# Distributed Speculative Program Parallelization

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

## Abstract

Most computing users today have access to clusters of multi-core computers. To fully utilize a cluster, a programmer must combine two levels of parallelism: shared-memory parallelism within a machine and distributed memory parallelism across machines. Such programming is difficult. Either a user has to mix two programming languages in a single program and use fixed computation and data partitioning between the two, or the user has to rewrite a program from scratch. Even after careful programming, a program may still have hidden concurrency bugs. Users who are accustomed to sequential programming do not find the same level of debugging and performance analysis support especially for a distributed environment.

In this paper we present a distributed parallelization system named *D-BOP* , which enables software speculative parallelization to cross multiple machines. *D-BOP* provides the same guarantee of preserving the outputs of the original sequential program as some of the shared memory speculative parallelization systems do. The execution of a *D-BOP* parallelized program is data race free, deadlock free and deterministic.

## 1. Introduction: From *BOP* To *D-BOP*

Computer users are increasingly interested in parallel programming because they want to utilize clusters of multi-core processors, which is capable, in theory, of performance tens or hundreds of times of a single personal computer. Although compiler techniques are effective in exploiting loop parallelism in scientific code written in Fortran they are not a sufficient solution for C/C++ programs where both the degree and the granularity of parallelism are not guaranteed or even predictable.

Manual parallel programming is becoming easier. There are ready-to-use parallel constructs in mainstream languages such as Fortran, C, C++ and Java, in threading libraries such as Microsoft .NET, Intel Thread Building Blocks, in domain-specific languages such as Google's Map Reduce and Intel Concurrent Collection for C++. Still, writing parallel code is considerably harder than sequential programming because of non-determinism. A program may run fine in one execution but generate incorrect results or run into a deadlock in the next execution because of a different thread interleaving. It may acquire new data races when ported to a machine with a different hardware memory consistency model. In addition, important types of computations such as mesh refinement, clustering, image segmentation, and SAT approximation cannot be parallelized without speculation since conflicts are not known until the computation finishes [7, 8].

In our PLDI07 paper [4] we present *BOP*, a speculative parallelization system that enables usual programmers with limited parallel programming experiences to transform sequential programs into parallel. *BOP*'s goal is to free programmers from heavy duties of code rewriting and debugging in demand of parallelization while at same time to obtain significant coarse-grained parallelisms at run time.

*BOP* is both a parallelization language and a runtime system. The *BOP* language is essentially comprised of three types of primitives: *parallelism hints*, which mark the possibly parallel regions, *dependence hints*, which express possible dependencies between parallel tasks, and *data-checking hints*, which verify "private" data by value-based checking.

The *BOP* runtime is a process-based speculation management system where speculation is realized by forking a new process and run into the "future". Data accesses are monitored on page granularity so that we can leverage OS-supported page protection mechanism to avoid code instrumentation. Modern OS performs copy-on-write and replicates pages on demand. In *BOP*, such replication removes false dependence conflicts and makes error recovery trivial to do. We can simply discard an erroneous speculative task by terminating the process.

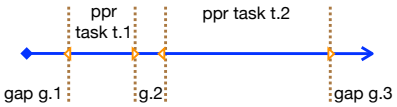
Next we extend the process-based design to enable program parallelization in a distributed environment.

## 2. Distributed Speculation

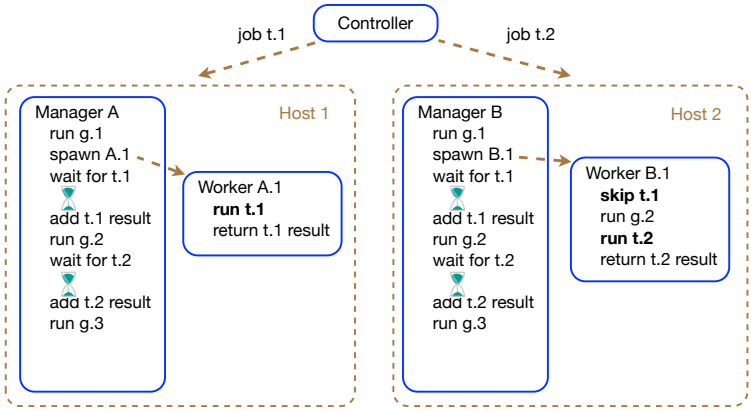
The *D-BOP* run-time system has three types of processes.

