Safe Parallel Programming using Dynamic Dependence Hints

Chuanle Ke and Lei Liu
Institute of Computing Technology, Chinese Academy of Sciences
{kechuanle,liulei}@ict.ac.cn

Chao Zhang
Intel Labs, China
zhangchaospecial@gmail.com

Tongxin Bai, Bryan Jacobs, and Chen Ding
University of Rochester
{bai,jacobs,cding}@cs.rochester.edu

Abstract

Speculative parallelization divides a sequential program into possibly parallel tasks and permits these tasks to run in parallel if and only if they show no dependences with each other. The parallelization is safe in that a speculative execution always produces the same output as the sequential execution.

In this paper, we present the dependence hint, an interface for a user to specify possible dependences between possibly parallel tasks. Dependence hints may be incorrect or incomplete but they do not change the program output. The interface extends Cytron's do-across and recent OpenMP ordering primitives and makes them safe and safely composable. We use it to express conditional and partial parallelism and to parallelize large-size legacy code. The prototype system is implemented as a software library. It is used to improve performance by nearly 10 times on average on current multicore machines for 8 programs including 5 SPEC benchmarks.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming

General Terms Languages, Performance

Keywords do-across parallelism, post-wait, speculative parallelization, safe parallel programming

1. Introduction

Speculative parallelization divides a sequential program into possibly parallel tasks—for example as safe futures [33], ordered transactions [32] or *PPR*s [12]—and uses a run-time system to ensure sequential equivalence. Speculation is useful in addressing the problems of uncertain parallelism due to either implementation or program input. It enables safe parallelization of programs that use legacy code and programs that have frequent but not definite parallelism.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.
Copyright © 2011 ACM 978-1-4503-0940-0/11/10. . . $10.00

Most previous systems allow speculation to succeed only if program tasks are completely independent (also called do-all parallelism or embarrassingly parallel). Two tasks are serialized if they have a conflict. However, in many cases tasks are partially parallel (called do-across parallelism). An example is result collection, where a set of tasks compute on different data and then combine their results into a shared counter. Another one is pipelined parallelism, where each task is divided into stages, and parallelism exists between stages rather than tasks as a whole.

To safely express partial parallelism, we present the dependence hint, an interface for a user to suggest post-wait dependences between possibly parallel tasks. It enables tasks to speculatively synchronize with each other. For example, a user can parallelize a compression utility by dividing the input into chunks and then use dependence hints to assemble the compressed chunks after parallel compression.

Dependence annotations have been studied in the past, including post-wait by Cytron for static parallelization [10], signal-wait by Zhai et al. for thread-level speculation (TLS) [35], and flow by von Praun et al. for ordered transactions [32]. In these systems, parallel tasks shared data directly. In the latter two, speculation required special hardware.

Speculation support may be implemented on conventional hardware, using a two-step strategy. The first is copy-on-write in speculative tasks to isolate them from each other. The second is serial commit to merge concurrent changes and resolve conflicts.

This software-only strategy is used by *BOP* [12, 17, 36] and Cord [30] for speculative parallelization, Grace [7] for race-free threaded execution, isolation and revision types in C# [8] and implicit copying and explicit commit in C [13] for safe parallel programming, DoublePlay [31] for deterministic replay, CoreDet [6] and Determinator [3] for deterministic parallel execution.

While these systems eliminate the interaction between parallel tasks to ensure safety, these tasks cannot coordinate with each other during execution. Any interaction, e.g. enforcing a dependence, would require explicit communication.

The dependence hint uses a *channel* to express communication between copy-on-write tasks. The channel abstraction has three benefits. First, a channel communicates dynamically allocated data, i.e. a task can receive new data from a peer task without knowing the address. Second, a channel communicates aggregate data, which allows staged execution such as pipelining. Finally, channels may be chained to handle conditional dependences.

We design the dependence hint with the following goals:

- *Safety.* Dependence hints are added to a sequential program. The program with hints always produces the identical result as the program without hints (hence no need for parallel debugging). When the hints are wrong, the program should not run slower than the sequential version.
- *Expressiveness*. The hint can express regular and conditional dependences, on data allocated before or during parallel execution.
- *Concision*. Not all dependences need hints. The needed ones can be combined in a few hints or expressed through high-level constructs.

These properties by themselves do not imply that dependence hints can be effectively used by average programmers. There are inherent difficulties in reasoning about concurrent value flows on replicated data. Our focus is the safe implementation that ensures correct program output against all possible errors. A system may help a user to correct erroneous hints or insert hints automatically, although these extensions are outside the scope of this paper.

Because of the cost of software implementation, we focus on coarse-grained task parallelism in sequential code. We target small-scale parallelism to utilize spare cores on today's workstations rather than for scalable performance on massively parallel machines.

The rest of the paper is organized as follows. Section 2 introduces software speculative parallelization. Section 3 describes the dependence hint: the interface, the safe implementation, the conditional and high-level hints. Section 4 demonstrates their use on four examples. Section 5 shows the performance. Finally, the last two sections discuss related work and summarize.

2. Background on The Parallelism Hint

Software speculative parallelization is pioneered by Rauchwerger and Padua [24]. Our work is based on behavior-based parallelization (*BOP*), which provided a manual interface to suggest *possibly parallel regions (*PPR*)* [12]:

- bop_ppr{ *PPR* code } marks a block of code and suggests task parallelism—the code in the *PPR* block may be parallel with the code after the *PPR* block.

BOP uses a process to implement a *PPR* task. A process is heavy weight, but it is flexible and fully protected. A *PPR* task can be forked anywhere in a program and can be aborted by simply killing it. More interesting is the effect on data sharing. When a process writes to a page for the first time, it makes a copy of the page. This is known as *copy-on-write*. Data copying insulates a *PPR* task against changes happening in other tasks. More importantly, data copying removes all false dependences—write-after-read, write-after-write conflicts—between *PPR* tasks.

When *PPR* tasks finish, *BOP* checks their data access for true dependences (read-after-write) and if none found, merges the data changes. Since a *PPR* task may be wrong, an understudy process re-executes speculative work non-speculatively. The understudy is always correct because it is the same as sequential execution.

Listing 1 shows a simple program with two *PPR* tasks.

Listing 1: Two *PPR* tasks

```
# try setting g[x] and using g[y]
#     in parallel
bop_ppr {
  g[x] = foo( x )
}
bop_ppr {
  bar( g[y] )
}
```

Listing 2: Implementation using 3 processes

```
t1 = fork {
  # copy-on-write g[x]
  g[x] = foo( x )

  # understudy for error recovery
  t2_undy = fork {
    bar( g[y] )  # safe re-execution
    kill t2      # abort speculation
  }
}
t2 = fork {
  bar( g[y] )    # speculation

  # wait for t1
  join(t1)
  # check correctness of t2
  if g[x] and g[y] are the same array cell
     exit            # abort speculation
  else
     copy g[x] from t1 to t2 # commit
     kill t2_undy      # abort understudy
  end
}
# either t2 or t2_undy succeeds/continues
```

Listing 2 shows the *BOP* implementation with three processes: the first and the third fork operations are for the two *PPR*s and the second fork starts the understudy. The implementation is simplified for the purpose of illustration and does not represent the complete design [12].

Safe parallelization relies on the following:

- *fork-without-join.* A *PPR* task commits when it finishes. A user does not specify when a task should finish.
- *copy-on-write data replication. PPR* tasks are isolated from each other. All false dependences are eliminated.
- *access monitoring.* A *PPR* task uses virtual memory support to monitor access (to global and heap data) at page granularity. The first read and write access to each page is recorded. The read and write sets are compared after the parallel execution to check for dependences.
- *recovery via understudy.* The understudy process runs the original code and aborts *PPR* tasks if they are wrong or too slow.

BOP supports only iterative parallelism with no nesting. A *PPR* is ignored if it is encountered in a *PPR* execution. Nesting can be supported using established techniques in race checking [5, 20].

In addition to *BOP*, a number of speculation systems are based on Unix processes, including Grace [7], SMTX [23] and recently DoublePlay [31]. Process-based speculation has been extended to support speculative memory allocation in Grace [7], irregular task sizes and adaptive speculation in *BOP* [36**?**].

As an interface, the parallelism hint has a major limitation—*PPR* tasks cannot collaborate with each other while they are executing. Dependences between active *PPR* tasks have to be enforced by rollbacks and sequential re-execution. We next show the dependence hint to allow tasks to utilize partial parallelism and avert rollbacks.

