Access Annotation for Safe Program Parallelization

Chen Ding, Lei Liu

Access Annotation for Safe Program Parallelization

Chen Ding[†] and Lei Liu[‡]

[†]Department of Computer Science, University of Rochester, Rochester, USA
[‡]Institute of Computing Technologies, Chinese Academy of Sciences, Beijing, P.R. China

Abstract. The safety of speculative parallelization depends on monitoring all program access to shared data. The problem is especially difficult in software-based solutions. Till now, automatic techniques use either program instrumentation, which can be costly, or virtual memory protection, which incurs false sharing. In addition, not all access requires monitoring. It is worth considering a manual approach in which programmers insert access annotations to reduce the cost and increase the precision of program monitoring.

This paper presents an interface for access annotation and two techniques to check the correctness of user annotation, i.e. whether all parallel executions are properly monitored and guaranteed to produce the sequential result. It gives a quadratic-time algorithm to check the exponential number of parallel interleavings. The paper then uses the annotation interface to parallelize several programs with uncertain parallelism. It demonstrates the efficiency of program monitoring by a performance comparison with OpenMP, which does not monitor data access or guarantee safety.

1 Introduction

With the advent of multicore processors, existing applications written in Fortran/C/C++ are increasingly adapted to parallel execution using programming interfaces such as OpenMP, Cilkplus and TBB. They are efficient but do not guarantee correctness. The correctness problem is more serious when parallelizing large program code, mainly due to several issues of uncertainty:

- *Complex code.* A task may execute low-level or dynamic code not amenable to static analysis. Example problems include exceptions in control flow, indirections in data access, dynamic memory allocation and custom memory management.
- *Partial information.* A programmer may read and understand only a part but not the whole program. The program may use separately compiled libraries.
- *Uncertain parallelism.* Parallelism may exist in most but not all iterations of a loop or in some but not all inputs of a program. Important irregular computing tasks such as mesh refinement, clustering, image segmentation, and SAT approximation cannot be fully parallelized without speculation [16].

Speculative parallelization is a technique to guard parallelism against uncertainty. To make it programmable, a number of systems provide primitives to mark a speculative task as a safe future [26], an ordered transaction [25], or a possibly parallel region (*PPR*) [8]. In this paper, we call a speculative task a *PPR* task and use the support system called *BOP* for speculation on commodity multicore machines [8, 14].

Safe parallelization depends on monitoring data access, which in software is typically done in two ways. The first is program instrumentation. The cost is high if too many data accesses are instrumented. A compiler may remove unnecessary instrumentation but only for applications amenable to static analysis [21, 24]. The other solution is virtual-memory support to monitor data at page granularity, which incurs false sharing [5, 8]. While these solutions are automatic and transparent to a user, they are not most efficient or precise especially when monitoring complex code.

The monitoring problem may be better solved with programmer control. In this paper, we define an interface for access annotation and integrate it into the *BOP* system. The new interface provides two primitives for marking data writes and data reads. The user annotation may be too much or too little, causing four possible problems:

- *Insufficient annotation*. Some reads and writes should be annotated but not. Incompleteness can lead to three types of errors:
  - *Incorrect output*. Speculation does not generate sequential output.
  - *Nondeterministic output*. Speculation generates different outputs depending on *PPR* task interleaving.
  - *Partial correctness*. The annotation is sufficient for some but not all inputs.
- *Redundant annotation*. The same access may be repeatedly annotated.

To check completeness, we describe two techniques, to be applied in order:

1. *Semantic checking*, which runs the *PPR* tasks in the sequential order and checks whether the output is the same as the sequential execution. The sequential *PPR* execution is the *canonical execution*, which is different from the sequential execution as we will explain later.
2. *Determinism checking*, which uses a quadratic number of tests to check whether all parallel interleavings produce the same result as the canonical execution, which means the sequential output if applied after the first step.

Semantic checking finds missing write annotations, while determinism checking looks for missing read annotations. The time cost is linear to the number of *PPR* tasks in the former and quadratic in the latter. If both checks are passed, the annotation is sufficient for the given test input. For this input, all parallel executions are guaranteed to produce the sequential result.

For the problem of redundant annotation, we view the new interface as a solution rather than a new cause. In the program examples shown throughout the paper, we will see how the interface enables a programmer to insert minimal annotation. Still, a user may over optimize. Then the problem becomes a completeness issue, which is the same as a user being too lazy or lacking knowledge. It will require annotation checking, which is the subject of this paper.

