Parallel Programming

To understand and evaluate design decisions in a parallel machine, we must get an idea of the software that runs on a parallel machine.

--Introduction to Culler et al.’s Chapter 2, beginning 192 pages on software
Outline

• Applications

• Creating Parallel Programs

• Programming for Performance

• Synchronization Basics

Applications

• Scientific
  – Simulation of natural phenomena (protein folding, planetary motion, molecular interactions, etc.)
  – Large mathematical problems

• Commercial
  – Online transaction processing (OLTP)
  – Decision support systems (DSS)
  – Web serving (e.g., Apache)
  – Application serving (a.k.a. middleware)

• Multimedia/home
  – Audio/video, games, word processing, speech recognition, etc.
Scientific: The SPLASH2 Benchmarks

- **Kernels**
  - Complex 1D FFT
  - Blocked LU Factorization
  - Blocked Sparse Cholesky Factorization
  - Integer Radix Sort

- **Applications**
  - **Barnes-Hut**: interaction of N bodies
  - Adaptive Fast Multipole (FMM): interaction of bodies
  - Ocean Simulation
  - Hierarchical Radiosity
  - Ray Tracer (Raytrace)
  - Volume Renderer (Volrend)
  - Water Simulation with Spatial Data Structure (Water-Spatial)
  - Water Simulation without Spatial Data Structure (Water-Nsquared)

Scientific: The SpecOMP Benchmarks

- **Parallel scientific benchmarks based on Spec CPU**
- **Written in OpenMP (shared memory “library”)**
  - wupwise quantum chromodynamics
  - swim shallow water modeling
  - mgrid multi-grid solver in 3D potential field
  - applu parabolic/elliptic partial differential equations
  - galgel fluid dynamics analysis of oscillatory instability
  - art neural net simulation of adaptive resonance theory
  - equake finite element simulation of earthquake modeling
  - ammp computational chemistry
  - fma3d finite-element crash simulation
  - apsi solves problems regarding temperature, wind, etc.
  - gafort genetic algorithm code
**PARSEC**

- **Diverse, emerging, non-HPC benchmarks**
  - blackscholes - Option pricing with Black-Scholes Partial Differential Equation (PDE)
  - bodytrack - Body tracking of a person
  - canneal - Simulated cache-aware annealing to optimize routing cost of a chip design
  - dedup - Next-generation compression with data deduplication
  - facesim - Simulates the motions of a human face
  - ferret - Content similarity search server
  - fluidanimate - Fluid dynamics for animation purposes with Smoothed Particle Hydrodynamics (SPH) method
  - freqmine - Frequent itemset mining
  - raytrace - Real-time raytracing
  - streamcluster - Online clustering of an input stream
  - swaptions - Pricing of a portfolio of swaptions
  - vips - Image processing
  - x264 - H.264 video encoding

---

**Ocean Simulation**

- **Simulate ocean currents**
- **Discretize in space and time**
N-body: Barnes-Hut

- Computing the mutual interactions of N bodies
  - N-body problems
  - Stars, planets, molecules...
- Can approximate influence of distant bodies

Online Transaction Processing: TPC-C

- TPC-C is a standard OLTP benchmark

- Models database transactions for company
  - Customers make orders of company
  - Company orders from suppliers to stock warehouses

- Goal: high transaction throughput

- Specifics
  - Specifies database size, number of clients, etc.
  - Does not specify implementation!
Outline

- Applications

- Creating Parallel Programs
  - In general
  - Two examples

- Programming for Performance

- Scaling

- Synchronization Basics

Creating a Parallel Program

- In theory, can be done by programmer, compiler, run-time system, or OS

- In practice, parallel programs created with
  - Explicitly parallel language (e.g., High Performance Fortran)
  - Library for implementing a programming model
    » Shared memory library (POSIX, PARMACS, OpenMP)
    » Message passing library (Message Passing Interface)

- What will you realize at end of this section?
  - Parallel programming is difficult!
A Little Terminology

• A **Task** is a piece of work
  – Ocean: grid point, row, plane
  – Apache: single query

• **Task granularity**
  – Small → fine-grain task
  – Large → coarse-grain task

• A process (thread) performs tasks
  – According to OS: process = thread(s) + address space

• A **process** is executed on a **processor**

Steps for Creating a Parallel Program

• **Decomposition** into tasks
• **Assignment** of tasks to processes (threads)
• **Orchestration** of data access, communication, etc.
• **Mapping** processes to processors
Decomposition