3. The Dependence Hint

This section presents the interface, the safe implementation, and improvements in efficiency and expressiveness. If the basic interface in Section 3.1 feels too low-level for manual use, treat it as the implementation interface for building high-level hints in Section 3.4.

3.1 The Interface

The dependent hint has three primitives, which are matched at run time by a unique *channel* identifier. The identifier is either a number or a string.

- bop_fill(channel_id, addr, size) is called by a sender task to register an address range with a channel. The data is copied when the channel is posted.
- bop_post(channel_id [, ALL]) is called to post the channel and its data. An optional ALL, if specified, means to post all modified data.[1] A channel is *single post* — posting to a posted channel has no effect (a no-op).
- bop_wait(channel_id) stalls the receiver task until the channel is posted. The received data is placed in the receiver task *in the same address* as in the sender. If a task waits for the same channel multiple times, later waits have no effect (no-ops). Multiple tasks may wait for the same channel and receive data from the same sender.

A sender task fills a channel with data and then posts the channel. A receiver task waits for a channel and then copies the channel data. The hint represents both synchronization and communication: the post is a non-blocking send, and the wait is a blocking receive.

An example Listing 3 shows a processing loop that computes on each input from a queue and inserts the result to an output queue. The processing step may be parallel, but the queuing step is not—each iteration adds a new node to the output queue. Speculation in Listing 3 would always fail because of the dependence.[2]

The solution in Listing 4 serializes the queuing step using a dependence hint. Each (non-first) iteration calls bop_wait to wait until the previous iteration posts the result. Then it receives the queue tail, appends a new node, and calls bop_fill and bop_post to send the new tail to the next iteration. The variable cid is used to create a new channel id at each iteration, so there is no reuse of channels and no cross-talk between non-adjacent iterations.

Listing 4 uses three primitives. To make the coding simpler, we can replaced them by a high-level hint, bop_ordered, which we show in Listing 5 and will describe in Section 3.4.

Next we explain the three features of the dependence hint necessary for this example to be parallelized correctly.

Sender-side addressing As a sender fills a channel, it records both the data and its address. A receiver does not say what data to receive or where to place the received data—the data from the channel is placed in the same address as the sender specified. Sender-side addressing has three benefits.

First, it simplifies the receiver interface and avoids the problems of mismatch. An example mismatch, as can happen in MPI, is when a task sends a message of size n, but the receiver expects a message of size $2n$. Sender-side addressing removes all possible sender-receiver disagreements.

Second, sender-side addressing allows a task to communicate dynamically allocated data (of a dynamic size). This is necessary for the solution in Listing 4. The output queue grows node by node. The next iteration can receive the new tail, even though it did not have the node in its address space and had no knowledge of its allocation in the previous iter-

[1] which means all data modified since the last fill/post/wait call or if there is none, all data modified since the start of the task.

[2] Note that the speculation would still fail if the queue insertion is moved outside the *PPR* block, since the insertion needs the datum computed inside the *PPR*.

Listing 3: A possibly partially parallel loop

```
while (has_more(inputs)) begin
  w = get_next(inputs)
  # try computing w in parallel
  bop_ppr {
    t = compute(w)
    # allocate a new node n
    n = new_qnode(t)
    # make n the new tail
    append(outputs, n)
  }
end
```

Listing 4: Safe parallelization using basic primitives

```
cid = 0    # channel id
while (has_more(inputs)) begin
  w = get_next(inputs)
  bop_ppr {
    t = compute(w)
    n = new_qnode(t)
    # wait for the last tail
    bop_wait ( cid - 1 ) if cid>0
    append(outputs, n)
    # send the new tail
    bop_fill( cid, n, sizeof(qnode) )
    bop_post( cid )
  }
  cid ++
end
```

Listing 5: An equivalent solution using a high-level hint

```
while (has_more(inputs)) begin
  w = get_next(inputs)
  bop_ppr {
    t = compute(w)
    bop_ordered {
      n = new_qnode(t)
      append(outputs, n)
    }
  }
end
```

ation. Communicating dynamic data is as simple to code as communicating static data.

Third, the run-time system can dynamically change the content of communication. This feature is critical in the safe implementation which we will describe in Section 3.2.

Selective dependence marking Enumerating all dependences is impracticable because there may be n^2 dependences in an n-statement program. The dependence hint is for selective marking, i.e. for only dependences from a *PPR* task to its continuation. We call these PPR *dependences*. Other dependences do not require hints, including dependences within a *PPR* task, dependences within inter-*PPR* code and from inter-*PPR* to *PPR* code.

PPR dependences do not need hints if they are too infrequent to affect performance. The rest, more regularly occurring dependences can be divided into two types: short range and long range. Short-range dependences happen between nearby *PPR* tasks, which are likely to require hints for coordination. Long-range dependences happen between distant *PPR* tasks, which are most likely already serial and do not need hints. For example when parallelizing a loop, hints are needed for short-range dependences between consecutive iterations but not for long-range dependences, e.g. between the loop and the subsequent code.

Furthermore, multiple dependences can share a single channel and be marked by a single hint. For example, in pipelining, each stage needs just one post and one wait. Another example of enforcement en masse is to suggest a join point to serialize two *PPR* tasks and satisfy all dependences between them. For these reasons, the number of hints can be few even though the dependences may be many.

A careful reader may note that long-range dependences, although they do not need synchronization, still need communication. Such communication is done at the commit time when a *PPR* task finishes and its modified data copied into later tasks. Data commits and dependence hints are the two ways by which *PPR* tasks share data. Data commits move data asynchronously and do not block an active task. Dependence hints are synchronous and may stall the receiver task. Dependence hints require an explicit hint, while data commits do not.

Now we can explain a subtlety in the solutions in Listing 4 and Listing 5. The last *PPR* task, the one to create the last node, is supposed to have the full queue, but it does not. From the dependence hint, it has only the node it creates and the one before it—just the two nodes, not the full queue. How and when is the entire queue assembled? The rest of the queue is pieced together by data commits. As tasks finish, their data is copied out and merged. The construction happens asynchronously as the loop progresses.

Selective dependence marking benefits both programmability and performance. In the example, parallelization is simple since only the tail node requires a hint. It is also efficient and more scalable since the communication is constant size rather than linear size. Each node is copied just once.

Safety and determinism Unlike in non-speculative systems where communication primitives must be perfectly paired, dependence-hint primitives are suggestions and may mismatch.

To ensure determinism, a channel accepts at most one post. If we were to allow multiple posts, we would be uncertain how many of the posts had happened at the time of

a bop_wait. In addition, the restriction means that a channel cannot be filled by multiple tasks. If two *PPR* tasks could fill the same channel, we would be uncertain which task finished first placing data in the channel. A third benefit is efficiency — bop_fill is a task-local operation since a channel is never used by more than one writer.

A programmer may not know completely about the code she is parallelizing. For example the loop shown in Listing 3 may have abnormal entries and early exits in the compute call. The call may have hidden dependences including access to the output queue. Next we describe the safe implementation to guard against potential errors.

3.2 Safe Implementation

Incomplete knowledge of a program can cause three types of errors in hint specification: under-specification, where an actual dependence is not hinted; over-specification, where a hinted dependence does not happen in execution; or incorrect specification, where the location or the cause of a dependence is wrong. Some errors (e.g. a missed dependence) may cause the speculation to fail, some (e.g. a widow wait) may delay a task unnecessarily, and some (e.g. a widow post and intra-task post-wait in our example) may add unnecessary overhead but do not otherwise harm parallel execution.

To present our implementation and show its correctness, we first define a *conflict*, which happens when a task in a parallel execution reads a value different from the would-be value in a sequential execution. It follows that a parallel execution produces the sequential result if there is no conflict.

The absence of conflicts are ensured by filtering and three checks as follows:

- *filtered posting* — At bop_post, only modified data in the channel is posted. The filtering avoids useless data transfer and simplifies correctness checking.
- *the sender conflict* —The conflict happens when a task modifies the data it has already posted. The check detects the transmission of stale data.
- *the receiver conflict* — There is a conflict if a task accesses (reads or modifies) some data and then later receives it by a wait. The check detects premature use of dependent data.
- *the last-writer check* — For every piece of data received by a task p, its sender must be the last writer before p in the sequential program order.

The filtering and the first two checks are task local and performed during speculation. The last-writer check involves potentially all speculation tasks. It is performed during the commit process.

Consider an example in Figure 1. There are 3 tasks: T1 and T2 write x, and T3 reads x. The correct post-wait is to pass x from T1 to T2 and then from T2 to T3. In the example, however, T1 sends x to T2 and T3.

