm for optimal barrier placement. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*. Association for Computing Machinery, New York, NY, USA, 26–35. https://doi.org/10.1145/1065944.1065949

[10] Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. 2016. A Survey on Thread-Level Speculation Techniques. *ACM Comput. Surv.* 49, 2, Article 22 (June 2016), 39 pages. https://doi.org/10.1145/2938369

[11] Rob Farber. 2016. *Parallel Programming with OpenACC* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[12] Wanrong Gao, Jianbin Fang, Chun Huang, Chuanfu Xu, and Zheng Wang. 2021. Optimizing Barrier Synchronization on ARMv8 Many-Core Architectures. https://doi.org/10.1109/Cluster48925.2021.00044

[13] Robert Griesemer, Rob Pike, and Ken Thompson. 2009. The Go Programming Language. https://golang.org. Accessed: 2024-11-09.

[14] Rajiv Gupta. 1989. The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*. Association for Computing Machinery, New York, NY, USA, 54–63. https://doi.org/10.1145/70082.68187

[15] Rajiv Gupta and Charles R. Hill. 1989. A Scalable Implementation of Barrier Synchronization Using an Adaptive Combining Tree. *International Journal of Parallel Programming* 18, 3 (June 1989), 161–180. http://dx.doi.org/10.1007/BF01407897

[16] Suyash Gupta and V. Krishna Nandivada. 2015. IMSuite. *J. Parallel Distrib. Comput.* 75, C (Jan 2015), 1–19. https://doi.org/10.1016/j.jpdc.2014.10.010

[17] Suyash Gupta, Rahul Shrivastava, and V Krishna Nandivada. 2017. Optimizing Recursive Task Parallel Programs. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. Association for Computing Machinery, New York, NY, USA, Article 11, 11 pages. https://doi.org/10.1145/3079079.3079102

[18] B. Hamidzadeh and D.J. Lilja. 1994. Self-Adjusting Scheduling: An On-Line Optimization Technique for Locality Management and Load Balancing. In *1994 Internatonal Conference on Parallel Processing Vol. 2*, Vol. 2. 39–46. https://doi.org/10.1109/ICPP.1994.179

[19] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[20] Debra Hensgen, Raphael Finkel, and Udi Manber. 1988. Two algorithms for barrier synchronization. *International Journal of Parallel Programming* 17 (02 1988), 1–17. https://doi.org/10.1007/BF01379320

[21] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. 1992. Factoring: a method for scheduling parallel loops. *Commun. ACM* 35, 8 (Aug 1992), 90–101. https://doi.org/10.1145/135226.135232

[22] Inbum Jung, Jongwoong Hyun, Joonwon Lee, and Joongsoo Ma. 2001. Two-Phase Barrier: A Synchronization Primitive for Improving the Processor Utilization. *International Journal of Parallel Programming* 29 (12 2001), 607–627. https://doi.org/10.1023/A:1013153020460

[23] A. Kejariwal, Alexandru Nicolau, and Constantine Polychronopoulos. 2006. History-aware Self-Scheduling. *Proceedings of the International Conference on Parallel Processing*, 185–192. https://doi.org/10.1109/ICPP.2006.49

[24] Amy N. Langville and Carl D. Meyer. 2012. *Google's PageRank and Beyond: The Science of Search Engine Rankings.* Princeton University Press, USA.

[25] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[26] Steven Lucco. 1992. A dynamic scheduling method for irregular parallel programs. *SIGPLAN Not.* 27, 7 (jul 1992), 200–211. https://doi.org/10.1145/143103.143134

[27] Hongtu Ma, Rongcai Zhao, Xiang Gao, and Youwei Zhang. 2009. Barrier Optimization for OpenMP Program. 495 – 500. https://doi.org/10.1109/SNPD.2009.16

[28] S. P. Midkiff and D. A. Padua. 1987. Compiler algorithms for synchronization. *IEEE Trans. Comput.* 36, 12 (dec 1987), 1485–1495. https://doi.org/10.1109/TC.1987.5009499

[29] Marvin Minsky and Seymour Papert. 1969. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, Cambridge, MA, USA.

