Module 8: "Performance Issues"

Lecture 14: "Load Balancing and Domain Decomposition"

Performance Issues

[From Chapter 3 of Culler, Singh, Gupta]

Previous Next

Agenda

- Partitioning for performance
- Data access and communication
- Summary
- Goal is to understand simple trade-offs involved in writing a parallel program keeping an eye on parallel performance
  - Getting good performance out of a multiprocessor is difficult
  - Programmers need to be careful
  - A little carelessness may lead to extremely poor performanc

Partitioning for perf.

- Partitioning plays an important role in the parallel performance
  - This is where you essentially determine the tasks
- A good partitioning should practise
  - Load balance
  - Minimal communication
  - Low overhead to determine and manage task assignment (sometimes called extra work)
- A well-balanced parallel program automatically has low barrier or point-to-point synchronization time
  - Ideally I want all the threads to arrive at a barrier at the same time

Load balancing

- Achievable speedup is bounded above by
  - Sequential exec. time / Max. time for any processor
  - Thus speedup is maximized when the maximum time and minimum time across all processors are close (want to minimize the variance of parallel execution time)
  - This directly gets translated to load balancing
- What leads to a high variance?
  - Ultimately all processors finish at the same time
  - But some do useful work all over this period while others may spend a significant time at synchronization points
  - This may arise from a bad partitioning
  - There may be other architectural reasons for load imbalance beyond the scope of a programmer e.g., network congestion, unforeseen cache conflicts etc. (slows down a few threads)
- Effect of decomposition/assignment on load balancing
  - Static partitioning is good when the nature of computation is predictable and regular
  - Dynamic partitioning normally provides better load balance, but has more runtime overhead for task management; also it may increase communication
  - Fine grain partitioning (extreme is one instruction per thread) leads to more overhead, but better load balance
  - Coarse grain partitioning (e.g., large tasks) may lead to load imbalance if the tasks are not well-balanced

Module 8: "Performance Issues"

Lecture 14: "Load Balancing and Domain Decomposition"

Dynamic task queues

- Introduced in the last lecture
- Normally implemented as part of the parallel program
- Two possible designs
  - Centralized task queue: a single queue of tasks; may lead to heavy contention because insertion and deletion to/from the queue must be critical sections
  - Distributed task queues: one queue per processor
- Issue with distributed task queues
  - When a queue of a particular processor is empty what does it do? Task stealing

Task stealing

- A processor may choose to steal tasks from another processor's queue if the former's queue is empty
  - How many tasks to steal? Whom to steal from?
  - The biggest question: how to detect termination? Really a distributed consensus!
  - Task stealing, in general, may increase overhead and communication, but a smart design may lead to excellent load balance (normally hard to design efficiently)
  - This is a form of a more general technique called Receiver Initiated Diffusion (RID) where the receiver of the task initiates the task transfer
  - In Sender Initiated Diffusion (SID) a processor may choose to insert into another processor's queue if the former's task queue is full above a threshold

Architect's job

- Normally load balancing is a responsibility of the programmer
  - However, an architecture may provide efficient primitives to implement task queues and task stealing
  - For example, the task queue may be allocated in a special shared memory segment, accesses to which may be optimized by special hardware in the memory controller
  - But this may expose some of the architectural features to the programmer
  - There are multiprocessors that provide efficient implementations for certain synchronization primitives; this may improve load balance
  - Sophisticated hardware tricks are possible: dynamic load monitoring and favoring slow threads dynamicall

Partitioning and communication

- Need to reduce inherent communication
  - This is the part of communication determined by assignment of tasks
  - There may be other communication traffic also (more later)
- Goal is to assign tasks such that accessed data are mostly local to a process
  - Ideally I do not want any communication
  - But in life sometimes you need to talk to people to get some work done!

Previous Next

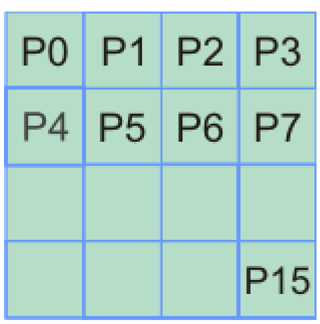
Domain decomposition

- Normally applications show a local bias on data usage
  - Communication is short-range e.g. nearest neighbor
  - Even if it is long-range it falls off with distance
  - View the dataset of an application as the domain of the problem e.g., the 2-D grid in equation solver
  - If you consider a point in this domain, in most of the applications it turns out that this point depends on points that are close by
  - Partitioning can exploit this property by assigning contiguous pieces of data to each process
  - Exact shape of decomposed domain depends on the application and load balancing requirements

Comm-to-comp ratio

- Surely, there could be many different domain decompositions for a particular problem
  - For grid solver we may have a square block decomposition, block row decomposition or cyclic row decomposition
  - How to determine which one is good? Communication-to-computation ratio

Assume P processors and NxN grid for grid solver



Size of each block: N/vP by N/vP

Communication (perimeter): 4N/vP

Computation (area): N^2/P

Comm-to-comp ratio = 4vP/N

Sq. block decomp. for P=16

- For block row decomposition
  - Each strip has N/P rows
  - Communication (boundary rows): 2N
  - Computation (area): N^2/P (same as square block)
  - Comm-to-comp ratio: 2P/N
- For cyclic row decomposition
  - Each processor gets N/P isolated rows
  - Communication: $2N^2/P$
  - Computation: N^2/P
  - Comm-to-comp ratio: 2
- Normally N is much much larger than P
  - Asymptotically, square block yields lowest comm-to-comp ratio
- Idea is to measure the volume of inherent communication per computation

- In most cases it is beneficial to pick the decomposition with the lowest comm-to-comp ratio
- But depends on the application structure i.e. picking the lowest comm-to-comp may have other problems
- Normally this ratio gives you a rough estimate about average communication bandwidth requirement of the application i.e. how frequent is communication
- But it does not tell you the nature of communication i.e. bursty or uniform
- For grid solver comm. happens only at the start of each iteration; it is not uniformly distributed over computation
- Thus the worst case BW requirement may exceed the average comm-to-comp ratio

Previous Next

Extra work

- Extra work in a parallel version of a sequential program may result from
  - Decomposition
  - Assignment techniques
  - Management of the task pool etc.
- Speedup is bounded above by
  Sequential work / Max (Useful work + Synchronization + Comm. cost + Extra work)
  where the Max is taken over all processors
- But this is still incomplete
  - We have only considered communication cost from the viewpoint of the algorithm and ignored the architecture completely

Data access and communication

- The memory hierarchy (caches and main memory) plays a significant role in determining communication cost
  - May easily dominate the inherent communication of the algorithm
- For uniprocessor, the execution time of a program is given by useful work time + data access time
  - Useful work time is normally called the busy time or busy cycles
  - Data access time can be reduced either by architectural techniques (e.g., large caches) or by cache-aware algorithm design that exploits spatial and temporal locality

Data access

- In multiprocessors
  - Every processor wants to see the memory interface as its own local cache and the main memory
  - In reality it is much more complicated
  - If the system has a centralized memory (e.g., SMPs), there are still caches of other processors; if the memory is distributed then some part of it is local and some is remote
  - For shared memory, data movement from local or remote memory to cache is transparent while for message passing it is explicit
  - View a multiprocessor as an extended memory hierarchy where the extension includes caches of other processors, remote memory modules and the network topology

Previous Next