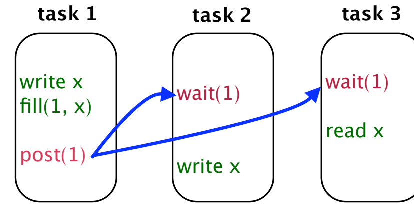


Figure 1: A misuse of post-wait: task 3 reads incorrect x.

The three checks detect the error as follows. The sender-conflict check ensures that T1 send last modified version of x. The receiver-conflict check ensures that T2 and T3 do not use x before getting its new value. The last-writer check ensures the right pairing of communication. In T2, the last writer and the sender of x are both T1, so the check passes. In Task 3, the sender is T1, but the last writer is T2, so the check fails. As a result, the system aborts and re-executes T3. Consider an extension of this example in which T2 sends correct x to T3, but T3 does not wait for it. The last-writer check still fails since T3 consumed the wrong version of x.

Next we show the correctness formally using a proof similar to that of the Fundamental Theorem of Dependence (Sec. 2.2.3 in [2]) and the one in [12].

Theorem 3.1. *The three checks, if all passed, rule out all possible conflicts in a parallel execution.*

Proof We assume that *PPR* tasks are numbered by their sequential order, all inter-task dependences are marked by hints, and all tasks have passed the three checks. We show that the parallel execution has no conflict. We prove by contradiction.

Let task t_q be the first task to have a conflict—it reads location x but the value is different from reading x in the sequential execution. Since all values seen by t_q before this read were correct, the incorrect value must come from outside t_q. Let's assume that in the sequential execution, task t_p is the last to write x before t_q. Since t_q is the first to have a conflict, t_p must be correct and write x correctly. The question is whether t_p communicates to t_q properly and whether other tasks may interfere by interjecting some other x to t_q.

Because all three checks are passed, t_p and t_q must communicate properly. First, t_p sends x after its last write, t_q receives x from t_p before its first read. Second, tasks after t_p do not post x because t_p is the last writer. Third, a task before t_p may send x to t_q. However, t_q must read the version of x from t_p (to pass the last-writer check). Therefore, there is no way for t_q to receive an incorrect x, which contradicts the assumption that x is incorrect. ∎

Progress guarantee Usually a blocking receive such as bop_wait may entangle a set of tasks into a deadlock. As a dependence primitive, *BOP* communication flows in one direction in the increasing order of the task index. In addition, *BOP* uses an understudy process to execute speculative

tasks non-speculatively. Understudy does not speculate and ignores all hints. If there is a wait with no matching post, the unmatched wait is dropped when the understudy finishes the re-execution of the waiting *PPR*.

Dependence hints are free of concurrency errors such as deadlocks, live locks, and data races. The implementation is wait free, which means that all tasks finish within bounded time—the time taken by the sequential execution.

Conflict handling If *PPR* i incurs a sender conflict, we abort and re-start all *PPR* j for $j > i$. With sufficient book-keeping, we can identify and rollback only those tasks that are affected by the offending send. In addition, a sender can remove and update data in a posted channel if it has not been received, or if it is received but not consumed. To avoid recurring conflicts, the *BOP* runtime can ignore a post operation if it caused a sender conflict before, leveraging the learning strategies studied by Jiang and Shen [**?**].

At a receiver conflict, the faulting task is aborted. A special case is when a task receives the same data, say x, from different senders (note that it cannot receive multiple x from the same sender because of the modification and sender-conflict checks). We rank the recency of received values by the task index of their sender. A larger index means a later sender and a newer value. There are three cases. First, x is first accessed before the first receive, which means a receiver conflict. Second, x is first accessed between two receives. If the value from the first receive is less recent than that of the second receive, a receiver conflict is triggered. If the value from the first receive is newer, the second receive can be silently dropped without raising a conflict. In the third case, x is first accessed after two or more receives. We keep the newest value of x and continue. We call the last two extensions *silent drop* and *reordered receive*.

Inter-*PPR *post-wait A post in an inter-*PPR* gap is unnecessary since its writes are visible to all later execution. bop_fill is ignored, and bop_post marks the channel as posted without communicating any data. Consider for example a dependence between two inter-*PPR* gaps. The dependence is always satisfied since the gaps run sequentially. There is no need to send any data, and there will be none sent. We call it *inter-*PPR *hint override*. Finally, a wait in an inter-*PPR* gap is treated normally since it may be the target of a *PPR* dependence.

Outdated post and channel de-allocation If a post-wait pair spans many *PPR* tasks, it is possible that the receiver already has the sent data. The channel data is freed when the sender commits and all active tasks have the data. The system still stores the identifier of posted channels in case a task later waits for one of these channels. When it happens, the task does not block and does not receive the data again. We call it *distant-*PPR *hint override*. The storage cost is small since the system stores only channel identifiers not channel data.

It is worth noting that most techniques in this section—filtered posting, silent drop, reordered receive, the inter-*PPR* and the distant-*PPR* override—dynamically change the content of a channel. As mentioned before, they are possible because of sender-side addressing. The feature benefits both programming and performance. For example with filtered posting, a task may change sub-parts of an array, and the programmer can simply post the whole array without incurring unneeded communication.

3.3 Conditional Dependence by Channel Chaining

If every task modifies shared data, we enforce in-order access by all of them. Sometimes however, there may be *non-contributing* tasks that decide not to write the shared data. Assuming the decision is dynamic, how should the next task know whether to wait for its predecessor? There are two simple solutions: the next task waits until the previous task finishes, or the previous task tells the next task not to wait. The first loses parallelism. The second requires a communication. Neither does any better whether there is a dependence or not. We note that this problem does not exist in concurrency constructs such as transactions and atomic sections. It is unique with dependence annotations because of their adherence to sequential semantics.

An efficient solution is to dynamically chain multiple channels to bypass non-contributing tasks. We call it *channel chaining*. The primitive is bop_cc:

- bop_cc(channel_1, channel_2) is called by a task to chain two channels. After chaining, a receiver of either channel waits for both channels, and a sender of either channel posts to both channels. The two channels are effectively aliases.

We revise the previous example. In Figure 2, an iteration outputs to the queue only conditionally in some cases; otherwise, it calls bop_cc to chain cid minus 1 with cid, so the next iteration waits for the post by the previous iteration. If only a few iterations generate output, most channels will be chained, and only the generating tasks will synchronize and communicate queue data.

Channel chaining has four benefits. First, any number of tasks may decide not to contribute. More than two channels may be chained. Second, a task may decide at any time not to contribute, regardless whether the peer tasks have posted or waited. Third, no data is transferred to non-contributing tasks. Last, incorrect chaining will be detected, for example, when a task mistakenly connects two channels. The implementation in Section 3.2 is safe against all misuses of channel chaining.

3.4 The Ordered Hint

Dependence, although fundamental, may be too low level for a programmer. In this section, we build a high-level hint as follows:

Listing 6: A loop with a conditional dependence

```
while (has_more(inputs)) begin
  w = get_next(inputs)
  # try computing w in parallel
  bop_ppr {
    t = compute(w)
    # conditional queue insertion
    if t ≠ nil begin
      n = new_qnode(t)
      append(outputs, n)
    end
  }
end
```

Listing 7: Parallelization with channel chaining

```
cid = 0    # channel id
while (has_more(inputs)) begin
  w = get_next(inputs)
  bop_ppr {
    t = compute(w)
    n = new_qnode(t)
    if t ≠ nil begin
      ... # same as Listing 4
    else
      # bypass the sync/comm
      bop_cc( cid -1, cid )
    end
  }
  cid ++
end
```

Figure 2: Channel chaining to handle a conditional dependence. Communication happens only between iterations that perform queue insertion.

- bop_ordered{ code } marks a block of code and suggests an ordered execution—*PPR* tasks running the code should execute one at a time and in their sequential order.

Figure 3(a) shows an ordered block in a function called foo, and the function is called in a while-loop. Figure 3(b,c) show two implementations of bop_ordered depending on whether the number of foo calls is known. When the last call is known, we post after the last call; otherwise, we post at the end of the task. A special case is when foo is not called at all. bop_cc is used.

OpenMP provides ordered as a directive for loop parallelization [21]. Gossamer introduced it as a general directive for task parallelism [26]. The examples in these papers show a single ordered call in each task or loop iteration. In fact, the OpenMP standard requires that ordered be used exactly once in each iteration—*"a conforming example ... each iteration will execute only one ordered region."* [21] It is unclear how errors are handled, e.g. dependence between ordered and unordered code, and how they may affect other parallel primitives in the language.

