The Parallel Persistent Memory Model

The Parallel Persistent Memory Model

Guy E. Blelloch* Phillip B. Gibbons* Yan Gu* Charles McGuffey* Julian Shun†

*Carnegie Mellon University †MIT CSAIL

ABSTRACT

We consider a parallel computational model, the *Parallel Persistent Memory model*, comprised of P processors, each with a fast local ephemeral memory of limited size, and sharing a large persistent memory. The model allows for each processor to fault at any time (with bounded probability), and possibly restart. When a processor faults, all of its state and local ephemeral memory is lost, but the persistent memory remains. This model is motivated by upcoming non-volatile memories that are nearly as fast as existing random access memory, are accessible at the granularity of cache lines, and have the capability of surviving power outages. It is further motivated by the observation that in large parallel systems, failure of processors and their caches is not unusual.

We present several results for the model, using an approach that breaks a computation into *capsules*, each of which can be safely run multiple times. For the single-processor version we describe how to simulate any program in the RAM, the external memory model, or the ideal cache model with an expected constant factor overhead. For the multiprocessor version we describe how to efficiently implement a work-stealing scheduler within the model such that it handles both soft faults, with a processor restarting, and hard faults, with a processor permanently failing. For any multithreaded fork-join computation that is race free, write-after-read conflict free and has W work, D depth, and C maximum capsule work in the absence of faults, the scheduler guarantees a time bound on the model of $O\left(\frac{W}{P_A} + \frac{DP}{P_A}\left\lceil \log_{1/(Cf)} W \right\rceil\right)$ in expectation, where P is the maximum number of processors, P_A is the average number, and $f \leq 1/(2C)$ is the probability a processor faults between successive persistent memory accesses. Within the model, and using the proposed methods, we develop efficient algorithms for parallel prefix sums, merging, sorting, and matrix multiply.

ACM Reference Format:
Guy E. Blelloch* Phillip B. Gibbons* Yan Gu* Charles McGuffey* Julian Shun† . 2018. The Parallel Persistent Memory Model. In *SPAA '18: 30th ACM Symposium on Parallelism in Algorithms and Architectures, July 16–18, 2018, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3210377.3210381

SPAA '18, July 16–18, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5799-9/18/07…$15.00
https://doi.org/10.1145/3210377.3210381

1 INTRODUCTION

In this paper, we consider a parallel computational model, the *Parallel Persistent Memory (Parallel-PM) model*, that consists of P processors, each with a fast local ephemeral memory of limited size M, and sharing a large slower persistent memory. As in the external memory model [4, 5], each processor runs a standard instruction set from its ephemeral memory and has instructions for transferring blocks of size B to and from the persistent memory. The cost of an algorithm is calculated based on the number of such transfers. A key difference, however, is that the model allows for individual processors to fault at any time. If a processor faults, all of its processor state and local ephemeral memory is lost, but the persistent memory remains. We consider both the case where the processor restarts (soft faults) and the case where it never restarts (hard faults).

The model is motivated by two complimentary trends. Firstly, it is motivated by upcoming non-volatile memories that are nearly as fast as existing random access memory (DRAM), are accessed via loads and stores at the granularity of cache lines, have large capacity (more bits per unit area than existing random access memory), and have the capability of surviving power outages and other failures without losing data (the memory is *non-volatile* or *persistent*). For example, Intel's 3D-Xpoint memory technology, currently available as an SSD, is scheduled to be available as such a random access memory in 2019. While such memories are expected to be the pervasive type of memory [50, 52, 56], each processor will still have a small amount of cache and other fast memory implemented with traditional *volatile* memory technologies (SRAM or DRAM). Secondly, it is motivated by the fact that in current and upcoming large parallel systems the probability that an individual processor faults is not negligible, requiring some form of fault tolerance [17].

In this paper, we first consider a single processor version of the model, the *PM model*, and give conditions under which programs are robust against faults. In particular, we identify that breaking a computation into "capsules" that have no write-after-read conflicts (writing a location that was read earlier within the same capsule) is sufficient, when combined with our approach to restarting faulting capsules from their beginning, due to its idempotent behavior. We then show that RAM algorithms, external memory algorithms, and cache-oblivious algorithms [31] can all be implemented asymptotically efficiently on the model. This involves a simulation that breaks the computations into capsules and buffers writes, which are handled in the next capsule. However, the simulation is likely not practical. We therefore consider a programming methodology in which the algorithm designer can identify capsule boundaries, and ensure that the capsules are free of write-after-read conflicts.

We then consider our multiprocessor counterpart, the Parallel-PM described above, and consider conditions under which programs are correct when the processors are interacting through the shared memory. We identify that if capsules are free of write-after-read

conflicts and atomic, in a way that we define, then each capsule acts as if it ran once despite many possible restarts. Furthermore we identify that a compare-and-swap (CAS) instruction is not safe in the PM model, but that a compare-and-modify (CAM), which does not see its result, is safe.

The most significant result in the paper is a work-stealing scheduler that can be used on the Parallel-PM. Our scheduler is based on the scheduler of Arora, Blumofe, and Plaxton (ABP) [5]. The key challenges in adopting it to handle faults are (i) modifying it so that it only uses CAMs instead of CASs, (ii) ensuring that each stolen task gets executed despite faults, (iii) properly handling hard faults, and (iv) ensuring its efficiency in the presence of soft or hard faults. Without a CAS, and to avoid blocking, handling faults requires that processors help the processor that is part way through a steal. Handling hard faults further requires being able to steal a thread from a processor that was part way through executing the thread.

Based on the scheduler we show that any race-free, write-after-read conflict free multithreaded fork-join program with work W, depth D, and maximum capsule work C will run in expected time:

$$O\left(\frac{W}{P_A} + D\left(\frac{P}{P_A}\right)\left\lceil \log_{1/(Cf)} W \right\rceil\right).$$

Here P is the maximum number of processors, P_A the average number, and $f \leq 1/(2C)$ an upper bound on the probability a processor faults between successive persistent memory accesses. This bound differs from the ABP result only in the $\log_{1/(Cf)} W$ factor on the depth term, due to faults along the critical path.

Finally, we present Parallel-PM algorithms for prefix-sums, merging, sorting, and matrix multiply that satisfy the required conditions. The results for prefix-sums, merging, and sorting are work-optimal, matching lower bounds for the external memory model. Importantly, these algorithms are only slight modifications from known parallel I/O efficient algorithms [15]. The main change is ensuring that they write their partial results to a separate location from where they read them so that they avoid write-after-read conflicts.

Related Work. Because of its importance to future computing, the computer systems community (including companies such as Intel and HP) have been hard at work trying to solve the issues arising when fast nonvolatile memories (such as caches) sit between the processor and a large persistent memory [10, 11, 19–21, 23, 28, 30, 32, 33, 36–39, 44–47, 51, 53, 55]. Standard caches are write-back, meaning that a write to a memory location will make it only as far as the cache, until at some later point the updated cache line gets flushed out to the persistent memory. Thus, when a processor crashes, some writes (those still in the cache) are lost while other writes are not. The above prior work includes schemes for encapsulating updates to persistent memory in either *transactions* or *lock-protected failure atomic sections* and using various forms of (undo, redo, resume) logging to ensure correct recovery. The intermittent computing community works on the related problem of small systems that will crash due to power loss [7, 16, 25, 26, 35, 48, 49, 54]. Lucia and Ransford [48] describe how faults and restarting lead to errors that will not occur in a faultless setting. Several of these works [25, 26, 48, 49, 54] break code into small chunks, referred to as *tasks*, and work to ensure progress at that granularity. Avoiding write-after-read conflicts is often the key step towards ensuring that tasks are idempotent. Because these works target intermittent computing systems, which are designed to be small and energy efficient, they do not consider multithreaded programs, concurrency, or synchronization. In contrast to this flurry of recent systems research, there is relatively little work from the theory/algorithms community aimed at this setting [27, 40, 41, 52]. David et al. [27] presents concurrent data structures (e.g., for skip-lists) that avoid the overheads of logging. Izraelevitz et al. [40, 41] presents efficient techniques for ensuring that the data in persistent memory captures a consistent cut in the happens-before graph of the program's execution, via the explicit use of instructions that flush cache lines to persistent memory (such as Intel's `CLFLUSH` instruction [38]). Nawab et al. [52] defines *periodically persistent* data structures, which combine mechanisms for tracking proper write ordering with a periodic flush of all cache lines to persistent memory. None of this work defines an algorithmic cost model, presents a work-stealing scheduler, or provides the provable bounds in this paper.

