Data and Next Generation Scalable Applications

Michael A. Heroux, Sandia National Laboratories

Large-scale parallel applications have traditionally stored and retrieved persistent data using logically bulk-synchronous interfaces and models. An emerging application architecture based on task-centric/dataflow concepts will require different interfaces, execution patterns and state management for persistent data and may suggest disruptive and complementary persistent data system architecture changes.

Task-centric/dataflow Application Architectures

Task-centric/dataflow application architectures are based on the over-decomposition of global application data structures relative to the number of computing cores.

A *patch* is a logically cohesive collection of application data such that the computational models of the application can be expressed locally on a patch. Examples include creating many sub-domains for the spatial domain of a PDE application, or a set of graph partitions for the connectivity graph of an inherently discrete application such as circuits. In most cases, except for some compute-rich, regular grid PDE applications, data should be allocated for each patch contiguously in memory, independent of data for other patches, for improved cache and memory access performance.

A *task* is a unit of computation defined on a patch. Tasks include the setup and evaluation of the computational model on a patch, for example the evaluation of a stencil computation on a subdomain. Application execution is then the aggregate of task computations combined with parallel computational patterns that treat task computations as generic operations.

Domain scientists write functionality for the application solely in terms of task code. Task source code should expose vectorization to the compiler, if the computational and data access patterns permit, and should support modest thread parallelism on multicore processors with a shared low-level cache. The domain scientist must also encode inter-task dependencies. These dependencies can be expressed in terms of futures, input and output variable dependencies, or through explicit dependency graph management.

Task-centric/dataflow architectures have many advantages:

1. **SPMD compatibility:** To the domain scientist, this model is very similar to the classic SPMD approach.
2. **Separation of domain science and parallel execution concerns:** Permits use of Fortran, C and OpenMP for writing task source code, while the task

management framework can, and will likely, be written using other programming environments that provide more natural parallel expressiveness.

3. **Enables manytasking execution models:** Tasks can be launched asynchronously, providing natural latency hiding, higher network injection rates and better dynamic load balancing.
4. **Resilience:** Each task has a parent such that, if a task fails or times out, the parent can regenerate the child. Parent data and execution integrity can be calibrated to match the robustness of the system.
5. **Accelerators:** Patch sizes can be calibrated for current and future target computing systems. For accelerators, patch sizes should be very large, permitting the accelerator runtime system (e.g., the thread block manager of a GPU) to manage work partitioning and latency hiding.
6. **Heterogeneous systems**: Patches and tasks can be of various types and scheduled by the task management layer to use heterogeneous processors if available and appropriate on a given system.
7. **Universal portability**: This architecture is adaptable to current and future computing systems.

Implications for data management

Task-centric/dataflow applications present new requirements for data management systems. In particular:

1. **Read and write functions must be task-compatible:** Data read and write operations will be performed by asynchronously executing tasks. There will be many task instances writing data dynamically. Concerns include thread-safety and thread-scalable execution and protocols for determining data integrity.
2. **Data versioning:** Persistent storage facilities must support versioning of data. When tasks execute asynchronously, several versions of a variable may exist across tasks. For example, in a transient simulation a velocity value at the grid point (i,j,k) may exist at different time steps during execution. Therefore storage and retrieval of a value must support multiple version of variable state.
3. **Global operations:** Task reads and writes are primarily local events. Coordinating these events for global objects will be more complicated. In particular, knowing when all stores to a global data object are complete represents a kind of Byzantine generals problem, where determining completion of the operation requires explicit coordination.

Some next generation extreme-scale applications will have very different execution and data access patterns. Control flow will be more localized and execution will be dynamic and asynchronous. These changes will lead to increased demands on persistent data management systems and may lead us to consider new data management architectures with similar behaviors and traits.