Use Cases for Large Memory Appliance/Burst Buffer

Lawrence Livermore National Laboratory

Rob Neely
Bert Still
Ian Karlin
Adam Bertsch

LLNL-PRES-648613

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344.
Lawrence Livermore National Security, LLC

Assumptions in these use cases

- Large/huge amounts of memory are directly available to applications.
- Access (latency and bandwidth) to this memory is slower than to regular “main memory”.
- Memory may or may not be persistent (few of these use-cases rely on it)
- There may or may not be dedicated compute resources near/ in the memory
- Memory is possibly shared by multiple compute nodes
- Access is either via block device interface or byte addressed interface (use-case dependent)

Each use case that follows lists a “priority”. This is our best attempt at guessing how likely we would be use pursue this case, and is the opinion of LLNL only.

Use case #1: Defensive I/O (checkpointing) Priority: 10/10

Use for fast checkpointing using standard techniques

- Use LLNL SCR (Scalable Checkpoint-Restart) package or System API
  - System API must maintain low barrier to entry for apps
  - SCR uses Memory-mapped files
- Drain every Nth checkpoint to filesystem asynchronously
- Use for regular defensive I/O
- If compute were attached, could have a user-defined management thread running asynchronously
- Allows for rapid rollback/recovery to be implemented in code

Pros:
- Concept and software (SCR) exist today
- Low barrier to entry for apps
- Non-volatile memory allows for potential fast restart if job is re-launched

Cons:
- Taking advantage of non-volatile properties would require re-launch on same set of nodes (or clever shuffling/ unified view of data)
- POSIX filesystem security required to protect NV data

Use case #2: Visualization I/O
Priority: 9/10

Use for fast output using standard techniques

- Drain plot files to filesystem asynchronously
- If compute were attached, could have a user-defined management thread (e.g. generate movie frame) running asynchronously
- Allows for asynchronous use of permanent storage

Pros:
- Should co-exist with defensive I/O case
- Asynchronous use of permanent storage improves ratio of time spent on compute
- Asynchronous computation (if available) could save a lot of time and space in generating plots

Cons:
- Managing asynchronous computation (if available) could add complexity to workflow

Use case #3: Accelerated reads
Priority: 8/10

Use for fast application startup using standard read techniques

- Stage data in advance of application requirements/launch
- Use for large or commonly read data, such as
  - Shared libraries
  - Input decks
  - Material databases

Pros:
- Probably a low barrier to entry for apps
- Some effort in place (SPINDLE) at LLNL to automate some of this for dynamic libraries

Cons:
- Not clear how storage management would be handled (staging, persistence, …)

Use case #4: In-situ visualization
Priority: 6/10

Do "on the fly" staged visualization and/or analysis

- Keep copy of application state in memory for in-situ processing
- Feature extraction / topological analysis
- Preprocessing or data reduction before writing to long term storage
- Create movie frames asynchronously while calculation continues
- Pause simulation and examine
- If compute is available on storage, process asynchronously

Pros:
- Existing research (e.g. VisIt) could take advantage
- Greatly reduces amount of state written to disk

Cons:
- Asynchrony provides performance gains, but adds complexity
- Research required before production use

Use case #5: Data lookup server
Priority: 4/10

Storage of large shared databases/tables

- Easily allow sharing of large data tables (e.g. EOS, cross-sections, opacities) across multiple nodes
- With attached compute, can do calculations (e.g. interpolations) locally and minimize data motion
- Large storage allows for additional pre-calculation of lookup coefficients, derivatives, etc… Faster lookup times

Pros:
- Clear breakdown of client/ server roles – well-understood programming model
- Tables could be preinitialized and kept in memory across runs – avoid reinitialization costs.

Cons:
- Bursty access will align across nodes in an SPMD model.
- Potential for hot spots and/or not enough compute
- May not be enough room to overlap request/use to make async access pay off

Use case #6: Large out-of-core storage
Priority: 3/10

Use as extension of main memory

- Allow for greater physics fidelity
  - For example, greatly increase the number of particles in a Monte Carlo simulation, staging them into "near" memory in a pipelined fashion
- "Near" memory becomes another level of cache – could be managed automatically through the OS via small pages
  - E.g DI-MMAP project at LLNL
- Reduce dynamic memory management
  - Just keep temporary arrays around
  - Malloc/free are slow and usually done in serial (non-threaded) sections of code

Pros:
- Memory-capacity intensive applications not required to strong-scale as much
- Cache-approach would be largely invisible to the user

Cons:
- Programmer-managed placement and movement between memory is as-yet undefined
- Many problems will want additional compute resources to go with all that additional memory

Use case #7: Speculative Execution
Priority: 2/10

Keep micro-checkpoints in memory to support transaction-like rollback

- Most multi-physics codes have physics-based triggers that will cause the code to self-abort. E.g.
  - Over-advection of materials
  - Failure to converge in a solver
- Allowing a code to rollback to an earlier part of the timestep would allow for a re-try
  - E.g. Loosen constraints on the problem, reduce timestep, etc…

Pros:
- More robust code behavior
- Much smaller and faster than a full checkpoint/restart

Cons:
- Memory capacity may not be what's preventing us from doing this today. It's just simply hard to implement

Use case #8: Master task / worker queue model Priority: 1/10

Keep state in NVRAM, fire off worker-tasks on nodes

- Task queue managed on NVRAM node – assumes some level of compute
- Potentially good model for older/legacy codes that did lots of work on task 0

Pros:
- Good for priority task queue designed software

Cons:
- Tasks must be long-running enough to absorb latency cost of spawning
- Still must manage distributed state of task queue at scale

Open Questions

- How does one access data in NVRAM?
  - Block or byte addressable?
  - Memory map?
  - Other?
- What is the security model, to ensure persistent data in NVRAM can be accessed by the user in subsequent runs, but not by other users?
  - Perhaps solvable with block-storage via file system permissions. Much harder problem for byte-addressable storage models
  - This is a big-deal issue for machines running in a classified environment
- What is the potential impact of accessing data in a memory space largely dedicated to another user?
- What's a notional ratio of memory between "near" and "far"?
- If we use NVRAM simply as additional capacity without commensurate boost in compute, aren't we greatly exacerbating the data motion / memory bandwidth bottleneck?
- For the UQ example, what's the advantage over having a global view of the data set on disk?