Chunking Loops with non-Uniform Workloads

Indu K. Prabhu
IIT Madras, India
indukprabhu@gmail.com

V. Krishna Nandivada
IIT Madras, India
nvk@iitm.ac.in

ABSTRACT

Task-parallel languages such as X10 implement dynamic lightweight task-parallel execution model, where programmers are encouraged to express the ideal parallelism in the program. Prior work has used loop chunking to extract useful parallelism from ideal. Traditional loop chunking techniques assume that iterations in the loop are of similar workload, or the behavior of the first few iterations can be used to predict the load in later iterations. However, in loops with non-uniform work distribution, such assumptions do not hold. This problem becomes more complicated in the presence of atomic blocks (critical sections).

In this paper, we propose a new optimization called *deep-chunking* that uses a mixed compile-time and runtime technique to chunk the iterations of the parallel-for-loops, based on the runtime workload of each iteration. We propose a parallel algorithm that is executed by individual threads to efficiently compute their respective chunks so that the overall execution time gets reduced. We prove that the algorithm is correct and is a 2-factor approximation. In addition to simple parallel-for-loops, the proposed deep-chunking can also handle loops with atomic blocks, which lead to exciting challenges. We have implemented deep-chunking in the X10 compiler and studied its performance on the benchmarks taken from IMSuite. We show that on an average, deep-chunking achieves 50.48%, 21.49%, 26.72%, 32.41%, and 28.84% better performance than un-chunked (same as work-stealing), cyclic-, block-, dynamic-, and guided-chunking versions of the code, respectively.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**; • **Theory of computation** → **Program analysis**; • **Software and its engineering** → **Compilers**.

KEYWORDS

Concurrent programs, loop chunking, parallel loop optimization

ACM Reference Format:
Indu K. Prabhu and V. Krishna Nandivada. 2020. Chunking Loops with non-Uniform Workloads. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3392717.3392763

1 INTRODUCTION

Modern languages such as Cilk [3], Chapel [5], HJ [14], X10 [6], and so on, employ dynamic lightweight task-parallel execution models. These languages help the programmers express the ideal parallelism inherent in the underlying application logic, and the challenging task of extracting the useful parallelism is left to the compiler and/or the runtime system. Chunking [22, 29] is a popular mechanism to bridge the gap between the ideal and useful parallelism.

Loop-chunking is the process of dividing the iterations of a parallel-for-loop into C number of chunks, so that each chunk of iterations is executed by a different thread, at runtime. Typically C is set to #runtime-threads (= #hardware-cores). The elements in each chunk are decided by the underlying chunking policy: for example, block-chunking assigns equal number of contiguous iterations to each chunk, and cyclic-chunking assigns iteration `i` to chunk `i mod C`. It has been found that chunking significantly reduces the overheads to create, terminate, and synchronize tasks [26].

Languages such as OpenMP [27], and Intel Thread Building Blocks [32] allow the programmer to specify/tune the chunking policy. Nandivada et al. [26] show that automated loop-chunking is effective for parallel-for-loops in X10 programs. Prior work [34, Section 5.2] has suggested that not only chunking is very effective, the chosen chunking policy greatly determines the gains obtained from loop-chunking, and its choice depends on the specific application under consideration; however, they do not provide any solution to automate this choosing process. Further, the problem of specifying the right chunking policy (by the compiler/programmer) becomes harder as that also depends on input dependent parameters. We first show an example to illustrate this latter dependence.

Figure 1 shows a code snippet (written in a language similar to X10 [33]), from the Byzantine benchmark (from IMSuite [12]). At line 1, the `forall` construct creates `nodes.size()` number of tasks, and each task may execute the body in parallel. The Byzantine kernel achieves consensus in a byzantine network. Here the execution-time (workload) of the iterations of the `forall` loop, depend on the predicate of the `if`-statement (Line 2), and loop-bounds of the two nested loops (Lines 4 and 7). These parameters are input dependent and may vary during the program execution. Thus, it is hard to identify an optimal chunking policy for parallel-for-loops with such non-uniform workloads.

To address these issues, it is vital to go beyond distributing the iterations equally and instead focus on optimal distribution of the workload. Many prior works take cognizance of this insight and dynamically distribute the iterations. For example, OpenMP supports *dynamic* and *guided* scheduling wherein the worker threads continuously retrieve blocks of loop iterations from a common work-queue. Similarly, the Cilk and X10 runtime schedulers use work stealing queues to achieve a balanced workload among threads. Though, as shown by Nandivada et al. [26], the default (work-stealing) runtime performs much worse than even the simple block-chunked code. This is because all of these runtime based techniques incur synchronization related overheads (to different degrees). Further, since these are runtime techniques, they do not take into consideration

ICS '20, June 29-July 2, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7983-0/20/06…$15.00
https://doi.org/10.1145/3392717.3392763

```
1  forall(i=0; i<nodes.size(); i++) {
2   if(nodes(i).msgHolder.size() > 0){
3    var m:Message = new Message();
4    for(j=0;j<nodes(i).msgHolder.size();j++){
5     m=nodes(i).msgHolder.get(j);
6     nei=nodes(i).neighbors;
7     for(k=0;k<nei.size();k++){
8      sendMsg(nei(k), m.source, m.vote);}}}
9   else {...}  }
```

Figure 1: Snippet from Byzantine kernel [12].

the overall program behavior; consequently the task scheduling decisions are local in nature. Such decisions can be highly suboptimal, especially in the presence of non-uniform workloads.

In this paper, we present a new optimization called *deep-chunking* to dynamically chunk parallel-for-loops with non-uniform workloads. Deep-chunking is based on a combined compile-time and runtime approach. It uses a compile-time analysis to emit code to estimates the workload (at runtime) and perform chunking dynamically. It uses a profile guided compile-time analysis to analyze each parallel-for-loop and formulate a cost-expression for the workload of each iteration therein; these cost-expressions are efficiently evaluated at runtime to obtain the workloads of all the iterations. These workloads are then used by the individual threads to compute the respective chunks at runtime, with minimal overheads.

The problem of efficient chunking becomes more challenging in the presence of atomic blocks (enforce global mutual-exclusion in X10). Increasing number of threads participating in atomic blocks increase the overheads associated with the atomic blocks [11]. Thus sometimes, it may be beneficial to create fewer chunks than the number of runtime threads. In other words, in the presence of atomic blocks, we have to carefully calculate the number of chunks to create, along with the distribution. Deep-chunking proposes a scheme (inspired by prior work [11, 35, 41]) to model the behavior of atomic blocks, in order to identify the optimal number of chunks to be created and then perform the actual chunking effectively. For example, for the Byzantine kernel (referred in Figure 1), on a 64 core AMD system, for an input size of 512, the execution times of block-, cyclic-, dynamic-, guided- and deep-chunking versions, normalized with respect to that of the default work-stealing version, are 0.95, 0.94, 0.94, 0.86, and 0.28, respectively. This clearly shows the effectiveness of deep-chunking.

Even though we present the idea of deep-chunking in the context of X10, it can be used in the context of other task-parallel languages (such as HJ, Chapel and OpenMP) that admit similar parallel loops.

Contributions:

- We propose a new optimization called deep-chunking to chunk parallel-for-loops with non-uniform workloads. For each parallel-for-loop, we emit application specific chunking code that is parametric on the input values. We compute the values of these parameters at runtime with low overheads.
- We propose a concurrent lock-free algorithm to efficiently divide the iterations of the parallel-for-loop among the threads at runtime, based on their estimated workloads. We prove that the proposed algorithm 1) leads to a valid set of chunks and 2) is a 2-factor approximation algorithm.
- We extend deep-chunking to handle atomic blocks by modeling the impact of atomic blocks on the overall workload of the corresponding parallel-for-loops.
- We have implemented deep-chunking in the X10 compiler and studied its performance on IMSuite [12] benchmarks. We show that on average, deep-chunking achieves significantly better performance than the work-stealing, cyclic-, block-, dynamic-, and guided-chunking versions of the code.

1.1 Related Works

Loop chunking for X10: Nandivada et al. [26] describe how to chunk (X10) parallel-for-loops that contain synchronization constructs like barriers. Similarly, Gupta et al. [13] describe the chunking of recursive task-parallel programs. Although these works help in chunking the loops and realizing improved performance (compared to the un-chunked code), they do not identify the most appropriate chunking policy. In contrast, our proposed deep-chunking uses a mixed compile-time and runtime approach to efficiently chunk parallel-for-loops, without any programmer intervention.

Loop Scheduling: In the context of parallel programs, loop-chunking is a form of loop scheduling [21] where the iterations of a chunk execute sequentially. The scheduling can be done both statically (for example, block-, and cyclic-scheduling) or at runtime [17, 22, 23, 29, 40]. BinLPT [28] is a loop scheduler for irregular parallel loops, that requires user-supplied estimation of workload to partition the iteration space. Lazy Binary Splitting [39] is a user-level scheduler for programs with nested parallelism, where it uses dynamic conditions to decide on whether to fork new threads/-tasks or not. Compared to all these techniques, deep-chunking uses a mixed compile-time and runtime approach to chunk parallel-for-loops with non-uniform workloads, and needs no programmer intervention. Further, our handling of atomic blocks is novel. It would be interesting to extend our work with some of the prior techniques of loop scheduling [1, 8, 10, 15, 16, 24, 36, 37] to improve processor affinity and cache locality.