Annotation checking is a form of debugging. It is not guaranteed to find all errors. This is a familiar limitation to the programmer. If a program runs on an untested input and produces an error, the two-step checking can be re-applied to add overlooked annotations. With the checking support, parallel coding is similar to sequential coding. Systematic testing can be used to gain a high degree of certainty that the parallelized program has the right semantics and is safe.

The new model maintains and extends the benefits of speculative parallelization. Because of the sequential semantics, parallelized code is easier to write and understand. It can be composed with other sequential code or automatically parallelized code. There is no need for parallel debugging—the parallelized code produces the correct output if the sequential code does. A program may be fully annotated, so it no longer needs speculation. Finally, an *BOP* program can run on a cluster of machines without shared memory [12].

Next, we describe in detail the need of access annotation and the two checking techniques, before we evaluate and discuss related work.

2 Access Annotation

To properly insert access annotation, a programmer needs to understand which tasks are parallel and how they share data. Since not all tasks are parallel, not all data access requires annotation.

2.1 The Execution Model

The parallelism hint is as follows:

- `bop_ppr{X}Y.` The `bop_ppr` block suggests possible parallelism—X is likely parallel with the subsequent code Y. Y is also known as the *continuation* of X.

At run time, the *PPR* hints divide a sequential execution into a series of *PPR* tasks, numbered in an increasing order starting from 0. Any two tasks have a sequential order between them, so we can refer to them as the earlier task and the later task. We assume no nesting.[1]

A *PPR* task is dynamic. It has two parts. The *link* is the initial execution after the previous *PPR* and before the next *PPR* hint. It executes the code between two *PPR* hints. After the link, the *body* is the execution of the code inside the *PPR* hint. We call these two parts the *link* PPR and the *body* PPR. The parallelism happens between a body and all later links, and between all bodies. We call all links collectively as the *backbone*. The backbone is sequential, and the bodies are limbs hanging on the backbone. Next we show how *PPR* tasks share data and how *BOP* ensures correctness.

2.2 Data Copy and Merge

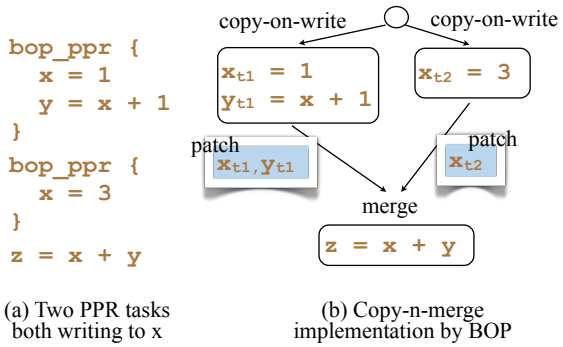
Logically, a *BOP* program is sequential, and all data is shared. In implementation, *BOP* uses a two-step strategy to share data correctly:

1. *Data copy-on-write during parallel execution*. *BOP* runs each task as a Unix process. For each page that the task modifies, the OS makes a private copy and hence removes all interference between *PPR* tasks.

[1] In *BOP*, a `bop_ppr` hint is ignored if encountered inside a *PPR* task. Nested hints can be supported by assigning a linear ordering of nested tasks [17] and checking them in a way similar to checking non-nested tasks.

2. *Sequential merge after parallel execution*. After a *PPR* task finishes, the changes are collected as a *PPR patch*. *BOP* merges the patches from multiple *PPR* tasks to ensure that the results be the same as the would-be sequential execution.

Figure 1 shows two *PPR* tasks. Although they both modify x, they do not conflict because each will do copy-on-write and then write to its private copy. After they finish, they will bundle the changes into two patches. At the merge, the x value from the later *PPR* task is kept to maintain sequential semantics.



(a) Two PPR tasks both writing to x

(b) Copy-n-merge implementation by BOP

Fig. 1: Illustration of copy and merge: two *PPR* tasks shown in (a), data copy-on-write during and merge after the parallel execution shown in (b). After a *PPR* task, its changes are bundled into a *patch* and used by the merge step.

The two steps can be called copy-n-merge. The reason for using the strategy is compelling: copy-n-merge removes all false dependences in a parallel execution and all data races in a concurrent execution. A necessary condition for a dependence or a race is that two operations access the same memory location, and at least one is a write. By copy-on-write, no two tasks can access the same location unless it is read only. Removing this necessary condition removes any false dependence or data race, just as removing oxygen extinguishes the fire.

