Sharing Thread Pools and Caches for Inter-library Composition and Multicore Performance

**Jed Brown**, Shrirang Abhyankar, Barry Smith

Mathematics and Computer Science Division, Argonne National Laboratory

SIAM CSE, 2013-02-28
- Parallel computing used to be about computing.
- It's increasingly about data movement.
Parallel computing used to be about computing.
It’s increasingly about data movement.
Libraries and threads

- **Purpose of threads**
  - Reduce memory usage for executable code and shared/global data structures
  - Reduce resource contention (network, filesystem)
  - Encourage cache and bandwidth sharing

- **Different ways to use threads**
  - Large dense linear algebra: use threads internally. User only interacts with serial interface.
  - OMP parallel at `main` and shared nothing by default.
  - `MPI_Comm_split_type()` and `MPI_Win_allocate_shared()`

- **Competing standards: OpenMP, TBB, Pthreads, OpenCL, ...**
  - Targeted at applications, not libraries
  - Poor support for sharing

- Unfriendly to require `MPI_THREAD_MULTIPLE`
Maintaining libraries

- PETSc developers receive about 100 user messages per day
  - Configuration/installation (with broken environment)
  - API (mis)usage
  - Understanding performance/variability
  - Solver convergence, selection of methods, and
- > 10% of PETSc is pure input validation and debuggability
  - Diagnose bugs in user code over email from our error messages
  - Valgrind-like memory tracing and sentinels, explicit stack for signal handlers, pointer testing
  - Compiled out in optimized builds
- 3% of PETSc is profiling/performance diagnostics
- Memory-related performance problems are difficult to debug
- Thread placement and affinity is fragile
Maintaining libraries

- PETSc developers receive about 100 user messages per day
  - Configuration/installation (with broken environment)
  - API (mis)usage
  - Understanding performance/variability
  - Solver convergence, selection of methods, and
- $>10\%$ of PETSc is pure input validation and debuggability
  - Diagnose bugs in user code over email from our error messages
  - Valgrind-like memory tracing and sentinels, explicit stack for signal handlers, pointer testing
  - Compiled out in optimized builds
- $3\%$ of PETSc is profiling/performance diagnostics
- Memory-related performance problems are **difficult to debug**
- Thread placement and affinity is fragile
Maintaining libraries

- PETSc developers receive about 100 user messages per day
  - Configuration/installation (with broken environment)
  - API (mis)usage
  - Understanding performance/variability
  - Solver convergence, selection of methods, and
- > 10% of PETSc is pure input validation and debuggability
  - Diagnose bugs in user code over email from our error messages
  - Valgrind-like memory tracing and sentinels, explicit stack for signal handlers, pointer testing
  - Compiled out in optimized builds
- 3% of PETSc is profiling/performance diagnostics
- Memory-related performance problems are difficult to debug
- Thread placement and affinity is fragile
Virtual addressing and “first touch”

- Virtual memory is unavoidable for NUMA with shared memory programming.
  - Can speculate that Blue Gene/Q is UMA because of TLB allergies (preference for offset-mapped shared memory)

- Most systems with virtual memory do not find physical pages when you call `malloc()`.

- The kernel finds physical pages when you trigger a page fault, usually “close to” the thread causing the page fault.

- cache and TLB information (2):
  - 0x5a: data TLB: 2M/4M pages, 4-way, 32 entries
  - 0x03: data TLB: 4K pages, 4-way, 64 entries

- Inspecting or changing location of physical pages is not portable (`hwloc` does the best they can).

- Implicitness is bad for libraries and bad for support
Virtual addressing and “first touch”

- Virtual memory is unavoidable for NUMA with shared memory programming.
  - Can speculate that Blue Gene/Q is UMA because of TLB allergies (preference for offset-mapped shared memory)
- Most systems with virtual memory do not find physical pages when you call malloc().
- The kernel finds physical pages when you trigger a page fault, usually “close to” the thread causing the page fault.
- cache and TLB information (2):
  - 0x5a: data TLB: 2M/4M pages, 4-way, 32 entries
  - 0x03: data TLB: 4K pages, 4-way, 64 entries
- Inspecting or changing location of physical pages is not portable (hwloc does the best they can).
- Implicitness is bad for libraries and bad for support
What can go wrong?