There is a very large body of research on models and algorithms where processors and/or memory can fault, but to our knowledge, none of it (other than the works mentioned above) fits the setting we study with its two classes of memory (local volatile and shared nonvolatile). Papers focusing on memory faults (e.g., [1, 22, 29] among a long list of such papers) consider models in which individual memory locations can fault. Papers focusing on processor faults (e.g., [6] among an even longer list of such papers) either do not consider memory faults or assume that all memory is volatile.

Write-back Caches. Note that while the PM models are defined using explicit external read and external write instructions, they are also appropriate for modeling the (write-back) cache setting described above, as follows. Explicit instructions, such as `CLFLUSH`, are used to ensure that an external write indeed writes to the persistent memory. Writes that are intended to be solely in local memory, on the other hand, could end up being evicted from the cache and written back to persistent memory. However, for programs that are race-free and well-formed, as defined in Section 3, our approach preserves its correctness properties.

2 THE PERSISTENT MEMORY MODEL

Single Processor. We assume a two-layer memory model with a small fast *ephemeral memory* of size M (in words) and a large slower *persistent memory* of size $M_p \gg M$. The two memories are partitioned into blocks of B words. Instructions include standard RAM instructions that work on single words within the processor registers (a processor has $O(1)$ registers) and ephemeral memory, as well as two *(external) memory transfer* instructions: an *external read* that transfers a block from persistent memory into ephemeral memory, and an *external write* that transfers a block from ephemeral memory to persistent memory. We assume that the words contain $\Theta(\log M_p)$ bits. These assumptions are effectively the same as in the (M, B) external memory model [2].

We further assume that the processor can *fault* between any two instructions,[1] and that after faulting, the processor *restarts*. On restart, the ephemeral memory and processor registers can be in an arbitrary state, but the persistent memory is in the same state as immediately before the fault. To enable forward progress, we assume

[1] For simplicity, we assume that individual instructions are atomic.

there is a fixed memory location in the persistent memory referred to as the *restart pointer location*, containing a *restart pointer*. On restart, the processor loads the restart pointer from the persistent memory into a register, which we refer to as the *base* register, and then loads the location pointed to by the restart pointer (the *restart instruction*) and jumps to that location, i.e., sets it as the program counter. The processor then proceeds as normal. As it executes, the processor can update the restart pointer to be the current program counter, at the cost of an external write, in order to limit how far the processor will fall back on a fault. We refer to this model as the (single processor) (M, B) persistent memory (PM) model.

The basic model can be parameterized based on the cost of the various instructions. Throughout this paper, and in the spirit of the external memory model [2] and the ideal cache model [31], we assume that external reads and writes take unit cost and all other instructions have no cost.[2] We further assume that the program is constant size and that either the program is loaded from persistent memory into ephemeral memory at restart, or that there is a small cache for the program itself, which is also lost in the case of a fault. Thus, faulting and restarting (loading the base register and jumping to the restart instruction, and fetching the code) takes a constant number of external memory transfers.

The processor's computation can be viewed as partitioned into *capsules*: each capsule corresponds to a maximally contiguous sequence of instructions running on the processor while the restart pointer location contains the same restart pointer. The last instruction of every capsule is therefore a write of a new restart pointer. We refer to writing a new restart pointer as *installing* a capsule. We assume that the next instructions after this write, which are at the start of the next capsule, do exactly the same as a restart does—i.e., load the restart pointer into the base pointer, load the start instruction pointed to by base pointer, and jump to it. The capsule is *active* while its restart pointer is installed. Whenever the processor faults, it will restart using the restart pointer of the active capsule, i.e., the capsule will be restarted as it was the first time. We define the *capsule work* to be the number of external reads and writes in the capsule, assuming no faults. Note that, akin to checkpointing, there is a tension between the desire for high work capsules that amortize the capsule start/restart overheads and the desire for low work capsules that lessen the repeated work on restart.

In our analysis, we consider two ways to count the total cost. We say that the *faultless work* (or *work*), W, is the number of external memory transfers assuming no faults. We say that the *total work* (or *fault-tolerant work*), W_f, is the number of external transfers for an actual run including all transfers due to having to restart. W_f can only be defined with respect to an assumed fault model. In this paper, for analyzing costs, we assume that the probability of faulting by a processor between any two consecutive non-zero cost instructions (i.e., external reads or writes) is bounded by $f \leq 1/2$, and that faults are independent events. We will specify f to ensure that a maximum work capsule fails with at most constant probability.

We assume throughout the paper that instructions are deterministic, i.e., each instruction is a function from the values of registers and memory locations that it reads to the registers and memory locations that it writes.

[2]The results in this paper can be readily extended to a setting (an *Asymmetric PM model*) where external writes are more costly than external reads, as in prior work on algorithms for NVM [8, 9, 12, 13, 18, 42]; for simplicity, we study here the simpler PM model because such asymmetry is not the focus of this paper.

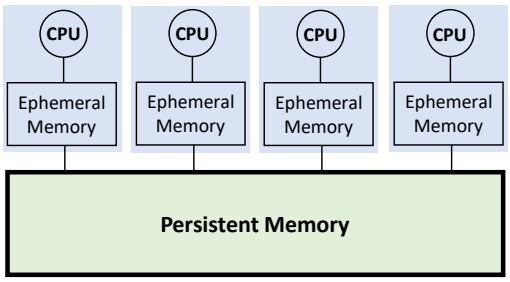

Figure 1: The Parallel Persistent Memory Model

Multiple Processors. The Parallel-PM consists of P processors each with its own fast local ephemeral memory of size M, but sharing a single slower persistent memory of size M_p (see Figure 1). Each processor works as in the single processor PM, and the processors run asynchronously. Any processor can fault between two of its instructions, and each has its own restart pointer location in the persistent memory. When a processor faults, the processor restarts like it would in the single processor case. We refer to this as a *soft fault*. We also allow for a *hard fault*, in which the processor faults and then never restarts—we say that such a processor is *dead*. We assume that other processors can detect when a processor has hard faulted using a liveness oracle `isLive(procId)`. We allow for concurrent reads and writes to the shared persistent memory, and assume that all instructions involving the persistent memory are sequentially consistent.

The Parallel-PM includes a compare-and-swap (CAS) instruction. The CAS takes a pointer to a location of a word in persistent memory and two values in registers. If the first value equals the value at the location, it atomically swaps the value at the location and the value in the second register, and the CAS is *successful*. Otherwise, no swap occurs and the CAS is *unsuccessful*. Even though the persistent memory is organized in blocks, we assume that the CAS is on a single word within a block.

The (faultless) work W and the total work W_f are as defined in the sequential model but summed across all processors. The (faultless) *time* T (and the *fault-tolerant* or *total time* T_f) is the maximum faultless work (total work, respectively) done by any one processor. Without faults, this is effectively the same as the parallel external memory model [4]. In analyzing correctness, we allow for arbitrary delays between any two successive instructions by a processor. However, for our time bounds and our results on work stealing we make similar assumptions as made in [5]. These are described in Section 6.

Multithreaded Computations. Our aim is to support multithreaded dynamic parallelism layered on top of the Parallel-PM. We consider the same form of multithreaded computations as considered by Arora, Blumofe, and Plaxton (ABP) [5]. In the model, a computation starts as a single thread. On each step, a thread can run an instruction, fork a new thread, or join with another thread. Such a computation can be viewed as a DAG, with an edge between instructions, a pair of out-edges at a fork, and a pair of in-edges at a join. As with ABP, we assume that each node in the DAG has out-degree at most two. In the *multithreaded model*, the (faultless) work

W is the work summed across all threads in the absence of faults, and the total work W_f is the summed work including faults. In addition, we define the (faultless) *depth* D (and the *fault-tolerant* or *total depth* D_f) to be the maximum work (total work, respectively) along any path in the DAG. The goal of our work-stealing scheduler (Section 6) is to efficiently map computations in the multithreaded model into the Parallel-PM.