• Decompose computation into set of tasks
• Could be dynamic
• Maximize concurrency
• Minimize overhead of managing tasks
• Remember Amdahl’s Law!

\[
\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \text{Fraction}_{\text{enhanced}}} \text{Speedup}_{\text{enhanced}}
\]

Assignment

• Assign tasks to processes (static vs. dynamic)
• Balance workload (load balancing)
• Reduce communication
• Minimize overhead
• Assignment + Decomposition = Partitioning
**Orchestration**

- Choreograph data access, communication, and synchronization
- Reduce cost of communication and synchronization
- Preserve data locality (data layout)
- Schedule tasks (order of execution)
- Reduce overhead of managing parallelism
- Must have good primitives (architecture and model)

**Mapping**

- Map processes to physical processors
- Static
- Dynamic
  - Processes migrate
  - What about orchestration (data locality)
  - Task queues
OS Effects on Mapping

- Ability to bind process to processor

- Space Sharing
  - Physical partitioning of machine

- Gang Scheduling
  - All processes context switched simultaneously

Outline

- Applications

- Creating Parallel Programs
  - In general
  - Two examples

- Programming for Performance

- Scaling

- Synchronization Basics
**Data Parallel Example: Ocean Simulation**

- Contains equation solver “kernel”
  - Kernel = small piece of important code (not OS “kernel”)
- Update each point based on adjacent neighbors
  - Gauss-Seidel (update in place)
- Compute average difference per element
- Convergence when diff small $\rightarrow$ exit

**Equation Solver Decomposition**

```plaintext
while (!converged){
    for{}  // over all points in x-dimension
        for{}  // over all points in y-dimension
            • The loops are not independent! But …
            • Exploit properties of problem
              - Don’t really need up-to-date values (approximation)
              - May take more steps to converge, but exposes parallelism
            • Red-Black
              - Like checkerboard: update of red point depends only on black points
              - Alternate iterations over red, then black
            • Asynchronous
              - Each processor updates its region independent of other’s values
              - Global synch at end of iteration to keep things somewhat up-to-date
```
Decomposition: The FORALL Statement

while (!converged)
for all // execute all iterations in parallel
for all // execute all iterations in parallel

• Data parallel execution, like in HPF (High Perf Fortran)
• Decomposition: tasks = loop iterations
  – Can execute the iterations in parallel
• Each grid point computation (n² parallelism)
  while (!converged)
  for all

• Computation for rows is independent (n parallelism)
  – Less overhead

Equation Solver Assignment

- Each process gets a contiguous block of rows
Writing Shared Memory Code with Pthreads

- Library of shared memory routines
  - Portable across most platforms
- For programming assignment #1, you’ll program with pthreads
- Some of the included routines:
  - `pthread_create` and `pthread_exit`
  - `pthread_mutex_init` – create a mutex (lock)
  - `pthread_mutex_lock` – lock a mutex
  - `pthread_mutex_unlock` – unlock a mutex
- Other shared memory libraries
  - “OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs.”
  - Solaris System V Shared Memory, PARMACS, MM for Linux
  - Goals: simplify programming, abstract away hardware

Equation Solver: The Ugly Code (PARMACS)

```c
main()
A = G_MALLOC(size of big array);
CREATE(nprocs-1,Solve, A);
Solve(A)
  WAIT_FOR_END;
end main

Solve(A)
  while (!done)
    for i = my_start to my_end
      for j = 1 to n
        compute new_A[i, j];
        mydiff += abs(new_A[i, j] - old_A[i, j]);
      LOCK(diff_lock);
      diff += mydiff;
      UNLOCK(diff_lock);
      if (convergence_test) then done = 1
      BARRIER
```

(C) 2012 Alvin R. Lebeck from Adve, Falsafi, Hill, Reinhardt, Singh, Sorin
Compsci 221 / ECE 259
**SM/MP Example: Standard Cell Router**

- **LocusRoute (VLSI standard cell router)**

```plaintext
while (route_density_improvement > threshold)
{
    for (i = 1 to num_wires) do
    {
        rip old wire out
        explore new route
        place wire using best new route
    }
}
```

**Shared Memory Implementation**

- **Shared memory algorithm**
  - Divide cost array into regions
    - Logically assign regions to processors
  - Assign wires to procs based on the region in which center lies
  - Do load balancing using stealing when local queue empty