Like Gossamer ordered, bop_ordered is a general primitive and can be used to serialize code in different functions. Unlike in OpenMP and Gossamer, however, bop_ordered is a safe hint and can be used amidst uncertain control flow. For example a program may branch into or out of an ordered block, which would be illegal in OpenMP. This shows the dependence hint as a firm foundation for the high-level hints. A user can define high-level hints using *BOP* post-wait to define desirable semantics especially with conditional executions (as we have done in Figure 3). The high-level hints are safe in composite uses and against misuse.

4. Programming with Dependence Hints

We show four examples of parallelization. Because of hidden or unpredictable dependences, the first two are difficult to express, and the second two are difficult to implement safely, if we were to use conventional methods.

4.1 String Substitution

Consider the task of sequentially replacing all occurrences of the 3-letter pattern "aba" with "bab". The process may be parallel, e.g. when the input has no "aba". But it may be completely sequential, e.g. when the string is "abaa...a" and should become "bb...bab" after conversion. For safe parallelization, we mark the inner loop a possibly parallel task. To count the number of substitutions, we use an ordering hint to add per *PPR* task counts into a global counter `num`. The program in Listing 8 processes the input in m-letter blocks. The code uses the range syntax. For example, `str[lo...hi]` refers to the series of letters starting from `str[lo]` and ending at (and including) `str[hi]`.

4.2 Time skewing

Iterative solvers are widely used to compute fixed-point or equilibrium solutions. Listing 9 shows the structure of a typical solver as a two-level loop. An outer iteration, often called a time step, computes on domain data first and then checks for convergence. There is no static or dynamic parallelism between time steps: the convergence cannot be checked until all computations finish, and the next step cannot start until the convergence check is done. But there is speculative parallelism—the computations do not converge until the last iteration, so time steps may overlap before then. The transformation is known as *time skewing* [34]. Previous literature shows large performance benefits for both sequential [27, 34] and parallel [18] executions.

Parallelization hints can express time skewing with two *PPR*s and two ordering hints. The first *PPR* (safely) parallelizes the inner loop. The second *PPR* makes the convergence check asynchronous, so the program can skip the convergence check and start the next time step, allowing two

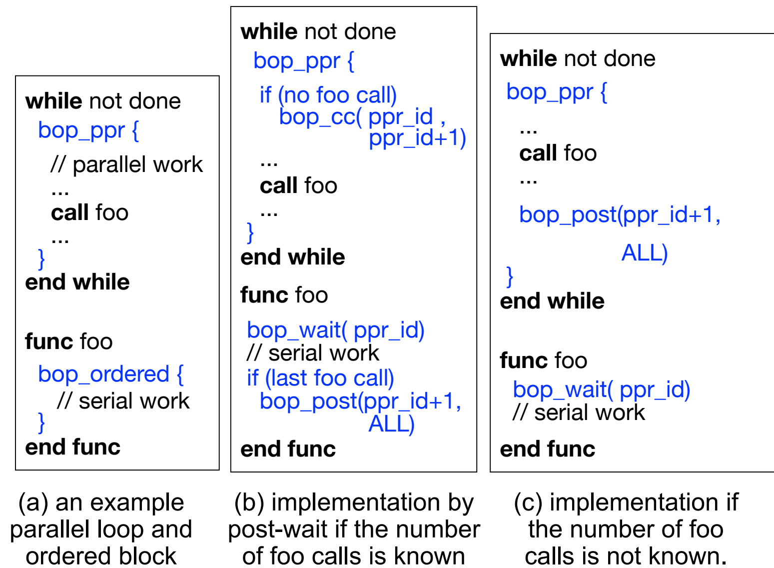


Figure 3: Using post-wait to implement the bop_ordered region hint.

Listing 8: string scanning and pattern conversion

```
str[0...n], src=''aba'', target=''bab''
num = 0    # number of substitutions

for ii in 2...n with step b do
  # try a string block in parallel
  bop_ppr {
    for i = ii...min(ii+b-1, n)
      cnt = 0
      if matches(str[i-2 ... i], src)
         str[i-2 ... i] = target
         cnt ++
      end
    end
    # update num sequentially
    bop_ordered {
      num += cnt
    }
  }
end
```

time steps to overlap. The convergence check must wait for the computations to finish. This is done by two ordering hints: the results of the domain computation is combined in the first ordered region, and the convergence check is then made in the second ordered region. In the last iteration, the write to the converged variable triggers a (true-dependence) conflict with the speculative execution of the next time step. The speculation is then rolled back, and the loop finishes normally as if by a sequential execution.

Listing 9: Time skewing to overlap consecutive time-step executions

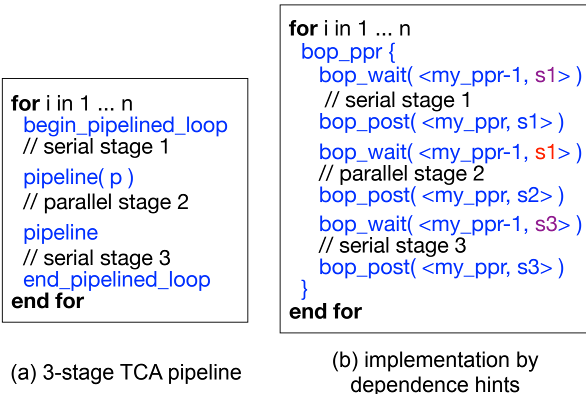
```
converged = false
while not converged

  for i in 1...n
    # try inner loop in parallel
    bop_ppr {
      r = compute( data[i] )
      bop_ordered {
        s = s.add_result( r )
      }
    }
  end

  # try next time step in parallel
  bop_ppr {
    bop_ordered {
      if good_enough?( s )
        converged = true
      end
    }
  }
end
```

4.3 TCA Pipelining

Thies, Chandrasekhar, and Amarasinghe defined an interface for expressing pipeline parallelism in a loop [29]. We refer to the interface as *TCA pipeline* after the initials of the authors. The original version is not speculative, and a recent

system called SMTX added the speculation support [23]. The body of a pipeline loop is divided into stages. Each stage is separated from the preceding stage by a pipeline label. By default, a stage is sequential and its label takes no parameter. A parallel stage has a parameter p to indicate the number of parallel processors to use for the stage.

Figure 4 (a) shows an example TCA pipeline with 3 stages: stages 1 and 3 are sequential, and stage 2 is parallel. The implementation uses one process running each sequential stage and p processes running the parallel stage. Thies et al. developed profiling support to identify and transfer shared data and to divide the stages evenly so that all processes are fully utilized in the steady state.



(a) 3-stage TCA pipeline (b) implementation by dependence hints

Figure 4: Using post-wait to safely implement the pipeline loop construct of Thies et al. [29]

The pipeline parallelism can be implemented by post-wait, as shown in Figure 4 (b). Each stage starts with a wait and ends with a post. The channel identifiers are set up to wait for the same stage in the previous task, if the stage is sequential. A parallel stage has two cases. If it is the first stage, it should not wait for anyone; otherwise, it waits for the previous stage in the previous task. Note that the implementation can be encapsulated so the programmer is provided with the same interface as the TCA pipeline, e.g. through a bop_pipeline hint.

The *BOP* pipeline exploits the same parallelism as TCA pipeline and its safe version SMTX [23], but the implementation is different. In *BOP*, the same task uses the same process, which simplifies error recovery. In TCA and SMTX, the same stage uses the same process(es), which reuses processes. The TCA and SMTX pipelines are likely more efficient when computations are regular and regularly chunked into stages. On the other hand, the fixed stage partition has trouble handling variable length iterations or dependence between non-consecutive tasks. In implementation, TCA and SMTX have the advantage of process reuse over the original *BOP* [12]. The current *BOP* also reuses processes, which we will discuss in Section 5.1.