[30] Prasoon Mishra and Krishna Nandivada. 2023. COWS for High Performance: Cost Aware Work Stealing for Irregular Parallel Loops. *ACM Transactions on Architecture and Code Optimization* 21 (11 2023). https://doi.org/10.1145/3633331

[31] V. Krishna Nandivada, Jun Shirako, Jisheng Zhao, and Vivek Sarkar. 2013. A Transformation Framework for Optimizing Task-Parallel Programs. *ACM Trans. Program. Lang. Syst.* 35, 1, Article 3 (April 2013), 48 pages. https://doi.org/10.1145/2450136.2450138

[32] Ramachandra Nanjegowda, Oscar Hernandez, Barbara Chapman, and Haoqiang Jin. 2009. Scalability Evaluation of Barrier Algorithms for OpenMP, Vol. 5568. 42–52. https://doi.org/10.1007/978-3-642-02303-3_4

[33] Alexandru Nicolau. 1985. *Percolation Scheduling: A Parallel Compilation Technique.* Technical Report. Ithaca, NY, USA.

[34] Alexandru Nicolau, Guangqiang Li, and Arun Kejariwal. 2009. Techniques for efficient placement of synchronization primitives. *SIGPLAN Not.* 44, 4 (Feb. 2009), 199–208. https://doi.org/10.1145/1594835.1504207

[35] Alexandru Nicolau, Guangqiang Li, Alexander V. Veidenbaum, and Arun Kejariwal. 2009. Synchronization optimizations for efficient execution on multi-cores. In *Proceedings of the 23rd international conference on Supercomputing (ICS '09).* ACM, New York, NY, USA, 169–180. https://doi.org/10.1145/1542275.1542303

[36] A Nougrahiya and K Nandivada. 2018. IIT Madras OpenMP (IMOP) Framework.

[37] A Nougrahiya and K Nandivada. 2019. IMOP: A Source-to-Source Compiler Framework for OpenMP C Programs.

[38] Aman Nougrahiya and V. Krishna Nandivada. 2024. Homeostasis: Design and Implementation of a Self-Stabilizing Compiler. *ACM Transactions on Programming Languages and Systems* 46, 2, Article 6 (May 2024), 58 pages. https://doi.org/10.1145/3649308

[39] M. O'Boyle and E. Stohr. 2002. Compile time barrier synchronization minimization. *IEEE Transactions on Parallel and Distributed Systems* 13, 6 (2002), 529–543. https://doi.org/10.1109/TPDS.2002.1011394

[40] Fatma Omara and Doaa Abdelkader. 2012. Dynamic Task Scheduling Algorithm with Load Balancing for Heterogeneous Computing System. *The Egyptian Informatics Journal* 13 (07 2012), 135–145. https://doi.org/10.1016/j.eij.2012.04.001

[41] Manuel Pedrero, Eladio Gutierrez, and Oscar Plata. 2018. TMbarrier: Speculative Barriers Using Hardware Transactional Memory. , 214-221 pages. https://doi.org/10.1109/PDP2018.2018.00036

[42] Manuel Pedrero, Ricardo Quislant, Eladio Gutierrez, Emilio L. Zapata, and Oscar Plata. 2022. Speculative Barriers With Transactional Memory . *IEEE Trans. Comput.* 71, 01 (Jan. 2022), 197–208. https://doi.org/10.1109/TC.2020.3044234

[43] D. Peleg. 2000. *Distributed computing: A Locality-Sensitive Approach.* Society for Industrial and Applied Mathematics.

[44] Chuck Pheatt. 2008. Intel® threading building blocks. *J. Comput. Sci. Coll.* 23, 4 (April 2008), 298.

