Recap of Last Lecture

- The actual performance of a simple program can be a complicated function of the architecture
- Slight changes in the architecture or program may change the performance significantly
- Since we want to write fast programs, we must take the architecture into account, even on uniprocessors
- Since the actual performance is so complicated, we need simple models to help us design efficient algorithms
- We illustrated with a common technique for improving cache performance, called blocking, applied to matrix multiplication
  - Blocking works for many architectures, but choosing the blocksize depends on the architecture
Outline

- Parallel machines and programming models
- Steps in writing a parallel program
- Cost modeling and performance trade-offs
A generic parallel architecture

Parallel Programming Models

° Control
  • how is parallelism created
  • what orderings exist between operations
  • how do different threads of control synchronize

° Naming
  • what data is private vs. shared
  • how logically shared data is accessed or communicated

° Set of operations
  • what are the basic operations
  • what operations are considered to be atomic

° Cost
  • how do we account for the cost of each of the above
Trivial Example

- $\sum_{i=0}^{n-1} f(A[i])$

- **Parallel Decomposition:**
  - Each evaluation and each partial sum is a task

- **Assign n/p numbers to each of p procs**
  - each computes independent “private” results and partial sum
  - one (or all) collects the p partial sums and computes the global sum

=> **Classes of Data**

- **Logically Shared**
  - the original n numbers, the global sum

- **Logically Private**
  - the individual function evaluations
  - what about the individual partial sums?

Programming Model 1

- **Shared Address Space**
  - program consists of a collection of threads of control,
  - each with a set of private variables
    - e.g., local variables on the stack
  - collectively with a set of shared variables
    - e.g., static variables, shared common blocks, global heap
  - threads communicate implicitly by writing and reading shared variables
  - threads coordinate explicitly by synchronization operations on shared variables
    - writing and reading flags
    - locks, semaphores

- **Like concurrent programming on uniprocessor**
**Machine Model 1**

- A shared memory machine
- Processors all connected to a large shared memory
- "Local" memory is not (usually) part of the hardware
  - Sun, DEC, Intel "SMPs" (Symmetric multiprocessors) in Millennium; SGI Origin
- Cost: much cheaper to cache than main memory

![Diagram of machine model 1](image)

- Machine model 1a: A Shared Address Space Machine
  - replace caches by local memories (in abstract machine model)
  - this affects the cost model -- repeatedly accessed data should be copied
  - Cray T3E

---

**Shared Memory code for computing a sum**

**Thread 1**

- \[ s = 0 \text{ initially} \]
- \[ \text{local}_s 1 = 0 \]
- \( i = 0, n/2 - 1 \)
  - \[ \text{local}_s 1 = \text{local}_s 1 + f(A[i]) \]
- \( s = s + \text{local}_s 1 \)

**Thread 2**

- \[ s = 0 \text{ initially} \]
- \[ \text{local}_s 2 = 0 \]
- \( i = n/2, n - 1 \)
  - \[ \text{local}_s 2 = \text{local}_s 2 + f(A[i]) \]
- \( s = s + \text{local}_s 2 \)

**What could go wrong?**
### Pitfall and solution via synchronization

- **Pitfall in computing a global sum** \( s = \text{local}_s1 + \text{local}_s2 \)

  **Thread 1 (initially \( s=0 \))**
  - load \( s \) [from mem to reg]
  - \( s = s + \text{local}_s1 \) [in reg]
  - store \( s \) [from reg to mem]

  **Thread 2 (initially \( s=0 \))**
  - load \( s \) [from mem to reg; initially 0]
  - \( s = s + \text{local}_s2 \) [in reg]
  - store \( s \) [from reg to mem]

- Instructions from different threads can be interleaved arbitrarily
- What can final result \( s \) stored in memory be?
- **Race Condition**
- Possible solution: **Mutual Exclusion with Locks**

  **Thread 1**
  - lock
  - load \( s \)
  - \( s = s + \text{local}_s1 \)
  - store \( s \)
  - unlock

  **Thread 2**
  - lock
  - load \( s \)
  - \( s = s + \text{local}_s2 \)
  - store \( s \)
  - unlock

- Locks must be **atomic** (execute completely without interruption)

---

### Programming Model 2

- **Message Passing**
  - program consists of a collection of **named** processes
    - thread of control plus local address space
    - local variables, static variables, common blocks, heap
  - processes communicate by **explicit data transfers**
    - matching pair of send & receive by source and dest. proc.
  - coordination is implicit in every communication event
  - logically shared data is partitioned over local processes

- **Like distributed programming**

- **Program with standard libraries:** MPI, PVM
Machine Model 2

- A distributed memory machine
  - Cray T3E (too!), IBM SP2, NOW, Millennium
- Processors all connected to own memory (and caches)
  - cannot directly access another processor’s memory
- Each “node” has a network interface (NI)
  - all communication and synchronization done through this

![Diagram showing memory nodes connected through NI and an interconnect.]