The OpenMP [27] language supports dynamic and guided scheduling. The dynamic- and guided-chunking involve high overheads, which are proportional to the number of iterations in the parallel-for-loop. Further, the chunking policies of OpenMP are oblivious to the structure of the input program and the input parameters. Similar to OpenMP, Intel TBB [32] supports controlled and automatic chunking for load balancing. Languages such as Cilk [3] and X10 [6] supports work stealing schedulers (oblivious to the workload of the task) that allow stealing of tasks at runtime, in order to keep the load balanced. Nandivada et al. [26] show the inefficiency of the default work-stealing based scheduling and instead showed that basic chunking policies (like blocked-/cyclic-chunking) lead to significant benefits. In contrast, deep-chunking is an application and input aware loop-chunking technique, where at runtime, the chunks are computed (based on the workloads) before starting the parallel-for-loop, which in turn avoids the overheads of continuous synchronization to "steal tasks" during the execution of the loop.

Recently Thoman et al. [38] present a smart scheme that uses a combined compiler and runtime scheme to schedule OpenMP

loops. They use a compiler pass that computes 'effort estimation functions' for the parallel loop bodies, and these are passed to the runtime system (a separate parallel thread) to derive optimal loop schedule. Their compiler pass requires that the loops, if-conditions, and the indices of loops must be affine in nature, and there should be no use of heap data structures. Further, they do not handle critical sections. For example, for the code shown in Figure 1, their technique would lead to chunks assuming that the workload of each iteration is the same; thereby leading to inefficiency similar to block-chunking. In contrast, our techniques can handle "extractable" (see Section 5) non-affine loops and conditions; and we do not have any restriction on the data-structures used. We do not require a separate monitoring thread, thereby avoid any associated overheads (synchronization, thread contention, and so on). Importantly, we handle atomic blocks, which is essential in many of the real-world parallel programs.

Profile Guided Optimization: Profile guided optimization has also been used in the context of improving the performance of parallel programs. For example, Chen et al. [7] use the profile data to improve the mapping of processes to different processors. Bull [4] uses the (online) profiling information collected from the execution of one instance of a parallel-loop to aid in loop-scheduling in the future instances. Kejriwal et al. [20] use the history of currently executing loop to guide the chunk size of the later iterations. Duran et al. [9] use online profiling to compute an optimal adaptive cut-off to decide (using the level of recursion) if the tasks should be serialized. In this paper, we use offline profiling to compute the cost of input independent parts of the code, to help in efficient computation of the workload at runtime to improve loop-chunking.

2 BACKGROUND

We now give some brief background of i) LX10 (a subset of X10) over which we describe our schemes, ii) PSGb (an extension to the Program Structure Graph [26]), and iii) different chunking policies.

LX10 Language is a strict subset of X10 [33], with added syntactic sugar to specify parallel-for-loops. LX10 admits only one other concurrency related construct: `atomic`. A parallel loop in LX10 can be derived from the following grammar:

```
PLoop   ::=  /* Parallel loop */   forall (loop-header) S
S       ::=  /* Statements */      atomic S | seq(S) | ϵ
```

For a non-terminal `X`, we use *seq*(`X`) to denote the program formed from `X` by closing under sequential constructs, such as assignments, declarations, if statement, function-calls, loops, and so on. A method body can be derived from *seq*(`PLoop`).

The `forall` statement (example in Figure 2(a)) derives a parallel loop that creates `n` number of tasks, where each task may execute the body `S` in parallel. Each task waits for each other at the end of the loop. `forall` terminates only after all the tasks have completed. LX10 prohibits the nesting of `forall` loops, at runtime. Without any loss of generality, we assume that each parallel loop is in normal form [25] – the loop-index starts at 0, increases by 1 in each successive iteration, and the loop upper bound is loop invariant.

The `atomic` statement (of the form `atomic S`) realizes global mutual exclusion. Nesting of `atomic` blocks is prohibited in X10.

Program Structure Graph with blocks (PSGb) is a minor extension to the Program Structure Graph (PSG) representation [26]. A

```
forall(j=0 to n−1){ S }
```

(a) Normalized loop

```
forall(i=0 to T-1) {
  for (j=i;j<n;j+=T){S}}
```

(b) Cyclic-chunked loop

```
forall(i=0 to T-1){
 for(j=bS*i;
     j<min(n,(i+1)*bS);
     j++) { S } }
```

(c) Block-chunked loop

Figure 2: Example input loop and code after chunking

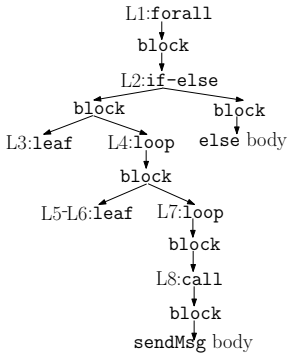


Figure 3: PSGb for the code shown in Figure 1.

PSGb is a rooted graph (N, E), where each node $o \in N$ can have one of the following types: `root`, `forall`, `atomic`, `loop`, `if-else`, `block`, `call` and `leaf`. The `root` represents the start of a method. The `loop` represents any serial loop. The `leaf` represents a sequence of statements except `forall`, `atomic`, `loop`, `if-else`, `block`, and `call`. The `block` represents a sequence of zero or more nodes of PSGb (except `root`). The `call` node represents a function-call; the child of a `call` node is a `block` node representing the body of the callee, with additional prologue code to copy the actual arguments to the formal ones. All of the following nodes will have only one child node: `forall`, `atomic`, `loop` and `call`. Similarly, an `if-else` node has two child nodes.

The set E contains the edges obtained by collapsing the abstract syntax tree of the program into the above seven node types (except `call`), along with special call edges (from `call` node to its body).

Figure 3 shows the PSGb for the sample code snippet shown in Figure 1; we treat the unknown (library) calls as simple statements.

We now list some attributes (a.k.a fields) of the PSGb nodes. (i) `if-else` node o: The fields $o.then$ and $o.else$ give the nodes representing the true and false branches, respectively. Further $o.condition$ gives the LX10 expression corresponding to the predicate of the if-else construct. (ii) `loop` node o: It includes four fields *prologue*, *epilogue*, *body* and *loopBound*. For a loop of the form `for(init; expr;update) S`, *prologue* refers to the part of the loop consisting of the code for `init` and `expr`; *epilogue* refers to the part of the loop consisting of the code for `expr` and `update`; *body* refers to the loop body (`S`); and *loopBound* is the expression given by `expr`. In case of a while loop, the `init` and `update` are considered to be 'nop's. (iii) `call` node o: $o.calee$ gives the body of the callee. (iv) `block` node o: $o.nodes$ gives the list of nodes in o.

Chunking policies. Cyclic-chunking: The iteration j of a parallel-for-loop is added to the chunk $j \% T$, where T=#runtime threads. Figure 2(b) shows the cyclic-chunked code for Figure 2(a).

Block-chunking: We divide the iterations of a parallel-for-loop into chunks such that each chunk gets contiguous iterations. Let n be the number of iterations of the parallel-for-loop and the number of chunks be set to T. If n is divisible by T, each chunk gets n/T number of iterations. If n is not divisible by T, then the first $T-1$ chunks get $\lceil n/(T) \rceil$ iterations and the last chunk gets the remaining

iterations. Figure 2(c) shows a sample block-chunked code for the loop shown in Figure 2(a). Here bS refers to the block size = n/T.

Dynamic-chunking (pure self scheduling, with chunksize=K): A centralised queue is maintained so that whenever a worker is idle, K number of iterations from the queue are assigned to it.

Guided-chunking (guided self scheduling): Here, the number of iterations assigned to an idle worker is given by a formula: max(K, (#remaining iterations)/(#worker threads)), where K is a constant.

3 DEEP CHUNKING

In this section, we describe our proposed deep-chunking (Dynamic EfficiEnt Parallel/Parametrized chunking) technique. Our objective is to transform an input LX10 program P to a semantically equivalent LX10 program P', such that the parallel-for-loops in P' are efficiently chunked at runtime; we will assume that there are T number of threads and hence the number of chunks is also equal to T. A chunk is a sequence of zero or more contiguous iterations and each chunk can be represented by a range $[s, e]$, where s and e mark the starting and ending iterations (both inclusive) of the chunk. A chunk $[s, e]$ is considered *non-empty* if $s \leq e$, else it is considered *empty*. Thus, the task of chunking a sequence of iterations $1, \cdots, n$ into T chunks is to compute an ordered list L of T ranges $\{[1, e_1], [s_2, e_2], \ldots, [s_T, n]\}$, such that (a) $\forall_{1 \leq i \leq T}\ 1 \leq s_i \leq n+1$, (b) $\forall_{1 \leq i \leq T}\ 1 \leq e_i \leq n$, and (c) $\forall_{1 \leq i \leq T-1}\ s_{i+1} = 1 + e_i$. The list L may contain empty chunks.