[45] C. D. Polychronopoulos and D. J. Kuck. 1987. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput.* 36, 12 (Dec 1987), 1425–1439. https://doi.org/10.1109/TC.1987.5009495

[46] Indu K. Prabhu and V. Krishna Nandivada. 2020. Chunking Loops with Non-Uniform Workloads. In *Proceedings of the 34th ACM International Conference on Supercomputing (ICS '20).* Association for Computing Machinery, New York, NY, USA, Article 40, 12 pages. https://doi.org/10.1145/3392717.3392763

[47] Lutz Prechelt and Stefan U. Hánßgen. 2002. Efficient Parallel Execution of Irregular Recursive Programs. *IEEE Trans. Parallel Distrib. Syst.* 13, 2 (feb 2002), 167–178. https://doi.org/10.1109/71.983944

[48] Harenome Razanajato, Cédric Bastoul, and Vincent Loechner. 2017. Lifting Barriers Using Parallel Polyhedral

Regions. *2017 IEEE 24th International Conference on High Performance Computing (HiPC)* (2017), 338–347. https://api.semanticscholar.org/CorpusID:3423661

[49] Martin C. Rinard. 2007. Using early phase termination to eliminate load imbalances at barrier synchronization points. *SIGPLAN Not.* 42, 10 (Oct 2007), 369–386. https://doi.org/10.1145/1297105.1297055

[50] Michael Scott and John Mellor-Crummey. 2000. Fast, Contention-Free Combining Tree Barriers for Shared-Memory Multiprocessors. *International Journal of Parallel Programming* 22 (02 2000). https://doi.org/10.1007/BF02577741

[51] Jun Shirako, Jisheng M. Zhao, V. Krishna Nandivada, and Vivek N. Sarkar. 2009. Chunking Parallel Loops in the Presence of Synchronization. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*. Association for Computing Machinery, New York, NY, USA, 181–192. https://doi.org/10.1145/1542275.1542304

[52] Rahul Shrivastava and V. Krishna Nandivada. 2017. Energy-Efficient Compilation of Irregular Task-Parallel Loops. *ACM Trans. Archit. Code Optim.* 14, 4, Article 35 (Nov 2017), 29 pages. https://doi.org/10.1145/3136063

[53] Elena Stöhr and Michael F. P. O'Boyle. 1997. Barrier Synchronisation Optimisation. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking (HPCN Europe '97)*. Springer-Verlag, Berlin, Heidelberg, 791–800.

[54] Tatiana Tabirca, Len Freeman, Sabin Tabirca, and Laurence Yang. 2001. Feedback guided dynamic loop scheduling; A theoretical approach. 115 – 121. https://doi.org/10.1109/ICPPW.2001.951913

[55] Peter Thoman, Herbert Jordan, Simone Pellegrini, and Thomas Fahringer. 2012. Automatic OpenMP Loop Scheduling: A Combined Compiler and Runtime Approach, Vol. 7312. 88–101. https://doi.org/10.1007/978-3-642-30961-8_7

[56] Chau-Wen Tseng. 1995. Compiler optimizations for eliminating barrier synchronization. *SIGPLAN Not.* 30, 8 (Aug 1995), 144–155. https://doi.org/10.1145/209937.209952

[57] T. H. Tzen and L. M. Ni. 1993. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. *IEEE Trans. Parallel Distrib. Syst.* 4, 1 (jan 1993), 87–98. https://doi.org/10.1109/71.205655

[58] Akshay Utture and V Krishna Nandivada. 2019. Efficient Lock-Step Synchronization in Task-Parallel Languages. *Software: Practice and Experience* 49, 9 (2019), 1379–1401. https://doi.org/10.1002/spe.2726 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2726

[59] Naoki Yonezawa and Koichi Wada. 2005. Eliminating barrier synchronizations in OpenMP programs for PC clusters. *PACRIM. 2005 IEEE Pacific Rim Conference on Communications, Computers and signal Processing, 2005.* (2005), 273–276. https://api.semanticscholar.org/CorpusID:31204416

[60] Naoki Yonezawa, Koichi Wada, and Takahiro Aida. 2006. Barrier elimination based on access dependency analysis for OpenMP. In *Proceedings of the 4th International Conference on Parallel and Distributed Processing and Applications (ISPA'06)*. Springer-Verlag, Berlin, Heidelberg, 362–373. https://doi.org/10.1007/11946441_36