2.3 Access Annotation

There are two types of annotations:

- ppr_write(*addr*, *size*) says that the *PPR* task produces a new value for data at memory *addr* for *size* bytes.
- ppr_read(*addr*, *size*) says that the *PPR* task needs the latest value produced before the *PPR* task for data at *addr* for *size* bytes.

Semantically, an annotation is a requirement on the information flow. The read annotation means that the reading *PPR* task must have the up-to-date value, the value from the last write before the *PPR* task (in the sequential execution). The write annotation means that the new value should be delivered to the remaining program execution for use by later reads of the data. These two requirements are intuitive, although not entirely precise.

Operationally, the annotations are used by *BOP* to create the *PPR* patch. After a task finishes, its patch includes the set of memory locations read and modified by the task and for each modified location, the last written value, i.e. the value it has at the end of the *PPR* execution. These patches are used at the merge time for conflict checking and data commit.

An annotation may be placed anywhere and can be executed more than once during the task execution. Because *PPR* tasks do not physically share data during the execution, the annotations specify only the values coming in and going out. As a result, the same data needs at most two annotations in a *PPR* task, one for the read, and one for the write, regardless how many times the task reads or writes the data. For the writes, only the last value will be copied into the *PPR* patch. For the reads, the task must read the correct value at the *first* time.

The read annotation is needed for the reads in both the link and the body of a *PPR* task, while the write annotation is needed only for the writes in the *PPR* body. To understand this slight asymmetry, we use Table 1 to enumerate all cases of true dependences between *PPR* link and body pairs. Of the four possible (source, sink) pairings, only the last two—(body i, body j) and (body i, link j), $i < j$ —are parallel. Since both types of dependences begin at a *PPR* body, it is the only place we need to be annotate for writes.

Table 1: Annotation is needed only for true dependences between body-body and body-link pairs, $i < j$.

source, sink	parallel?	data sharing	annotation
link i, link j	no	direct	none
link i, body j			
body i, body j	yes	patching	ppr_write at i,
body i, link j			ppr_read at j

Access annotation is used to monitor dependences. It is indirect compared to annotating dependences explicitly. The indirect approach has several benefits. First, the number of dependence annotations may be quadratic to the length of the code, while the number of access annotations is at most linear. Second, access annotations are modular. They can be specified function by function and aggregated in larger modules without any extra effort. Finally, the source and sink locations of a dependence may be dynamic.

3 Correctness Checking

We use the following notations. *PPR* tasks are numbered from 0 to $n-1$. Each *PPR* task, $ppr[i]$, has a link $link[i]$ and a body $body[i]$. If a program ends in a link task, for symmetry we assume there is an empty $body[n-1]$ after $link[n-1]$. For each task, we record two sets of memory locations, $read, write$, which are the memory locations annotated by *ppr_read* and *ppr_write*. Also recorded are the values, the value at the first read for each memory location in the *read* set and the value at the last write for each location in the $write$ set.

```
Algorithm 1: semantics checking

// 1. sequential execution
run ppr[0..n-1] in a single process
treat the data in output stmts as annotated reads
record the patches as seq˙ppr[0..n-1].read/write

// 2. canonical execution
for i=1 to n-1
  run link[i]
  fork a process p to run body[i]
  wait for p to finish and get the patch
  raise error if either
     cano˙ppr[i].read != seq˙ppr[i].read
     cano˙ppr[i].write != seq˙ppr[i].write
  copy in body[i].write
end

End Algorithm 1
```

Fig. 2: The algorithm for semantics checking

3.1 Semantics Checking

The results of a *PPR* task are communicated in a patch. If a *PPR* task writes to x, and x is not annotated by a *ppr_write*, the new value will not be included in the patch and will not be seen by computation outside the task. If the new value is only used inside the *PPR* task and not needed outside the task, the write annotation is extraneous. Omitting an extraneous annotation improves efficiency.

However, if the new value of the write is needed, directly or indirectly, to produce a program output later, then the write annotation is necessary. The absence of the annotation is an error. We call it a missing write annotation. The purpose of semantics checking is to examine the program execution for a given input and ensure that the program has no write annotation missing for this input.

The algorithm is given in Figure 2. Each *PPR* task is run twice, first in the sequential execution and second in the canonical execution. To distinguish their results, the algorithm refers to the first as $seq\_ppr[i]$ and the second as $cano\_ppr[i]$.

