System Comparisons

CS315B

Lecture 14
Recap

- We’ve looked at a variety of parallel/distributed system designs
  - SPMD
    - MPI, Charm++
  - Tasking
    - Regent, StarPU
  - Thread-based
    - Chapel, X10

- There are also data analytics systems such as Spark and TensorFlow
How Do We Compare Systems

• Benchmarks!

• Implement program X on systems A and B
  • Compare performance!

• Major pitfall: Making the comparison fair
  • Is it really apples to apples?

• Practical problem:
  • Expensive to write many X’s
  • For many A’s and B’s
Is there a better way?
The Focus

• We want to
  • Compare the programming systems
  • Not the applications themselves
Benchmarks are expensive to implement
  • For $N$ benchmarks and $M$ programming systems, $O(NM)$ effort
  • Must be tuned for performance

Could try proxy apps
  • Cut down benchmarks

Or microbenchmarks
  • But not benchmarks
  • Consequence: few papers evaluate many systems
Task Bench

Task Bench reduces the effort to $O(N + M)$
Model Space of Application Behaviors

• Model application as a **task graph**
  • Task: units of code with no communication

• **Parameterize** the task graph to explore a space of modeled behaviors
  • Set of tasks
  • Dependencies between tasks
  • Kernels executed by each task
  • Data produced by each task
  (and communicated by dependencies)
Task Graphs

• Task graph is product of:
  • Iteration space
  • Dependence relation

• An extensible set
Task = Kernel

• Executed at every point in a task graph.

• Examples
  • Empty
  • Compute-bound (achieves peak compute)
  • Memory-bound (achieves peak memory BW)
  • Load-imbalanced (randomly varying duration)
  • Also extensible

• Implemented once for all systems
  • exposed by the core API
Implementations

• 15 parallel and distributed systems:
  • Traditional HPC: MPI and MPI+OpenMP, MPI+CUDA
  • PGAS/Actor: Chapel, Charm++ and X10
  • Task-based: OmpSs, OpenMP 4, PaRSEC, Realm, Regent, and StarPU
  • Data analytics, machine learning, workflows: Dask, Spark, Swift/T, and TensorFlow

• Implementations are tuned
  • With help from the system developers
Tasks in MPI?!

• A “task” is a communication-free section of application code

```c
RECEIVE(...)  -- Dependence
for(...) {
    ... application code ...  -- Task
}
SEND(...)  -- Dependence
```
Metrics

• Task Bench makes it easy to gather data
  • Lots of data

• But how do we compare systems?
  • What is the metric(s)?
Idea #1: Tasks/Second

• Problem: How big are the tasks?

• Most common choice: Empty tasks
  • Intuition: Measures only runtime overhead
  • Problem: All resources can be devoted to the runtime system

• Other extreme: Huge tasks
  • But that minimizes runtime costs
  • Any amount of overhead can be hidden by some task size
Idea #2: Weak Scaling

- Keep problem size the same per processor
  - Double number of processors, double problem size

- Problem
  - Runtime system performance sensitive to choice of problem size
  - Double problem size => halve runtime overhead
Idea #3: Strong Scaling

• Problem size stays fixed as processor numbers scale
  • Double parallel resources
  • Problem size per processor is halved

• Plus: Strong scaling limit captures when overheads become dominant

• Minus: But overheads are not just from the programming system
  • Communication costs increase with strong scaling
Discussion

• We want a metric that measures the cost of a runtime system

• Must constrain efficiency
  • Minimum amount of application work must be getting done
  • Avoids problems of TPS w/empty tasks, weak scaling

• Want point at which efficiency goal is just met
  • Least application work that achieves the efficiency metric
  • Avoids problems of TPS w/huge tasks, weak scaling
Minimum Effective Task Granularity (METG)

• METG(50%) is the smallest task granularity where an application achieves 50% efficiency

• Parameterized on the efficiency metric:
  • E.g.: machine’s peak performance is 1.2 TFLOP/s, so 50% is 0.6 TFLOP/s
  • E.g.: application’s peak is $1 \times 10^9$ mesh cells/s, so 50% is $0.5 \times 10^9$

• Efficiency constrained: useful work is performed
• Exposes overhead: the limit of a system under load
Calculating METG(50%)

- **Step 1**: measure performance with decreasing problem size

- **Step 2**: convert to efficiency vs. task granularity and intersect with 50%

Performance drops with problem size

Intersection with 50% efficiency (METG(50%) is 4.6 μs)
Understanding METG(50%)

• METG is a *minimum*

• Two systems with METG of 1 ms and 5 ms

• Application has an average task granularity of
  • 100 ms: doesn’t matter
  • 10 ms: matters a little
  • 1 ms: huge difference
  • METG imposes a floor on task granularity that is efficient
Evaluation

• 15 programming systems
  • On up to 256 nodes

• Cori Supercomputer
  • Cray XC40
  • 2× Intel Xeon E5-2698 v3 processors (32 physical cores/node)
  • 128 GB RAM
  • Cray Aries interconnect
  • GCC 7.3.0, Cray MPICH 7.7.3
Results
Overhead and Scalability

METG(50%) vs node count, stencil pattern

Lower is better
More Complicated Dependencies

METG(50%) vs node count, 4 nearest neighbors, 4 task graphs

Gap shrinks or even closes with more complicated dependence pattern

Lower is better
METG vs Dependencies/Task
METG vs Bytes/Dependence

(a) 16 bytes per task dependency.

(b) 65536 bytes per task dependency.
Badwidth Bound Kernels
Task-Based Systems

METG(50%) vs node count, stencil pattern

Scaling gap between PaRSEC, StarPU implicit and explicit programming models

Regent has flat scaling using control replication

Lower is better
Overlap Communication and Computation

Efficiency vs task granularity, 4 distant neighbors, 4 task graphs

Asynchronous systems gain advantage when computation and communication are balanced...

... as long as METG(50%) is lower than about 100 μs
Impact

- Task Bench has already made some of these systems better
  - Intensive effort to find and fix performance issues

- Should provide a new “microscope” for future work
Limitations

- Only compare on the intersection of features
  - CPU-only workloads
  - Dense problems

- Performance only
  - Not productivity

- Kernels
  - Single kernel is good, but also hides differences in writing kernels for specific systems
Summary

• Lowest overhead systems get METG(50%) of about 100μs at >= 100 nodes

• Asynchronous systems:
  • Better overlap between computation and communication
  • Better for complex task graphs
  • As long as they’re not too slow! (METG about 100μs)

• Task-based systems:
  • Scaling bottlenecks not entirely resolved by task pruning
  • Regent’s control replication does solve the scaling bottleneck