ECE5610/CSC6220
Introduction to Parallel and Distribution Computing

Lecture 3: Programming for Performance
Performance Goal: Speedup

- Architects’ Goal
  - Observe how program uses machine and improve the system design to enhance performance
  - Solutions: high-performance interconnect, efficient implementation of coherence protocol and synchronization primitives …

- Programmers’ Goal
  - observe how the program uses the machine and identify and remove performance bottlenecks.
What Limits Performance of Parallel Programs?

• Available parallelism
• Load balance
  - Some processors do more work than others
  - Some work while others wait
  - Remote resource contention (such as I/O services)
• Communication
• Extra work
  - Management of parallelism
  - Redundant computation
An Analogy: Limit in Speedup due to Available Parallelism

Suppose a person wants to travel from city A to city C via city B. The routes from A to B are in mountains and the routes from B to C are in desert. The distances from A to B, and from B to C are 80 miles and 200 miles, respectively.
From A to B, have to walk at speed of 4 mph
From B to C, walk or drive

• Question 1: How long will it take for the entire trip by walking?
  ➢ Answer: \( \frac{80}{4} + \frac{200}{4} = 70 \) hours

• Question 2: How much faster if the trip from B to C is by a car as opposed to walk? (at speed of 100 mph)
  ➢ Answer: \( \frac{70}{\left(\frac{80}{4} + \frac{200}{100}\right)} = 3.18 \)

• Question 3: What is the maximum speedup by increasing driving speed?
  ➢ Answer: \( \frac{70}{\left(\frac{80}{4} + \frac{200}{\text{infinite}}\right)} \) ➔ 3.5
**Limited Concurrency: Amdahl’s Law**

- Most fundamental limitation on parallel speedup
- Amdahl’s law: if fraction \( f \) of sequential execution is inherently serial, speedup \( \leq 1/f \)

Assuming ideal speedup for the non-serial part: (\( p \) is \# of processors.)

Speedup factor is given by:

\[
S(p) = \frac{t_s}{ft_s + (1 - f)t_s/p} = \frac{p}{1 + (p - 1)f} \leq 1/f
\]

Efficiency factor is given by:

\[
\text{Efficiency}(p) = S(p)/p \leq 1/(fp)
\]
Limited Concurrency: Amdahl’s Law

Even with infinite number of processors, maximum speedup limited to $1/f$.

**Example:** With only 5% of computation being serial, maximum speedup is 20, irrespective of number of processors.
The Implication of Amdahl’s Law

\[ S(p) = \frac{1}{1 - f} \]

\[ f = 0\% \]
\[ f = 5\% \]
\[ f = 10\% \]
\[ f = 20\% \]
Gustafson’s Law

Parallel processing is to solve larger programs in a fixed time
Let $t_s$ be sequential execution time,
We assume serial work $f$ (not fraction) is fixed when the total work of a parallel program increases:
Scaled speedup factor: $S(p) = f + p*(1-f)$
($S(p)$ increases linearly with $p$)

Suppose a serial section of 5% and 20 processors
According to Amdahl’s law, the speedup is 10.26
According to Gustafson’s law, the speedup is 19.05
Overhead of Parallelism

• Given enough parallel work, this is the most significant barrier to getting desired speedup.

• Parallelism overheads include:
  ➢ cost of starting a thread or process
  ➢ cost of communicating shared data
  ➢ cost of synchronizing
  ➢ extra (redundant) computation

• Each of these could be in the range of milliseconds on some systems

• Tradeoff: Algorithm needs sufficiently large units of work to run fast in parallel (i.e. large granularity), but not so large that there is not enough parallel work.
High-Performance Parallel Programs

• Program tuning as successive refinement

  ➢ Architecture-independent partitioning
    • View machine as a collection of communicating processors
    • Focus: balancing workload, reducing inherent communication & extra work

  ➢ Architecture-dependent orchestration
    • View machine as extended memory hierarchy
    • Focus: reduce artificial communication due to architectural interactions, cost of communication depends on how it is structured
    • May inspire changes in partitioning
Partitioning for Performance

• Three major areas
  ➢ Balancing the workload + reducing wait time at synchronization points
  ➢ Reducing inherent communication
  ➢ Reducing extra work

• Tradeoff between these algorithmic issues
  ➢ Minimize communication ➔ run on one processor ➔ extreme load imbalance
  ➢ Maximum load balance ➔ random assignment of tiny work ➔ no control over communication
  ➢ Good partition may imply extra work to compute or manage it

