Parallel Programming Principle and Practice

Lecture 4 — Parallel Programming Methodology

Jin, Hai

School of Computer Science and Technology
Huazhong University of Science and Technology
Outline

- Motivating Problems
- Steps in Creating a Parallel Program
- What a Simple Parallel Program Looks Like
Parallel programming methodology

MOTIVATING PROBLEMS
Motivating Problems

- Simulating Ocean Currents
  - Regular structure, scientific computing

- Simulating the Evolution of Galaxies
  - Irregular structure, scientific computing

- Rendering Scenes by Ray Tracing
  - Irregular structure, computer graphics
Simulating Ocean Currents

- Model as **two-dimensional grids**
- Discretize in space and time
  - finer spatial and temporal resolution -> greater accuracy
- Many different computations per time step
  - set up and solve equations
- Concurrency across and within grid computations
Simulating Galaxy Evolution

- Simulate the interactions of many stars evolving over time
- Computing forces is expensive
- $O(n^2)$ brute force approach
- Hierarchical methods take advantage of force law: $G \frac{m_1 m_2}{r^2}$

- Many time-steps, plenty of concurrency across stars within one
Rendering Scenes by Ray Tracing

- Shoot rays into scene through pixels in image plane
- Follow their paths
  - They bounce around as they strike objects
  - They generate new rays: ray tree per input ray
- Result is color and opacity for that pixel
- Parallelism across rays

All case studies have abundant concurrency
Creating a Parallel Program

- **Assumption**: Sequential algorithm is given
  - Sometimes need very different algorithm, but beyond scope

- **Pieces of the job**
  - Identify work that can be done in parallel
  - Partition work and perhaps data among processes
  - Manage data access, communication and synchronization
  - *Note*: work includes computation, data access, and I/O

- **Main goal**: Speedup (plus low prog. effort and resource needs)
  \[
  \text{Speedup} (p) = \frac{\text{Performance}(p)}{\text{Performance}(1)}
  \]

- **For a fixed problem**
  \[
  \text{Speedup} (p) = \frac{\text{Time}(1)}{\text{Time}(p)}
  \]
Parallel programming methodology

STEPS IN CREATING PARALLEL PROGRAM
Some Important Concepts

- **Task**
  - Arbitrary piece of undecomposed work in parallel computation
  - Executed sequentially; concurrency is only across tasks
  - e.g. a particle/cell in Barnes-Hut, a ray or ray group in Raytrace
  - Fine-grained versus coarse-grained tasks

- **Process (thread)**
  - Abstract entity that performs the tasks assigned to processes
  - Processes communicate and synchronize to perform their tasks

- **Processor**
  - Physical engine on which process executes
  - Processes virtualize machine to programmer
    - first write program in terms of processes, then map to processors
Limited Concurrency: Amdahl’s Law

- Fundamental limitation on parallel speedup

- If $s =$ fraction of sequential execution that is inherently serial

  then $\text{speedup} \leq \frac{1}{s}$
Amdahl’s Law Example

- 2-phase computation over an $n$-by-$n$ grid
  - **Phase 1**: perform an independent computation on each grid element
    - easy to parallelize
  - **Phase 2**: add a value from each grid element into a global sum
    - more difficult to parallelize; serial by default

- **Sequential Execution**
  - both phases take $n^2$ time; $2n^2$ total
First Attempt at Parallelization

- **Strategy**
  - **Phase 1**: execute in parallel
    - time for phase 1 = \( \frac{n^2}{p} \)
  - **Phase 2**: execute serially
    - time for phase 2 = \( n^2 \)

- **Overall Performance**
  - Speedup \( \leq \frac{2n^2}{n^2/p + n^2} \)
  - i.e. no more than 2
Parallelizing Phase 2

- **Trick:** divide second phase into two steps
  - Step 1: accumulate into *private* sum during sweep
  - Step 2: add per-process private sum into *global* sum

- **Overall Performance:**
  - Parallel time = \( \frac{n^2}{p} + \frac{n^2}{p} + p \)
  - Speedup
    \[ \leq \frac{p2n^2}{2n^2 + p^2} \]
    close to \( p \) if \( n \gg p \)
Concurreny Profiles

- Cannot usually divide into serial and fully parallel parts

Area under curve is total work done, or time with 1 processor

Horizontal extent is lower bound on time (infinite processors)

Speedup is the ratio: \[
\frac{\sum_{k=1}^{\infty} f_k \frac{k}{p}}{\sum_{k=1}^{\infty} f_k \left\lfloor \frac{k}{p} \right\rfloor}, \text{ base case: } \frac{1}{s + \frac{1-s}{p}}
\]

