The Parallel Persistent Memory Model

Guy E. Blelloch\textsuperscript{1} Phillip B. Gibbons\textsuperscript{1} Yan Gu\textsuperscript{3} Charles McGuffey\textsuperscript{1} Julian Shun\textsuperscript{2}
\textsuperscript{1}Carnegie Mellon University \textsuperscript{2}MIT CSAIL \textsuperscript{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), 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 equivalence 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
new
xxxx
= xxxx
jobId
jobId
xxxx
0
target = &
currjob
0xDEAD
jobId
old new target 0xDEAD currjob
xxxx xxxx xxxx 0 xxxx
0
3
0xDEAD
0 → 3
0
3
0xDEAD
3
Processor 3
void claimOwnership() {
    int jobId; 
    int old = defaults[style]; 
    int new = getProcNum(); 
    int* target = &jobOwners[jobId]; 
    CAM(target, old, new); 
    currentJob = jobId; 
}