A missing write annotation may cause the canonical execution to generate a different output than the sequential execution. This is detected because some of the reads will be wrong. A user can examine the canonical execution in a debugger. Debugging is conventional since there is no concurrency or non-determinism. We note that if an error is detected at $ppr[j]$, the missing write annotation may be anywhere in $ppr[0..j-1]$. Not detecting an error at $ppr[i]$ does not mean $ppr[0..i]$ has no missing annotation. Only when the algorithm finishes, will we know whether the program has all necessary write annotations for the test input.

Correctness proof . We prove by contradiction. Assume that semantics checking detects no error but there is an unannotated write in a *PPR* task that is actually necessary because

its value is used to produce a later output. Since the write is not annotated, its value is not visible outside the *PPR*. If the value is needed to produce an output, the output in the canonical execution must be incorrect, which contradicts the assumption that the checking has succeeded.

3.2 Determinism Checking

If a task $ppr[i]$ writes x, and $ppr[j]$ reads and prints x, $i < j$, the canonical execution will be correct since the $ppr[j].write$ is merged into the backbone before $ppr[j]$ starts. $ppr[j]$ does not need a read annotation. In a real run, however, $ppr[j]$ must exhibit a true dependence to $ppr[i]$. This can be done by adding to $ppr[j]$ a read annotation for x. Parallel semantics checking is to ensure that there are enough read annotations to mark all the true dependences between *PPR* tasks.

In a parallel execution, *PPR* bodies may finish in any order. In the absence of true dependences, their patches may be merged at different times. There are two restrictions.

- *Merge time*. A patch can only be merged at the juncture between the end of a *PPR* body and the start of the next *PPR* link. Any number of patches can be merged at the same juncture.
- *Merge order*. The patches can only be added in the sequential order of the *PPR* tasks. For example, the patch for $ppr[i]$, i.e. $ppr[i].write$, can be merged only after $ppr[0..i-1].write$ have all been merged first.

Under these two restrictions, there is still much freedom in when patches are merged. For example, if $ppr[0..i-1].write$ are merged before $link[j]$, then $ppr[i].write$ may be merged before any $link[k]$ for $k \geq j$.

We can view each *PPR* body as a single unit, and the merge time as its execution time on the backbone. Then all link and body executions are serialized. The number of possible orders is the number of all legal interleavings subject to three constraints: an earlier link happens before a later link, an earlier body happens before a later body, and the link happens before the body for the same *PPR*.

The interleaving problem is similar to arranging parentheses correctly in a sequence. Let $link[i]$ and $body[i]$ be a pair of parentheses. A sequence of n open-close parentheses corresponds to the sequential execution of $2n$ link and body tasks. Any sequence of properly nested parentheses corresponds to a possible parallel interleaving. The number of different sequences is $\frac{2n!}{n!(n+1)!}$, known as the Catalan number.

A read annotation is missing in a *PPR* task if its absence can cause one or more of its annotated writes to produce a value different from the canonical execution. Determinism checking detects missing read annotations inductively for each *PPR* in sequential order.

The algorithm for determinism checking is given in Figure 3. The checking is done in a nested loop. The iterations of the outer loop are the inductive steps. At step i, $ppr[i]$ is checked to have sufficient read annotations such that its patch is always the same as its patch in the canonical execution, which is the same as its patch in the sequential execution.

```
Algorithm 2: determinism checking

for i=1 to n-1
  run link[1..i]
  for j=1 to i-1
    run in a new process p
    copy in body[1..j].write
    run body[i] to produce the patch
    raise error if
      ppr[i].read != cano˙ppr[i].read
      ppr[i].write != cano˙ppr[i].write
    terminate process p
  end
end

End Algorithm 2
```

Fig. 3: The algorithm for determinism checking

For the first *PPR*, there is nothing to check. For the second *PPR*, there are two cases. The $body[0].write$ may be merged before or after $link[1]$. The checking procedure would run both cases and check whether $ppr[1]$ produces the same result as it does in the canonical execution. Suppose there is a true dependence from $ppr[0]$ to $ppr[1]$, but the absence of a read annotation in $ppr[1]$ causes this to be missed at the merge time. The checking procedure would detect an error and stop.

The inductive process checks $link[i]$ and $body[i]$ after checking all previous links and bodies. As in the canonical execution, the algorithm checks the *PPR*s sequentially in the outer loop and applies a different number of patches in each step of the inner loop. First, it runs $body[i]$ with no prior patches. Then for $j = 1, \ldots, i-1$, it includes the patches of *PPR* 0 through j.