- **Pros:**
  - Good load balancing on average
  - Mostly local accesses
  - High cache hit ratio

- **Cons:**
  - Non-deterministic (why is this bad?)
  - Potential for hot spots
  - Amount of parallelism
Message Passing Implementations

- **Method 1:**
  - Distribute wires and cost array regions as in SM implementation
  - When wire-path crosses to remote region
    » Send computation to remote PE, or
    » Send message to access remote data

- **Method 2:**
  - Distribute only wires as in SM implementation
  - Fully replicate cost array on each PE
    » One owned region, and potential stale copy of others
    » Send updates so copies are not too stale
  - Consequences:
    » Waste of memory in replication
    » Stale data → poorer quality results or more iterations

- Both methods require lots of thought for programmer

---

MPI: Message Passing Interface

- **From the MPI website:**
  
  “MPI is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users.”

- Popular and portable message passing library for
  - Massively parallel machines
  - Clusters of PCs or workstations

- For programming assignment #2, you will write a message passing program with MPI that runs a cluster
Review: Creating a Parallel Program

• Can be done by programmer, compiler, run-time system or OS

• Steps for creating parallel program
  – Decomposition of work into tasks
  – Assignment of tasks to processes
  – Orchestration of processes
  – Mapping of processes to processors

• In practice, parallel programs created with
  – Explicitly parallel language (HPF, Split-C)
  – Shared memory library (pthreads, PARMACS)
  – Message passing library (MPI)

Outline

• Applications

• Creating Parallel Programs

• Programming for Performance

• Scaling

• Synchronization Basics
Programming for Performance

• Partitioning, Granularity, Communication, etc.

• Caches and Their Effects

Aside on Cost-Effective Computing

• Isn’t Speedup(P) < P inefficient?
• If only throughput matters, use P computers instead?

• But much of a computer’s cost is NOT in the processor [Wood & Hill, *IEEE Computer*, Feb 95]
• Let Costup(P) = Cost(P)/Cost(1)
• Parallel computing is cost-effective if Speedup(P) > Costup(P)
• E.g., for SGI PowerChallenge w/ 500MB:
  Costup(32) = 8.6
Where Do Programs Spend Time?

- **Sequential**
  - Busy computing
  - Memory system stalls

- **Parallel**
  - Busy computing
  - Stalled for local memory
  - Stalled for remote memory (communication)
  - Synchronizing (load imbalance and operations)
  - Overhead

- **Speedup** \( p = \frac{\text{time}(1)}{\text{time}(p)} \)
  - Amdahl’s Law
  - Could even be superlinear

Partitioning for Performance

- **Balance workload**
  - Reduce time spent at synchronization

- **Reduce communication**

- **Reduce extra work**
  - Determining and managing good assignment

- **These goals are at odds with each other**
**Programming for Performance**

- Identifying concurrency

- Managing concurrency
  - Static
  - Dynamic

- Granularity of concurrency

- Serialization and synchronization costs

---

**Identifying Concurrency**

- Data parallelism
  - Same ops on different data items

- Functional (control, task) parallelism
  - Pipeline

- Impact on load balancing?

- Functional is more difficult
  - Longer running tasks
Managing Concurrency

- **Static**
  - Cannot adapt to changes

- **Dynamic**
  - Can adapt
  - Cost of management increases
  - Self-scheduling (guided self-scheduling)
  - Centralized task queue
    » Contention
  - Distributed task queue
    » Can steal from other queues
    » Architecture: Name data associated with stolen task

Granularity of Concurrency

- **Granularity = Amount of work associated with task**

- **Large tasks**
  - Worse load balancing
  + Lower overhead
  + Less contention
  + Less communication

- **Small tasks**
  + Better load balancing
  - More synchronization
  - More management overhead
  - Might have too much communication (affinity scheduling)
Impact of Synchronization and Serialization

• Too coarse synchronization
  – Barriers instead of point-to-point synch
  – Poor load balancing
• Too many synchronization operations
  – Lock each element of array
  – Costly operations
• Coarse grain locking
  – Lock entire array
  – Serialize access to array
• Architectural aspects
  – Cost of synchronization operation
  – Synchronization name space
• Transactional Memory

Architectural Support for Dynamic Task Stealing

• How can architecture help?
  • Communication
    – Support for transfer of small amount of data and mutual exclusion
    – Can make tasks smaller
    – Better load balance
  • Naming
    – Make it easy to name data associated with stolen task
  • Synchronization
    – Support point-to-point synchronization
    – Better load balancing
Reducing Inherent Communication