• Goal is to compromise
  ➢ Fortunately, often not difficult in practice
Focus 1: Load Balance and Synchronization Time

• Limits on speedup

\[
\text{Speedup}_{\text{problem}}(p) \leq \frac{\text{Sequential Work}}{\text{Max Work on any Processor}}
\]

- Work includes data access and other costs
- not just equalizing work, but must keep processor busy at the same time

• Four parts to the problem
  - Identify enough concurrency
  - Decide how to manage it (statically or dynamically)
  - Determine the granularity at which to exploit it
  - Reduce serialization and cost of synchronization
Identifying Concurrency

• Techniques used for the equation solver kernel
  ➢ loop structure
  ➢ fundamental dependencies (not constrained by sequential algorithm) ➔ new algorithms

• In general, two orthogonal levels of parallelism
  ➢ Function (Task) parallelism
    • Entire large tasks (procedures) can be done in parallel (e.g., in encoding a sequence of video frames: prefiltering, convolution, quantization, entropy coding ..)
    • Difficult to load balance
  ➢ Data parallelism
    • More scalable: proportional to input size
    • mostly used on large-scale parallel machines
Managing Concurrency

Static versus dynamic techniques

• Static techniques
  ➢ Algorithmic assignment based on input: does not change
  ➢ low run-time overhead, but requires predictable computation
  ➢ Preferable when applicable

  Caveat: in multi-programmed/heterogeneous environments

• Dynamic techniques
  ➢ Adapt at run time to balance load
  ➢ But, can increase communication and task management overheads
Managing Concurrency (cont’d)

• Dynamic techniques
  ➢ Profile-based (semi-static)
    • Profile work distribution at run time and repartition dynamically
  ➢ Dynamic tasking
    • Pool of tasks: remove takes and add tasks until done
    • Scheduling with Task Queues: Centralized versus distributed queues

(a) Centralized task queue

(b) Distributed task queues (one per process)
Determining Task Granularity

• Task granularity: amount of work associated with a task
  ➢ should scale with respect to parallelism overheads in the system
    • communication, synchronization, etc.

• General rule
  ➢ coarse-grained ➔ often poor load balance
  ➢ fine-grained ➔ more overhead, often more communication, requires more synchronization (contention)
Reducing Serialization

• Influenced by assignment and orchestration (including how tasks are scheduled on physical resources)

• Event synchronization
  ➢ Conservative (global) versus point-to-point synchronization
    • e.g. barriers versus lock
  ➢ However, fine-grained sync more difficult to program and can produce more synchronization operations

• Mutual exclusion
  ➢ Main goal is to reduce contention: separate locks for separate data
    • e.g. locking records in a database: lock per process, record, or field
    • lock per task in task queue, not per queue
    • finer grain => less contention/serialization, more space, less reuse
  ➢ Smaller critical sections
    • don’t do reading/testing in critical section, only modification
  ➢ Stagger critical sections in time
Implications of Load Balance

- Extends speedup limit expression to
  \[ \text{Speedup}_{\text{problem}}(p) \leq \frac{\text{Sequential Work}}{\max (\text{Work on any processor} + \text{Sync Wait Time})} \]

- Each processor should do the same amount of work

- Instantaneous load imbalance revealed at barriers, mutex, or send/receive
  - Cost of load imbalance = wait time

- Load balance is the responsibility of programmers
Focus 2: Reducing Inherent Communication

• Communication is expensive!

\[
\text{Speedup} \leq \frac{\text{Sequential Work}}{\text{Max} (\text{Work on any processor} + \text{Synch Wait Time} + \text{Comm Cost})}
\]

• Metric: \textit{communication to computation ratio}

• Controlled primarily by the assignment of tasks to processes

• Assign tasks that access the same data to the same process

• Some simple heuristic partitioning methods work well

\(\Rightarrow\) \textit{Domain decomposition}
Domain Decomposition for Load Balancing and Inherent Communication

- Exploits locality nature of physical problems
  - Information requirements often lie in a small localized region
  - or long-range but fall off with distance
- Simple example: nearest-neighbor grid computation

> Perimeter-to-area relationship represents comm-to-comp ratio: $4n/\sqrt{p} : n^2/p$
  (area to volume in 3-d)
> the ratio depends on $n, p$: decreases with $n$, increases with $p
Domain Decomposition (cont’d)

• Best domain decomposition depends on information requirements
• Nearest neighbor example: block versus strip decomposition:

Comm-to-comp ratio: \(\frac{4\sqrt{p}}{n}\) for block, \(\frac{2p}{n}\) for strip