Computing $s = x(1)+x(2)$ on each processor

- First possible solution

  Processor 1
  send $x_{local}$, proc2
  [$x_{local} = x(1)$]
  receive $x_{remote}$, proc2
  $s = x_{local} + x_{remote}$

  Processor 2
  receive $x_{remote}$, proc1
  send $x_{local}$, proc1
  [$x_{local} = x(2)$]
  $s = x_{local} + x_{remote}$

- Second possible solution - what could go wrong?

  Processor 1
  send $x_{local}$, proc2
  [$x_{local} = x(1)$]
  receive $x_{remote}$, proc2
  $s = x_{local} + x_{remote}$

  Processor 2
  send $x_{local}$, proc1
  [$x_{local} = x(2)$]
  receive $x_{remote}$, proc1
  $s = x_{local} + x_{remote}$

- What if send/receive act like the telephone system? The post office?
Programming Model 3

- **Data Parallel**
  - Single sequential thread of control consisting of parallel operations
  - Parallel operations applied to all (or defined subset) of a data structure
  - Communication is implicit in parallel operators and “shifted” data structures
  - Elegant and easy to understand and reason about
  - Not all problems fit this model

- **Like marching in a regiment**

  \[ A = \text{array of all data} \]
  \[ fA = f(A) \]
  \[ s = \text{sum}(fA) \]

- **Think of Matlab**

Machine Model 3

- **An SIMD (Single Instruction Multiple Data) machine**

- A large number of small processors

- A single “control processor” issues each instruction
  - each processor executes the same instruction
  - some processors may be turned off on any instruction

- **Machines not popular (CM2), but programming model is**
  - implemented by mapping n-fold parallelism to p processors
  - mostly done in the compilers (HPF = High Performance Fortran)
Machine Model 4

° Since small shared memory machines (SMPs) are the fastest commodity machine, why not build a larger machine by connecting many of them with a network?

° CLUMP = Cluster of SMPs

° Shared memory within one SMP, message passing outside

° Millennium, ASCI Red (Intel), ...

° Programming model?
  • Treat machine as “flat”, always use message passing, even within SMP (simple, but ignore important part of memory hierarchy)
  • Expose two layers: shared memory and message passing (higher performance, but ugly to program)

Programming Model 5

° Bulk synchronous

° Used within the message passing or shared memory models as a programming convention

° Phases separated by global barriers
  • Compute phases: all operate on local data (in distributed memory)
    - or read access to global data (in shared memory)
  • Communication phases: all participate in rearrangement or reduction of global data

° Generally all doing the “same thing” in a phase
  • all do f, but may all do different things within f

° Simplicity of data parallelism without restrictions
Summary so far

- Historically, each parallel machine was unique, along with its programming model and programming language.
- You had to throw away your software and start over with each new kind of machine - ugh.
- Now we distinguish the programming model from the underlying machine, so we can write portably correct code, that runs on many machines.
  - MPI now the most portable option, but can be tedious.
- Writing portably fast code requires tuning for the architecture.
  - Algorithm design challenge is to make this process easy.
  - Example: picking a blocksize, not rewriting whole algorithm.

Steps in Writing Parallel Programs
Creating a Parallel Program

° Pieces of the job
  • Identify work that can be done in parallel
  • Partition work and perhaps data among processes=threads
  • Manage the data access, communication, synchronization

° Goal: maximize Speedup due to parallelism

\[
\text{Speedup}_{\text{prob}}(P \text{ proc}) = \frac{\text{Time to solve prob with “best” sequential solution}}{\text{Time to solve prob in parallel on P processors}}
\]

\[\leq P \quad \text{(Brent’s Theorem)}\]

\[
\text{Efficiency}(P) = \frac{\text{Speedup}(P)}{P}
\]

\[\leq 1\]

° Key question is when you can solve each piece
  • statically, if information is known in advance
  • dynamically, otherwise

Steps in the Process

° Task: arbitrarily defined piece of work that forms the basic unit of concurrency

° Process/Thread: abstract entity that performs tasks
  • tasks are assigned to threads via an assignment mechanism
  • threads must coordinate to accomplish their collective tasks

° Processor: physical entity that executes a thread

Overall Computation
Grains of Work
Assignment
Orchestration
Mapping
Processes/Threads
Processes/Threads
Processors
Decomposition

° Break the overall computation into grains of work (tasks)
  • identify concurrency and decide at what level to exploit it
  • concurrency may be statically identifiable or may vary dynamically
  • it may depend only on problem size, or it may depend on the particular input data

° Goal: enough tasks to keep the target range of processors busy, but not too many
  • establishes upper limit on number of useful processors (i.e., scaling)

Assignment

° Determine mechanism to divide work among threads
  • functional partitioning
    - assign logically distinct aspects of work to different threads
      - eg pipelining
  • structural mechanisms
    - assign iterations of “parallel loop” according to simple rule
      - eg proc j gets iterates j*n/p through (j+1)*n/p-1
    - throw tasks in a bowl (task queue) and let threads feed
  • data/domain decomposition
    - data describing the problem has a natural decomposition
    - break up the data and assign work associated with regions
      - eg parts of physical system being simulated