- Communication required for parallel program
- Communication to Computation Ratio
  - (bytes / time) or (bytes / instruction)
- Affected by assignment (task → process)
- Domain decomposition
  - Interact with neighbors in space
  - Good for simulation of physical

\[
\text{Speedup} \leq \frac{\text{Sequential Work}}{\max(\text{Work on any processor})}
\]

\[
\text{Speedup} \leq \frac{\text{Sequential Work}}{\max(\text{Work + Synch Wait + Communication})}
\]
Reducing Extra Work

- **Redundant Computation**
  - If node would be idle anyway, compute data to avoid communication
  - Creating processes (high cost)

\[
\text{Speedup} \leq \frac{\text{Sequential Work}}{\max(\text{Work} + \text{Synch Wait} + \text{Communication} + \text{work})}
\]

Inherent vs. Artifactual Communication

- **Potential causes of artifactual communication**
  - Poor allocation of data
  - Unnecessary data in transfer
  - Unnecessary data transfer because of system granularity
  - Redundant communication
  - Limited capacity for replication
Cache Memory 101

• Locality + smaller HW is faster = memory hierarchy
  – Levels: each smaller, faster, more expensive/byte than level below
  – Inclusive: data found in top also found in the bottom

• Definitions
  – Upper is closer to processor
  – Block: minimum unit of data present or not in upper level
  – Frame: HW (physical) place to put block (same size as block)
  – Address = Block address + block offset address
  – Hit time: time to access upper level, including hit determination

• 3C Model (Cold/Compulsory, Capacity, Conflict)
• Add communication/coherence misses

Cache Coherent Shared Memory

(C) 2012 Alvin R. Lebeck from Adve, Falsafi, Hill, Reinhardt, Singh, Sorin
Compsci 221 / ECE 259
Cache Coherent Shared Memory

Orchestration for Performance

- **Exploit Temporal and Spatial Locality**
  - Temporal locality affects replication
  - Touch too much data → capacity misses
- **Computation Blocking**
Spatial Locality

• Granularities
  – Communication grain
  – Allocation grain
  – Coherence grain (for cache coherent shared memory)

• What benefits do you get from larger block size?

• Potential disadvantage is **false sharing**
  – Two or more processors accessing same cache block but don’t share any of the data

Poor Data Allocation

![Diagram of elements on the same page and cache block](image-url)
Data Blocking

Review: Programming for Performance

- Partitioning for Performance
  - Identify concurrency
  - Managing concurrency
    - Static
    - Dynamic
  - Granularity of concurrency
  - Serialization and synchronization costs
  - Communication

- Orchestration for Performance
  - Exploit Locality
  - Data and Computation Blocking
  - Match system (page size, cache block size)
Outline

- Applications
- Creating Parallel Programs
- Programming for Performance
- Scaling
- Synchronization Basics

Scaling: Why Talk About it?

- Speedup: change in performance as system parameter is scaled (e.g., number of processors, P)

- New problems on new machines
  - Problem scaling
  - Data set size
  - Algorithmic complexity

- Scaling is natural when simulating physical phenomena
  - Space is grid
  - Refine grid size
  - Larger grid
Questions in Scaling

• Fundamental question:

  What do real users actually do when they get access to larger parallel machines?

• Constant problem size
  – Just add more processors to speed up execution

• Memory constrained scaling
  – Scale data size linearly with # of processors
  – Can significantly increase execution time

• Time constrained scaling
  – Keep same wall clock time as processors are added
  – Solve largest problem in same amount of time

How to scale?

• Not just data

• Must consider application constraints
  – E.g., error scaling

• Equal error scaling
  – Scale all sources of error so they have equal contribution to total error
Example: Barnes-Hut Galaxy Simulation

- Different parameters govern different sources of error
  - Number of bodies \((n)\)
  - Time-step resolution \((dt)\)
  - Force calculation accuracy \((fa)\)

- Scaling Rule
  All components of simulation error should scale at the same rate

- Result: If \(n\) scales by a factor of \(s\)
  - \(dt\) must scale by \(s^{1/4}\)
  - \(fa\) must scale by \(s^{1/4}\)

Demonstrating Scaling Problems

- Small & big Ocean problems on SGI Origin2000

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>31</td>
<td>31</td>
</tr>
</tbody>
</table>