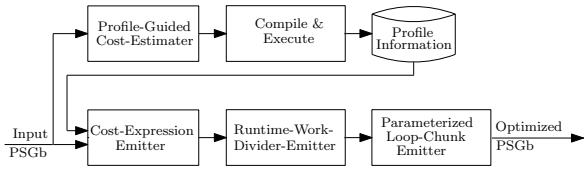
Ideally, loop-chunking should result in a set of chunks, such that each chunk has the same execution time – referred to as *balanced* chunking. If every iteration of the parallel-for-loop does the exact same amount of work, then we can get a (near) balanced set of chunks by simply partitioning the iterations equally (using block- / cyclic-distribution, see Section 2); However, for parallel-for-loops with non-uniform workloads, balanced chunking is challenging.

To handle parallel-for-loops with non-uniform workloads, instead of partitioning the iterations equally, we focus on the *workload* of each chunk. Workload of an iteration is an approximation of its execution time, and the workload of a chunk is the sum of the workloads of all the iterations assigned to the chunk. Among the chunks created, the chunk with maximum workload (= the critical path [2] of the parallel-for-loop) is called the *max-chunk*. We aim to create the chunks such that the workload (*a.k.a* cost) of our estimated max-chunk is as close to that of the hypothetical optimum.

Our proposed approach has two components: a) the compile-time component that emits parametrized chunked loops, and b) the runtime-component where each thread computes the values of these parameters and passes them to the parametrized code to obtain/execute efficient loop-chunks. For the ease of explanation, in this section, we will assume that the parallel-for-loops in the input LX10 programs do not have `atomics`. Later in Section 4, we extend our scheme to handle `atomics`. We now elaborate on the above two components.

3.1 Compile-time component

Figure 4(a) shows the overall block diagram of our compile-time component. Note that the workload of any iteration may consist of two parts: input independent part and input dependent part. Thus the workload of each iteration can be represented as an expression



(a) Compile-time component

(b) Run-time component

Figure 4: Overall technique

(called cost-expression) of input dependent parameters and some constants (corresponding to the input independent parts). Profile-Guided-Cost-Estimator processes each parallel-for-loop in the input PSGb to generate an instrumented code, which is then executed (on a small input) to estimate the workloads of the input independent (serial) parts. The estimated workloads and the input PSGb are fed to Cost-Expression-Emitter, which generates cost-expressions for each iteration (parameterized by the iteration index) and emits code to evaluate the same at runtime. Work-Divider-Emitter emits code that is to be executed by each thread. This code uses the list of workloads (obtained by evaluating the cost-expressions at runtime) to generate a chunk for each thread. Parameterised-Loop-Chunk-Emitter emits chunked loops parametrized by the list of chunks. We now describe each of these sub-components.

3.1.1 Profile-Guided-Cost-Estimator . This phase first generates a serial version of the input PSGb by replacing all the `forall` keywords with `for`. Then it instruments each of the leaf nodes, such that on executing the serial code, we get the execution times of each leaf nodes in the input parallel program. For each such leaf node B, we compute the average of the execution times (denoted by $Profile(B)$). Note that, the number and types of instructions executed in each of these leaf nodes is independent of the program input. We use a serial version of the input parallel-for-loop for instrumentation, so as to avoid interference/overheads due to the parallel threads. Also note that naively assigning unit cost to each leaf node (and thereby avoiding profiling) may lead to highly inaccurate cost-estimates as the number of instructions in the leaf nodes may vary significantly.

3.1.2 Cost-Expression-Emitter. The PSGb corresponding to the input program P along with the generated profile information is given as input to the Cost-Expression-Emitter. It processes each parallel-for-loop of P independently to generate the cost-expression for each iteration thereof. For any given parallel-for-loop $\mathcal{L}$, the cost-expression for any of its iteration is given by a unique syntactic expression, parameterised by the loop-iteration index. For example, for the code in Figure 1 the cost-expression for iteration `i` is given by: `nodes(i).msgHolder.size()>0 ?` C_3 `+ nodes(i).msgHolder.size()*(`C_{5_6}`+`C_8`*nodes(i).neighbors.size()):`Str_9, where C_x is the workload of statement(s) labeled x (as computed by the Profile-Guided-Cost-Estimator), and Str_j is the cost expression of the statement

```
1  Function String costFunc(ℒ, S)
       Input: ℒ: PSGb node for a parallel-for-loop, S: a child of ℒ
2      switch type of S do
3          case loop node do
4              LBound = extractExpr(ℒ, S.loopBound) ;
5              if LBound≠null then // extraction successful
6                  C_e = Profile(S.prologue) || "+" || LBound || "*(" ||
                       costFunc(ℒ, S.body)||"+"||Profile(S.epilogue)||")";
7              else C_e = "K";// K: a large constant ;
8          case if-else node do
9              cond = extractExpr(ℒ, S.condition);
10             if cond ≠ null then // extraction success
11                 C_e = Profile(S.condition) || "+(" || cond || "?" ||
                       costFunc(ℒ, S.then) || ":" || costFunc(ℒ, S.else) || ")";
12             else
13                 C_e = Profile(S.condition) || "+max(" costFunc(ℒ, S.then)
                       || "," || costFunc(ℒ, S.else) || ")";
14         case call node do
15             if S does not correspond to a recursive call then
16                 C_e = costFunc(ℒ, S.callee);
17             else C_e = "K" ;
18         case block node do
19             for s in S.nodes do C_e=C_e||"+"||costFunc(ℒ, s) ;
20         otherwise do C_e = Profile(S) ; // leaf node
21     return C_e
```

Figure 5: Recursive function for cost-expression generation

labeled j. After these expressions are generated, Cost-Expression-Emitter emits the appropriate LX10 code (just before the parallel-for-loop) to evaluate the cost-expressions at runtime. These computed workloads are used for chunking.

Generating the Cost-expressions: We perform a post-order traversal of the input PSGb. Since the workload of the leaf nodes has already been obtained from the profile information, we compute the workload of each intermediate node as an expression of some input dependent parameters and workloads of the children nodes. Figure 5 shows how we generate the cost-expressions at different intermediate nodes. For each parallel-for-loop node $\mathcal{L}$ in the PSGb, the function *costFunc* is invoked, by passing $\mathcal{L}$ and the child node S of $\mathcal{L}$. Note: S corresponds to the body of the $\mathcal{L}$. The function *costFunc* returns a string, denoting the cost-expression for S.

The cost-expression generated by *costFunc* depends on the type of S. If S is a `loop` node, then it first invokes the function *extractExpr* by passing $\mathcal{L}$ and *S.loopBound* as arguments. Note that we cannot simply use profiling to estimate the value of loop-bound using profiling, as it may be input dependent. The *extractExpr* function (code not shown) tries to map the expression in *S.loopBound* to another expression in terms of variables live immediately before $\mathcal{L}$. For example, in Figure 1, for the `loop` at Line 7, the function *extractExpr* returns the string `"nodes(i).neighbors.size()"`. If the extraction succeeds, we generate a string that represents the cost of the `loop` body, executed *LBound* number of times. For example, in Figure 1, for the `loop` at Line 7 the generated cost-expression string is: `"`C_{p_7}`+nodes(i).neighbors.size()*(`C_8`+`C_{e_7}`)"`, where C_{p_7}, C_8, and C_{e_7} represent the costs of executing the *S.prologue*, *S.body* (Statement at Line 8) and *S.epilogue* (refer Section 2), respectively. The function *extractExpr* returns *null*, if it fails to map the expression (say, in case of for-loops whose loop-bound changes within $\mathcal{L}$, or while-loops). In such a case, we set the variable C_e (denoting the cost expression) to a conservative value K (a large constant). Section 5 discusses the implementation details of *extractExpr* and the impact of the value of K on chunking.

If S is an `if-else` node, then as in the case of a `loop` node we first try to extract the predicate *S.condition* (by calling *extractExpr*). If the extraction is successful, C_e is set to a conditional expression which uses *S.condition* to choose the appropriate cost-expression from *S.then* and *S.else*. Otherwise, C_e is set to an expression to compute the maximum of the costs of *S.then* and *S.else*.

If S is a `call` node then we recursively call the *costFunc* on the body of *S.callee*, if S does not correspond to a recursive call. If S is a `block` node then the cost-expressions of all the constituent nodes are added to obtain the cost-expression for S. Finally, if S is a `leaf` node then we set C_e to the cost of S as *Profile*(S).

The intuition behind our handling of serial-loops and if-else statements is that precisely modeling the loop-bounds and predicates can help obtain precise workloads, which in turn can help in better chunking. Note that in the presence of loops and conditionals, it is not possible to compute precise workload statically. Hence, we emit additional code to compute the workload at runtime.

Emitting workload computation code: After computing the cost-expressions, for each loop $\mathcal{L}$ (say, for a target with #runtime-threads=T), we emit the code shown in Figure 6 (inspector loop), immediately before $\mathcal{L}$. We assume that the variable $C_{\mathcal{L}}$ contains the cost-expression string as computed by *costFunc* and the operator $[\![X]\!]$ emits the value of the variable X. We also assume the availability of a global variable T containing the number of runtime-threads (specified in the X10_NTHREADS environment variable). This variable gives an upper limit to the number of chunks created in the output program.

```
1  long WLArray[];//array of size n=#iterations of ℒ.
2  long partialSum[];//An array of size T
3  long blkSz = (n+T -1 )/ T;
4  forall(it = 0 ... T){
5   for(i=T*blkSz...min(n, T*(blkSz+1))){
6    WLArray(i) = [[C_ℒ]]; // C_ℒ may be parmetric on i
7    pSum =+ WLArray(i); }
8   partialSum(it)= pSum; }
```

Figure 6: Inspector loop emitted by the Cost-Expression-Emitter, immediately before a parallel-for-loop $\mathcal{L}$.

