

CMSC 714

Lecture 7

Titanium and MPI vs. OpenMP

Alan Sussman

Notes

- OpenMP project due Monday, March 8
 - Use the same account info for the Sun SMP (tau.umiacs.umd.edu) as for your bug cluster account
 - don't try until after class!
- More readings posted later today
 - send email with preferences – first shot to those (few) of you who haven't done 1 yet

Titanium

Titanium Features

- Based on Java, so object-oriented
 - Easy to extend, since (relatively) small and clean
 - Easy to learn (if you know C/C++/Java)
 - Safe language – better for programmer and compiler
- Explicitly parallel
- SPMD execution model
- Global address space
- Zone-based memory management
- Runs on both shared-memory and distributed-memory parallel architectures – with different language features performing better on different architectures
- Compiler implementation translates to C

Titanium Goals

- **Performance**
 - Targeted at high-end scientific applications – but still not too many of these written in Java ...
 - Programmer has control over parallelism (no heroic compiler), and distribution of data across processes
 - Type modifiers for variables to specify locality that may be hard to find with a compiler
- **Safety**
 - Static detection of errors - from Java plus execution model
 - Accurate reporting of runtime errors – mainly from Java exception model
- **Expressiveness**
 - Real multi-dimensional arrays and iterators, point and index sets are first class objects, and global references (pointers)
 - Less important than the first 2

Language Extensions

- **Immutable classes**
 - Don't extend any class (not even Object), and can't be extended
 - All non-static fields final
 - So compiler can pass by value and stack (not heap) allocate
 - Makes these objects look like Java primitive types or C structs
- **Parallelism**
 - Global synchronization – a barrier (see Figure 1 in paper)
 - Correct synchronization checked by compiler, using single-value variable analysis
 - Local and global references
 - Object is allocated in a region (1 per process)
 - Local variables and dynamically allocated objects are in region of process that allocates them
 - Refs to objects in other regions obtained via comm. primitives
 - Default for refs is global, can declare that a variable can only point to local objects with **local** modifier

Language Extensions (cont.)

- **Communication**
 - Via reads and writes of object fields, or cloning an object, or copying array parts
 - *Broadcast* method for one-to-all, *exchange* method for all-to-all, and both require global synchronization
 - *Barrier* method for global synchronization
 - Synchronization between processes uses Java *synchronized* methods and statements
- **Zone-based memory management**
 - Allocations into zones
 - Entire zone freed with one explicit *delete-zone* operation
 - Reference counts used by runtime system to keep a zone from being deleted while there are still refs into the zone
 - Other papers for sequential C programs show this is faster than malloc/free and garbage collection, in most cases

Language Extensions (cont.)

- **Arrays, points, domains**
 - True multi-dimensional arrays, unlike Java
 - Constructed using *domains* for index set, and indexed by *points* (not lists of integers)
 - A *point* is an integer tuple and a *domain* is a set of points

```
// Construct a 2D array
Point<2> l = [1, 1];
Point<2> r = [10, 20];
RectDomain<2> r = [l : u];
double [2d] A = new double[r];

// Iterate over the array elements
foreach (p in A.domain()) {
    A[p] = 42;
}
```

Applications

- **AMR3D**
 - Adaptive mesh refinement Poisson solver
 - Multiple nested grids cover the domain, and can change over time
 - Need dynamic load balancing to keep number of points per process near equal
- **EM3D**
 - Kernel from application that models EM wave propagation through objects in 3D
 - Turns into a bipartite grid/graph of electric and magnetic field nodes
 - each point is computed as a function of its neighbors, alternating E and M nodes
- **Overall performance of Titanium code is comparable to C/C++/Fortran code versions, for small problems on Sparc and Pentium machines**
 - They don't do a good job of comparing against other systems because of the weaknesses of the other compilers
 - EM3D shows perfect speedup, and AMR3D shows some parallel speedup, on 8-way Sparc SMP – don't show absolute times (probably worse for parallel versions only running on 1 processor compared to sequential versions)

OpenMP vs. MPI

OpenMP + MPI

- **Some applications can take advantage of both message passing and threads**
 - Questions is what to do to obtain best overall performance, without too much programming difficulty
 - Choices are all MPI, all OpenMP, or *both*
 - For *both*, common option is outer loop parallelized with message passing, inner loop with directives to generate threads
- **Applications studied:**
 - Hydrology – CGWAVE
 - Computational chemistry – GAMESS
 - Linear algebra – matrix multiplication and QR factorization
 - Seismic processing – SPECseis95
 - Computational fluid dynamics – TLNS3D
 - Computational physics - CRETIN

