Wrap-Up

CS315B

Lecture 15
Topics

• Presentations

• Key Ideas

• Predictions
Presentations
Your Presentation Should Include

• Brief problem description
  • Enough for everyone to understand what the computation does

• Parallelization strategy
  • What are the tasks and what are the dependencies?

• Mapping strategy
  • Where did you put tasks and data?
  • If different from the default mapper

• Issues

• Performance results
  • Graphs up and to the right!
  • Profiles
  • Comparisons with reasonable baselines if possible
Your Presentation Should Not Include

• Disproportionate discussion of related work
  • Some context is good, of course

• Gory details
  • Don’t need to see your command line flags

*Remember: You have 15 minutes*
What is Due?

• Your slide deck
  • Updated with any new results since your presentation

• Your code
Key Ideas:
Parallel Programming

Prof. Aiken   CS 315B  Lecture 15  7
Amdahl’s Law

\[ \text{Speedup} = \frac{1}{(1 - p) + (p / s)} \]

where

\( p \) = portion of the program sped up
\( s \) = factor improvement of that portion
Parallelism: Speed vs. # of Processors for Different Values of $p$
Examples

• What are some examples of Amdahl’s Law?

• Bonus: Have you come across an instance yourself?
Locality

• Machines are hierarchically constructed
  • Small and fast at finest scale
  • Big and slow at coarsest scale
  • Each level is at least 10X

• Locality matters
  • Data and associated compute should be co-located
  • Not a small effect
Locality: Examples

• What are some computations/algorithms with good or bad locality?
Overhead

• Overhead = anything that isn’t application code

• Any system overheads limit scalability
Weak and Strong Scaling

• Weak scaling
  • Increase problem size with node count
    • Problem size per node is constant
  • Characterizes communication behavior

• Strong scaling
  • Problem size is fixed
  • Tests minimum granularity & communication
Surface Area to Volume

• A partitioning into N pieces is better if it requires less communication

• For stencils, communication is proportional to the surface area of a piece

• The volume of a piece represents the total work in that piece
Metaprogramming

• Not specifically for parallelism
  • Or even for performance

• Just a useful idea
  • That is not as well known as it should be
Key Ideas: Tasking
Task-Based Programming

• Tasks = parallel functions

• Collection arguments

• Program is a directed acyclic graph of tasks
  • Edges indicate ordering relationships
  • Can program graphs directly
  • Or write a program to generate graphs
Mapping

• Selecting
  • Where tasks run
  • Where data is placed

• Very important to performance
  • Significant improvements/penalties possible
Partitioning

- To distribute data, it must be partitioned

- Two issues
  - How partitions are named
  - What partitioning operators are available

- Overpartitioning

- Underexplored aspect of parallel programming
The Argument

• Tasking is compositional
  • Natural to compose programs/libraries that use tasks
  • Runtime can extract parallelism across abstraction boundaries

• Mapping is fundamentally not compositional
  • Adding a component may change the mapping for the whole program
  • A resource optimization problem
Predictions
Hardware

• Hardware drives the programming model

• Trends
  • More specialized accelerators
  • More reconfigurable processors
  • Decreasing (or not increasing) memory/thread

• Implication
  • Data movement and placement will be key
Applications

• Who will be the programmers?

• Options
  • Traditional HPC
  • Data analytics

• Likely data analysis >> HPC
  • Even within traditional HPC communities
Programming Systems

• MPI, OpenMP, CUDA are here to stay
  • Nothing goes away
  • E.g., Fortran

• One or two tasking systems will survive
  • And likely succeed
  • Building on top of MPI, OpenMP, CUDA
Why?

• Compositionality
  • Clear composition model
  • Clear mechanism for optimizing whole programs
    • Scheduling ahead
    • Mapping
Cloud vs. Supercomputer

• For small/short projects, the cloud will rule
  • Removes fixed overheads of obtaining and running machines

• For large/long projects, less clear
  • Compute intensive applications can be competitive in the cloud
  • Data intensive applications tend to be too expensive
  • If a project is large enough, it will benefit from its own hardware resources
Open Questions for Tasking Systems

- How well will composing task systems really work?
  - Few actual demonstrations as yet

- How important is resilience?

- Can mapping be automated?

- Can partitioning be automated?

- How low can the overheads be?

- Others?