3 ROBUSTNESS ON A SINGLE PROCESSOR

In this section, we discuss how to run programs on the single processor PM model so that they complete the computation properly.

Our goal is to structure the computation and its partitioning into capsules in a way that is sufficient to ensure correctness regardless of faults. Specifically, our goal is that each capsule is a sequence of instructions that will look from an external view like it has been run exactly once after its completion, regardless of the number of times it was partially run due to faults and restarts. We say that a capsule is *idempotent* if, when it completes, regardless of how many times it faults and restarts, all modifications to the persistent memory are consistent with running once from the initial state (i.e., the state of the persistent memory, the ephemeral memory, and the registers at the start of the capsule).

There are various means to guarantee that a capsule is idempotent, and here we consider a natural one. We say that a capsule has a *write-after-read conflict* if the first transfer from a block in persistent memory is a read (called an "exposed" read), and later there is a write to the same block. Avoiding such a conflict is important because if a location in the persistent memory is read and later written, then on restart the capsule would see the new value instead of the old one. We say a capsule is *well-formed* if the first access to each word in the registers or ephemeral memory is a write. Being well-formed means that a capsule will not read the undefined values from registers and ephemeral memory after a fault. We say that a capsule is *write-after-read conflict free* if it is well-formed and had no write-after-read conflicts.

Theorem 3.1. *With a single processor, all write-after-read conflict free capsules are idempotent.*

Proof. On restarting, the capsule cannot read any persistent memory written by previous faults on the capsule, because we restart from the beginning of the capsule and the exposed read locations are disjoint from the write locations. Moreover, the capsule cannot read the state of the ephemeral memory because a write is required before a read (well-formedness). Therefore, the first time a capsule runs and every time a capsule restarts it has the same visible state, and because the processor instructions are deterministic, will repeat exactly the same instructions with the same results. □

An immediate question is whether a standard processing model such as the RAM can be simulated efficiently on the PM model. The following theorem, whose proof is in the full version of the paper [14], shows that the PM can simulate the RAM model with only constant overheads.

Theorem 3.2. *Any RAM computation taking t time can be simulated on the $(O(1), B)$ PM model with $f \leq 1/c$ for some constant $c \geq 2$, using $O(t)$ expected total work, for any B ($B = 1$ is sufficient).*

Although the RAM simulation is linear in the number of instructions, our goal is to create algorithms that require asymptotically fewer reads and writes to persistent memory. We therefore consider efficiently simulating external memory algorithms in the model.

Theorem 3.3. *Any (M, B) external memory computation with t external accesses can be simulated on the $(O(M), B)$ PM model with $f \leq B/(cM)$ for some constant $c \geq 2$, using $O(t)$ expected total work.*

Proof. The simulation consists of rounds each of which has a *simulation* capsule and a *commit* capsule. It maps the ephemeral memory of the source program to part of the ephemeral memory, and the external memory to the persistent memory. It keeps the registers in the ephemeral memory, and keeps space for two copies of the simulated ephemeral memory and the registers in the persistent memory, which it swaps back and forth between.

The simulation capsule simulates some number of steps of the source program. It starts by reading in one of the two copies of the ephemeral memory and registers. Then during the simulation all instructions are applied within their corresponding memories, except for writes from the ephemeral memory to the persistent memory. These writes, instead of being written immediately, are buffered in the ephemeral memory. This means that all reads from the external memory have to first check the buffer. The simulation also maintains a count of the number of reads and writes to the external memory within a capsule. When this count reaches M/B, the simulation "closes" the capsule. The closing is done by writing out the simulated ephemeral memory, the registers, and the write buffer to persistent memory. For ephemeral memory and registers, this is the other copy from the one that is read. The capsule finishes by installing a commit capsule.

The commit capsule reads in the write buffer from the closed capsule to ephemeral memory, and applies all the writes to their appropriate locations of the simulated external memory in the persistent memory. When the commit capsule is done, it installs the next simulation capsule.

This simulation is write-after-read conflict free because the only writes during a simulation capsule are to the copy of ephemeral memory, registers, and write buffer. The write buffer has no conflicts since it is not read, and the ephemeral memory and registers have no conflicts since they swap back and forth. There are no conflicts in the commit capsules because they read from write buffer and write to the simulated external memory. The simulation is therefore write-after-read conflict free.

To see the claimed time and space bounds, we note that the ephemeral memory need only be a constant factor bigger than the simulated ephemeral memory because the write buffer can only contain M entries. Each round requires only $O(M/B)$ reads and writes to the persistent memory because the simulating capsules only need the stored copy of the ephemeral memory, do at most M/B reads, and then do at most $O(M/B)$ writes to the other stored copy. The commit capsule does at most M/B simulated writes, each requiring a read from and write to the persistent memory. Because each round simulates M/B reads and writes to external memory at the cost of $O(M/B)$ reads and writes to persistent memory, the faultless work across all capsules is bounded by $O(t)$. Because the probability that a capsule faults is bounded by the maximum capsule work, $O(M/B)$, when $f \leq B/(cM)$, there is a constant c such that

the probability of a capsule faulting is less than 1. Since the faults are independent, the expected total work is a constant factor greater than the faultless work, giving the stated bounds. □

It is also possible to simulate the ideal cache model [31] in the PM model. The ideal cache model is similar to the external memory model, but assumes that the fast memory is managed as a fully associative cache. It assumes a cache of size M is organized in blocks of size B and has an optimal replacement policy. The ideal cache model makes it possible to design cache-oblivious algorithms [31]. Due to the following result, whose proof is in the full version of the paper [14], these algorithms are also efficient in the PM model.

Theorem 3.4. *Any (M, B) ideal cache computation with t cache misses can be simulated on the $(O(M), B)$ PM model with $f \leq B/(cM)$ for a constant $c \geq 2$, using $O(t)$ expected total work.*

4 PROGRAMMING FOR ROBUSTNESS

This simulation of the external memory is not completely satisfactory because its overhead, although constant, could be significant. It can be more convenient and certainly more efficient to program directly for the model. Here we describe one protocol for this purpose. It can greatly reduce the overhead of using the PM model. It is also useful in the context of the parallel model.

Our protocol is designed so capsules begin and end at the boundaries of certain function calls, which we refer to as *persistent function calls*. Non-persistent calls are ephemeral. We assume function calls can be marked as persistent or ephemeral, by the user or possibly compiler. Once a persistent call is made, the callee will never revert back further than the call itself, and after a return the caller will never revert back further than the return. All persistent calls require a constant number of external reads and writes on the call and on the return. In an ephemeral function call a fault in a callee can roll back to before the call, and similarly a fault after a return can roll back to before the return. All ephemeral calls are handled completely in the ephemeral memory and therefore by themselves do not require any external reads or writes. In addition to the persistent function call we assume a `commit` command that forces a capsule boundary at that point. As with a persistent call, the commit requires a constant number of external reads and writes.

We assume that all user code between persistent boundaries is write-after-read conflict free, or otherwise idempotent. This requires a style of programming in which results are copied instead of overwritten. For sequential programs, this increases the space requirements of an algorithm by at most a factor of two. Persistent counters can be implemented by placing a commit between reading the old value and writing the new. In the algorithms that we describe in Section 7, this style is very natural.

Implementing persistent function calls requires some care with a standard stack protocol. Here we outline one way to modify a standard stack discipline to work. We describe how to do this is some detail using closures [3] in the full paper [14].

The stack is organized in stack frames stored in the persistent memory. The first location in each stack frame is a pointer to the first instruction to run, and it is followed by slots for its arguments, for values returned to it, a pointer to the parent frame, and a pointer to code to execute for the parent when the function returns. When making a call, the parent can fill in the frame of the child. In particular the instruction to start at (in the first location), the arguments, a pointer to itself, and the instruction to run on return. As in standard protocols it must also save local variables to its own frame it needs on return. When making a call, the parent can install a the child frame as the new capsule.

On return, the child can fill in the result in the parent frame, and also fill in the instruction to run on return in the first slot of the parent frame. It can then install the parent frame as the new capsule. Arguments should not be modified in place since this would not be write-after-read conflict free. Local variable and return results that are available when returning from a function call must also not be modified for the same reason. A `commit` command can be implemented by creating a function for the code after the commit, and calling it. Standard tail-recursion optimizations can then return directly to the parent of the caller. The main difference of this calling convention from a standard one is keeping an instruction pointer with each frame, and ensuring local variables do not have any write-after-read conflicts. It also means the built in call/ret instruction on certain architectures likely cannot be used.