Correctness Proof . Prove by contradiction. If the determinism checking succeeds, but in one of the speculative executions, the output is incorrect. Since all the writes are annotated, there is a read missing. The missing read at ppr[y] and the matching write is in body[x] ($x < y$). Consider the inner loop at iteration i=y. For all iterations $j < x$, the execution of ppr[y] does not have the result of body[x]. There will be an error raised. Contradiction to the assumption that the determinism checking has succeeded.

4 Discussion

Composability of annotations Given the program, if we fix the hints and test for annotation correctness, we have the property that as new annotations are added for later tests, they preserve the correctness of earlier tests. New annotations do not break the correctness of previously passed tests. This property helps to bring down the cost of concurrency error testing to a level closer to sequential error testing.

Composability of annotated code Multiple *BOP* tasks can be grouped to form a single task the same way sequential tasks are stringed together. In addition, *BOP* tasks may

run with auto-parallelized code, since both have sequential semantics. However, when new parallelism is introduced, e.g. by adding a task or removing a barrier, old access annotations need to be checked for completeness.

Automation Automatic techniques may be used to identify shared data accesses and annotate them using the annotation interface. Such analysis includes type inference as in Jade [22], compiler analysis as in CorD [24], and virtual memory support as in *BOP* [8]. A user may use automatic analysis in most of the program and then manually annotate critical loops or functions. The hybrid solution lets a programmer lower the monitoring cost while letting a tool perform most of the annotation work.

Shared vs. private by default Most costs of speculation come from monitoring, checking, and copying shared data. *BOP* chooses to provide interface to specify shared data access because a user can minimize the monitoring cost by specifying only the data that has to be shared. Furthermore, the user can annotate data by regions rather than by elements, reducing both the number of annotation calls and the size of meta-data that the speculation system has to track and process.

Data access vs. data identity Data identity takes just one annotation per variable. Data access takes up to two annotations per datum per *PPR* (one per link task). A benefit, however, is the uniform treatment of global and heap data. A declaration-based method would have difficulties regarding dynamic data: heap data often has no static address, the access is often conditional, and the data location is often dynamically computed. Access annotation is also dynamic in that the role of data, whether shared or private, is allowed to change in different program phases. Finally, it is also more precise since the annotation can be inside arbitrary control flow to capture the condition of data access and avoid redundant annotation calls. Access annotation, however is harder to ensure completeness.

A comparison Table 2 compares *BOP* with other annotation schemes: annotation of private data (the rest is shared) as in OpenMP, annotation of shared data as in Treadmarks [2], annotation of shared data access as in DSTM [11], and annotation (registration) of files in distributed version control as in Mercurial.

Table 2: Comparison of five annotation schemes.

data sharing	annotation unit	frequency	safety
BOP copy-n-merge	access annotation	$\leq$ 2 per datum per task	sequential
private by default	declaration, e.g. OpenMP	1 per variable	no
shared by default	allocation, e.g. Treadmarks	1 per variable	no
	access, e.g. DSTM	1 per access	transaction
version control	file, e.g. Mercurial	1 per file	no

Like DSTM, *BOP* annotation is based on data access rather than data identity. Unlike DSTM, *BOP* uses copy-n-merge (Section 2.2), which requires annotation per *PPR* not per access. In fact, the annotation can be reduced to two per task pair as shown later in an example in Section 5. Because of copy-n-merge, annotations in *BOP* are entirely

local operations. Synchronization happens at the end of a task. *BOP* is similar to check-in and check-out in distributed version control, which has one copy-in and one copy-out per datum per parallel execution. Unlike check-in/check-out, not all data sharing, i.e. link-link and link-body in Table 1, requires annotation. *BOP* is also distinct in its safety guarantee and the need for correctness checking as described in this paper.

5 Evaluation

Access annotation was used in our earlier paper, which describes the interface (including the ordered block) and the safe implementation of dependence hints [14]. As mentioned in a paragraph in Section 5.1 Experimental Setup, "*BOP* provides an annotation interface for recording data access at byte granularity." The interface described in this paper was necessary to parallel seven of the eight tests in that paper. The programs include string substitution (Section 5 of this paper), two clustering algorithms, and five SPEC benchmark programs: art, bzip2, hmmer, parser, and sjeng. The parallel speedup for these programs ranges from 5.8 to a factor of 14 when running on a 16-core machine.

Next we show performance for two of the tests. The first is string substitution, whose parallelization requires byte-granularity annotation. The second is k-means clustering, for which we compare the manual annotation with paging-based monitoring as used in the previous paper [14]. We also compare with OpenMP, which has no access annotation. OpenMP (or any other non-speculative system) cannot parallelize string substitution.