The main purpose of the code shown in Figure 6 is to compute (at runtime) the workload of each iteration (and store in the array `WLArray`). In addition, we use Lines 2, 7, and 8 to populate another helper array `partialSum`, to maintain the sum of workloads of blocks of iterations; max iterations in a block = `blkSz`. In Section 5 we discuss how this array is used by the Work-Divider-Emitter to emit efficient code to divide the loop-iterations among the threads.

3.1.3 Work-Divider-Emitter. To partition the iterations of a given parallel-for-loop into different chunks, this phase emits a function

workDivider. We use *WLSum* to denote the total workload of the parallel-for-loop and *avgCost* to denote the average workload per chunk (= Σ_i WLArray(i)/T). We first list some of the goals behind the design of our heuristic: (1) *Correctness*: Each iteration must be part of exactly one chunk. (2) *Efficiency*: Each chunk should have contiguous iterations. This makes it easy to traverse the iterations of the chunk and may also helps preserve cache locality. (3) *Mimicking equal partition property I*: If a chunk contains $k(>1)$ iterations, then the sum of workloads of the first $k-1$ iterations must be $< avgCost$, and the k^{th} iteration should be such that the cost of the chunk should not be more than $2 \times avgCost$. (4) *Mimicking equal partition property II*: If the workload of an iteration is considered "large-and-overflowing" (defined below), then that iteration should be in its own chunk. This ensures that iterations with overly large workloads are not combined with other iterations – otherwise, it leads to an increase in the length of the critical path. (5) *Concurrency*: The algorithm should be designed in a way such that each thread may be able to run the algorithm in parallel and compute their respective chunks.

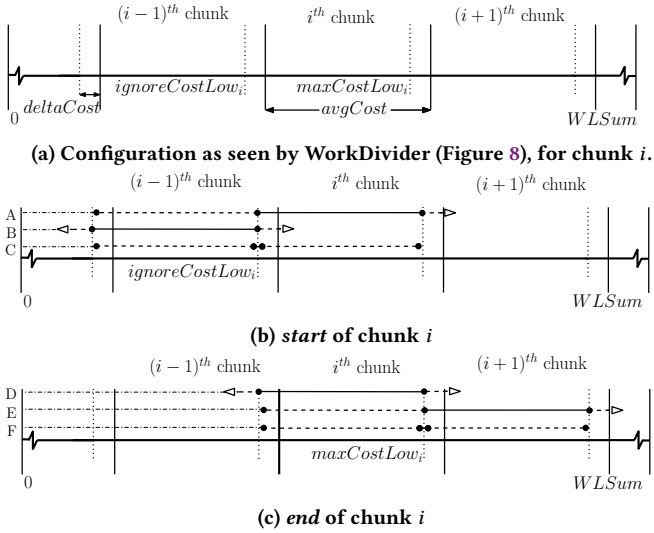
As discussed earlier in this section, in the case of balanced chunking, the workload of each chunk will be equal to *avgCost*. Considering that we may not achieve this ideal scenario (in general), we instead propose to make the chunks elastic. That is, instead of suggesting that the desired workload of a chunk is *avgCost*, we set the desired workload of a chunk to vary between $avgCost \times (1-\delta)$ to $avgCost \times (1+\delta)$, where δ is a constant ($0 \leq \delta < 1$).

The function *workDivider* (Figure 8) is invoked by each thread in parallel. To enable such parallelism, for each thread i, we enforce an additional property that the sum of the workloads of the threads $0 \cdots i-1$ would be at least $(avgCost \times i) - deltaCost$, where $deltaCost = \delta \times avgCost$. We store this lower limit in a variable called $ignoreCostLow_i$. Consider an iteration j of the input parallel-for-loop , such that $\Sigma_{k=0..j-1}$WLArray(k) $< ignoreCostLow_i$ and $\Sigma_{k=0..j}$WLArray(k) $\geq ignoreCostLow_i$. Here, we say that iteration j *crosses* $ignoreCostLow_i$. The variable *costTillNow* is set to $\Sigma_{k=0..j-1}$WLArray(k) (Line 9). Figure 7(a) depicts it graphically.

Since the iteration j crosses $ignoreCostLow_i$, it is either part of the previous thread ($i-1$), or the current thread (i). Or in other words for the chunk Q_i (of the thread i), $Q_i.start$ is either j or $j+1$. Similarly, if iteration j crosses $maxCostLow_i$ then $Q_i.end$ is j or $j-1$. We propose a heuristic to make a decision that is consistent across multiple parallel threads.

Consider the Figure 7(b). The figure depicts three possible scenarios based on the location of the end-points of j. In each scenario, we use ⋯⋯ to depict *costTillNow* (spanning $j-1$ iterations). The end-points of the j^{th} iteration are shown using either a dot (•) or an arrow (◃ or ▹). A •– – –• segment indicates that the end-point can be anywhere on that segment. A •– – –▹ or a ◃– – –• indicates that the end-point can be anywhere in that direction. We define an iteration j to be *large-and-overflowing*, if ∃ chunk k, such that iteration j crosses both $ignoreCostLow_{k-1}$ and $ignoreCostLow_k$. There are three scenarios:

Scenario A and B: j is a large-and-overflowing iteration: If iteration j crosses $ignoreCostLow_i$ and $ignoreCostLow_{i+1}$, but does not cross $ignoreCostLow_{i-1}$ then we set $Q_i.start = j$ (Scenario A). If iteration j also crosses $ignoreCostLow_{i-1}$ then some other thread



(a) Configuration as seen by WorkDivider (Figure 8), for chunk i.

(b) *start* of chunk i

(c) *end* of chunk i

Figure 7: Work division heuristics

(with id $< i$), would have made j a part of its chunk, and hence we set $Q_i.start = j+1$ (Scenario B).

Scenario C: j is not a large-and-overflowing iteration: In such a case, since j is crossing $ignoreCostLow_i$, we leave this iteration for the $i-1^{th}$ chunk and set $Q_i.start = j+1$.

Note that for the first chunk, we skip all calculations and just set $Q_i.start$=0. These heuristics to set $Q_i.start$ are coded in Lines 10-16 (Figure 8).

To set $Q_i.end$, similar to the how we set $Q_i.start$, we find an iteration j that crosses $maxCostLow_i$; we again have three scenarios:

Scenario D and E: j is a large-and-overflowing iteration: If iteration j crosses $ignoreCostLow_i$ and $maxCostLow_i$ then we set $Q_i.end = j$, indicating that the current chunk will not include any iteration beyond j. Note: the chunk Q_i includes the iteration j, only if $Q_i.start \leq j$. If iteration j does not cross $ignoreCostLow_i$, but crosses $maxCostLow_{i+1}$ then we set $Q_i.end = j-1$, indicating that the current chunk will not include the iteration j.

Scenario F: j is not a large-and-overflowing iteration: Since j is crossing the $maxCostLow_i$, we set $Q_i.end = j$.

Similar to the first chunk, in case of the last chunk, we skip all calculations and just set $Q_i.end$=$n-1$. All these heuristics to set the $Q_i.end$ are expressed in Lines 19-25 (Figure 8).

Note: if a large-and-overflowing iteration crosses a series of $ignoreCostLow_i$ ... $ignoreCostLow_{i+x}$ then all the chunks $i+1$ to $i+x-1$ will be empty.