Memory allocation can be implemented in various ways in a write-after-read conflict free manner. One way is for the memory for a capsule to be allocated starting at a base pointer that is stored in the closure. Memory is allocated one after the other, using a pointer kept in local memory (avoiding the need for a write-after-read conflict to persistent memory in order to update it). In this way, the allocations are the same addresses in memory each time the capsule restarts. At the end of the capsule, the final value of the pointer is stored in the closure for the next capsule. For the Parallel-PM, each processor allocates from its own pool of persistent memory, using this approach. In the case where a processor takes over for a hard-faulting processor, any allocations while the taking-over processor is executing on behalf of the faulted processor will be from the pool of the faulted processor.

5 ROBUSTNESS ON MULTIPLE PROCESSORS

With multiple processors our previous definition of idempotent is inadequate since the other processors can read or write persistent memory locations while a capsule is running. For example, even though the final values written by a capsule c might be idempotent, other processors can observe intermediate values while c is running and therefore act differently than if c was run just once. We therefore consider a stronger variant of idempotency that in addition to requiring that its final effects on memory are if it ran once, requires that it acts as if it ran atomically. The requirement of atomicity is not necessary for correctness, but sufficient for what we need and allows a simple definition. We give an example of how it can be relaxed at the end of the section.

More formally we consider the history of a computation, which is an interleaving of the persistent memory instructions from each of the processors, and which abides by the sequential semantics of the memory. The history includes the additional instructions due to faults (i.e., it is a complete trace of instructions that actually happened). A capsule within a history is *invoked* at the instruction it is installed and *responds* at the instruction that installs the next capsule on the processor. All instructions of a capsule, and

possibly other instructions from other processors, fall between the invocation and response.

We say that a capsule in a history is *atomically idempotent* if

(1) (atomic) all its instructions can be moved in the history to be adjacent somewhere between its invocation and response without violating the memory semantics, and
(2) (idempotent) the instructions are idempotent at the spot they are moved to—i.e., their effect on memory is as if the capsule ran just once without fault.

As with a single processor, we now consider conditions under which capsules are ensured to be idempotent, in this case atomically. Akin to standard definitions of conflict, race, and race free, we say that two persistent memory instructions on separate processors *conflict* if they are on the same block and one is a write. For a capsule within a history we say that one of its instructions has a *race* if it conflicts with another instruction that is between the invocation and response of that capsule. A capsule in a history is *race free* if none of its instructions have a race.

Theorem 5.1. *Any capsule that is write-after-read conflict free and race free in a history is atomically idempotent.*

Proof. Because the capsule is race free we can move its instructions to be adjacent at any point between the invocation and response without affecting the memory semantics. Once moved to that point, the idempotence follows from Theorem 3.1 because the capsule is write-after-read conflict free. □

This property is useful for user code if one can ensure that the capsules are race free via synchronization. We use this extensively in our algorithms. However the requirement of being race free is insufficient in general because synchronizations themselves require races. In fact the only way to ensure race freedom throughout a computation would be to have no processor ever write a location that another processor ever reads or writes. We therefore consider some other conditions that are sufficient for atomic idempotence.

Racy Read Capsule. We first consider a *racy read capsule*, which reads one location from persistent memory and writes its value to another location in persistent memory. The capsule can have other instructions, but none of them can depend on the value that is read. A racy read capsule is atomically idempotent if all its instructions except for the read are race free. This is true because we can move all instructions of the capsule, with possible repeats due to faults, to the position of the last read. The capsule will then properly act like the read and write happened just once. Because races are allowed on the read location, there can be multiple writes by other processors of different values to the read location, and different such values can be read anytime the racy read capsule is restarted. However, because the write location is race free, no other processor can "witness" these possible writes of different values to the write location. Thus, the copy capsule is atomically idempotent. A copy capsule is a useful primitive for copying from a volatile location that could be written at any point into a processor private location that will be stable once copied. Then when the processor private location is used in a future capsule, it will stay the same however many times the capsule faults and restarts. We make significant use of this in the work-stealing scheduler.

Racy Write Capsule. We also consider a *racy write capsule*, for which the only instruction with a race is a write instruction to persistent memory, and the instruction races only with either read instructions or other write instructions, but not both kinds. Such a capsule can be shown to be atomically idempotent. In the former case (races only with reads), then in any history, the value in the write location during the capsule transitions from an old value to a new value exactly once no matter how many times the capsule is restarted. Thus, for the purposes of showing atomicity, we can move all the instructions of the capsule to immediately before the first read that sees the new value, or to the end of the capsule if there is no such read. Although the first time the new value is written (and read by other processors) may be part of a capsule execution that subsequently faulted, the effect on memory is as if the capsule ran just once without fault (idempotency). In the latter case (races only with other writes), then if in the history the racy write capsule is the last writer before the end of the capsule, we can move all the instructions of the capsule to the end of the capsule, otherwise we can move all the instructions to the beginning of the capsule, satisfying atomicity and idempotency.

Compare-and-Modify (CAM) Instruction. We now consider idempotency of the CAS instruction. Recall that we assume that a CAS is part of the machine model. We cannot assume the CAS is race free because the whole purpose of the operation is to act atomically in the presence of a race. Unfortunately it seems hard to efficiently simulate a CAS at the user level when there are faults. The problem is that a CAS writes two locations, the two that it swaps. In the standard non-faulty model one is local (a register) and therefore the CAS involves a single shared memory modification and a local register update. Unfortunately in the Parallel-PM model, the processor could fault immediately before or after the CAS instruction. On restart the local register is lost and therefore the information about whether it succeeded is lost. Looking at the shared location does not help since identical CAS instructions from other processors might have been applied to the location, and the capsule cannot distinguish its success from their success.

Instead of using a CAS, here we show how to use a weaker instruction, a *compare-and modify (CAM)*. A CAM is simply a CAS for which no subsequent instruction in the capsule reads the local result (i.e., the swapped value).[3] Furthermore, we restrict the usage of a CAM. For a capsule within a history we say a write w (including a CAS or CAM) to persistent memory is *non-reverting* if no other conflicting write between w and the capsule's response changes the value back to its value before w. We define a *CAM capsule* as a capsule that contains one non-reverting CAM and may contain other write-after-read conflict free and race free instructions.

Theorem 5.2. *A CAM capsule is atomically idempotent.*

Proof. Assume that the CAM is non-reverting and all other instructions in the capsule are write-after-read conflict free and race free. Due to faults the CAM can repeat multiple times, but it can only succeed in changing the target value at most once. This is because the CAM is non-reverting so once the target value is changed, it could not be changed back. Therefore if the CAM ever succeeds, for the purpose of showing atomicity, in the history we move all the

[3] Some CAS instructions in practice return a boolean to indicate success; in such cases, the boolean cannot be read either.

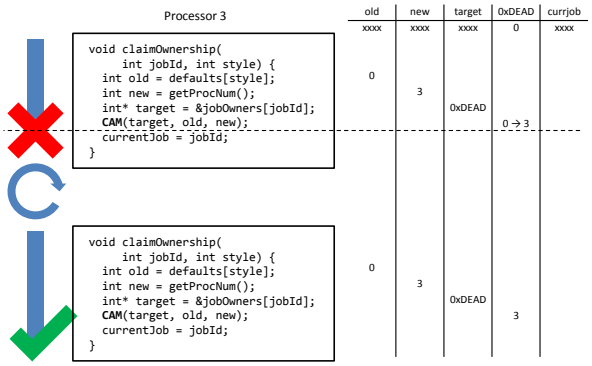

Figure 2: CAM Capsule Example. In CAM capsules, earlier faulting runs of the capsule may perform work that is visible to the rest of the system.

instructions of the capsule (including the instructions from faulty runs) to the point of the successful CAM. This does not affect the memory semantics because none of the other instructions have races, and any of the other CAMs were unsuccessful and therefore also have no affect on memory. At the point of the successful CAM the capsule acts like it ran once because it is write-after-read conflict free—other than the CAM, which succeeded just once. If the CAM never succeeds, the capsule is conflict free and race free because the CAM did not do any writes, so Theorem 5.1 applies. □

The example CAM capsule in Figure 2 shows one of the interesting properties of idempotence: unlike transactions or checkpointing, earlier runs that faulted can make changes to the memory that are seen or used by other processes. Similarly, these earlier runs can affect the results of the successful run, as long as the result is equivalent to a non-faulty run.