For demonstration, we test two different multi-core machines: k-means on a machine with two 2.3GHz quad-core Intel Xeon (E5520) processors with 8MB second-level cache per processor, and string substitution on a machine with four 2.5GHz quad-core AMD Opteron (8380) processors with 512KB cache per core. Both are compiled by by GCC 4.4 with "-g3". The performance is measured by the speedup over the sequential version of the program and shown in Figure 4.

String substitution The test program finds and replaces a pattern in 557MB of text, divided into 55,724 *PPR*s. The sequential run time is 4.4 seconds. Given the small size of each *PPR*, *BOP* uses a pool of processes rather than starting a process for each *PPR*. It uses a manager process to check correctness. When it has no checking work, the manager computes on the next *PPR*. At a conflict for safe recovery, *BOP* resumes from the last correct *PPR* and starts a new process pool.

We test the program with 5 different levels of conflicts: no conflict, 1%, 5%, 10%, and 50% conflicts. With no conflicts, the speed is improved by 94% to a factor of 5.5 with 2 to 9 processors, as shown in Figure 4. The execution time is reduced from 4.4 seconds to 0.8 second. The improvement decreases in the presence of conflicts, as the four other curves show. As expected, parallel performance is sensitive of the frequency of conflicts. The maximal speedup drops to 4.9 for 1% (551) conflicts, to 2.7 for 5% (2653) conflicts, and to 1.8 for 10% (5065) conflicts.

In the case of 50% conflicts, every other *PPR* fails the correctness check and requires a rollback. The parallel execution is slower by 6% to 19%. It shows the efficiency of understudy-based error recovery in *BOP*.

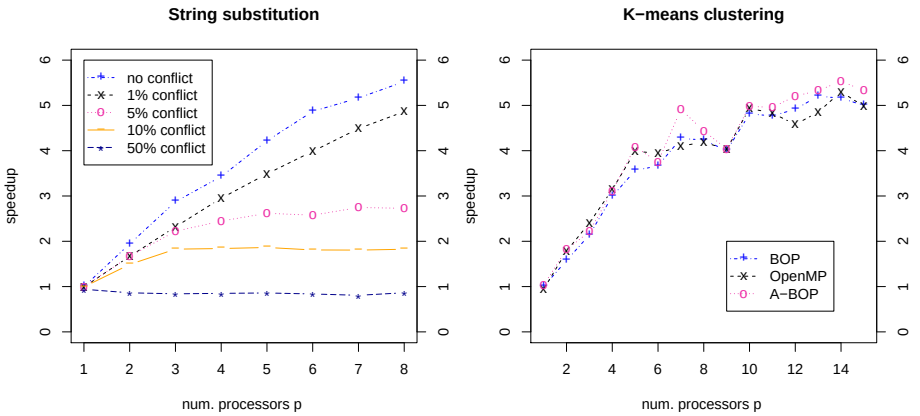


Fig. 4: Demonstration of *BOP* performance. The test of string substitution shows that *BOP* exploits speculative parallelism in the presence of unknown conflicts. The test of k-means clustering shows that *BOP*, which monitors select data, reduces the overhead of *BOP*, which monitors all data, and performs similarly to OpenMP, which does not use speculation or guarantee sequential equivalence. OpenMP cannot parallelize the string substitution.

K-means clustering The program clusters N points into k clusters iteratively. It starts with k centroids. In each step, it assigns each point to the closest centroid, and recomputes the center of each cluster as new centroids. In this test, we use 8 million points in 20 dimension space, 10 clusters, and 10 iterations. The sequential time is 110 seconds.

The original *BOP* uses page protection for all global and heap data, which includes 640MB for coordinating data, 32MB for storing cluster assignments (both old and new assignments), and 2480 bytes for the centroids. It uses padding to separate the three arrays. To avoid false sharing, it blocks the loop so each *PPR* computes on 409,600 points and use different memory pages.

The coarse granularity limits the amount of parallelism—there are 190 *PPR*s, every 19 of them are parallel. As a result, the performance does not increase linearly with the number of processors. For example, a clustering iteration takes about the same amount of time using 8 and 9 processors.

Two access annotations are used, one for the centroid array, and the other for the (new) assignment. It monitors access to 16MB data, 2.3% of total data. The speedup, shown as A-BOP in the figure, is consistently better than *BOP*, although the difference is less than 10%. The OpenMP version has a higher sequential overhead (4%). It is about the same speed as A-BOP, although at finer granularity, the OpenMP version ran faster. Overall for k-means, the needed annotation is small (2 annotations), the performance is improved over automatic monitoring and comparable to OpenMP.