Example. To illustrate the algorithm described in Figure 8, consider an example where the workloads of the iterations of a parallel-for-loop are given as follows: WLArray = {4, 4, 22, 1, 3, 2, 4}. Say T=4 and δ=0.25. We choose such an example to illustrate the behavior of deep-chunking in cases where loops are not balanced (due to non-uniform load). We now examine how the chunks are assigned.
Thread 0: Due to Line 16, iteration 0 will be part of Q_0. Since iteration 1 satisfies the conditions at Line 24, we set $Q_0.end = 1$.
Thread 1: Since iteration with index 1 satisfies the conditions at Line 14, we set $Q_1.start = 2$. Iteration 2 is a large and overflowing

```
1  Function Chunk workDivider(T, i, n, WLArray)
       Input: T: Number of threads, i: thread id, n: total no of iterations,
              WLArray: Contains estimated workload of each iteration
2      Chunk Q_i;// to be returned
3      double WLSum = Σ_{k=0}^{n-1}(WLArray(k));
4      double avgCost = WLSum/T; deltaCost = δ * avgCost;
5      double costTillNow = 0 ;
6      ignoreCostLow_i = (avgCost * i) − deltaCost;
7      maxCostLow_i = (avgCost * (i + 1)) − deltaCost;
8      Find the iteration j which crosses ignoreCostLow_i ;
9      costTillNow = sum of the cost of all iteration till j excluding j;
10     if i ≠ 0 then
11         if costTillNow< ignoreCostLow_{i−1} then          // Scenario A
12             Q_i.start = j + 1;
13         else
14             if costTillNow + WLArray(j) ≥ maxCostLow_i then
                   Q_i.start = j; ;                          // Scenario B
15             else Q_i.start = j + 1; ;                     // Scenario C
16     else Q_i.start = 0 ;
17     Find the iteration j which crosses maxCostLow_i ;
18     costTillNow = sum of the cost of all iteration till j excluding j;
19     if i ≠ T − 1 then
20         if costTillNow< ignoreCostLow_i then              // Scenario D
21             Q_i.end = j;
22         else
23             if costTillNow + WLArray(j) ≥ maxCostLow_{i+1} then
                   Q_i.end = j − 1 ;                         // Scenario E
24             else Q_i.end = j ;                            // Scenario F
25     else Q_i.end = n − 1 ;
26     return Q_i;
```

Figure 8: Algorithm for deep-chunking

```
1  forall(i = 0 to T) {
2    Chunk myChunk=workDivider(T,i,n,WLArray);
3    int myChunkStart=myChunk.start;
4    int myChunkEnd=myChunk.end;
5    for(j = myChunkStart to myChunkEnd){ S } }
```

Figure 9: Loop emitted by Parameterised-Loop-Chunk-Emitter

iteration and it satisfies the conditions at Line 23. Hence we set $Q_1.end = 1$. Thus, chunk Q_1 is an empty chunk.
Thread 2: Iteration 2 satisfies the conditions at Lines 14 and 21. Thus, we set $Q_2.start$=$Q_2.end$=2. Note: Q_2 has only one large and overflowing iteration, and the workload of chunk Q_2 is $22 > 2 \times avgCost$.
Thread 3: The iteration 2 satisfies the condition at Lines 12. Hence, we set $Q_3.start = 3$ Similarly, we set $Q_3.end = 6$ (Line 25). Thus the chunks are $Q_0 = [0, 1], Q_1 = [2, 1], Q_2 = [2, 2]$, and $Q_3 = [3, 6]$. And the workloads are 8, 0, 22, and 10, respectively.

3.1.4 Parameterised-Loop-Chunk-Emitter. The Parameterised-Loop-Chunk-Emitter transforms each normalized parallel-for-loop (of the form shown in Figure 2(a)) to a normalized-chunked parallel loop, as shown in Figure 9. It can be seen that the transformed loop creates only T tasks (as compared the n tasks created in the input program). Each task is executed by a different thread. The number and range of iterations of each chunk (executed part of a different task) are obtained from invoking the function *workDivider* at runtime.

3.2 Runtime Component

For each transformed parallel-loop, Figure 4(b) shows the flow diagram of the three different steps that get executed at runtime, as part of our transformation. (1) *Load-Computation.* This phase executes the inspector loop emitted by the Cost-Expression-Emitter (shown in Figure 6) and populates the elements of `WLArray`, in parallel. (2) *Work-Division.* After each thread has completed the Load-computation step, they invoke the *workDivider* function (in parallel). This function takes `WLArray` (computed in the previous step) as an argument. The *workDivider* function returns the chunk to be executed by each thread. (3) *Deep-chunking.* Each thread executes the iterations of its chunk in parallel.

3.3 Correctness and bound on the algorithm

We now present an argument about the correctness and efficiency of our proposed *workDivider* algorithm (Figure 8), which may be called by T number of threads concurrently.

Theorem 3.1. **Correctness**: *Every iteration of the input parallel-for-loop gets assigned to exactly one chunk.*

Lemma 3.2. *If the workload of a chunk Q_i is greater than or equal to $2 \times avgCost$, then the number of iterations assigned to chunk Q_i is exactly one.*

Theorem 3.3. **Efficiency**: *A chunk Q_i, either contains exactly one iteration, or its workload is less than $2 \times avgCost$.*

Corollary 3.4. *The concurrent invocation of workDivider function leads to a 2-approximate solution.*

Detailed proofs [30] omitted for space. □

4 ATOMIC BLOCKS AND DEEP CHUNKING

We now extend the deep-chunking techniques presented in Section 3 to handle `atomics`, which bring in new challenges. In the absence of any `atomic` blocks, our aim was to distribute the overall estimated workload (across all the iterations of the parallel-for-loop) as evenly as possible among the chunks. However, in the presence of `atomic` blocks, this goal becomes more challenging as 1) the overheads associated with the `atomic` blocks increases with the increase in the number of threads, and 2) the worst-case waiting-time at any `atomic` block includes the time taken by other parallel tasks to execute the `atomic` blocks. Thus, while the execution time of the non `atomic` blocks decreases as the number of threads increase (assuming $T \leq$#cores), the execution time of the code with `atomic` blocks increases with the number of threads [11].

The goal of our approach is to extract useful parallelism (from the programmer specified ideal parallelism) while ensuring that overheads due to the `atomic` blocks are minimal. Our approach is based on an observation that if the ratio of the workload of `atomic` blocks to the total workload inside the parallel-for-loop is "high" then it is beneficial to spawn a "small" number of tasks. To leverage this observation, we propose a scheme (inspired by prior work [11, 35, 41]) to model the workload of the parallel-for-loop,

in terms of the workloads of the atomic and non-atomic regions. Our overall scheme remains similar to the one described in Figure 4. We now explain extensions to the compile-time components below. Note: the steps followed during the runtime (Figure 4(a)) remain unchanged for handling atomics.

4.1 Compile-time Component in the presence of Atomic Blocks

We reuse the Profile-Guided-Cost-Estimator discussed in Section 3.1.1. We now present the extensions to the Cost-Expression-Emitter and Work-Divider-Emitter to handle atomic blocks.

Cost-Expression-Emitter in the presence of atomics We first invoke the *costFunc* function (Figure 5), by ignoring the atomic blocks – this returns a string denoting the cost-expression for the non-atomic regions. After that, we invoke the *costFunc-At* function (Figure 10), which returns a pair of strings denoting cost-expression for the atomic blocks (C_A) and an expression N_A that can compute the number of atomic blocks (at runtime).

The functionality of *costFunc-At* (Figure 10) is similar to that of *costFunc*. It mainly aims at generating an expression representing the workload associated with the statements inside atomic blocks (as if all the non-atomic input independent part is ignored). To achieve this *costFunc-At* takes an additional boolean argument, which indicates if the current node S is a descendent of an atomic node. We now discuss the main differences between *costFunc* and *costFunc-At*.

If S is an atomic node we recursively invoke *costFunc-At*, by passing a 'true' value for the third argument. Since LX10 (similar to X10) does not allow the nesting of atomic blocks, we set N_A=1.

If S is a loop node and if the *extractExpr* fails then we set C_A and N_A to "0". These are set to zero so that we do not overestimate the cost due to atomic blocks; as we discuss below, overestimating these costs may lead to underestimating the amount of useful parallelism. If *extractExpr* succeeds, then depending on whether the loop is inside an atomic block or not, we handle it differently. In the first case, the cost-expression C_A is computed, as discussed in Figure 5. Otherwise, the cost of atomics C_A in the loop is given by the product of the *S.loopBound* with cost C'_A of atomics of the loop-body; N_A is set similarly.

We handle if-else nodes similarly: the cost of the *S.condition* is added to C_A, if *inAtomic* is true. If *extractExpr* succeeds, then the computation of C_A and N_A are similar to the computations shown in Figure 5. Otherwise, C_A and N_A are set to zero (like in the case of the loop node). Similarly, call and block nodes are handled in a straightforward manner. If S is a leaf node, we set the value of C_A to the value of *Profile*(S), only if S is inside an atomic block. Or else, we set C_A to "0".

Similar to the code shown in Figure 6, we emit additional code to compute $\texttt{WLArray}_A$ and *totAtCost*. At run-time, for any given parallel-for-loop $\mathcal{L}$, the array $\texttt{WLArray}_A$ gives the total workload of atomic blocks in each iteration of $\mathcal{L}$. Similarly, *totAtCost* gives the total workload of atomic blocks across all the iterations of $\mathcal{L}$.