4.4 Hmmer from SPEC 2006

Hmmer is a genetic search program developed at Washington University with nearly 36,000 lines of C code. Most of the execution happens in two steps: calibration and search. The calibration loop is shown below. Most of the time is spent in the function P7Viterbi. The loop traverses through a series of genetic sequences. It is parallel as far as we know except in the call to AddToHistogram_zc, which adds the result computed in each iteration to a histogram. It can be parallelized by a bop_ppr and a bop_ordered hint as shown below. The entire histogram data (2 memory pages in the test) is marked for posting in every task.

```
for (i = 0; i < parallelism; i++) {

  bop_ppr {  // begin possibly parallel region (PPR)

    mx = CreatePlan7Matrix(1, hmm->M, 25, 0);
    for (idx=i*temp; idx<(i+1)*temp && idx<nsample; idx++) {
      dsq = DigitizeSequence(seq[idx], sqlen[idx]);

      if (P7ViterbiSize(sqlen[idx], hmm->M) <= RAMLIMIT)
        score = P7Viterbi(dsq, sqlen[idx], hmm, mx, NULL);
      else
        score = P7SmallViterbi(dsq, sqlen[idx], hmm, mx, NULL);

      hhu[idx%temp] = score;
      free(dsq); free(seq[idx]);
    }
    FreePlan7Matrix(mx);

    bop_ordered {  // implemented by post-wait

      for (idx=i*temp; idx<(i+1)*temp && idx<nsample; idx++) {
        length_zc = AddToHistogram_zc(&(post_zc.a), hhu[idx%temp]);
        if (hhu[idx%temp] > post_zc.b) post_zc.b = hhu[idx%temp];
      }
    } // end bop_ordered

  } // end bop_ppr
 }
```

The search loop has more dependent operations at the end of each iteration to perform a significance test and add significant matches to a result list. The serial block is several times longer in code and transfers 40 times more data (about 79 pages in the test run) than in the calibration loop. The matched genes are inserted into the result list in the same order as they were read from the input file.

5. Evaluation

5.1 Experimental Setup

BOP implementation *BOP* hints are implemented as run-time library calls. We have completely re-designed and re-implemented the system three times to improve its efficiency. The current design has three important features:

- *Process reuse*. Instead of forking a process for each *PPR*, we fork a set of processes at the first *PPR*. Each one is assigned the next unexecuted *PPR* and returns for a new assignment after finishing. We designate a *main* process to serve as the understudy and always maintain a correct state. In case of a speculation error, the offending pro-

Table 1: The 8 test programs

test	source	code lines/changes			num.	seq.
		orig	bop	omp	*PPR*s	time
str-sub	Section 4.1	80	9	-	279	4.4s
k-means	textbook	260	8	3	200	110s
qt-cluster	[15] & Section 4.2	303	11	4	1,600	438s
art	SPEC 2k	1,270	29	707	2,480	1211s
bzip2	SPEC 2k	4,649	19	-	373	115s
hmmer	SPEC 06 & Section 4.4	33,992	22	62	85,000*	93s
parser	SPEC 2k	11,391	4	-	7,756	140s
sjeng	SPEC 06	13,847	15	-	15	500s

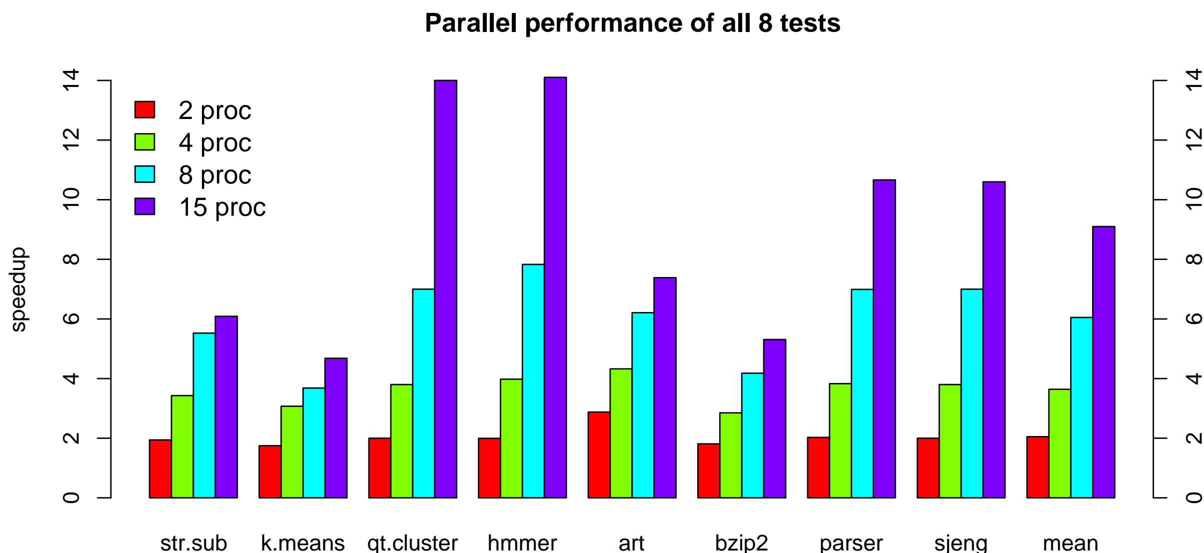


Figure 5: Summary of *BOP* performance. On average, the reduction on end-to-end, wall-clock execution time is a factor of 2.1 on 2 processors, 3.6 on 4 processors, 6.1 on 8 processors, and 9.1 on 15 processors.

cesses are killed, and new ones are forked (from the main process) in their place. In case of unrecoverable operations such as system calls (e.g. mmap), only the main process survives. It starts a new pool at the start of the next *PPR*.

- *Dynamic load balancing*. *PPR* tasks have unpredictable sizes. For load balancing, we use *delegated correctness checking*. When a process finishes a *PPR* task, it submits the changes and then returns to run the next *PPR* task. Correctness checking is delegated to the main process and by doing so, all processes are busy working as long as there is enough parallelism.
- *Byte-granularity checking*. *BOP* provides an annotation interface for recording data access at byte granularity [11]. If a programmer or a compiler knows all the places a variable is accessed, the data can be monitored precisely, which avoids false sharing and the page-protection overhead. Otherwise, page-level protection is used as in the base *BOP*. We use the byte-granularity interface to evaluate the cost of page-level protection and to implement fine-grained *PPR* tasks.

The use of a main process complicates task counting. Most of the times the main process does not contribute to actually executing a program. Therefore we count only speculative processes as tasks.

Test Machines We test two machine platforms. One has four 2.5GHz quad-core AMD Opteron (8380) processors with 512KB cache per core. The other has two 2.3GHz

quad-core Intel Xeon (E5520) processors with 8MB second-level cache per processor. The Intel processors are hyper-threaded, so we run our tests up to 15 tasks. The test programs and the *BOP* code are compiled by GCC 4.1 with "-O3" on the AMD machine and (due to errors when compiled with "-O3") by GCC 4.4 with "-g3" on the Intel machine. The two machines have 32GB and 8GB of physical memory respectively. Both run Linux, Red Hat 4.1.2 and Linux 2.6.30. We run each version of a test on each task count from 1 to 15 for three times (for a total of 45 runs per test) and report the average result.

Test programs Table 1 shows the eight test programs, including the three examples in Section 4 and five full-size SPEC benchmarks. The programs have between 80 and 34-thousand lines of C code. Except for the two clustering tests, all make heavy use of pointer data. To parallelize them, *BOP* adds between 4 and 29 lines of hints and access annotations. Most tests use dependence hints except for *parser* and *sjeng*, which are included to compare with the previous *BOP* [12]. The number of *PPR*s in these programs ranges from 186 to 85 thousand. We test two *PPR* counts for *hmmer*, marked by a star in the table. The average length of a parallel task is as small as 1 millisecond and as large as half second.

Not all code is included in the source form. The test *bzip2* is a block-sorting data compressor. Through binary instrumentation we found that up to 25% of executed instructions are inside the `glibc` library. We use full address space protection for correctness.

OpenMP parallelization For comparison, we test the OpenMP version for half of the tests. The OpenMP code of *Art* comes from SPEC OMP 2001. A "diff" between SPEC 2K and SPEC OMP 2001 shows 707 lines of difference. Much of the code changes in the OpenMP version are due to new data structures needed to implement reduction. The two clustering tests have a regular loop structure and are easy to parallelize using OpenMP. We also created an OpenMP version for *hmmer*. OpenMP cannot parallelize string substitution, which has input-dependent parallelism. The remaining three programs would require significant changes to the source code to use OpenMP (e.g. some loops have early exits), which we did not endeavor to perform.

5.2 BOP Performance

Figure 5 shows an overview of *BOP* performance. On average, *BOP* parallelization reduces the end-to-end, wall-clock program run time by a factor of 2.1 on 2 processors, 3.6 on 4 processors, 6.1 on 8 processors, and 9.1 on 15 processors. We next discuss the programs, the cost of dependence hints, and the comparison with OpenMP.