6 Related Work

Software speculative parallelization Software speculative parallelization was pioneered by Rauchwerger and Padua [21]. While most techniques automatically parallelized do-all loops (with limited potential on full applications [15]), several techniques provided

a safe interface for expressing possible parallelism [25, 26] and likely dependence [27]. *BOP* used Unix processes to implement parallelism and dependence hints for sequential C/C++ programs [8, 12]. Process-based systems use the virtual memory protection mechanism, which transparently provides strong isolation, on-demand data replication, and complete abortability. Similar benefits can be realized for threads using the Copy-or-Discard model with compiler and run-time support [24]. Raman et al. presented a software system called SMTX that supports pipelined execution of sequential loops [19]. Recently, Feng et al. generalized the model as SpiceC (scalable parallelism via implicit copying and explicit commit) to support either speculative parallelization by default or other types of commits defined by the user [9].

The original *BOP* divides program data into three categories — shared, checked, and likely private —for complete monitoring [8]. Shared data is monitored at page granularity. Value-based checking is precise and has no false sharing. SMTX uses value-based checking for all shared data to eliminate the false sharing (at the cost of per-access monitoring and logging) [19]. Instead of automatic monitoring, this paper describes an interface for a user to control access monitoring. It shows when annotations are necessary and how to ensure their correctness. A manual interface may leverage user knowledge and enable more efficient and precise monitoring than what is possible with automatic methods alone.

The framework of task isolation and ordered commit has been used to ensure deterministic semantics and eliminate concurrency errors in multi-threaded code. Grace used processes to implement threads to eliminate deadlocks and data races [5]. Burckhardt et al. defined isolation and revision types in C# to buffer and merge concurrent data changes in r-fork/r-join threads [6]. Determinator was developed as a new operating system that buffers processes and threads in private workspaces and terminates an execution if concurrent data writes are detected [3]. CoreDet ensured determinism in threaded execution using versioned memory and a deterministic commit protocol [4]. The access annotation of *BOP* may help to make program monitoring more precise and generally applicable in these systems. The concept of executable declaration is applicable, so is the use of error recovery and speculative synchronization.

Scott and Lu gave five definitions of determinism and showed their containment relationships [23]. A language-level definition is $ExternalEvents$, which requires that the observable events in two executions be the same. An implementation level definition is $Dataflow$, which requires that two executions follow the same "reads-see-writes" relationship. *BOP* lets a programmer define external events and relies on speculation to preserve data flow. The combination enables user control over both the semantics and the cost of its enforcement.

Race detection in fork-join parallelism On-the-fly race detection can be done efficiently for perfectly nested fork-join parallelism [17, 20]. Callahan and others showed that at the program level, the problem of post-wait race checking is co-NP hard and gave an approximate solution based on dataflow analysis [7]. They used the term *canonical execution* to mean the sequential execution of fork-join parallel constructs. The post-wait race checking can be done at run time in $O(np)$ time, where n is the number of synchronization operations and p is the number of tasks [18]. These and other results are summarized in [10]. The execution model of *BOP* is speculative, and the primitives of

*PPR*s and ordered sections are hints and do not affect program semantics. The access annotation in *BOP* is used both at debugging time and at run time, so it needs the maximum efficiency. *BOP* gains efficiency from its data sharing model, which is copy-n-merge. As discussed in Section 4, *BOP* needs at most two annotations per datum per task and does not synchronize at every access. Race checkers, like DSTM mentioned in Section 4, use shared memory and have to monitor all accesses, and except for the recent Tardis system [13], update a shared data structure at each shared-data access [20].

7 Summary

We have presented a new interface for access annotation. As access monitoring becomes programmable, it becomes part of program semantics. This paper provides two techniques to check correctness: canonical execution to ensure sequential semantics and check for missing write annotations, and a quadratic-time algorithm to ensure determinism and check for missing read annotations. In addition, the paper demonstrates efficient and safe parallelization of non-trivial programs. Some of them, string substitution and time skewing, cannot be parallelized by a conventional interface like OpenMP. Others, when parallelized safely, have a similar performance as OpenMP.

The new interface gives a programmer direct control over the cost and precision of access monitoring. Much of the cost can be saved by leveraging user knowledge. It enables a user to control both the semantics and its enforcement. Finally, the interface may be used by an automatic tool, allowing the mixed use of manual and automatic parallelization.