Work-Divider-Emitter We extend the *workDivider* function (Figure 8) to take two more parameters ($\texttt{WLArray}_A$ and *totAtCost*). Here, the function *workDivider* executes (after Line 3, Figure 8) the code shown in Figure 11 to determine the useful-parallelism and T.

```
1  Function String costFunc-At(L, S, inAtomic)
       Input: L: a parallel-for-loop node, S: a descendent of L in the PSGb;
              inAtomic: boolean
2      N_A = "0";
3      switch type of S do
4          case atomic node do
5              (C_A, N'_A) = costFunc-At(L, S.body, true); N_A = "1";
6          case loop node do
7              LBound = extractExpr(L, S.loopBound) ;
8              (C'_A, N'_A) = costFunc-At(L, S.body, inAtomic);
9              if LBound≠null then // extraction successful
10                 if inAtomic == true then
11                     C_A = Profile(S.prologue) || "+" || LBound||
                           "*(" ||C'_A|| "+" || Profile(S.epilogue) || ")";
12                 else
13                     C_A =LBound || "*" ||C'_A;
14                     N_A =LBound || "*" ||N'_A;
15             else {C_A="0"; N_A="0"};
16         case if-else node do
17             cond = extractExpr(L, S.condition);
18             if inAtomic == true then C_A = Profile(S.condition) ;
19             (C^t_A, N^t_A) = costFunc-At(L, S.then, inAtomic);
20             (C^e_A, N^e_A) = costFunc-At(L, S.else, inAtomic);
21             if cond ≠ null then // extraction successful
22                 C_A = C_A|| "+" || cond || "?" ||C^t_A || ":" ||C^e_A;
23                 N_A = cond || "?" ||N^t_A || ":" ||N^e_A ;
24             else {C_A="0"; N_A="0"};
25         case call node do
26             (C_A, N_A) = costFunc-At(L, S.callee, inAtomic);
27         case block node do
28             for s_i in S.nodes() do
29                 (C'_A, N'_A) = costFunc-At(L, s_i, inAtomic);
30                 C_A = C_A|| "+" ||C'_A; N_A = N_A|| "+" ||N'_A;
31         otherwise do // leaf node
32             if inAtomic == true then C_A = Profile(S);
33             else C_A = "0";
34     return (C_A, N_A);
```

Figure 10: Cost-expression generation in the presence of atomic blocks

The presented extension computes the estimated workload of the critical thread for varying number of runtime threads (from 1 to T) and sets T to that number of threads, which leads to minimum cost. The workload of the critical thread is computed to include (1) the cost due to non-atomic region (shared equally among all the threads – *parCost*), (2) worst-case cost arising out of atomic blocks (*totAtCost*), and (3) overhead incurred due to interactions between the threads, while executing the atomic blocks ($UnitOverhd \times maxInteractions$). Note I: In LX10, atomic blocks enforce mutual-exclusion, and hence in the worst-case the critical thread will execute the atomic block, only after all the other atomic blocks have been executed. And hence we set *totAtCost* as the sum of all the elements of $\texttt{WLArray}_A$. Note II: in the worst-case, all the threads reach the atomic blocks at the same time – leading to $maxInteractions = totAtCost \times (i-1)$ interactions.

```
1 long minCost = Long.MAX_VALUE;
2 long UnitOvrhd = K_d; // Overhead of one atomic op.
3 long totAtCost = Σ_{k=0}^{n-1} WLArray_A(k); // total atomic cost
4 for i : 1 to T do // compute the cost assuming #threads=i
5     long parCost = WLSum/i ;
6     long maxInteractions = totAtCost * (i − 1) ;
7     long Cost=parCost+totAtCost+UnitOvrhd × maxInteractions;
8     if minCost > Cost then {minCost=Cost; nproc=i;};
9 T = nproc;
```

Figure 11: Determine useful parallelism in the presence of atomic blocks

The algorithm chooses T to be that value of i where the estimated workload is the minimum. Note that the *parCost* decreases as the number of threads increase, and the *maxInteractions* increase as the number of threads increase. In the limiting case, where the number of `atomic` blocks is 0, our algorithm will leave the variable T untouched. Similarly, if the body of the parallel-for-loop contains `atomic` block only, then we will obtain $T = 1$. The computation of the variable K_d, which indicates the overhead associated with one atomic interaction, is discussed in the next section.

5 DISCUSSION

The extractExpr function. The function *extractExpr*(L, e), described in Section 3.1.2, tries to map the input expression e to another expression e_1, in terms of variables that are live before the parallel-for-loop L. To achieve this, we first identify the loop (parallel or serial) surrounding e; say it is L_1. If L_1 is not a for-loop, then we mark e as non-extractable and return. We compute a chop [18] from the predecessor of L_1 to e. If the chop includes a loop, then we mark the expression non-extractable. Else, we follow the use-def chains to compute an expression e_2, which is the 'avatar' of e, in terms of variables visible before L_1. If $L_1 = L$ then, we return e_2. Otherwise, it indicates that e is nested insides a loop L_1. The algorithms given in Figures 5 and 10 are written assuming that e is not nested inside any loop. However, our implementation does handle the case with loops (details skipped for space).

Impact of the value of K in Figure 5. If *extractExpr* fails to extract a valid loop-bound for a loop, we set the cost-expression to be a constant K. The value of K is conservatively chosen such that it is very large and it subsumes any other costs that may be part of the parallel for loop. Hence the parallel-for-loop will be treated as a balanced loop, and the *workDivider* will distribute the iterations to chunks in such a way that it mimics block-chunking (far better than the default work-stealing based option). Similarly, our worst-case assumption in if-then-else statements may lead to inaccuracies; this, in the worst-case, may again mimic block-chunking.

In such conservative scenarios, even though we mimic block-chunking, the performance may not match exactly that of block-chunking, as (i) our scheme allows some minor variation due to the delta-cost variation for each chunk, and so on. Hence, there is a chance that a very few iterations may end up in a different chunk than as decided by block-chunking. (ii) our scheme incurs some minor overheads to perform deep-chunking at runtime.

Value of the constant δ, described in Section 3.1.3. The constant δ is one of the factors that affect the distribution of the iterations among chunks. It helps in expanding the desired capacity of a chunk from *avgCost* to a maximum of *avgCost* + *deltaCost*. The optimal value of *deltaCost* may depend on the value of T, the specific input (value of n) to the benchmark, and many input application dependent factors such as the granularity of the workloads of the iterations, and the distribution of the workload among all the parallel iterations. We have experimented with different values of δ and found that δ=0.01 gave the best results.

Value of the constant K_d. We calculated (using profiling) the value of the constant K_d, which denotes the overheads associated with a single atomic operation to be 7 μsec.

Impact on code size. Deep-chunking has a negative impact (albeit small) on the code size. In terms of the number of lines in a high-level language like Java, deep-chunking adds around ten lines per loop. Such an overhead has to be kept in mind when compiling with hard code size restrictions.

Optimizations: We now discuss four additional optimizations that we perform, on top of the proposed chunking technique.

- If the compiler can guarantee that the computed values of the `WLArray` does not change across multiple instances of a parallel-for-loop, then the code to compute the `WLArray` and create chunks can be executed before the first such instance and the computed chunks can be re-used. We identify such a parallel-for-loop by analyzing the cost-expression.
- We first perform a preprocessing step to identify if a parallel-for-loop $\mathcal{L}$ has inner serial for-loops with parallel-loop-variant bounds; that is, if $\mathcal{L}$ has potential workload imbalance. If so, we invoke our proposed optimization on $\mathcal{L}$. Otherwise, we chunk them using a default policy (block-chunking).
- When the function *workDivider* has to find an iteration j that crosses $ignoreCostLow_i$, iterating through each element of `WLArray` can be quite expensive. Instead, we divide the work of summing the elements of `WLArray` among the threads. Each thread i populates `partialSum`(i) with the sum of its *block* of elements from `WLArray`; this is done along with the calculation of the elements of `WLArray` (Figure 6). This array `partialSum` is used by *workDivider* to traverse through the iterations efficiently. To find an iteration j such that j crosses $ignoreCostLow_i$, we traverse the `partialSum` elements to find a "block" b such that the sum of the costs of the blocks till b (but excluding b) is less than $ignoreCostLow_i$ and the sum of the cost of the blocks including b is greater than $ignoreCostLow_i$. Then to find the exact iteration j, we traverse through the iterations of b, which significantly reduces the number of comparisons needed to find the iteration j.

6 IMPLEMENTATION AND EVALUATION

We have implemented our proposed fully automated approach in the X10 compiler; the complete source code can obtained from GitHub [31]. The profiling pass and the actual optimization pass are invoked using two different compiler switches designed by us (-profileForChunking, and -chunkUnbalanced). We evaluated our proposed techniques on a set of eight shared-memory benchmarks taken from IMSuite [12] – these are the benchmarks with non-uniform workloads in at least one of their parallel-for-loops.

Bench	LOC	SL	SA	I/P	DL	DA
1. BFS-BellmanFord (BF)	386	2	1	32K	8	1M
2. BFS-Dijkstra (DST)	556	7	3	1K	1K	18K
3. Byzantine (BY)	570	3	1	512	325	18M
4. DominatingSet (DS)	721	9	2	1K	171	6K
5. KCommitte (KC)	844	11	2	2K	246	9M
6. LeaderElect-DP (DP)	676	4	2	8K	56	3M
7. MIS	609	5	6	32K	21	12M
8. VertexCol (VC)	641	5	0	16K	17	0

Figure 12: Characteristics of the benchmarks. Abbreviations: LOC: Lines of code, SL: #static parallel-for-loops, SA: #static `atomic` blocks, I/P: input size, DL: #dynamic parallel-for-loops, DA: #dynamic `atomic` blocks.

Figure 12 lists some static characteristics of the chosen set of benchmarks. Except VC, all other benchmarks have `atomic` blocks within one or more parallel-for-loops. Columns 2, 3, 4 list the number of lines of code, parallel-for-loops (static) and `atomic` blocks (static). For each of the benchmarks we fixed the largest input size such that the total execution can finish within 10 minutes (Column 5). The number of dynamic parallel-for-loops and `atomic` blocks, for the chosen input, are listed in columns 6 and 7. Except for BF, DST, and DS, in all the other cases, the *extractExpr* routine (Section 3) succeeded in mapping the loop-bounds and if-predicates appropriately. For profiling it was sufficient to use a very small input (size 16); the total compilation overhead was negligible.