° Goal
  • Balance the workload to keep everyone busy (all the time)
  • Allow efficient orchestration
Orchestration

° Provide a means of
  • naming and accessing shared data,
  • communication and coordination among threads of control

° Goals:
  • correctness of parallel solution
    - respect the inherent dependencies within the algorithm
  • avoid serialization
  • reduce cost of communication, synchronization, and management
  • preserve locality of data reference

Mapping

° Binding processes to physical processors
° Time to reach processor across network does not depend on which processor (roughly)
  • lots of old literature on “network topology”, no longer so important
° Basic issue is how many remote accesses
Example

° $s = f(A[1]) + \ldots + f(A[n])$

° Decomposition
  • computing each $f(A[j])$
  • $n$-fold parallelism, where $n$ may be $>> p$
  • computing sum $s$

° Assignment
  • thread $k$ sums $s_k = f(A[k \times n/p]) + \ldots + f(A[(k+1) \times n/p-1])$
  • thread 1 sums $s = s_1 + \ldots + s_p$ (for simplicity of this example)
  • thread 1 communicates $s$ to other threads

° Orchestration
  • starting up threads
  • communicating, synchronizing with thread 1

° Mapping
  • processor $j$ runs thread $j$

Administrative Issues

° Assignment 2 will be on the home page later today
  • Matrix Multiply contest
  • Find a partner (outside of your own department)
  • Due in 2 weeks

° Lab/discussion section will be 5-6pm Tuesdays

° Reading assignment
  • www.cs.berkeley.edu/~demmel/cs267/lecture04.html
  • Optional:
    - Chapter 1 of Culler/Singh book
    - Chapters 1 and 2 of www.mcs.anl.gov/dbpp
Cost Modeling and Performance Tradeoffs

Identifying enough Concurrency

- Parallelism profile
  - area is total work done

  Simple Decomposition:
  - \( f(\ A[i] ) \) is the parallel task
  - \( \sum n \) is sequential

- Amdahl’s law
  - let \( s \) be the fraction of total work done sequentially

\[
\text{Speedup}(P) \leq \frac{1}{s + \frac{1 - s}{P}} \leq \frac{1}{s}
\]

After mapping

\[
p \times \frac{n}{P} \times \text{time}(f)
\]

\[
\text{time}(\sum n)
\]
Algorithmic Trade-offs

- Parallelize partial sum of the f’s
  - what fraction of the computation is “sequential”
  - what does this do for communication? locality?
  - what if you sum what you “own”

- Parallelize the final summation (tree sum)
  - Generalize Amdahl’s law for arbitrary “ideal” parallelism profile

Problem Size is Critical

- Suppose Total work = n + P
- Serial work: P
- Parallel work: n
- s = serial fraction
  = P / (n+P)

In general seek to exploit a fraction of the peak parallelism in the problem.
Load Balance

° Insufficient Concurrency will appear as load imbalance

° Use of coarser grain tends to increase load imbalance.

° Poor assignment of tasks can cause load imbalance.

° Synchronization waits are instantaneous load imbalance

\[
\text{Speedup} (P) \leq \frac{\text{Work}(1)}{\max_p (\text{Work}(p) + \text{idle})}
\]

Extra Work

° There is always some amount of extra work to manage parallelism
  • e.g., to decide who is to do what

\[
\text{Speedup} (P) \leq \frac{\text{Work}(1)}{\max_p (\text{Work}(p) + \text{idle} + \text{extra})}
\]
There are many ways to reduce communication costs.

\[
\text{Speedup}(P) \leq \frac{\text{Work}(1)}{\max(\text{Work}(P) + \text{idle} + \text{extra} + \text{comm})}
\]

- Coordinating placement of work and data to eliminate unnecessary communication
- Replicating data
- Redundant work
- Performing required communication efficiently
  - e.g., transfer size, contention, machine specific optimizations
The Tension

\[
\text{Speedup}(P) \leq \frac{\text{Work}(1)}{\max(\text{Work}(P) + \text{idle} + \text{comm} + \text{extraWork})}
\]

- Fine grain decomposition and flexible assignment tends to minimize load imbalance at the cost of increased communication
  - In many problems communication goes like the surface-to-volume ratio
  - Larger grain \(\Rightarrow\) larger transfers, fewer synchronization events
- Simple static assignment reduces extra work, but may yield load imbalance

The Good News

- The basic work component in the parallel program may be more efficient that in the sequential case
  - Only a small fraction of the problem fits in cache
  - Need to chop problem up into pieces and concentrate on them to get good cache performance.
  - Similar to the parallel case
  - Indeed, the best sequential program may emulate the parallel one.
- Communication can be hidden behind computation
  - May lead to better algorithms for memory hierarchies
- Parallel algorithms may lead to better serial ones
  - Parallel search may explore space more effectively