References

1. R. Allen and K. Kennedy.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. In: Morgan Kaufmann Publishers, Oct. (2001)
2. C. Amza, A. L. Cox, S. Dwarkadas, P. J. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel.: Shared memory computing on networks of workstations.In: IEEE Computer, 29(2):18–28 (1996)
3. A. Aviram, S.-C. Weng, S. Hu, and B. Ford.:Efficient system-enforced deterministic parallelism. In: OSDI. (2010)
4. T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman.: a compiler and runtime system for deterministic multithreaded execution. In: ASPLOS, pp 53–64. (2010)
5. E. D. Berger, T. Yang, T. Liu, and G. Novark.: Grace: Safe multithreaded programming for C/C++. In: OOPSLA, pp 81–96. (2009)
6. S. Burckhardt, A. Baldassin, and D. Leijen. : Concurrent programming with revisions and isolation types. In: OOPSLA, pp 691–707. (2010)
7. D. Callahan, K. Kennedy, and J. Subhlok.: Analysis of event synchronization in a parallel programming tool. In: PPoPP, pages 21–30. ACM. New York, NY, USA. (1990)
8. C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. : Software behavior oriented parallelization. In: PLDI, pp 223–234, (2007)
9. M. Feng, R. Gupta, and Y. Hu.: SpiceC: scalable parallelism via implicit copying and explicit commit. In: PPoPP. pp 69–80. (2011)
10. D. P. Helmbold and C. E. McDowell.: A taxonomy of race detection algorithms. In: Technical Report UCSC-CRL-94-35, University of California, Santa Cruz. (1994)

11. M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III.: Software transactional memory for dynamic-sized data structures. In: Proc. of the 22nd ACM Symp. on Principles of Distributed Computing. pp 92–101, Boston, MA. (2003)
12. B. Jacobs, T. Bai, and C. Ding.: Distributive program parallelization using a suggestion language. In: Technical Report URCS #952, Department of Computer Science, University of Rochester. (2009)
13. W. Ji, L. Lu, and M. L. Scott.: Tardis: Task-level access race detection by intersecting sets.In: Workshop on Determinism and Correctness in Parallel Programming, (2013)
14. C. Ke, L. Liu, C. Zhang, T. Bai, B. Jacobs, and C. Ding.: Safe parallel programming using dynamic dependence hints. In: OOPSLA. pp 243–258. (2011)
15. A. Kejariwal, X. Tian, W. Li, M. Girkar, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos.: On the performance potential of different types of speculative thread-level parallelism. In ICS. pp 24. (2006)
16. M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew.: Optimistic parallelism requires abstractions. In: PLDI. pp 211–222. (2007)
17. J. M. Mellor-Crummey.: On-the-fly detection of data races for programs with nested fork-join parallelism. In SC. pp 24–33. (1991)
18. R. H. B. Netzer and S. Ghosh.: Efficient race condition detection for shared-memory programs with post/wait synchronization. In ICPP. pp 242–246. (1992)
19. A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August.: Speculative parallelization using software multi-threaded transactions. In ASPLOS. pp 65–76. (2010)
20. R. Raman, J. Zhao, V. Sarkar, M. T. Vechev, and E. Yahav.: Scalable and precise dynamic datarace detection for structured parallelism. In PLDI. pages 531–542. (2012)
21. L. Rauchwerger and D. Padua.: The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In PLDI, La Jolla, CA. (1995)
22. M. C. Rinard and M. S. Lam.: The design, implementation, and evaluation of Jade. In TOPLAS. ACM 20(3):483–545. (1998)
23. M. L. Scott and L. Lu.: Toward a formal semantic framework for deterministic parallel programming. In the Second Workshop on Determinism and Correctness in Parallel Programming. (2011)
24. C. Tian, M. Feng, and R. Gupta.: Supporting speculative parallelization in the presence of dynamic data structures. In PLDI. pp 62–73. (2010)
25. C. von Praun, L. Ceze, and C. Cascaval.: Implicit parallelism with ordered transactions. In: PPoPP. pp 79–89, Mar. (2007)
26. A. Welc, S. Jagannathan, and A. L. Hosking.: Safe futures for Java. In OOPSLA. pp 439–453. (2005)
27. A. Zhai, J. G. Steffan, C. B. Colohan, and T. C. Mowry.: Compiler and hardware support for reducing the synchronization of speculative threads. In ACM Trans. on Arch. and Code Opt. 5(1):1–33 (2008)