String substitution The test program finds and replaces a pattern in 558MB of text, with a block size of 2MB per *PPR* and a total of 279 *PPR*s. We test the program with 5 different levels of conflicts: no conflict, 1%, 5%, 10%, and 50% conflicts. The sequential run time ranges from 4.4 seconds with no conflict to 4.7 seconds with 50% conflicts.

The ordering hint is used to measure the total number of substitutions. When there is no match, the program is sped up by 1.9 to 5.1 times with 2 to 8 processors, as shown in Figure 6. The execution time is reduced from 4.4 seconds to 0.9 second. The improvement decreases when there are conflicts, as the four other curves show. The maximal speedup drops to 4.6 for 1% (3) conflicts, to 2.4 for 5% (14) conflicts, and to 1.6 for 10% (28) conflicts. With 50% conflicts, every other *PPR* fails and requires a rollback. The program runs 7% to 14% slower.

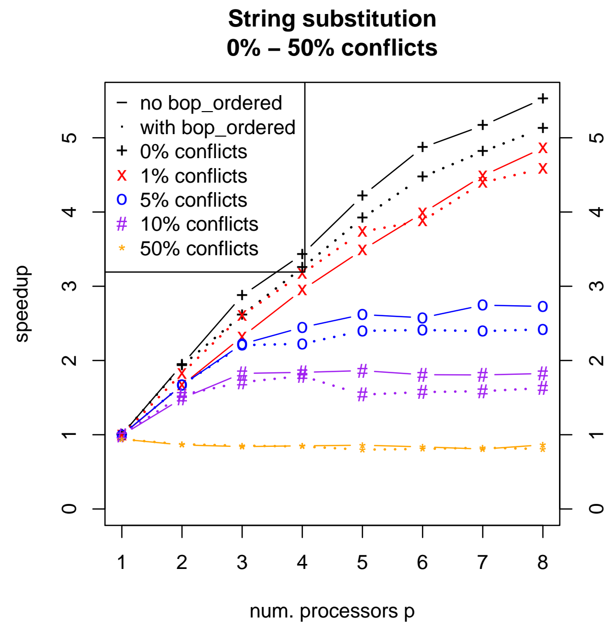


Figure 6: The speedup of parallel string substitution with and without the ordering hint.

K-means The program has regular loop parallelism and dependence inside each clustering step when it updates the centroids data. The ordering hint is used with the additional benefit of maintaining the same numerical precision for the centroid coordinates as in the sequential execution. For this test, the program divides 8 million points in a 20-dimensional space into 10 clusters in 10 steps.

BOP protects 15627 pages or 64MB data. It uses dependence hints at page granularity to serialize the centroid data updates. During commit and post-wait, it copies and transfers 168740 pages or 691MB data. As shown in Figure 5 and in more detail later in Figure 9, the performance increases linearly to a factor of 4 with five processors and then slowly by anther 20% to 15 processors.

Bzip2 We parallelized two versions of *bzip2*, . The original code is revision 0.1p12, dated 1997. In this version, post-wait calls are inserted in two separate functions in the code.

This cannot be done with any of the ordered hint. A newer version, 1.0.3, is included in SPEC 2K, which we use to report performance. In this version, the serialized code is placed in a single function and bracketed by bop_ordered.

The input to *bzip2* is a 300MB file (Intel Fortran compiler). The speedups are 1.8 by 2 tasks, 2.8 by 4, 4.1 by 8, and 5.3 by 15 tasks. The compression time was reduced from 85 seconds to 18 seconds, a significant reduction. Compared to others, however, the speedup is low. The reason is the large amount of disk reads. *BOP* runs a pool of processes, each of which has to read the file separately. The total amount of file reads multiply. 15 tasks would read 15 times 300MB (4.5GB). The problem may be solved by using copy-on-write in the file system, as advocated by the Determinator OS [3].

Art and Hmmer The program *art* is used to train a neural network for image recognition using the adaptive resonance theory. The training process is parallelized, and the results are combined using an ordering hint. As shown in Figure 5 and in more detail later in Figure 10, the performance of *art* increases linearly but at two different rates: a faster rate from 2 to 8 processors and then a slower rate from 8 to 15. The speedup is 6.2 at 8 processors and 7.4 at 15 processors.

For *hmmer*, we test the smallest and the largest granularity. The smallest is 85 thousand *PPR*s in 93 seconds or on average 1 millisecond per *PPR*. To support tasks at this granularity, we use the byte-granularity interface instead of page-based monitoring. The dependence hint is too costly so we move the serial code out of the parallel region. On the other extreme, we divide the loop into p *PPR*s when running with p tasks and use the dependence hint. *Hmmer* has near perfect scaling with 14 times performance on 15 processors.

QT-clustering Quality threshold (QT) clustering is an iterative algorithm proposed in 1999 for grouping related genes [15]. In each step, the algorithm finds the cluster around every node (unclustered neighbors within a distance) and picks the largest cluster. The next step repeats the process for the remaining nodes until all nodes are clustered. QT clustering overcomes two shortcomings of k-means clustering. It needs no a priori knowledge of the number of clusters. The result is deterministic and does not depend on the initial choices of cluster centroids. The drawback is that QT clustering is more computationally intensive.

We implemented the QT algorithm in C and tested both intra-time step parallelism (shown in Figure 5) and time skewing (see Section 5.5). There is sufficient parallelism, 400 *PPR*s, in each time step. *BOP* shows highly scalable performance, with speedups of 2.0, 3.8, 7.0, and 14 times for 2, 4, 8, and 15 parallel tasks, as shown in Figure 5.

Sjeng and parser *Sjeng* is a computer-chess program. The input is based on a chess-board file in the set of reference inputs provided as part of the SPEC 2006 benchmark package. We increased the number of tasks in the input. The improvement is a factor of 2.0 by 2 tasks, 3.8 by 4 tasks, 7.4 by 8 tasks, and 13 by 15 tasks. *Parser* obtains a speedup of 2.0 by 2 tasks, 3.8 by 4, 7.0 by 8, and 10.6 by 15 tasks. The two programs have *do-all* loops. They do not need dependence hints but show the benefits of the current *BOP* design especially process reuse. The average task size in *parser* is 0.02 second, which cannot be parallelized efficiently if a process is created for every task.

5.3 The Dependence-hint Overhead

K-means The dependence hint requires serialization of *PPR* tasks. It is implemented using a series of locks, one between each pair of tasks. To quantify the serialization cost, we measure the difference between the time when the last process in a group ends and the time when the whole group finishes. We call the time difference the *serial pause*. We collect the data from the test runs of *k-means*. We show in Figure 7 the min, mean, and max serial pauses for two versions: *BOP* and *skim* BOP. *BOP* communicates data, a total of 690MB, but *skim* BOP does not (and is incorrect as a result). The figure shows the mean pause using a solid bar, and the range from the min to the max pause using a vertical line.

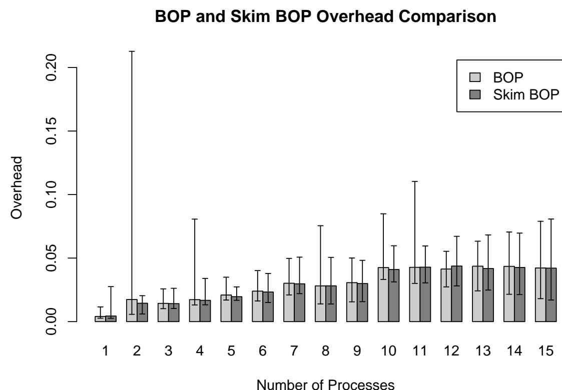


Figure 7: The time takes to serialize *BOP* parallel tasks, with and without data copying.

Without data copying, the average serial pause increases from 0.004 second when there is no parallel execution to 0.04 seconds when 15 tasks are running. With data copying, the average cost is within 3% of that without data copying, showing that data communication has a negligible effect on the length of a serial pause. The cost in a sequential run comes mostly from re-locking the memory pages accessed during the preceding task. The cost in parallel runs is due entirely to synchronization.

On average, the serialization takes up to 0.02 seconds for up to 6 tasks, 0.03 seconds for 7 to 9 task, and 0.04 seconds for 10 to 15 tasks. The pause time dictates the minimal task granularity when dependence hints are used. If a *PPR* takes half a second or more, the serialization overhead will be insignificant.