Types of parallelism in the codes

- **For message passing parallelism (MPI)**
 - Parametric – coarse-grained outer loop, essentially task parallel
 - Structured domains – domain decomposition with local operations – structured and unstructured grids
 - Direct solvers – linear algebra, lots of communication and load balancing required – message passing works well for large systems of equations
- **Shared memory parallelism (OpenMP)**
 - Statically scheduled parallel loops – one large, or several smaller loops, non-nested parallel
 - Parallel regions – merge loops into one parallel region to reduce overhead of directives
 - Dynamic load balanced – when static scheduling leads to load imbalance from irregular task sizes

CGWAVE

- Finite elements - MPI parameter space evaluation at outer loop, OpenMP sparse linear equation solver in inner loops
- Speedup using 2 levels of parallelism makes modeling larger bodies of water possible in a reasonable amount of time
- Master-worker strategy for dynamic load balancing in MPI part/component
- Solver for each component solves large sparse linear system with OpenMP to parallelize
- On SGI Origin 2000 (distributed shared memory machine), use first touch rule to migrate data for each component to the processor that uses it
- Performance results show that best performance obtained using both MPI and OpenMP, with a combination of MPI workers and OpenMP threads that depends on the problem/grid size
 - And for load balancing, a lot fewer MPI workers than components

GAMESS

- Computational chemistry – molecular dynamics – MPI across cluster, OpenMP within each node
- Built on top of Global Arrays package – for distributed array operations
 - Which in turn uses MPI (paper says PVM) and OpenMP
- Linear algebra solvers mainly use OpenMP for dynamic scheduling and load balancing
- MPI versions of parts of code are complex, but can provide higher performance for large problems
- Performance results on “medium” sized problem from SPEC (Standard Performance Evaluation Corp.) are for a small system (4 8-processor Alpha processors) connected by Memory Channel

Linear algebra

- Hybrid parallelism with MPI for scalability and OpenMP for load balancing, for MM and QR factorization
- On IBM SP system with multiple 4-processor nodes
- Studies tradeoffs of hybrid approach for linear algebra algorithms vs. only using MPI (running 4 MPI processes per node)
- Use OpenMP for load balancing and decreasing communication costs within a node
- Also helps to hide communication latency behind other operations – important for overall performance
- QR factorization results on “medium” sized matrices show that adaptive load balancing is better than dynamic loop scheduling within a node

SPECseis95

- For gas and oil exploration
 - Uses FFTs and finite-difference solvers
- Original message passing version (in PVM) is SPMD, OpenMP starts serial then starts an SPMD parallel section
 - In OpenMP version, shared data is only boundaries, everything else local (like PVM version)
 - OpenMP calls all in Fortran – no C OpenMP compiler – caused difficulties for privatizing C global data, and thread issues (binding to processors, OS calls)
- Code scales equally well for PVM and OpenMP, on SGI Power Challenge (a DSM machine)
 - This is a weak argument, because of likely poor PVM message passing performance (in general, and especially on DSM systems)

TLNS3D

- CFD in Fortran77, uses MPI across grids and OpenMP to parallelize each grid
- Multiple, non-overlapping grids/blocks that exchange data at boundaries periodically
- Static block assignment to processors – divide blocks into groups of about equal number of grid points for each processor
- Master-worker execution model for MPI level, then parallelize 3D loops for each block with OpenMP
 - Many loops, so need to be careful about affinity of data objects to processors across loops
- Hard to balance MPI workers vs. OpenMP threads per block – tradeoff minimizing load imbalance vs. communication and synchronization cost
- Seems to work best on DSMS, but can be done well on distributed memory systems
- No performance results!

CRETIN

- Physics application with multiple levels of message passing and thread parallelism
- Ported onto both distributed memory system (1464 4-processor nodes) and DSM (large SGI Origin 2000)
- Complex structure, with 2 parts discussed
 - Atomic kinetics – multiple zones with lots of computation per zone – maps to either MPI or OpenMP
 - Load balancing across zones is the problem – requires complex dynamic algorithm that benefits both versions
 - Radiation transport – mesh sweep across multiple zones, suitable for both MPI and OpenMP
 - Two MPI options to parallelize, which one works best depends on problem size – one needs a transpose operation for the MPI version
- No performance results