All the benchmarks were executed on an AMD Opteron 6320 processor, with 4 sockets each having 16 cores (total 64 cores), and total memory of 512GB. For each benchmark, besides the un-chunked (using the default work-stealing scheme) version, we evaluated five different chunked versions (see Section 2): i) Cyclic, ii) Block, iii) Dynamic, iv) Guided, and v) Deep. The implementations of block- and cyclic-chunking versions are derived from that of Nandivada et al. [26] and that of the dynamic- and guided-chunking policies are derived from that of OpenMP [27]. In order to execute a code with T number of worker-threads, we also fix the number of available cores to T. For the five chunking policies the underlying work-stealing has no effect, as the number of parallel-tasks (= #chunks) matches T. We plotted (in Figure 13) the execution times of the above mentioned versions (each normalized to the time taken by the un-chunked work-stealing based code), for varying number of worker-threads (2-64). The deep-chunking numbers refer to the total execution time including the time of inspection and deep-chunking runtime framework.

As can be seen, with increasing number of threads, the overall gains due to deep-chunking are increasing: the geometric mean of the normalized execution time reduces from 0.70 (at 2 threads) to 0.28 (at 64 threads). Overall, on an average, the proposed deep-chunking technique achieves 50.48%, 21.49%, 26.72%, 32.41%, and 28.84% better performance than un-chunked, cyclic-, block-, dynamic-, and guided-chunking techniques, respectively. The observed gains are due to better load-balancing, handling of `atomic` blocks, and low overheads during runtime.

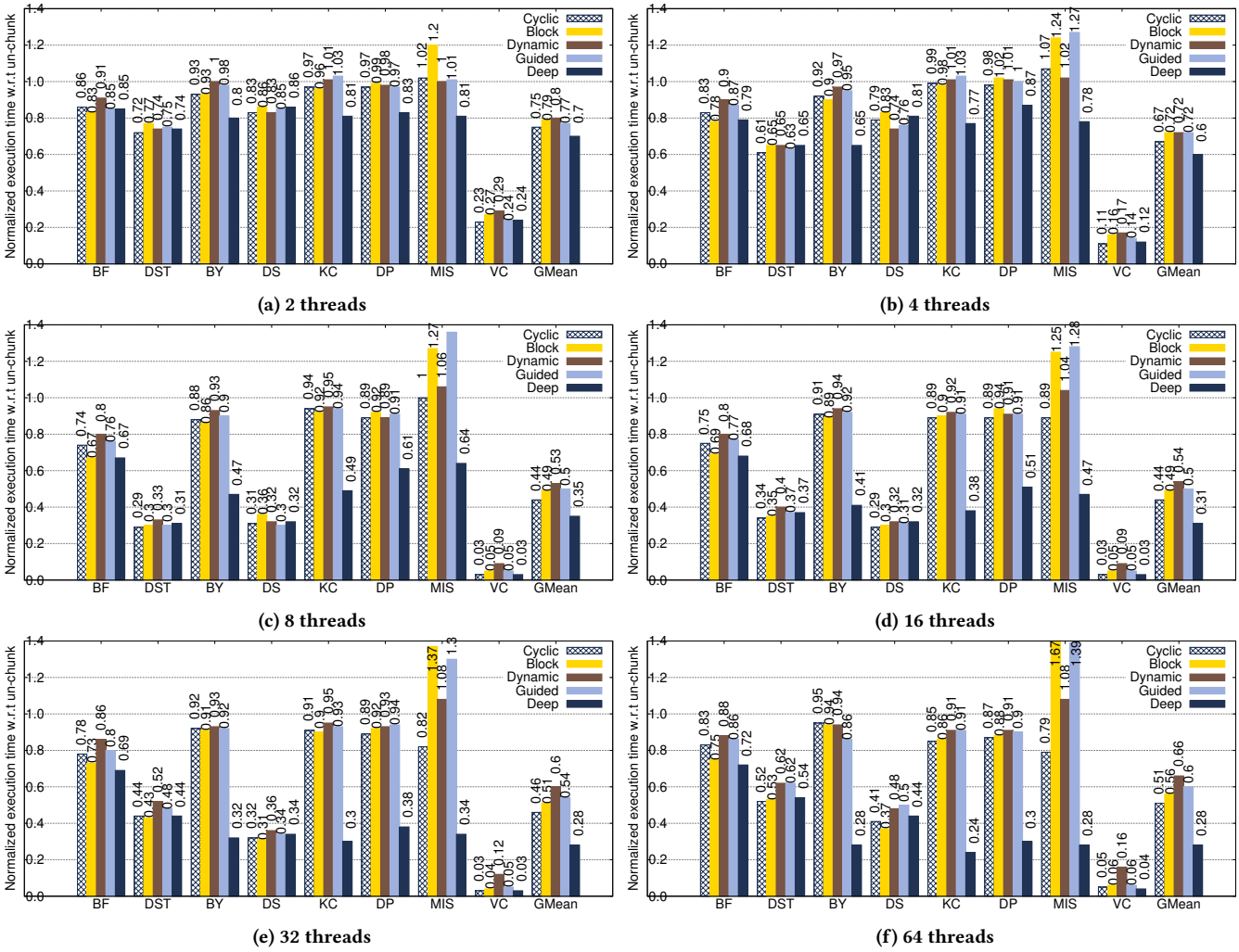
For smaller number of threads, it can be seen that our approach does not gain many benefits compared to versions chunked using block-, cyclic-, dynamic- and guided-chunking, as there is much less scope to divide the work. However, with the increasing number of threads, the gains mostly improve. As it can be seen in Figure 13, except DST and DS all other benchmarks show significant gains for the higher number of threads (32 or 64).

We analyzed the DST and DS benchmarks and found that the reason why deep-chunking does not fare well compared to block- and cyclic-chunking is that both of these benchmarks have conditionals that cannot be extracted by the *extractExpr* function. And these `if-else` statements have nested `for` loops with `atomic` blocks inside them. And our conservative approach did not pay off well, in these benchmarks. Here, even though the chunking of loops in the deep-chunked code mimics block-chunking (see Section 5), the minor performance loss the former incurs (compared to blocked-chunking) is due to the additional overheads incurred as part of the deep-chunking runtime infrastructure. However, in contrast, BF has a similar structured code, where our conservative approach is still able to perform better than other techniques. We conclude that the gains depend on the accuracy of workload calculation and the number of times the parallel-for-loops are executed; accurate prediction + large number of such dynamically executed parallel-for-loops lead to large gains. But inaccurate prediction + large number of such dynamically executed parallel-for-loops may impact the performance negatively.

We observed that for VC the performance gain compared to un-chunked version of the code is very high: normalized execution time with respect to un-chunked version is as low as 0.03. This is because the work to be executed by each iteration of the parallel for loop is very less compared to the number of iterations (16K). Hence the un-chunked version (using work-stealing scheduler in case of X10) creates 16K tasks and suffers from poor performance due to task creation and migration overheads. At the same time, for MIS, the un-chunked version performs better than Block or Guided strategies (Block and Guided with values as high as 1.67 and 1.40). We analyzed the benchmark and found that the workload of the iterations of the parallel-for-loop is very high for the initial iterations and very low for the later iterations. Because of this reason, the guided- and block-chunked codes perform worse than un-chunked. In contrast, the code generated by deep-chunking (workload aware) is able to achieve performance gains.

It can be seen that for BY, KC, DP, and MIS, our approach significantly outperforms the Block, Cyclic, Dynamic and Guided approaches. This is because of our handling of `atomic` blocks, where we are able to extract useful parallelism more accurately. To understand the impact of our proposed approach to handle `atomic` blocks (Section 4), we compared the execution times of these benchmarks with and without the extensions of Section 4. Figure 14 shows a comparative plot of these execution times, for varying number of threads. As can be seen, our proposed extensions are quite effective and lead to significant improvements to our base technique (up to 76.6%, geomean 42.8%).

Overheads due to the inserted code. We also computed the overheads incurred due to the execution of the additional code inserted by deep-chunking and found it to be quite low. For each benchmark, we computed the overhead as a percentage over the total execution time of that benchmark (=100×total-overhead/total-exectime). Due to lack of space, we avoided presenting the overheads of each benchmark for each configuration (2/4/8/..64 threads), especially



(a) 2 threads

(b) 4 threads

(c) 8 threads

(d) 16 threads

(e) 32 threads

(f) 64 threads

Figure 13: Chunking with varying number of threads

because the overheads were very low (most of them < 1%). Across all the benchmarks, for 2-64 threads, the geometric mean overhead was found to be 0.92%. Around 90% of this overhead was due to the inspector loop (which may increase the critical path) and the rest was from the invocation of the *workDivider* function.