String substitution The *k-means* analysis does not consider the effect of process reuse, which can hide the cost of serialization. In string substitution, the average *PPR* size is 0.016 second, but *BOP* shows scalable improvements up to 8 tasks. To quantify the cost of the ordering hint, we removed it and measured the performance, which is shown in Figure 6. Without the ordering hint, the program runs 3.5% faster on average overall and 6-7% faster on average for tests with 0%, 5%, and 10% conflicts.

5.4 Comparison with Original BOP

The original *BOP* does not support dependence hints. As a result, it can parallelize 2 of the 8 programs (it can run the other programs correctly but not in parallel). In addition, there are significant differences in the implementation. The original *BOP* creates a process for each *PPR* and uses only OS page-protection for access monitoring. It was implemented for only the 32-bit address space. The current version has process reuse and byte-granularity interface for access monitoring and supports the 64-bit address space.

Next we evaluate the overhead of page protection in two tests. They are short programs. We compare the automatic page-granularity monitoring with manually inserted byte-granularity monitoring.

String substitution In this test, the average length of a *PPR* task is 0.016 second. The cost of page protection is significant. Figure 8 shows that byte-granularity monitoring is 30% to 50% faster than page-granularity monitoring. For easy viewing, the graph does not show 1% and 10% conflict curves. Byte-granularity monitoring with 1% conflicts, shown previously in Figure 6, is about 25% faster than page-granularity monitoring with no conflict.

K-means In this test, the average length of a *PPR* task is 0.55 second. The cost of page protection is negligible. We have tested a version, *skim* BOP, which does not protect data and does not use dependence hints. It is incorrect parallelization. We use it for performance evaluation since skim *BOP* is free of most of the speculation overheads. Because of coarse granularity, we found that the overheads from data monitoring, commit, and post-wait costed no more than 2% of the overall performance.

5.5 Comparison with OpenMP

K-means We test two OpenMP versions. *OpenMP ordered* uses an ordered region to perform the reduction on the centroid data. *OpenMP atomic* uses a critical section instead. Figure 9 shows their difference, 1% to 4% at $p = 10$–15 except -2% at $p = 12$ and 11% at $p = 14$. There is a slight performance benefit from out of order access to shared data.

For *BOP*, however, its large arrays and large amounts of data writes make it challenging for copy-on-write to obtain good performance, especially in comparison with OpenMP which modifies data in place. In Figure 9, safe parallelization by *BOP* and unsafe parallelization by *OpenMP ordered* have less than 3% difference when running with 5 or fewer processors. Then OpenMP is faster on average by 7% for $p = 7$–12. *BOP* becomes faster by 1% to 2% for $p = 13$–15. The two have the same peak speedup of 4.7X.

Art and Hmmer Figure 10 shows that *BOP* has a similar performance as OpenMP. In *art*, as the number of tasks is increased from 1 to 15, the performance of both the *BOP* and the OpenMP versions increases almost identically to a factor

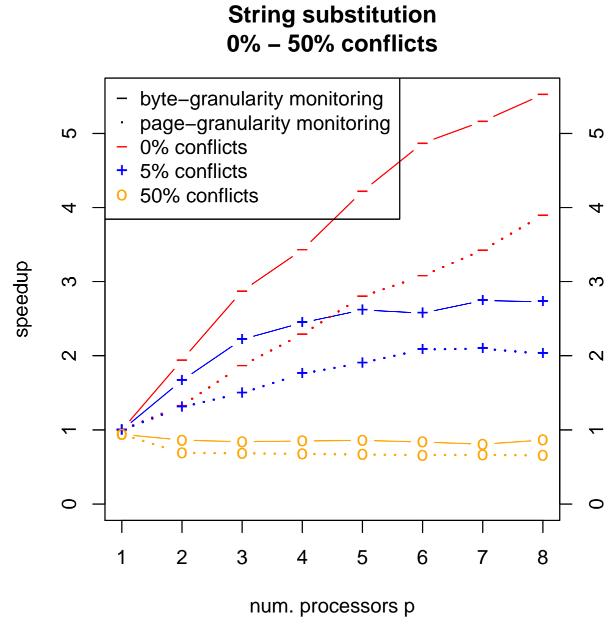


Figure 8: String substitution (without the ordering hint) with and without the page protection overhead.

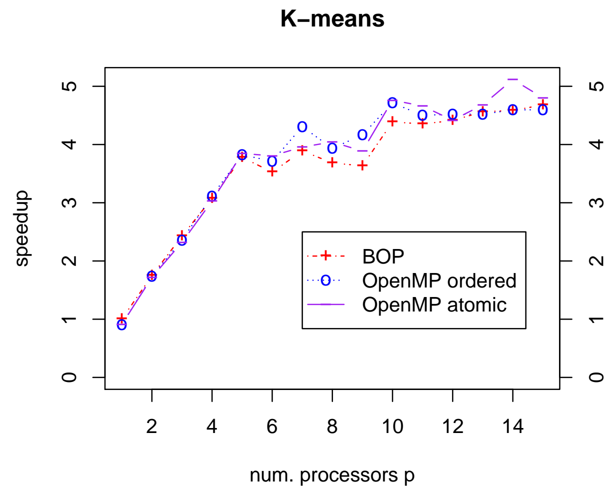


Figure 9: Comparison of *BOP* with OpenMP using ordered and critical sections.

of over 7. The execution time is reduced from 1,211 seconds to 164 seconds by *BOP* and 172 seconds by OpenMP.

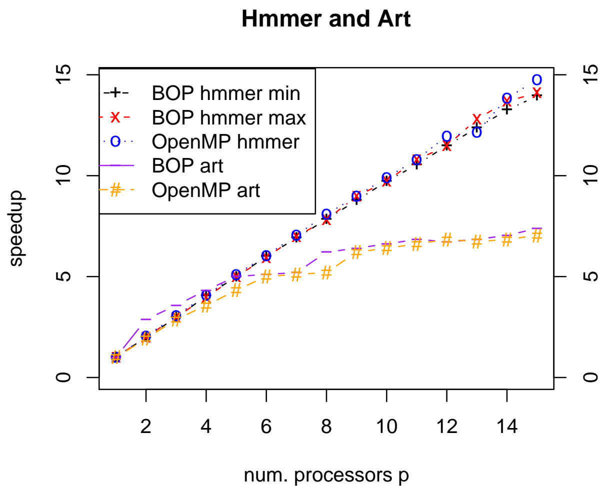


Figure 10: *BOP* reduces *hmmer* time from 93 seconds to 6.7 seconds and *art* time from 1,211 seconds to 164 seconds. Both improvements are similar to OpenMP.

The figure shows three versions of *hmmer*: coarse-grained *BOP* with dependence hints, fine-grained *BOP* without dependence hints, and fine-grained OpenMP without the critical section (moved out of the compute loop). All three versions show almost identical linear speedups. The running time is reduced from 93 seconds to 6.7 seconds by both *BOP* versions and 6.3 seconds by OpenMP.

QT-clustering The OpenMP version parallelizes within each time step, using a critical section to combine results and an implicit barrier to separate time steps. When there is sufficient parallelism, 400 *PPR*s, in each time step, both *BOP* and OpenMP obtain a highly scalable performance. The speedup by *BOP* is 2.0, 3.8, 7.0, and 14 times for 2, 4, 8, and 15 parallel tasks. The speedup by OpenMP is 2.0, 4.0, 7.9, and 15.6. When there is an insufficient number of *PPR* tasks in time steps, time skewing can help. We set the inner loop to have 15 iterations and run it on a machine with 8 processors. In OpenMP, the 15 iterations are divided into 3 groups when $p = 5, 6, 7$. As a result, OpenMP has less than 5% improvement from $p = 5$ to $p = 7$. With time skewing, however, different time steps may overlap, so time skewing is 7% and 18% faster when using 6 and 7 parallel tasks than OpenMP.

6. Related Work

Dependence in speculative parallelization Software speculative loop parallelization was pioneered by Rawchwerger and Padua in the LRPD test [24]. Java safe future and *BOP PPR* provided an interface for expressing possible parallelism but not dependence [12, 33]. In hardware thread-level speculation (TLS), a dependence can be specified using signal-wait [35]. Like Cytron's post-wait [10], signal and wait are paired by an identifier, which is usually a data address. Another construct is flow in an *ordered transaction*. It specifies that a variable read should wait until a new value is produced by the previous transaction [32]. In these systems, memory is shared, so a construct can implicitly synchronize dependences on other data as well. The correctness in guaranteed by the user or special hardware support.