A CAM can be used to implement a form of test-and-set in a constant number of instructions. In particular, we will assume a location can either be *unset*, or the value of a process identifier or other unique identifier. A process can then use a CAM to conditionally swap such a location from *unset* to its unique identifier. The process can then check if it "won" by seeing if its identifier is in the location. We make heavy use of this in the work-stealing scheduler to atomically "steal" a job from another queue. It can also be used at the join point of two threads in fork-join parallelism to determine who got there last (the one whose CAM from *unset* was unsuccessful) and hence needs to run the code after the join.

Racy Multiread Capsule. It is also possible to design capsules that are idempotent without the requirement of atomicity. By way of example, we discuss the *racy multiread capsule*. This capsule consists of multiple racy read capsules that have been combined together into a single capsule. Concurrent processes may write to locations that the capsule is reading between reads, which violates atomicity. Despite this, a racy multiread capsule is idempotent since the results of the final successful run of the capsule will overwrite any results of partial runs. We make use of the snapshot capsule in the work-stealing scheduler to reduce the number of capsules required. It is not needed for correctness.

6 WORK STEALING

We show how to implement an efficient version of work stealing (WS) in the Parallel-PM model. Our results are based on the work-stealing scheduler of Arora, Blumofe, and Plaxton (ABP) [5] and therefore work in a multiprogrammed environment where the number of active processors can change. As in their work, we require some assumptions about the machine, which we summarize here.

The schedule is a two-level scheduler in which the work-stealing scheduler, under our control, maps threads to processes, and an adversarial operating system scheduler maps processes to processors. The OS scheduler can change the number of allocated processors and which processes are scheduled on those processors during the computation, perhaps giving processors to other users. The number of processes and the maximum number of processors used is given by P. The average number that are allocated to the user is P_A.

The quanta for scheduling is at least the time for two scheduling steps where each step takes a small constant number of instructions. In our case we cannot guarantee that the quanta is big enough to capture two steps since the processor could fault. However it is sufficient to show that with constant probability two scheduling steps complete within the quanta, which we can show.

The available instruction set contains a yield-to-all instruction. This instruction tells the OS that it must schedule all other processes that have not hard faulted before (or at the same time) as the process that executes the instruction. It is used to ensure that processors that are doing useful work have preference over ones who run out of work and need to steal.

Our schedule differs from the ABP scheduler in some crucial ways since our model allowing processors to fault. First, our scheduler cannot use a CAS, for reasons described in Section 5, and instead must use a CAM. ABP uses a CAS and we see no direct translation to using a CAM. Second, our scheduler has to handle soft faults anywhere in either the scheduler or the user program. This requires some care to maintain idempotence. Third, our scheduler has to handle hard faults. In particular it has to be able to steal from a processor that hard faults while it is running a thread. It cannot restart the thread from scratch, but needs to start from the previous capsule boundary (a thread can consist of multiple capsules).

Our scheduler is also similar to the ABP scheduler in some crucial ways. In particular it uses a work-stealing double ended work queue and takes a constant number of instructions for the popTop, popBottom, and pushBottom functions. This is important in proving the performance bounds and allows us to leverage much of their analysis. An important difference in the performance analysis is that faults can increase both the total work and the total depth. Because faults can happen anywhere this holds for the user work and for the scheduler. The expected work is only increased by a constant factor, which is not a serious issue. However, for total depth, expectations cannot be carried through the maximum implied by parallel execution. We therefore need to consider high probability bounds.

6.1 The Scheduler Interface

For handling faults, and in particular hard faults, the interaction of the scheduler and threads is slightly different from that of ABP.

We assume that when a thread finishes it jumps to the `scheduler`.[4] When a thread forks another thread, it calls a `fork` function, which pushes the new thread on the bottom of the work queue and returns to the calling thread. When the scheduler starts a thread it jumps to it (actually a capsule representing the code to run for the thread). Recall that when the thread is done it jumps back to the scheduler. These are the only interactions of threads and the scheduler—i.e. jumping to a thread from the scheduler, forking a new thread within a thread, and jumping back to the scheduler from a thread on completion. All of these occur at capsule boundaries, but a thread itself can consist of many capsules. We assume that at a join (synchronization) point of threads whichever one arrives last continues the code after the join and therefore that thread need not interact with the scheduler. The other threads that arrive at the join earlier finish and jump to the scheduler. In our setup, therefore, a thread is never blocked, assuming the `fork` function is non-blocking.

6.2 WS-Deque

A work-stealing deque (WS-deque) is a concurrent deque supporting a limited interface. Here we used a similar interface to ABP. In particular the interface supports popTop, pushBottom, and popBottom. Any number of concurrent processors can execute popTop, but only one process can execute either pushBottom or popBottom. The idea is only the process owning the deque will work on the bottom. The deque is linearizable except that popTop can return empty even if the deque is not-empty. However this can only happen if another concurrent popTop succeeds with a linearization point when the popTop is live, i.e., from invocation to response.

We provide an implementation of a idempotent WS-deque in Figure 3. Our implementation maintains an array of tagged entries that refer to threads that the processor has either enabled or stolen while working on the computation. The tag is simply a counter that is used to avoid the ABA problem [34]. An *entry* consists of one of the following states:

- *empty*: An empty entry is one that has not been associated with a thread yet. Newly created elements in the array are initialized to empty.
- *local*: A local entry refers to a thread that is currently being run by the processor that owns this WS-Deque. We need to track local entries to deal with processors that have a hard fault (i.e., never restart).
- *job*: A job entry is equivalent to the values found in the original implementation of the WS-Deque. It contains a thread (i.e., a capsule to jump to start the thread).
- *taken*: A taken entry refers to a thread that has already been or is in the process of being stolen. It contains a pointer to the entry that the thief is using to hold the stolen thread, and the tag of that entry at the time of the steal.

The transition table for the entry states is shown in Figure 4.

In addition to this array of entries, we maintain pointers to the top and the bottom of the deque, which is a contiguous region of the array. As new threads are forked by the owner process, new entries will be added to the bottom of the deque using the pushBottom function. The bottom pointer will be updated to these new entries. The top pointer will move down on the deque as threads are stolen. This implementation does not delete elements at the top of the