- **A control process** manages shared system information and serves as the central point of communication. There is one control task in each *D-BOP* execution. We refer to it as *the controller*.
- **A host management process** manages parallel execution within a host and coordinates other hosts through the control process. There is one management process on each host. We refer to it as *a (host) manager*.
- **A worker process** runs one or a series of *PPR* tasks on a single processor. Work processes are dynamically created and terminated and may perform computation speculatively. We refer to it as *a worker* or *a risky worker* in the sense that the job is speculative.

The first two types are common in a distributed system such as the one used by MPI and Erlang. The unique problems in *D-BOP* design are how to support speculation including the ability to run a *PPR* task on an available machine, checking correctness and maintaining progress in a distributed environment. To do so, the three types




(a) A program execution consists of two PPR tasks, t.1 and t.2, separated by gaps g.1, g.2, and g.3.




(b) Distributed parallel execution of t.1 on Host 1 and t.2 on Host 2. Both hosts run gaps g.1, g.2 and g.3.

**Figure 1.** An example execution by *BOP*. The input program is a series of *PPR* tasks separated by gaps. The controller distributes *PPR* tasks to hosts. A host manager forks a worker process to execute a *PPR* task. Inter-*PPR* gaps are executed by the control and every manager.

of processes divide the work and coordinate with each other as follows.

- The control process distributes a group of *PPR* instances (or jobs) to the management process of each host.
- On a host, the management process starts one or more work processes based on the resources available on the host.
- After finishing its assigned *PPR* instances, a work process reports its results for verification.
- If a finished *PPR* instance has a conflict, the control process redistributes the *PPR* instance and its successors for re-execution. If all finished *PPR* instances are verified correct, the control process continues distributing subsequent *PPR* instances until the program finishes.

Figure 1 shows the distributed execution of an example program on two hosts. The program has four *PPR* tasks separated by gaps. The controller distributes *PPR* tasks to hosts. A host manager forks a worker process to execute a *PPR* task. Inter-*PPR* gaps are executed by the control and every manager.

To start a worker, *BOP* needs to create a process and initialize it to have the right starting state. There are two basic ways of creating an existing state: one is copying from the existing state, the other is re-computing it from the beginning. On the same host, copying can be easily done using Unix `fork`. Across hosts, we use a hybrid scheme we call *skeleton re-execution*. Each host manager executes all inter-*PPR* gaps, which is the "skeleton." When it reaches a *PPR* task, the manager waits for its successful completion (by some worker), copies the data changes, and skips to continue the skeleton execution at the next inter-*PPR* gap. With skeleton re-execution, a manager maintains a local copy of program state and use it to start worker tasks through `fork`.

An alternative to re-execution is to use remote checkpointing, for example, to use a system like Condor to implement a remote `fork`. Simple checkpointing would transfer the entire program state, which is unnecessary. Incremental checkpointing may alleviate the problem. Checkpointing is a more general solution than re-execution because it handles code that cannot be re-executed. In our current prototype, we allow only CPU, memory, and basic file I/O operations, where re-execution can be supported at the application level. Checkpointing support can eliminate these restrictions.

### 2.1 Correctness Checking and Data Update

There are two basic problems in speculation support: correctness checking and data update. Each problem has two basic solution choices.

- *Correctness checking* can be done by *centralized validation*, where checking is centralized in one process (the controller), or *distributed validation*, where checking work is divided among hosts.
- *Data update* can be done by *eager update*, where changes made by one host are copied to all other hosts, or *lazy update*, where changes are communicated only when they are needed by a remote host.

The problem of data updates is similar to the ones in software distributed shared memory (S-DSM), while the problem of correctness checking is unique to a speculative parallelization system. The above choices can be combined to produce four basic designs.