The flow construct is a data trigger and useful when a read needs to wait for an unidentified write in the predecessor task [32]. A problem may arise if the previous task does not write or writes multiple times. As shown in Section 3.4, *BOP* provides a more programmable solution. SMTX is an interface for software multi-threaded transactions. It provides primitives to accessed versioned data [23]. The read and write accesses specify a variable name and are matched by the name and version number. The channel identifier in *BOP* can serve the purpose of a version number. *BOP* channels use one-sided addressing (to allow dynamic changes to the channel content) and can communicate aggregate and dynamically-allocated data, which would require additional annotations if using flow or versioned access.

The Galois system lets a user express parallelism at an abstract level of optimistic iterators instead of the level of reads and writes [22]. Dependence hints also address the problem in complex code where access tracking is difficult to specify, but with a different solution which is to mark larger units of data and to combine with speculation. Optimistic iterators in Galois can specify semantic commutativity and inverse methods, which permits speculative parallelism beyond the limit of dependence hints. However, these extensions are not hints and must be used correctly.

An increasing number of software systems use copy-on-write data replication to implement speculation [12, 13, 23, 30] or race-free threaded execution [7, 8, 31]. Tasks do not physically share written memory and must exchange dependent data explicitly. A solution, transactional communicator, addresses the problem in transactional memory [19]. *BOP* hints provide a solution for parallelization and use channels supported by sender-side addressing, selective dependence marking, and channel chaining to reduce the programming effort and implementation cost.

Dependence in dynamic parallelization Instead of enumerating dependences, the Jade language uses data specification to derive dependence automatically [25]. Jade identifies all dependences without having to specify any of them. The dependence hint provide a different solution through partial dependence specification. Being speculative, dependence hints enforce all dependences without having to specify all of them. In comparison, Jade specifications are not hints and may lead to program error if used incorrectly. The two approaches are fundamentally different. Jade is aimed for automatically optimized parallelization. Depen-

dence hints focus on parallelization with incomplete program knowledge but with direct control.

In many programs especially irregular scientific code, dependences can be analyzed through the inspector-executor approach. Recent advances include leveraging the OpenMP interface [4] and utilizing powerful static tools such as the use of uninterpreted function symbols in an integer-set solver [28]. The combination of static techniques such pointer and inter-procedural analysis with run-time dependence counting has also enabled dynamic parallelization of Java programs by OoOJava [16].

Fork-join parallel languages Fork-join primitives, including spawn/sync in Cilk [14], future/get in Java, and async/finish in X10 [9], come in pairs: one for fork and one for join. *PPR* has fork but no explicit join. bop_ppr uses speculation to abandon a task if it does not finish in time, thus providing a safe join. It is useful when the join point of a task is unknown, unpredictable, or too complex to specify. In fact, not relying on user is a requirement for full safety. One use of the dependence hint is in suggesting a task join.

Parallel languages provide primitives for synchronization such as critical (atomic) section or transactions. New languages such as Fortress have primitives for reduction [1]. These primitives maximize parallelism in dependent operations and may be necessary for highly scalable performance. *BOP* cannot support these constructs because it cannot automatically guarantee sequential semantics for them. The ordering hint loses parallelism, but *BOP* recoups the loss by speculating on the later *PPR*s while waiting. Ordered operations are often useful. For example in *bzip2*, it ensures that the compressed data is generated in order, which cannot be done by a critical section or a transaction.

7. Summary

Copy-on-write data replication is increasingly used in safe program parallelization to isolate parallel tasks and eliminate their false dependences. In this paper, we have presented the dependence hint, an interface for a user to express partial parallelism so copy-on-write tasks can speculatively communicate and synchronize with each other.

We have presented single-post channel, sender-side addressing and selective dependence marking to simplify programming; channel chaining to express conditional dependences; the three correctness checks for safe implementation; and a set of techniques including filtered posting, silent drop, reordered receive, inter-*PPR* and distant-*PPR* hint overrides to enhance parallelism and reduce communication. We show that dependence hints can be used to build high-level constructs such as ordering and pipelining hints and make them safe and safely composable.

We have shown example uses of dependence hints and demonstrated their expressiveness and safety. In evaluation, we found that despite of the cost of data copying, access monitoring and task serialization, the safe parallelization by *BOP* achieves on average 9.1 times performance improvement for 8 programs on today's multicore computers.

Acknowledgments

The idea of process reuse was originated in a parallelization system developed by Long Chen and others at ICT. Long Chen also provided the initial code for several of the SPEC benchmark tests. The initial implementation of *BOP*-malloc was assisted by Jingliang Zhang. Yang Chen helped result processing. Xiaoming Gu helped with the implementation of an earlier version of the system. Finally, we wish to thank Michael Scott, Xipeng Shen, Kai Shen, Peng Wu, Christopher Hill, Ron Rostan, Mark Hapner, and others for the helpful discussions. The research is supported by NSF (Contract No. CCF-1116104, CCF-0963759, CNS-0834566, CNS-0720796), and IBM CAS Faculty Fellowships.

References

[1] E. Allen, D. Chase, C. Flood, V. Luchangco, J. Maessen, S. Ryu, and G. L. Steele. Project fortress: a multicore language for multicore processors. *Linux Magazine*, pages 38–43, September 2007.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, Oct. 2001.

[3] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2010.

[4] A. Basumallik and R. Eigenmann. Optimizing irregular shared-memory applications for distributed-memory systems. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–128, 2006.

[5] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 133–144, Barcelona, Spain, 2004.

[6] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–64, 2010.

[7] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *Proceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications*, 2009.

[8] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *Proceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications*, pages 691–707, 2010.

[9] 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 *Pro-*

ceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications, pages 519–538, 2005.

[10] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–844, St. Charles, IL, Aug. 1986.

[11] C. Ding. Access annotation for safe speculative parallelization: Semantics and support. Technical Report URCS #966, Department of Computer Science, University of Rochester, March 2011.

[12] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 223–234, 2007.

[13] M. Feng, R. Gupta, and Y. Hu. SpiceC: scalable parallelism via implicit copying and explicit commit. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 69–80, 2011.

[14] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[15] L. Heyer, S. Kruglyak, and S. Yooseph. Exploring expression data: Identification and analysis of coexpressed genes. *Genome Research*, 9:1106–1115, 1999.

[16] J. C. Jenista, Y. H. Eom, and B. Demsky. OoOJava: Software out-of-order execution. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 57–68, 2011.

[17] Y. Jiang and X. Shen. Adaptive software speculation for enhancing the cost-efficiency of behavior-oriented parallelization. In *Proceedings of the International Conference on Parallel Processing*, pages 270–278, 2008.

[18] L. Liu and Z. Li. Improving parallelism and locality with asynchronous algorithms. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 213–222, 2010.

[19] V. Luchangco and V. J. Marathe. Transaction communicators: enabling cooperation among concurrent transactions. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 169–178, 2011.

[20] J. M. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing*, pages 24–33, 1991.

[21] OpenMP application program interface, version 3.0, May 2008. http://www.openmp.org/mp-documents/spec30.pdf.

[22] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 12–25, 2011.

[23] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 65–76, 2010.

[24] L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[25] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, 1998.

[26] J. A. Roback and G. R. Andrews. Gossamer: A lightweight approach to using multicore machines. In *Proceedings of the International Conference on Parallel Processing*, pages 30–39, Washington, DC, USA, 2010. IEEE Computer Society.

[27] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 215–228, Atlanta, Georgia, May 1999.

[28] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–257, San Diego, CA, June 2003.

[29] W. Thies, V. Chandrasekhar, and S. P. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, pages 356–369, 2007.

[30] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or Discard execution model for speculative parallelization on multicores. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, pages 330–341, 2008.

[31] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: parallelizing sequential logging and replay. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 15–26, 2011.

[32] C. von Praun, L. Ceze, and C. Cascaval. Implicit parallelism with ordered transactions. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mar. 2007.

[33] A. Welc, S. Jagannathan, and A. L. Hosking. Safe futures for Java. In *Proceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications*, pages 439–453, 2005.

[34] D. Wonnacott. Achieving scalable locality with time skewing. *International Journal of Parallel Programming*, 30(3), June 2002.

[35] A. Zhai, J. G. Steffan, C. B. Colohan, and T. C. Mowry. Compiler and hardware support for reducing the synchronization of speculative threads. *ACM Transactions on Architecture and Code Optimization*, 5(1):1–33, 2008.

[36] C. Zhang, C. Ding, X. Gu, K. Kelsey, T. Bai, and X. F. 0002. Continuous speculative program parallelization in software. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 335–336, 2010. *poster paper*.