So how about the ratio for cyclic decomposition(row \(i\) assigned to process \(i \mod nprocs\))?
Focus 3: Reducing Extra Work

• Communication is expensive!

\[
\text{Speedup} \leq \frac{\text{Sequential Work}}{\text{Max} (\text{Work on any processor} + \text{Synch Wait Time} + \text{Comm Cost} + \text{Extra work})}
\]

• Common sources of extra work
  - Computing a good partition (e.g., in a sparse matrix computation)
  - Using redundant computation to avoid communication
  - Task, data, and process management overhead
    - Application, language, run-time systems, OS
  - Imposing structure on communication
    - Coalescing messages

• How can architecture help?
  - Effective support of communication and synchronization (orchestration)
Memory-oriented View of a Multiprocessor

• Multiprocessor as an extended memory hierarchy

➢ Levels: registers, caches, local memory, remote memory (topology)
  • glued together by communication architecture
  • at different levels communicate at a different granularity of data transfer
  • Differences in access costs and bandwidth

➢ Need to exploit spatial and temporal locality in hierarchy
  • Similar to uniprocessors: extra communication ➔ high communication costs

➢ Trade off with parallelism
Artifactual Communication Costs

Accesses not satisfied in local hierarchy levels cause communication

• Inherent
  ➢ Determined by program
  ➢ Assume unlimited capacity, small transfers, perfect knowledge

• Artifactual
  ➢ Determined by program implementation and interactions with architecture
  ➢ Some reasons
    – Poor allocation of data across distributed memories
    – Redundant communication of data
    – Unnecessary data in a transfer or unnecessary transfers (system granularities)
    – Finite replication capacity
      • Four kinds of cache misses: compulsory, capacity, conflict, coherence
      • Finite capacity affects: capacity and conflict misses
Orchestration for Performance

Two areas of focus:

• Reducing amount of communication
  ➢ Inherent: change logical data sharing patterns in an algorithm
  ➢ Artifactual: exploit spatial, temporal locality in extended hierarchy
    – Techniques often similar to those on uniprocessors
    – Shared address space machines support this in hardware, distributed memory machines support the same techniques in software

• Structuring communication to reduce cost
Structuring Communication to Reduce Cost

\[ \text{communication cost} = f \left( o + l + \frac{n_c}{m} + \frac{t_c - \text{overlap}}{B} \right) \]

- frequency of messages
- message overhead
- \( n_c \): total data sent
- \( m \): number of messages
- \( B \): bandwidth along path
- portion of latency that can be overlapped
- cost induced by contention
  - in the network
  - end-point contention
Reducing Contention

- All resources have nonzero occupancy
  - Memory, communication controller, network link, etc.
  - Can only handle limited transactions per unit time

- Effects of contention:
  - Increased end-to-end cost for messages
  - Reduced available bandwidth for individual messages
  - Causes imbalances across processors

- Insidious performance consequence
  - Easy to ignore when programming
  - Effect can be devastating: **Don’t flood a resource!**
Reducing Contention (cont’d)

• Two types of contentions:
  – Network contention
    • Can be reduced by mapping processes and scheduling communication appropriately
  – end-point contention (hot-spots)
    • Reduced by using tree-structured communication

• In general, the solution is to reduce burstiness;
  – By staggering messages to the same destination in time
  – However, this may conflict with the advantage of making messages larger
Overlapping Communication

• Cannot afford the stall due to communication of high latency
  ➢ Even on uniprocessors!
• Overlap computation with communication to hide latency
• Need extra concurrency (slackness) beyond the number of processors
• Techniques:
  ➢ Prefetching
  ➢ Using asynchronous/non-blocking communication
  ➢ Multithreading
Summary of Performance Tradeoffs

- **Load Balance**  
  
  - [synchronization wait time]
  
  - Fine-grained tasks
  
  - random or dynamic assignment

- **Inherent communication volume**  
  
  - [data access costs]
  
  - Coarse-grained tasks
  
  - Tension between locality and load balance

- **Extra work**  
  
  - [processor overhead + data access costs]
  
  - Coarse-grained tasks
  
  - Simple assignment

- **Artifactual Communication costs**  
  
  - [data access costs]
  
  - big transfers – amortize overhead and latency
  
  - small transfers – reduce overhead and contention

- **From the perspective of programming model:**
  
  - Advantage of implicit communication in the SAS: programming ease and performance in the presence of fine-grained data sharing
  
  - Advantage of explicit communication in the MP: block data transfer, implicit synchronization, better performance prediction ability, and ease of building machines.