The “Tilearchy” Programming Model

Charles R. Ferenbaugh
DOE Programming Environments Workshop
March 10, 2015
Part I
A Brief Overview of the Tilearchy

(Thanks to Eric Nelson for some of these slides)
What is this “tilearchy,” anyway?

- A model developed in LANL’s SoftWare Infrastructure for Future Technology (SWIFT) Project, 2011-2013

- A conceptual programming model for describing:
  - Hierarchical parallelism
  - Hierarchical data decomposition
  - Separation of physics kernels (algorithms) from CS drivers (portability/performance)

- Not a DSL, runtime library, or compiler technology
  - Implemented in SWIFT using explicit coding in C++, MPI, OpenMP, some OpenCL (and even some assembly language)
The basic idea: Match computation and data storage to hardware, OS hierarchy

- processing hardware
- software operating context
- SWIFT integrated code computation and storage hierarchy
- storage hardware

- machine
- job
- mesh/domain
- machine disk
- node disk
- node main memory
- NUMA domain main memory
- L3 cache
- L2 cache
- L1 cache
- vector register

- node
- process
- subdomain
- core
- thread
- tile
- subtile/chunk
- vector
- vector unit

UNCLASSIFIED
The basic idea: Match computation and data storage to hardware, OS hierarchy
Full problem domain is partitioned into subdomains

- Similar to what we currently do for MPI ranks, except:
  - only one rank per NUMA domain
  - not necessarily executed in bulk-synchronous mode
Subdomains are partitioned into tiles

- **Subdomain driver loops over tiles (threaded)**
- **Code at the tile level and below is single-threaded**
Tiles are sized to make best use of the memory hierarchy

- Tile size is chosen to (usually) fit in L2 cache
- Tiles may be divided into subtiles that fit in L1 cache, if needed
Kernels are executed on tiles

- Tile/subtile drivers execute highly optimized algorithm kernels on data in cache.
Tilearchery advantages beyond performance

- Driver/kernel separation facilitates development
  - Tile drivers and kernels written by domain scientists
  - Higher-level drivers written by CS/architecture experts
  - Nearly all performance and portability features (parallelism, fault-tolerance, …) reside in high-level drivers
    - Main exception: vectorization is in kernels

- Functional model
  - Kernels must be stateless, with separate inputs and outputs
  - Aids reasoning about composition, fault-tolerance, …

- Central data store is tile-based, not global
  - Supports data locality, migration, …
TN burn app scales to all 136288 cores on Cielo, 1.7T zones at 0.36 Pflop/s (28% of peak)

10K zone tiles, 360 zone subtiles

- 8518 nodes, to 1.70T zones
- 6500 nodes, to 1.43T zones
- 4096 nodes, to 885G zones
- 1024 nodes, to 221G zones
- 256 nodes, to 55.3G zones
- 64 nodes, to 13.8G zones
- 16 nodes, to 3.46G zones
- 4 nodes, to 864M zones
- 1 nodes, to 216M zones
Tilearchy examples

- Main SWIFT demo app: TN burn
  - Full app prototyped
  - Scales to all 133K cores of Cielo, runs at 28% of peak

- Other SWIFT apps: cell-centered hydro, rad diffusion
  - Lower-level drivers prototyped
  - Not developed as far as TN Burn, but enough for proof of concept

- ENAMR (Evaluating a New AMR)
  - Experiment underway now to extend tilearchy to Eulerian AMR
  - If successful, this approach will be used to modernize the ASC XRage code
Part II
Some Answers to Workshop Questions
(inspired by Tilearchy experience)
How are programming models differentiated from programming environments?

My (partial) answer:
Model is conceptual, environment is concrete…

<table>
<thead>
<tr>
<th>Programming model (conceptual)</th>
<th>Low-level Implementation</th>
<th>Higher-level Implementation (part of environment)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed parallel, message passing</td>
<td>TCP/IP, IBVerbs, …</td>
<td>MPI</td>
</tr>
<tr>
<td>Shared-memory parallel</td>
<td>Pthreads, PTX, …</td>
<td>OpenMP, OpenACC; RAJA, Kokkos, …</td>
</tr>
<tr>
<td>Hierarchical parallelism and data decomposition (tilearchy)</td>
<td>MPI + (OpenMP, OpenCL, CUDA, …)</td>
<td>[To be determined]</td>
</tr>
</tbody>
</table>
What are new abstractions for parallelism at exascale? How should parallelism be identified and concurrency managed?

- Need a way to naturally represent processing hierarchy
- Some approaches that do this:
  - OpenMP 4.0 (threads/teams/league)
  - OpenACC (workers/gangs)
  - Sequoia – Stanford
  - Hierarchically Tiled Arrays – Illinois
  - Tiling as a Durable Abstraction (TiDA) – LBNL
- Some (most?) of these have only a fixed set of levels – can they be generalized?
What are programming abstractions to represent data and its distribution?

- OpenMP 4.0, OpenACC have partial solutions
  - Distinction between host/device memories
  - Limited ability to specify tiling/locality
  - Needs to be more general?

- Some other approaches may be more complete
  - Kokkos (Tile<M, N>)
  - Sequoia
  - Hierarchically Tiled Arrays
  - TiDA
How should PM/E represent persistent objects and the storage system?

- Represent and store persistent objects in a hierarchical way, to match the data hierarchy.
- In particular, don’t assume that all problem data is stored in a single contiguous array!
  - Not even within a single rank, thread, or NUMA domain
  - For example, in SWIFT’s implementation, data is contiguous only within each individual tile
  - So we need persistence/storage interfaces that don’t require contiguous data (e.g., not POSIX fread/fwrite)
Are there innovative ideas for integrating resilience and debugging into the programming model?

- Resilience should be primarily a driver-level property, not kernel-level
  - Don’t burden the domain scientists with it
  - Have well-defined bounds on what code is executed, what data is modified by each driver
  - This means driver writers can reason more about how to recover from faults

- Debugging should be easier with driver/kernel separation
  - Debugging of algorithms mostly at kernel level
  - Debugging of CS implementation issues mostly at driver level
Are there lessons to be learned from other communities that we can apply?

- Find flexible ways to store/access state data in central DataStore
- Database community may be able to help here
  - SWIFT experimented with a relational database model; this merits further investigation
Questions?

cferenba@lanl.gov