[4]Note that jumping to a thread is the same as installing a capsule.

```
1  P = number of procs
2  S  = stack size

4  struct procState {
5    union entry = empty
6                | local
7                | job of continuation
8                | taken of ⟨entry*,int⟩

10   ⟨int,entry⟩ stack[S];
11   int top;
12   int bot;
13   int ownerID;

15   inline int getStep(i) { return stack[i].first; }

17   inline void clearBottom() {
18     stack[bot] = ⟨getStep(bot)+1, empty⟩; }

20   void helpPopTop() {
21     int t = top;
22     switch(stack[t]) {
23       case ⟨_, taken(ps,i)⟩:
24         // Set thief state.
25         CAM(ps, ⟨i,empty⟩, ⟨i+1,local⟩);
26         CAM(&top, t, t+1);    // Increment top.
27    } }

29   // Steal from current process, if possible.
30   // If a steal happens, location e is set to "local"
31   // & a job is returned. Otherwise NULL is returned.
32   continuation popTop(entry* e, int c) {
33     helpPopTop();
34     int i = top;
35     ⟨int, entry⟩ old = stack[i];
36     commit;
37     switch(old) {
38       // No jobs to steal and no ongoing local work.
39       case ⟨j, empty⟩: return NULL;
40       // Someone else stole in meantime.  Help it.
41       case ⟨j, taken(_)⟩:
42          helpPopTop(); return NULL;
43       // Job available, try to steal it with a CAM.
44       case ⟨j, job(f)⟩:
45         ⟨int, entry⟩ new = ⟨j+1, taken(e,c)⟩;
46         CAM(&stack[i], old, new);
47         helpPopTop();
48         if (stack[i] != new) return NULL;
49         return f;
50       // No jobs to steal, but there is local work.
51       case ⟨j, local⟩:
52         // Try to steal local work if process is dead.
53         if (!isLive(ownerID) && stack[i] == old) {
54           commit;
55           ⟨int, entry⟩ new = ⟨j+1,taken(e,c)⟩;
56           stack[i+1] = ⟨getStep(i+1)+1, empty⟩;
57           CAM(&stack[i], old, new);
58           helpPopTop();
59           if (stack[i] != new) return NULL;
60           return getActiveCapsule(ownerID);
61         }
62         // Otherwise, return NULL.
63         return NULL;
64   }  }

66   void pushBottom(continuation f) {
67     int b = bot;
68     int t1 = getStep(b+1);
69     int t2 = getStep(b);
70     commit;
71     if (stack[b] == ⟨t2, local⟩) {
72       stack[b+1] = ⟨t1+1, local⟩;
73       bot = b + 1;
74       CAM(&stack[b], ⟨t2, local⟩, ⟨t2+1, job(f)⟩)
75     } else if (stack[b+1].second == empty) {
76       states[getProcNum()].pushBottom(f);
77     }
78     return;
79   }
```

```
  continuation popBottom() {
    int b = bot;
    ⟨int, entry⟩ old = stack[b-1];
    commit;
    if (old == ⟨j, job(f)⟩) {
      CAM(&stack[b-1], old, ⟨j+1,local⟩);
      if (stack[b-1] == ⟨j+1, local⟩) {
        bot = b-1;
        return f;
      }   }
    // If we fail to grab a job, return NULL.
    return NULL;
  }

  ^ findWork() {
    // Try to take from local stack first.
    continuation f = popBottom();
    if (f) GOTO(f);
    // If nothing locally, randomly steal.
    while (true) {
      yield();
      int victim = rand(P);
      int i = getStep(bot);
      continuation g
          = states[victim].popTop(&stack[bot],i);
      if (g) GOTO(g);
    }
  }
}

procState states[P]; // Stack for each process.

// User call to fork.
void fork(continuation f) {
  // Pushes job onto the correct stack.
  states[getProcNum()].pushBottom(f);
}

// Return to scheduler when any job finishes.
^ scheduler() {
  // Mark the completion of local thread.
  states[getProcNum()].clearBottom();
  // Find work on the correct stack.
  GOTO(states[getProcNum()].findWork());
}
```

Figure 3: Fault-tolerant WS-Deque Implementation. Jumps are marked as GOTO and functions that are jumped to and do not return (technically continuations) are marked with a ^. All CAM instructions occur in separate capsules, similar to function calls.

		New State			
		Empty	Local	Job	Taken
Old State	Empty	-	✓		
	Local	✓	-	✓	✓
	Job		✓	-	✓
	Taken				-

Figure 4: Entry state transition diagram

deque, even after steals. This means that we do not need to worry about entries being deleted in the process of a steal attempt, but does mean that maintaining P WS-Deques for a computation with span T_∞ requires $O(PT_\infty)$ storage space.

Our implementation of the WS-Deque maintains a consistent structure that is useful for proving its correctness and efficiency. The elements of our WS-Deque are always ordered from the beginning to the end of the array as follows:

(1) A non-negative number of taken entries. These entries refer to threads that have been stolen, or possibly in the case of the last taken entry, to a thread that is in the process of being stolen.
(2) A non-negative number of job entries. These entries refer to threads that the process has enabled that have not been stolen or started since their enabling.
(3) Zero, one, or two local entries. If a process has one local entry, it is the entry that the process is currently working on. Processes can momentarily have two local entries during the pushBottom function, before the earlier one is changed to a job. If a process has zero local entries, that means the process has completed the execution of its local work and is in the process of acquiring more work through popBottom or stealing, or it is dead.
(4) A non-negative number of empty entries. These entries are available to store new threads as they are forked during the computation.

We can also relate the top and bottom pointers of the WS-Deque (i.e. the range of the deque) to this array structure. The top pointer will point to the last taken entry in the array if a steal is in process. Otherwise, it will point to the first entry after the taken entries. At the end of a capsule, the bottom pointer will point to the local entry if it exists, or the first empty entry after the jobs otherwise. The bottom pointer can also point to the last job in the array or the earlier local entry during a call to pushBottom.

6.3 Algorithm Overview and Rationale

We now give an overview and rationale of correctness of our work-stealing scheduler under the Parallel-PM.

Each process is initialized with an empty WS-Deque containing enough `empty` entries to complete the computation. The top and bottom pointers of each WS-Deque are set to the first entry. One process is assigned the root thread. This process installs the first capsule of this thread, and sets its first entry to `local`. All other processes install the `findWork` capsule.

Once computation begins, the adversary chooses processes to schedule according to the rules of the yield instruction described in ABP, with the additional restriction that dead processes cannot be scheduled. When a process is scheduled, it continues running its code. This code may be scheduler code or user code.

If the process is running user code, this continues until the code calls `fork` or terminates. Calls to `fork` result in the newly enabled thread being pushed onto the bottom of the process' WS-Deque. When the user code terminates, the process returns to the `scheduler` function.

The scheduler code works to find new threads for the process to work on. It begins by calling the `popBottom` function to try and find a thread on the owner's WS-Deque. If `popBottom` finds a thread, the process works on that thread as described above. Otherwise, the process begins to make steal attempts using the `popTop` function on random victim stacks. In a faultless setting, our work-stealing scheduler fuctions like that of ABP. We use the additional information stored in the WS-Deques and the configuration of capsule boundaries to provide fault tolerance.

We provide correctness in a setting with soft faults using idempotent capsules. Each capsule in the scheduler is an instance of one of the capsules discussed in Section 5. This means that processes can fault and restart without affecting the correctness of the scheduler.

Providing correctness in a setting with hard faults is more challenging. This requires the scheduler to ensure that work being done by processes that hard fault is picked up in the same capsule that the fault ocurred during by exactly one other process. We handle this by allowing thieves to steal `local` entries from dead processes. A process can check whether another process is dead using a liveness oracle `isLive(procId)`.

The liveness oracle might be constructed by implementing a counter and a flag for each process. Each process updates its counter after a constant number of steps (this does not have to be synchronized). If the time since a counter has last updated passes some threshold, the process is considered dead and its flag is set. If the process restarts, it can notice that it was marked as dead, clear its flag, and enter the system with a new empty WS-Deque. Constructing such an oracle does not require a global clock or tight synchronization.

By handling these high level challenges, along with some of the more subtle challenges that occur when trying to provide exactly-once semantics in the face of both soft and hard faults, we reach the following result.

Theorem 6.1. *The implementation of work stealing provided in Figure 3 correctly schedules work according to the specification in Section 6.*

The proof, appearing in the full version of the paper [14], deals with the many possible code interleavings that arise when considering combinations of faulting and concurrency. We discuss our methods for ensuring that work is neither duplicated during capsule retries after soft faults or dropped due to hard faults. In particular, we spend considerable time ensuring that recovery from hard faults during interaction with the bottom of the WS-Deque happens correctly.

6.4 Time Bounds

We now analyze bounds on runtime based on the work-stealing scheduler under the assumptions mentioned at the start of the section (scheduled in fixed quanta, and supporting a yield-to-all instruction).

As with ABP, we consider the total amount of work done by a computation, and the depth of the computation, also called the critical path length. In our case we have W, the work assuming no faults, and W_f, the work including faults. In algorithm analysis the user analyzes the first, but in determining the runtime we care about the second. Similarly we have both D, a depth assuming no faults, and D_f, a depth with faults.

For the time bounds we can leverage the proof of ABP. In particular as in their algorithm our popTop, popBottom, and pushBottom functions all take $O(1)$ work without faults. With our deque, operations take expected $O(1)$ work. Also as with their version, our popTop is unsuccessful (returns Null when there is work) only if another popTop is successful during the attempt. The one place where their proof breaks down in our setup is the assumption that a constant sized quanta can always capture two steal attempts. Because our processors can fault multiple times, we cannot guarantee this. However in their proof this is needed to show that for every P steal attempts, with probability at least $1/4$, at least $1/4$ of the non-empty deques are successfully stolen from ([5], Lemma 8). In our case a constant fraction $(1 - O(1) \cdot f)^2$ of adjacent pairs of steal attempts will not fault at all and therefore count as a steal attempt. For analysis we can assume that if either steals in a pair faults, then the steal is unsuccessful. This gives a similar result, only with a different constant, i.e., with probability at least $1/4$, at least $(1 - O(1) \cdot f)^2/4$ of the non-empty deques are successfully stolen from. We note that hard faults affect the average number of active processors P_A. However they otherwise have no asymptotic affect in our bounds because a hard fault in our scheduler is effectively the same as forking a thread onto the bottom of a work-queue and then finishing.