- Memory performance depends on socket connectivity
- Unbalanced prior allocations
- Cache coherence costs (e.g., STREAM at 50% of bus bandwidth)
- Thread can migrate away
- Linux-2.6.38 has transparent huge pages (2M/4M versus 4K)
- `libhugetlbfs` not widely installed
With all these problems, why use common allocation?

- Data structures are simpler, smaller, and share more easily.
  - Consider sparse matrix-matrix multiply
- Cache/bandwidth sharing are key reasons for threads in the first place
- Compatibility with user expectation
- Ability to mix optimized threaded code with legacy unthreaded
- Separate allocation is sometimes feasible and can work very well
Speed of light and cost of synchronization

- Fundamental lower bound: several clock cycles for light to make round trip across an Ivy Bridge die

<table>
<thead>
<tr>
<th>Operation (16-CPU X5550 Nehalem)</th>
<th>Time (ns)</th>
<th>Clocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock period (two packed FP instructions)</td>
<td>0.4</td>
<td>1</td>
</tr>
<tr>
<td>Best case CAS</td>
<td>12.2</td>
<td>33.8</td>
</tr>
<tr>
<td>Best-case lock/unlock</td>
<td>25.6</td>
<td>71.2</td>
</tr>
<tr>
<td>Single cache miss</td>
<td>12.9</td>
<td>35.8</td>
</tr>
<tr>
<td>CAS cache miss</td>
<td>7.0</td>
<td>19.4</td>
</tr>
<tr>
<td>Single cache-miss (off-core)</td>
<td>31.2</td>
<td>86.6</td>
</tr>
<tr>
<td>CAS cache miss (off-core)</td>
<td>31.2</td>
<td>86.5</td>
</tr>
<tr>
<td>Single cache miss (off-socket)</td>
<td>92.4</td>
<td>256.7</td>
</tr>
<tr>
<td>CAS cache miss (off-socket)</td>
<td>95.9</td>
<td>266.4</td>
</tr>
</tbody>
</table>

- From Paul McKenney, see
  http://www.rdrop.com/users/paulmck/RCU/
Synchronization mechanisms

- OpenMP over-synchronizes by default
- OMP `nowait` clause is global and of limited utility
- OMP `critical` is fundamentally not scalable
- OMP `atomic` cause excessive cache-line bouncing
- Collectives like `allreduce` and `scan` can be scalable
- Mechanisms like RCU (Read-Copy Update) allow safe, mostly-unstructured asynchronous shared mutable state
- TBB synchronization is either non-scalable (e.g., mutexes) or tightly coupled to tasks
Will Transactional Memory save the day?

- TM is good for large scattered writes over data structures that cannot be partitioned.
- TM is expensive relative to locks for small writes
- Implementations and performance is highly variable
- Non-idempotent operations may be applied multiple times on retry
- McKenney, Michael, Triplett, Walpole (2010) “Why the grass may not be greener on the other side: A comparison of locking versus transactional memory”.
PETSc: “E” is for *Extensible*

- **Ideal:** anything that can be developed in the library can also be developed as a plugin.
  - Matrix and vector formats
  - Preconditioners
  - Krylov methods
- **High-level plugins** do not want to think about threads
- **Low-level plugins** need low-level access
- **Ability to call internal functions** *from threads*
Thread Communicator design goals

- Run-time choice of common threading environments
- Ability to split communicators
- Non-blocking job submission of collective jobs (perhaps on subcomms)
- Thread collectives like reductions and scans decoupled from tasks
- Collective asynchronous and synchronous jobs
- Avoid over-synchronization: hazard pointers, RCU (unfortunately a patent minefield for non-LGPL)
- Library isolation, attribute caching
PetscThreadComm

- Attached to MPI_Comm which is used in existing interfaces
- Split and dup based on topology
  - runs asynchronously if thread rank 0 is not in comm
- Asynchronous reductions:

  ```c
  void VecDot_k(int thread_id, Vec X, Vec Y, PetscThreadReduction red) {
      int rstart, rend;
      const Scalar *x, *y;
      VecGetThreadOwnershipRange(X, thread_id, &rstart, &rend);
      VecGetArrayRead(X, &x);
      VecGetArrayRead(Y, &y);
      Scalar a = BLASdot_(x[rstart:rstart+rend], y[rstart:rstart+rend]);
      PetscThreadReductionPost_k(thread_id, red, &a);
  }
  void VecDot(Vec X, Vec Y, Scalar *a) {
      ...
      PetscCommRunKernel3(X->comm, VecDot_k, X, Y, red);
      PetscThreadReductionEnd(red, a); // or PetscThreadReductionEnd_k()
  }
  ```