Impact on codes with uniform workloads. Though the focus of deep-chunking is codes with non-uniform workloads, for codes with uniform workloads, where we can statically identify the best chunking scheme, deep-chunking may still be useful wherein we do not have to manually try the different popular chunking schemes and get more or less the best performance. As a minor test, we ran the X10 version of the LUFact code (from New Java Grande Forum suite [19]) that has triangular workload (the workload of the iterations of the parallel-for-loop decreases as the iteration index increases). We observed that deep chunking achieves 18% and 22% performance gains compared to Block and un-chunked (work-stealing), whereas the performance is comparable to Guided, and has negligible (5-6%) degradation when compared to Cyclic and Dynamic, respectively.

Overall summary. Figure 13 shows that the performance gains resulting from different chunking policies vary. The relative performance between un-chunked (using the default X10 work-stealing scheduler), blocked-chunking, cyclic-chunking, guided-chunking, and dynamic-chunking varies significantly depending on the specific benchmark, and the runtime configuration (number of cores). This makes it hard for a compiler to fix upon one of these policies. In contrast, our proposed technique is able to realize gains (compared to all the five techniques) in most of the cases, and even in benchmarks where our approach does not lead to gains, the performance is more or less comparable to the best of these approaches.

7 CONCLUSION

In this paper we presented an optimization (deep-chunking), to efficiently perform loop chunking on programs with non-uniform workloads. We presented a combined compile-time and runtime

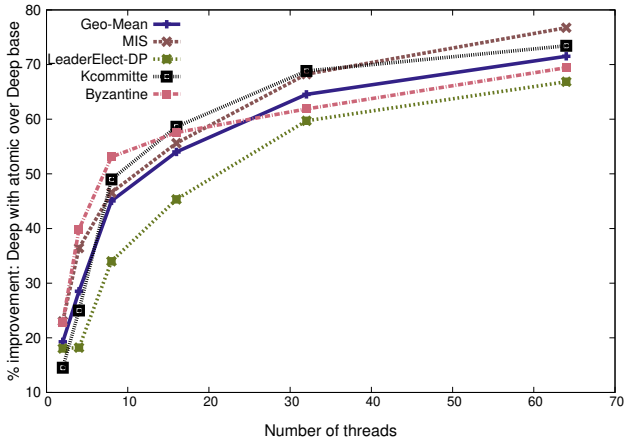


Figure 14: Performance improvements in deep-chunking due to the techniques discussed in Section 4.

approach, which studies the program structure at compile time and estimates the workload of iterations at runtime. We also presented an algorithm which can be invoked by the worker threads to obtain an optimal distribution of iterations of parallel loops with minimal synchronization overheads. We proved that the algorithm is a 2-factor approximation algorithm. Further, we showed that by modelling the behavior of `atomic` blocks, we can reduce the overheads associated with `atomic` blocks inside parallel loops. We evaluated our implementation on a set of benchmarks from IMSuite and found that on an average, deep-chunking achieves 50.48%, 21.49%, 26.72%, 32.41%, and 28.84% better performance than un-chunked, cyclic-, block-, dynamic-, and guided-chunking policies, respectively.

Acknowledgments. This work is partially supported by SERB CRG grant (sanction number CRG/2018/002488) and NSM research grant (sanction number MeitY/R&D/HPC/2(1)/2014).

REFERENCES

[1] A. Agarwal, D. A. Kranz, and V. Natarajan. Automatic Partitioning of Parallel Loops and Data Arrays for Distributed Shared-Memory Multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 6(9):943–962, September 1995.
[2] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, March 1996.
[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.
[4] J. Mark Bull. Feedback Guided Dynamic Loop Scheduling: Algorithms and Experiments. In *EUROPAR*, pages 377–382, 1998.
[5] Chapel. The Chapel language specification version 0.4. http://chapel.cray.com/, 2005.
[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *OOPSLA*, pages 519–538, New York, NY, USA, 2005. ACM.
[7] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn. MPIPP: An Automatic Profile-guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters. In *ICS*, pages 353–360. ACM, 2006.
[8] Q. Chen, M. Guo, and Z. Huang. CATS: Cache Aware Task-stealing Based on Online Profiling in Multi-socket Multi-core Architectures. In *ICS*, pages 163–172, New York, NY, USA, 2012. ACM.
[9] A. Duran, J. Corbalán, and E. Ayguadé. An Adaptive Cut-off for Task Parallelism. In *SC*, pages 36:1–36:11. IEEE Press, 2008.
[10] M. Durand, F. Broquedis, T. Gautier, and B. Raffin. An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines. In *IWOMP*, pages 141–155. Springer Berlin, 2013.
[11] S. Eyerman and L. Eeckhout. Modeling Critical Sections in Amdahl's Law and Its Implications for Multicore Design. In *ISCA*, pages 362–370, 2010.
[12] S. Gupta and V. K. Nandivada. IMSuite: A benchmark suite for simulating distributed algorithms. *Journal of Parallel and Distributed Computing*, 75(0):1 – 19, January 2015.
[13] S. Gupta, R. Shrivastava, and V. K. Nandivada. Optimizing Recursive Task Parallel Programs. In *ICS*, pages 11:1–11:11, 2017.
[14] Habanero. Habanero Java. http://habanero.rice.edu/hj, Dec 2009.
[15] B. Hamidzadeh, L. Y. Kit, and D. J. Lilja. Dynamic task scheduling using online optimization. *IEEE Trans. Parallel Distrib. Syst.*, 11(11):1151–1163, 2000.
[16] B. Hamidzadeh and D. J. Lilja. Self-Adjusting Scheduling: An On-Line Optimization Technique for Locality Management and Load Balancing. In *ICPP*, pages 39–46. IEEE Computer Society, 1994.
[17] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: A Practical and Robust Method for Scheduling Parallel Loops. In *Supercomputing*, pages 610–632, New York, NY, USA, 1991. ACM.
[18] D. Jackson and E. J. Rollins. Chopping: A Generalization of Slicing. Technical Report CMU-CS-94-169, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.
[19] JGF. The Java Grande Forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande/javag.html.
[20] A. Kejariwal, A. Nicolau, and C. D. Polychronopoulos. History-aware Self-Scheduling. In *ICPP*, pages 185–192, Aug 2006.
[21] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.
[22] C. Kruskal and A. Weiss. Allocating Independent Subtasks on Parallel Processors. *SE*, SE-11(10), October 1985.
[23] S. Lucco. A Dynamic Scheduling Method for Irregular Parallel Programs. *SIGPLAN Not.*, 27(7):200–211, July 1992.
[24] E. P. Markatos and T. J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-memory Multiprocessors. In *SC*, pages 104–113. IEEE Computer Society Press, 1992.
[25] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
[26] V. K. Nandivada, J. Shirako, J. Zhao, and V. Sarkar. A Transformation Framework for Optimizing Task-Parallel Programs. *ACM Trans. Program. Lang. Syst.*, 35(1):3:1–3:48, April 2013.
[27] OpenMP Application Program Interface Version 4.0. http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.
[28] P. H. Penna, A. T. A. Gomes, M. Castro, P. D.M. Plentz, H. C. Freitas, F. Broquedis, and J-F MÃĲhaut. A comprehensive performance evaluation of the binlpt workload-aware loop scheduler. *Concurrency and Computation: Practice and Experience*, 31(18):e5170, 2019. e5170 cpe.5170.
[29] C. D. Polychronopoulos and D. J. Kuck. Guided Self-scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Trans. Comput.*, 36(12):1425–1439, December 1987.
[30] Indu K. Prabhu and V. Krishna Nandivada. An extended report on chunking loops with non-uniform workloads. https://www.cse.iitm.ac.in/~krishna/preprints/ics20/ics20-full.pdf.
[31] Indu K. Prabhu and V. Krishna Nandivada. Deep Chunking Implementation. https://github.com/indukprabhu/deepChunking.
[32] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.
[33] V. Saraswat, B. Bard, P. Igor, O. Tardieu, and D. Grove. X10 Language Specification Version 2.4. Technical report, IBM, 2014.
[34] J. Shirako, J. M. Zhao, V. K. Nandivada, and V. N. Sarkar. Chunking Parallel Loops in the Presence of Synchronization. In *ICS*, pages 181–192. ACM, 2009.
[35] R. Shrivastava and V. K. Nandivada. Energy-efficient compilation of irregular task-parallel loops. *TACO*, 14(4):35:1–35:29, 2017.
[36] M. S. Squillante and E. D. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 4(2):131–143, February 1993.
[37] S. Subramaniam and D. L. Eager. Affinity Scheduling of Unbalanced Workloads. In *SC*, pages 214–226. IEEE Press, 1994.
[38] P. Thoman, H. Jordan, S. Pellegrini, and T. Fahringer. Automatic OpenMP Loop Scheduling: A Combined Compiler and Runtime Approach. In *IWOMP*, pages 88–101. Springer-Verlag, 2012.
[39] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin. Lazy Binary-Splitting: A Run-time Adaptive Work-stealing Scheduler. In *PPPoP*, pages 179–190, 2010.
[40] T. H. Tzen and L. M. Ni. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. *IEEE Trans. Parallel Distrib. Syst.*, 4(1):87–98, January 1993.
[41] Akshay Utture and V. Krishna Nandivada. Efficient lock-step synchronization in task-parallel languages. *Softw., Pract. Exper.*, 49(9):1379–1401, 2019.