ABP show that their work-stealing scheduler runs in expected time $O(W/P_A + DP/P_A)$. To apply their results we need to plug in W_f for W because that is the actual work done, and D_f for D because that is actual depth. While bounding W_f to be within a constant factor of W is straightforward, bounding D_f is trickier because we cannot sum expectations to get the depth bound (the depth is a maximum over paths lengths). Instead we show that with some high probability no capsule faults more than some number of times l. We then simply multiply the depth by l. By making the probability sufficiently high, we can pessimistically assume that in the unlikely even that any capsule faults more than l times then, the depth is as large as the work. This idea leads to the following theorem.

Theorem 6.2. *Consider any multithreaded computation with W work, D depth, and C maximum capsule work (all assuming no faults) for which all capsules are atomically idempotent. On the Parallel-PM with P processors, P_A average number of active processors, and fault probability bounded by $f \leq 1/(2C)$, the expected total time T_f for the computation is*

$$O\left(\frac{W}{P_A} + D\left(\frac{P}{P_A}\right)\left\lceil \log_{1/(Cf)} W \right\rceil\right).$$

Proof. We must account for faults in both the computation and the work-stealing scheduler. The work-stealing scheduler has $O(1)$ maximum capsule work, which we assume is at most C. Because we assume all faults are independent, the probability that a capsule will run l or more times is upper bounded by $(Cf)^l$. Therefore if there are κ capsules in the computation including the capsules executed as part of the scheduler, the probability that any one runs more than l times is upper bounded by $\kappa(Cf)^l$ (by the union bound). If we want to bound this probability by some ϵ, we have $\kappa(Cf)^l \leq \epsilon$. Solving for l and using $\kappa \leq 2W$ gives $l \leq \lceil \log_{1/(Cf)}(2W/\epsilon) \rceil$. This means that with probability at most ϵ, $D_f \leq D \log_{1/(Cf)}(2W/\epsilon)$. If we set $\epsilon = 2/W$ then $D_f \leq 2D \log_{1/(Cf)} W$. Now we assume that if any capsule faults l times or more that the depth of the computation equals the work. This gives $(P/P_A)(2/W)W + (1 - 2/W)2D\lceil \log_{1/(Cf)} W \rceil)$ as the expected value of the second term of the ABP bound, which is bounded by $O((P/P_A)D\lceil \log_{1/(Cf)} W \rceil)$. Because the expected total work for the first term is $W_f \leq (1/(1 - Cf))W$, and given $Cf \leq 1/2$, the theorem follows. □

This time bound differs from the ABP bound only in the extra $\log_{1/(Cf)} W$ factor. If we assume P_A is a constant fraction of P then the expected time simplifies to $O(W/P + D\lceil \log_{1/(Cf)} W \rceil)$.

7 FAULT-TOLERANT ALGORITHMS

In this section, we outline how to implement several algorithms for the Parallel-PM model. The algorithms are all based on binary fork-join parallelism (i.e., nested parallelism), and hence fit within the multithreaded model. We state all results in terms of faultless work and depth. The results can be used with Theorem 6.2 to derive bounds on time for the Parallel-PM. Recall that in the Parallel-PM model, external reads and writes are unit cost, and all other instructions have no cost (accounting for other instructions would not be hard). The algorithms that we use are already race-free. Making them write-after-read conflict free simply involves ensuring that reads and writes are to different locations. All capsules of the algorithms are therefore atomically idempodent. The base case for each of our variants of the algorithms is done sequentially within the ephemeral memory.

Prefix Sum. Given n elements $\{a_1, \cdots, a_n\}$ and an associative operator "+", the prefix sum algorithm computes a list of prefix sums $\{p_1, \cdots, p_n\}$ such that $p_i = \sum_{j=1}^{i} a_j$. Prefix sum is one of the most commonly-used building blocks in parallel algorithm design [43].

We note that the standard prefix sum algorithm [43] works well in our setting. The algorithm consists of two phases—the up-sweep phase and the down-sweep phase, both based on divide-and-conquer. The up-sweep phase bisects the list, computes the sum of each sublist recursively, adds the two partial sums as the sum of the overall list, and stores the sum in the persistent memory. After the up-sweep phase finishes, we run the down-sweep phase with the same bisection of the list and recursion. Each recursive call in this phase has a temporary parameter t, which is initiated as 0 for the initial call. Then within each function, we pass t to the left recursive call and t + *LeftSum* for the right recursive call, where *LeftSum* is the sum of the left sublist computed from the up-sweep phase. In both sweeps the recursion stops when the sublist has no more than B elements, and we sequentially process it using $O(1)$ memory transfers. For the base case in the down-sweep phase, we set the first element p_i to be $t + a_i$, and then sequentially compute the rest of the prefix sums for this block. The correctness of p_i follows from how t is computed along the path to a_i.

This algorithm fits the Parallel-PM model in a straightforward manner. We can place the body of each function call (without the recursive calls) in an individual capsule. In the up-sweep phase, a capsule reads from two memory locations and stores the sum back to another location. In the down-sweep phase, it reads from at most one memory location, updates t, and passes t to the recursive calls. Defining capsules in this way provides write-after-read conflict-freedom and limits the maximum capsule work to a constant.

Theorem 7.1. *The prefix sum of an array of size n can be computed in $O(n/B)$ work, $O(\log n)$ depth, and $O(1)$ maximum capsule work, using only atomically-idempotent capsules.*

Merging. A merging algorithm takes the input of two sorted arrays A and B of size l_A and l_B ($l_A + l_B = n$), and returns a sorted array containing the elements in both input lists. We use an algorithm on the Parallel-PM model based on the classic divide-and-conquer algorithm [15].

The first step of the algorithm is to allocate the output array of size n. Then the algorithm conducts dual binary searches of the arrays in parallel to find the elements ranked $\{n^{2/3}, 2n^{2/3}, 3n^{2/3}, \ldots, (n^{1/3}-1)n^{2/3}\}$ among the set of keys from both arrays, and recurses on each pair of subarrays until the base case when there are no more than B elements left (and we switch to a sequential version). We put each of the binary searches into a capsule, as well as each base case. These capsules are write-after-read conflict free because the output of each capsule is written to a different subarray. Based on the analysis in [15] we have the following theorem.

Theorem 7.2. *Merging two sorted arrays of overall size n can be done in $O(n/B)$ work, $O(\log n)$ depth, and $O(\log n)$ maximum capsule work, using only atomically-idempotent capsules.*

Sorting. Using the merging algorithm in Section 7, we can implement a fault-tolerant mergesort with $O((n/B)\log(n/M))$ work and maximum capsule work $O(\log n)$. However, this is not optimal. We now outline a samplesort algorithm with improved work $O(n/B \cdot \log_M n)$, based on the algorithm in [15].

The sorting algorithm first splits the set of elements into $\sqrt{n}$ subarrays of size $\sqrt{n}$ and recursively sorts each of the subarrays. The recursion terminates when the subarray size is less than M, and the algorithm then sequentially sorts within a single capsule. Then the algorithm samples every $\log n$'th element from each subarray. These samples are sorted using mergesort, and $\sqrt{n}$ pivots are picked from the result using a fixed stride. The next step is to merge each $\sqrt{n}$-size subarray with the sorted pivots to determine bucket boundaries within each subarray. Once the subarrays have been split, prefix sums and matrix transposes are used to determine the location in the buckets where each segment of the subarray is to be sent. After that, the keys need to be moved to the buckets, using a bucket transpose algorithm. We can use our prefix sum algorithm and the divide-and-conquer bucket transpose algorithm from [15], where the base case is a matrix of size less than M, and in the base case the transpose is done sequentially within a single capsule (note that this assumes $M > B^2$ to be efficient). The last step is to recursively sort the elements within each bucket. All steps can be made write-after-read conflict free by writing to locations separate than those being read. By applying the analysis in [15] with the change that the base cases (for the recursive sort and the transpose) are when the size fits in the ephemeral memory, and that the base case is done sequentially, we obtain the following theorem.