- Can also call VecDot_k() from another kernel
Expressing memory layout

- PETSc vectors and matrices have PetscLayout
- Provides sufficient local view of distribution across distributed memory
- Now includes thread ownership ranges
- Implementations can extend to richer descriptions
  - Grouping and interlacing
## Hardware Arithmetic Intensity

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arithmetic Intensity (flops/B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sparse matrix-vector product</td>
<td>1/6</td>
</tr>
<tr>
<td>Dense matrix-vector product</td>
<td>1/4</td>
</tr>
<tr>
<td>Unassembled matrix-vector product</td>
<td>≈ 8</td>
</tr>
<tr>
<td>High-order residual evaluation</td>
<td>&gt; 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor</th>
<th>BW (GB/s)</th>
<th>Peak (GF/s)</th>
<th>Balanced AI (F/B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E5-2670 8-core</td>
<td>35</td>
<td>166</td>
<td>4.7</td>
</tr>
<tr>
<td>Magny Cours 16-core</td>
<td>49</td>
<td>281</td>
<td>5.7</td>
</tr>
<tr>
<td>Blue Gene/Q node</td>
<td>43</td>
<td>205</td>
<td>4.8</td>
</tr>
<tr>
<td>Tesla M2090</td>
<td>120</td>
<td>665</td>
<td>5.5</td>
</tr>
<tr>
<td>Kepler K20Xm</td>
<td>160</td>
<td>1310</td>
<td>8.2</td>
</tr>
<tr>
<td>Xeon Phi</td>
<td>150</td>
<td>1248</td>
<td>8.3</td>
</tr>
</tbody>
</table>
Performance of assembled versus unassembled

- High order Jacobian stored unassembled using coefficients at quadrature points, can use local AD
- Choose approximation order at run-time, independent for each field
- Precondition high order using assembled lowest order method
- Implementation > 70% of FPU peak, SpMV bandwidth wall < 4%
Reducing memory bandwidth

- Sweep through “coarse” grid with moving window
- Zoom in on new slab, construct fine grid “window” in-cache
- Interpolate to new fine grid, apply pipelined smoother ($s$-step)
- Compute residual, accumulate restriction of state and residual into coarse grid, expire slab from window
Arithmetic intensity of sweeping visit

- Assume 3D cell-centered, 7-point stencil
- 14 flops/cell for second order interpolation
- $\geq 15$ flops/cell for fine-grid residual or point smoother
- 2 flops/cell to enforce coarse-grid compatibility
- 2 flops/cell for plane restriction
- Assume coarse grid points are reused in cache
- Fused visit reads $u^H$ and writes $\hat{I}_h^H u^h$ and $I_h^H r^h$
- Arithmetic Intensity

\[
\begin{align*}
\text{interp} & + \underbrace{2 \cdot (15 + 2)}_{\text{compatible relaxation}} + \underbrace{2 \cdot 15}_{\text{smooth}} + \underbrace{15}_{\text{residual}} + \underbrace{2}_{\text{restrict}} \\
& \geq 30 \quad (1)
\end{align*}
\]

- Still $\geq 10$ with non-compressible fine-grid forcing
Outlook

- PetscThreadComm with pthreads lower overhead than OpenMP
  - Weaker synchronization, fewer memory fences
- Enable better reuse of “kernels”
- Thread organization more explicit, can cross library boundaries
- Performance and correctness debuggability via email/error messages
- Allow transition from calling via outer interfaces to calling from threads
- Matrix-free methods reduce bandwidth requirements
  - can simplify memory management, but the user is no longer isolated from solvers
- Exotic algorithms can move us back to FPU-limited
  - Don’t have to worry so much about memory
  - Such algorithms are often cache-intensive so need to share
- Portable Hardware Locality
  http://open-mpi.org/projects/hwloc
- Concurrency Kit http://concurrencykit.org