- *centralized validation, eager update*. This design is similar to shared-memory speculation. With eager update, a worker incurs no network-related delays because all its data is available on the local host. However, the controller may become a bottleneck, since it must send all data updates to all hosts.
- *distributed validation, eager update*. With distributed validation, each host sends its data updates to all other hosts, which avoids the bottleneck in centralized validation but still retains the benefit of eager update. However, correctness checking is repeated on all hosts (in parallel), which increases the total checking and communication cost due to speculation.

- *distributed validation, lazy update*. Lazy update incurs less network traffic because it transfers data to a host only if it is requested by one of its workers. The reduction of traffic comes at an increase of latency. Instead of accessing data locally, a worker must wait for data to be fetched from the network, although the latency may be tolerated by creating more workers to maintain full progress when some workers wait for data. With distributed validation, the global memory state is distributed instead of centralized and replicated.
- *centralized validation, lazy update*. This scheme maintains a centralized global state. As the sole server of data requests, the controller should inevitably become a bottleneck as the number of lazy-update workers increases. This combination is likely not a competitive option.

There are hybrids among these basic combinations. For example, we may combine centralized and distributed validation by checking correctness in the controller but sending data updates from each host, in a way similar to the "migrate-on-read-miss" protocol in a distributed, cache coherent system [3]. We may also divide program data and use eager update in one set and lazy update in another.

## 3. Related Work

***Software speculative parallelization*** LRPD test provides loop-level software speculation for Fortran programs [14]. The support has been extended to C programs using annotations [2], compiler support [16, 18], and techniques developed for transactional memory [10, 11, 15]. Speculation support has been developed for Java to support safe future [17], return-value speculation [12], and speculative data abstractions [9].

Two recent systems [1, 13] use process-based approach to support multithreaded programs where processes are leveraged to enable strong isolation and to automate write buffer construction.

***Process-based Speculation*** Process-based approach has been used for speculative parallelization [1, 4, 5, 13]. It is used to implement pipeline stages in multi-threaded software transactional memory [13]. Similar controls can be imitated for threads [16]. It also supports speculative program optimization and its use in parallel program profiling and memory-access checking [6].

## 4. Summary

We have presented the design of the *D-BOP* distributed speculative parallelization system. With full support of automatic task management, distributed communication, speculation state maintanance and correctness checking, parallelization becomes so easy that it only requires moderate effort from the programmers by just putting suggestions in their sequential programs. Until recently a user faced with the task of extracting coarse-grained parallelism in complex code had to choose either a compiler to automate the task or a language extension to parallelize a program completely by hand. *D-BOP* provides a middle ground where parallelization is achieved far beyond the capabilities of a compiler while only simple code annotations are required.

## References


[1] 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.

[2] M. H. Cintra and D. R. Llanos. Design space exploration of a software speculative parallelization scheme. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):562–576, 2005.

[3] A. L. Cox and R. J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the International Symposium on Computer Architecture*, pages 98–108, 1993.

[4] 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.

[5] 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.

[6] K. Kelsey, T. Bai, and C. Ding. Fast track: a software system for speculative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 157–168, 2009.

[7] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 233–243, 2008.

[8] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 211–222, 2007.

[9] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *Communications of ACM*, 52(9):89–97, 2009.

[10] M. Mehrara, J. Hao, P.-C. Hsu, and S. A. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 166–176, 2009.

[11] C. E. Oancea, A. Mycroft, and T. Harris. A lightweight in-place implementation for software thread-level speculation. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 223–232, 2009.

[12] C. J. F. Pickett and C. Verbrugge. Software thread level speculation for the Java language and virtual machine environment. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, pages 304–318, 2005.

[13] 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.

[14] 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.

[15] M. F. Spear, K. Kelsey, T. Bai, L. Dalessandro, M. L. Scott, C. Ding, and P. Wu. Fastpath speculative parallelization. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 2009.

[16] 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.

[17] 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.

[18] A. Zhai, S. Wang, P.-C. Yew, and G. He. Compiler optimizations for parallelizing general-purpose applications under thread-level speculation. pages 271–272, New York, NY, USA, 2008. ACM.