Ocean: 258 x 258

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>31</td>
<td>31</td>
</tr>
</tbody>
</table>

Ocean: 12 K x 12 K

Ideal
Problem Constrained Scaling

- User wants to solve same problem, only faster
  - E.g., Video compression & VLSI routing

  \[ \text{Speedup}_{PC}(p) = \frac{\text{Time}(1)}{\text{Time}(p)} \]

- Assessment
  - Good: easy to do & explain
  - May not be realistic
  - Doesn’t work well for much larger machine
    (c.f., Amdahl’s Law)

Time Constrained Scaling

- Execution time is kept fixed as system scales
  - User has fixed time to use machine or wait for result

- Performance = Work/Time as usual, and time is fixed, so

  \[ \text{Speedup}_{TC}(p) = \frac{\text{Work}(p)}{\text{Work}(1)} \]

- Assessment
  - Often realistic (e.g., best weather forecast overnight)
  - Must understand application to scale meaningfully
    (would scientist scale grid, time step, error bound, or combination?)
  - Execution time on a single processor can be hard to get
    (no uniprocessor may have enough memory)
Memory Constrained Scaling

- Scale so memory usage per processor stays fixed
- Scaled Speedup: Is $\frac{\text{Time}(1)}{\text{Time}(p)}$?

$$\text{Speedup}_{\text{MC}}(p) = \frac{\text{Work}(p)}{\text{Time}(p)} \times \frac{\text{Time}(1)}{\text{Work}(1)} = \frac{\text{Increase in Work}}{\text{Increase in Time}}$$

- Assessment
  - Realistic for memory-constrained programs (e.g., grid size)
  - Can lead to large increases in execution time if work grows faster than linearly in memory usage
  - E.g., matrix factorization
    - 10,000-by 10,000 matrix takes 800MB and 1 hour on uniprocessor
    - With 1,000 processors, can run 320K-by-320K matrix
    - But ideal parallel time grows to 32 hours!

Scaling Down

- Scale down to shorten evaluation time on hardware and especially on simulators
- “Scale up” issues apply in reverse

- Must watch out if problem size gets too small
  - Communication dominates computation (e.g., all boundary elements)
  - Problem size gets too small for realistic caches, yielding too many cache hits
    - Scale caches down considering application working sets
    - E.g., if a on a realistic problem a realistic cache could hold a matrix row but not whole matrix
    - Scale cache so it hold only row or scaled problem’s matrix
Outline

- Applications
- Creating Parallel Programs
- Programming for Performance
- Scaling
- Synchronization Basics

A Hierarchy of Synchronization

- Application programmer uses high-level library
- Library programmer uses hardware instructions
- Hardware implements atomic primitives
What the Application Programmer Sees

- Application programmer uses synch libraries

- Machine-independent (i.e., portable) interfaces

- E.g., pthreads provides synch methods
  - Barriers, locks

- Barrier
  - All processors wait at barrier until all others have reached it

- Lock
  - Lock restricts access to shared data to enforce mutual exclusion

What the Library Programmer Sees

- Libraries implement high-level synch interface
  - Can implement locks with different algorithms
  - E.g., can try to acquire with test & set or test & test & set

- Synch libraries must deal with hardware specifics
  - E.g., CM-5 has hardware support for barriers
  - All machines have atomic operations, but they’re different
  - Synch implementation might depend on system
What the Hardware Does

- All systems implement atomic operations
  - SPARC: Compare & Swap
  - Alpha: Load linked / Store conditional

- Libraries use these primitives to implement synch
  - Test & Test & Set algorithm could use Compare & Swap

Transactional Memory (TM)

- Alternate model of synchronization
  - Very hot topic in architecture community

- Concept of transaction
  - Atomic chunk of code
  - Either completely executes or doesn’t execute at all
  - Effects of transaction (writes to shared memory) are either all seen (by other processors) or not seen at all

- Goal
  - Simplify programming → transactions are “easier” than locks
  - But can’t just naively replace locks with transactions

- Much more about this topic, but mostly how to support transactions in hardware and/or software
<table>
<thead>
<tr>
<th>Outline</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Applications</td>
</tr>
<tr>
<td>• Creating Parallel Programs</td>
</tr>
<tr>
<td>• Programming for Performance</td>
</tr>
<tr>
<td>• Scaling</td>
</tr>
<tr>
<td>• Synchronization Basics</td>
</tr>
</tbody>
</table>