Theorem 7.3. *Sorting n elements can be done in $O(n/B \cdot \log_M n)$ work, $O((M/B+\log n)\log_M n)$ depth, and $O(M/B)$ maximum capsule work, using only atomically-idempotent capsules.*

It is possible that the $\log n$ term in the depth could be reduced using a sort by Cole and Ramachandran [24].

Matrix Multiplication. Due to space constraints, our Parallel-PM algorithm for matrix multiply is given in the full version of this paper [14], and here we only introduce our result.

Theorem 7.4. *Multiplying two square matrices of size n can be done in $O(n^3/(B\sqrt{M}))$ work, $O(M/B + \log^2 n)$ depth, and $O(M/B)$ maximum capsule work, using only atomically-idempotent capsules.*

The algorithm is a slight modification of the classic 8-way divide-and conquer approach [31]. The computation is made race-free by setting correct capsule boundaries.

8 CONCLUSION

In this paper, we describe the Parallel Persistent Memory model, which characterizes faults as loss of data in individual processors and their associated volatile memory. For this paper, we consider an external memory model view of algorithm cost, but the model could easily be adapted to support other traditional cost models. We also provide a general strategy for designing programs based on capsules that perform properly when faults occur. We specify a condition of being atomically idempotent that is sufficient for correctness, and provide examples of atomic idempotent capsules that can be used to generate more complex programs. We use these capsules to build a work-stealing scheduler that can run programs in a parallel system while tolerating both hard and soft faults with only a modest increase in the total cost of the computation. We also provide several algorithms designed to support fault tolerance using our capsule methodology. We believe that the techniques in this paper can provide a practical way to provide the desirable quality of fault tolerance without requiring significant changes to hardware or software.

ACKNOWLEDGEMENTS

This work was supported in part by NSF grants CCF-1408940, CCF-1533858, and CCF-1629444.

REFERENCES

[1] Y. Afek, D. S. Greenberg, M. Merritt, and G. Taubenfeld. Computing with faulty shared memory. In *PODC*, 1992.
[2] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9), 1988.
[3] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *POPL*, 1989.
[4] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *SPAA*, 2008.
[5] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2), Apr 2001.
[6] Y. Aumann and M. Ben-Or. Asymptotically optimal PRAM emulation on faulty hypercubes. In *FOCS*, 1991.
[7] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters*, 7(1), 2015.
[8] N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. Parallel algorithms for asymmetric read-write costs. In *SPAA*, 2016.
[9] N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. Implicit decomposition for write-efficient connectivity algorithms. In *IPDPS*, 2018.
[10] R. Berryhill, W. Golab, and M. Tripunitara. Robust shared objects for non-volatile main memory. In *Conf. on Principles of Distributed Systems (OPODIS)*, volume 46, 2016.
[11] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *OOPSLA*, 2016.
[12] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, and J. Shun. Sorting with asymmetric read and write costs. In *SPAA*, 2015.
[13] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, and J. Shun. Efficient algorithms with asymmetric read and write costs. In *ESA*, 2016.
[14] G. E. Blelloch, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. The parallel persistent memory model. *arXiv preprint:1805.05580*, 2018.
[15] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low depth cache-oblivious algorithms. In *SPAA*, 2010.
[16] M. Buettner, B. Greenstein, and D. Wetherall. Dewdrop: an energy-aware runtime for computational RFID. In *NSDI*, 2011.
[17] F. Cappello, G. Al, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomput. Front. Innov.: Int. J.*, 1(1), Apr. 2014.
[18] E. Carson, J. Demmel, L. Grigori, N. Knight, P. Koanantakool, O. Schwartz, and H. V. Simhadri. Write-avoiding algorithms. In *IPDPS*, 2016.
[19] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *OOPSLA*, 2014.
[20] H. Chauhan, I. Calciu, V. Chidambaram, E. Schkufza, O. Mutlu, and P. Subrahmanyam. NVMove: Helping programmers move to byte-based persistence. In *INFLOW*, 2016.
[21] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7), 2015.
[22] B. S. Chlebus, A. Gambin, and P. Indyk. PRAM computations resilient to memory faults. In *ESA*, 1994.
[23] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, 2011.
[24] R. Cole and V. Ramachandran. Resource oblivious sorting on multicores. *ACM Transactions on Parallel Computing (TOPC)*, 3(4), 2017.
[25] A. Colin and B. Lucia. Chain: tasks and channels for reliable intermittent programs. *OOPSLA*, 2016.
[26] A. Colin and B. Lucia. Termination checking and task decomposition for task-based intermittent programs. In *International Conference on Compiler Construction*, 2018.
[27] T. David, A. Dragojevic, R. Guerraoui, and I. Zablotchi. Log-free concurrent data structures. EPFL Technical Report, 2017.
[28] M. A. de Kruijf, K. Sankaralingam, and S. Jha. Static analysis and compiler design for idempotent processing. In *PLDI*, 2012.
[29] I. Finocchi and G. F. Italiano. Sorting and searching in the presence of memory faults (without redundancy). In *STOC*, 2004.
[30] M. Friedman, M. Herlihy, V. J. Marathe, and E. Petrank. A persistent lock-free queue for non-volatile memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2018.
[31] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, 1999.
[32] D. Grove, S. S. Hamouda, B. Herta, A. Iyengar, K. Kawachiya, J. Milthorpe, V. Saraswat, A. Shinnar, M. Takeuchi, and O. Tardieu. Failure recovery in resilient X10. Technical Report RC25660 (WAT1707-028), IBM Research, Computer Science, 2017.
[33] R. Guerraoui and R. R. Levy. Robust emulations of shared memory in a crash-recovery model. In *Inter. Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2004.
[34] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.
[35] J. Hester, K. Storer, and J. Sorber. Timely execution on intermittently powered batteryless sensors. In *Proc. ACM Conference on Embedded Network Sensor Systems*, 2017.
[36] T. C.-H. Hsu, H. Bruegner, I. Roy, K. Keeton, and P. Eugster. NVthreads: Practical persistence for multi-threaded applications. In *EuroSys*, 2017.
[37] Intel. Intel NVM library. https://github.com/pmem/nvml/.
[38] Intel. Intel architecture instruction set extensions programming reference. Technical Report 3319433-029, Intel Corporation, April 2017.
[39] J. Izraelevitz, T. Kelly, and A. Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *ASPLOS*, 2016.
[40] J. Izraelevitz, H. Mendes, and M. L. Scott. Brief announcement: Preserving happens-before in persistent memory. In *SPAA*, 2016.
[41] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *DISC*, 2016.
[42] R. Jacob and N. Sitchinava. Lower bounds in the asymmetric external memory model. In *SPAA*, 2017.
[43] J. JaJa. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.
[44] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won. NVWAL: exploiting NVRAM in write-ahead logging. In *ASPLOS*, 2016.
[45] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch. High-performance transactions for persistent memories. In *ASPLOS*, 2016.
[46] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh. Wort: Write optimal radix tree for persistent memory storage systems. In *USENIX Conference on File and Storage Technologies (FAST)*, 2017.
[47] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren. DudeTM: Building durable transactions with decoupling for persistent memory. In *ASPLOS*, 2017.
[48] B. Lucia and B. Ransford. A simpler, safer programming and execution model for intermittent systems. *PLDI*, 2015.
[49] K. Maeng, A. Colin, and B. Lucia. Alpaca: intermittent execution without checkpoints. *OOPSLA*, 2017.
[50] J. S. Meena, S. M. Sze, U. Chand, and T.-Y. Tseng. Overview of emerging non-volatile memory technologies. *Nanoscale Research Letters*, 9, 2014.
[51] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson. Atomic in-place updates for non-volatile main memories with Kamino-Tx. In *EuroSys*, 2017.
[52] F. Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey III, and D. R. C. amd Michael L. Scott. Dali: A periodically persistent hash map. In *DISC*, 2017.
[53] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *ISCA*, 2014.
[54] J. Van Der Woude and M. Hicks. Intermittent computation without hardware support or programmer intervention. In *OSDI*, 2016.
[55] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS*, 2011.
[56] Yole Developpement. Emerging non-volatile memory technologies, 2013.