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 [3]UC Riverside

In this work, we present 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 [1, 2], each processor runs a standard instruction set on 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 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 [6, 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 [4].

In this work, 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 contiguous sections known as "capsules" that have no write-after-read conflicts (overwriting input data) results in idempotent behavior. When combined with our approach of restarting faulting capsules from their beginning using a restart pointer saved in persistent memory, this is sufficient. We then show that RAM algorithms, external memory algorithms, and cache-oblivious algorithms [5] can all be implemented asymptotically efficiently in the model. Our technique is a simulation that breaks the computations into capsules, with writes buffered in their original capsule and handled in the next. However, the simulation is likely not practical. We therefore consider a programming methodology in which the algorithm designer can identify capsule boundaries that ensure 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

*This work summarizes a paper at SPAA 2018 with the same name.

model, but that a compare-and-modify (CAM), which does not see its result, is safe.

The most significant result in the work 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) [2]. 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) preserving efficiency in the presence of soft or hard faults. To handle faults in a manner that avoids blocking without the use of CAS, failed steal attempts must help the processor that is part way through a steal. Furthermore, processors must be able to steal a thread from a processor that suffered a hard fault part way through executing a thread.

Using our 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 [3]. 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.

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, 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. Since locations in local memory are invisible to other processors, this does not affect correctness.

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, and H. V. Simhadri. Low depth cache-oblivious algorithms. In *ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2010.

[4] 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.

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

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

[7] 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.

[8] Yole Developpement. Emerging non-volatile memory technologies, 2013.