The Parallel Persistent Memory Model

Guy E. Blelloch[1] Phillip B. Gibbons[1] Yan Gu[3] Charles McGuffey[1] Julian Shun[2]
[1]Carnegie Mellon University [2]MIT CSAIL [3]U.C. Riverside

We present a parallel computational model called the *Parallel Persistent Memory (Parallel-PM) model*, which 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 [1, 2], each processor runs a standard instruction set on its ephemeral memory and has external write/read instructions for transferring blocks of size B to/from the persistent memory. The cost of an algorithm is calculated based on the number of persistent memory 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 recent byte-addressable non-volatile memories (NVRAMs) 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 Optane memory technology became available in a byte-addressable DIMM (NVRAM) form-factor early in 2019. While such memories are expected to be the pervasive type of memory [7, 8], 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 [5].

The Persistent Memory Model. In this work, we first consider a single processor version of the model, the *Persistent Memory (PM) model*, and give conditions under which programs are robust against faults. In particular, we identify that breaking a computation into contiguous segments, known as *capsules*, with the following properties results in idempotent behavior despite capsule restart. First, capsules must have no write-after-read conflicts, as such conflicts could cause the capsule's input data to be overwritten prior to the capsule restarting. Second, because all data in the ephemeral memory is lost on a fault, the capsule cannot read any ephemeral memory location that it has not already written. If the processor faults while executing a given capsule, we simply restart the capsule from its beginning using a capsule restart pointer saved in persistent memory. Because of the idempotent behavior, partial runs of capsules prior to a successful run maintains equivalance to a fault-free execution, thereby providing fault tolerance.

We then show that RAM algorithms, external memory algorithms, and cache-oblivious algorithms [6] can all be implemented asymptotically efficiently in the model. Our technique is a simulation that breaks the computation into contiguous segments. For each segment we create two capsules: one to perform the computation with output redirected to a scratch space, and one to move the data from the scratch space to its original destination. However, the simulation is likely not practical. We therefore consider a programming methodology in which the algorithm designer can identify capsule boundaries to avoid write-after-read conflicts.

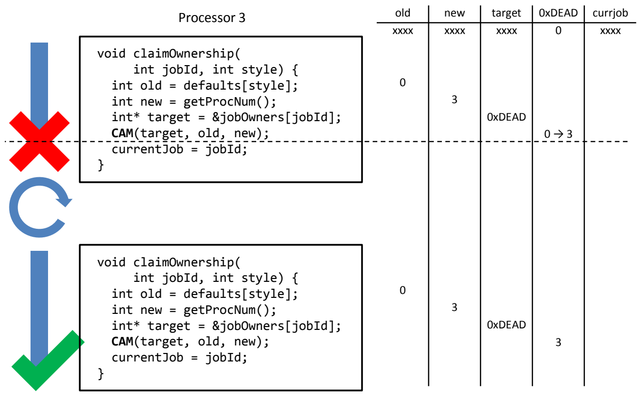
Our methodology has capsules begin and end at the boundaries of certain function calls, which we call *persistent function calls*. Once a persistent call is made, the callee will never revert back further than the call itself, and after a return the caller will never revert back further than the return. Persistent call boundaries require a constant number of external reads and writes. In a non-persistent function call faults can cause a roll back beyond the function boundaries. All non-persistent calls are handled completely in the ephemeral memory, without external reads or writes.

In addition, we assume a `commit` command that forces a capsule boundary. As with a persistent call, commit requires only a constant number of external reads and writes. We assume that all user code between persistent boundaries is idempotent, which can be achieved using a style of programming in which results are copied instead of overwritten.

The Parallel-PM Model. We then consider our multiprocessor counterpart, the *Parallel-PM*, and consider conditions under which programs are correct when the processors are interacting through the shared memory. We show that if capsules are "atomically idempotent", then each capsule acts as if it ran once despite many possible restarts.

We define a history as an interleaving of the persistent memory instructions from each of the processors and say that a capsule in a history is *atomically idempotent* if:

1. (atomic) all of its instructions can be moved in the history to be adjacent somewhere between the instruction that installs the capsule's restart pointer and the instruction that installs the next capsule's restart pointer without violating the memory semantics, and



```
void claimOwnership(
     int jobId, int style) {
  int old = defaults[style];
  int new = getProcNum();
  int* target = &jobOwners[jobId];
  CAM(target, old, new);
  currentJob = jobId;
}
```

```
void claimOwnership(
     int jobId, int style) {
  int old = defaults[style];
  int new = getProcNum();
  int* target = &jobOwners[jobId];
  CAM(target, old, new);
  currentJob = jobId;
}
```

Figure 1. An idempotent capsule containing a CAM. Earlier faulting runs may alter the shared memory is ways visible to the rest of the system (here, location 0xDEAD is set to 3 right before the processor faults). As long as no concurrent capsule causes an instance of the ABA problem, the faulting runs do not affect idempotence.

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.

We show that any capsule with no write-after-read conflicts and no races is atomically idempotent. This means that such capsules easily support fault tolerance. For many programs, however, the requirement of being race free is too restrictive. We identify several categories of capsules that safely contain races while maintaining idempotence.

We also note that a compare-and-swap (CAS) instruction is not safe in the Parallel-PM, but that a compare-and-modify (CAM), which is identical to a CAS except that it does not return a result, is safe. Figure 1 shows an example of an idempotent capsule that contains a racy CAM instruction.

Work Stealing. The most significant result in this work is a work-stealing scheduler that can be used on the Parallel-PM. Our scheduler is based on that of Arora, Blumofe, and Plaxton (ABP) [2]. The key challenges in adopting ABP to handle faults are (i) properly using CAMs in place of CASs, (ii) ensuring that each stolen task gets executed exactly once regardless of faults and restarts, (iii) properly handling hard faults, and (iv) preserving efficiency in the presence of faults.

To avoid blocking on a process that faulted in the middle of stealing, failed steal attempts must help any steal that is in progress. Furthermore, processors that suffer a hard fault part way through executing a thread must not cause the computation to fail. Accordingly, the threads that are being processed must be exposed in a way such that they can be stolen in the event that their owner faults. Stealing these in-process threads is complicated by the fact that the computation must be resumed at the capsule that was in progress when the fault occurred.

We show that any race-free, write-after-read conflict free fork-join program with work W, depth D, and maximum capsule work C will run on our scheduler 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 ABP only in the $\log_{1/(Cf)} W$ factor on the depth term, due to faults along the critical path.

Algorithms. We also 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 [4]. The main change is ensuring that partial results are written to separate locations from where they are read, in order to avoid write-after-read conflicts.

Write-back Caches. Finally, 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, 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. Because locations in local memory are invisible to other processors, this does not affect correctness.

Full Paper. See [3] for the complete paper.

Acknowledgements

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

References

[1] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2008.

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2), Apr. 2001.

[3] G. E. Blelloch, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. The parallel persistent memory model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[4] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010.

[5] F. Cappello, G. Al, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomputing Frontiers and Innovations*, 1(1), Apr. 2014.

[6] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1999.

[7] J. S. Meena, S. M. Sze, U. Chand, and T.-Y. Tseng. Overview of emerging nonvolatile memory technologies. *Nanoscale Research Letters*, 9, 2014.

[8] F. Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey III, D. R. Chakrabarti, and M. L. Scott. Dali: A periodically persistent hash map. In *International Symposium on Distributed Computing (DISC)*, 2017.