Amdahl’s law applies to any overhead, not just limited concurrency
Steps in Creating a Parallel Program

4 steps: Decomposition, Assignment, Orchestration, Mapping

➢ Done by programmer or system software (compiler, runtime, ...)
➢ Issues are the same, so assume programmer does it all explicitly
Decomposition

- Break up computation into tasks to be divided among processes
  - i.e. Identify concurrency and decide level at which to exploit it

- Tasks may or may not be defined statically
  - Tasks may become available dynamically
  - Lots of available tasks may vary with time

- Goal: Enough tasks to keep processes busy, but not too many
  - Lots of tasks available at a time is upper bound on achievable speedup
Steps in Creating a Parallel Program

- 4 steps: Decomposition, Assignment, Orchestration, Mapping
Assignment

- Specifying mechanism to divide work among processes
  - e.g. which process computes forces on which stars, or which rays
  - Together with decomposition, also called partitioning
  - Goals: balance workload, reduce communication and management cost

- Structured approaches usually work well
  - Code inspection (parallel loops) or understanding of application
  - Well-known heuristics
  - Static versus dynamic assignment

- As programmers, we worry about partitioning first
  - Usually independent of architecture or programming model
  - But cost and complexity of using primitives may affect decisions

- As architects, we assume program does reasonable job of it
Steps in Creating a Parallel Program

- 4 steps: Decomposition, Assignment, Orchestration, Mapping
Orchestration

- **Main task**
  - Naming data
  - Structuring communication
  - Synchronization
  - Organizing data structures and scheduling tasks temporally

- **Goals**
  - Reduce cost of communication and synchronization as seen by processors
  - Preserve locality of data reference (incl. data structure organization)
  - Schedule tasks to satisfy dependences early
  - Reduce overhead of parallelism management

- **Closest to architecture** (and programming model & language)
  - Choices depend a lot on communication abstraction, efficiency of primitives
  - Architects should provide appropriate primitives efficiently
Steps in Creating a Parallel Program

- 4 steps: Decomposition, Assignment, Orchestration, Mapping
Mapping

- After orchestration, already have parallel program
- Two aspects of mapping
  - Which processes will run on same processor, if necessary
  - Which process runs on which particular processor
    - mapping to a network topology
- One extreme: space-sharing
  - Machine divided into subsets, only one application at a time in a subset
  - Processes can be pinned to processors, or left to OS
- Another extreme: complete resource management control to OS
  - OS uses the performance techniques we will discuss later
- Real world is between the two
  - User specifies desires in some aspects, system may ignore
- Usually adopt the view: process <-> processor
Parallelizing Computation vs. Data

- Above view is centered around computation
  - Computation is decomposed and assigned (partitioned)

- Partitioning data is often a natural view too
  - Computation follows data: owner computes
  - Grid example; data mining; High Performance Fortran (HPF)

- But not general enough
  - Distinction between computation and data stronger in many applications
    - Retain computation-centric view
    - Data access and communication is part of orchestration
High-level Goals

- **High performance** (speedup over sequential program)
  - But low resource usage and development effort
  - Implications for algorithm designers and architects
    - Algorithm designers: high-performance, low resource needs
    - Architects: high-performance, low cost, reduced programming effort
      - e.g. gradually improving performance with programming effort may be preferable to sudden threshold after large programming effort
Parallel programming methodology

WHAT PARALLEL PROGRAMS LOOK LIKE
Parallelization of an Example Program

- Motivating problems all lead to large, complex programs

- Examine simplified version of a piece of Ocean simulation
  - Iterative equation solver

- Illustrate parallel program in low-level parallel language
  - C-like pseudocode with simple extensions for parallelism
  - Expose basic comm. and synch. primitives that must be supported
  - State of most real parallel programming today
Grid Solver Example

- Simplified version of solver in Ocean simulation
- Gauss-Seidel (near-neighbor) sweeps to convergence
  - interior $n$-by-$n$ points of $(n+2)$-by-$(n+2)$ updated in each sweep
  - updates done in-place in grid, and diff from previous value computed
  - accumulate partial diffs into global diff at end of every sweep
  - check if error has converged to (within a tolerance parameter)
  - if so, exit solver; if not, do another sweep

Expression for updating each interior point:

1. int n;
2. float **A, diff = 0;
3. main()
4. begin
5. read(n); /*read input parameter: matrix size*/
6. A ← malloc (a 2-d array of size n + 2 by n + 2 doubles);
7. initialize(A); /*initialize the matrix A somehow*/
8. Solve(A); /*call the routine to solve equation*/
9. end main

10. procedure Solve(A)
11. begin
12. float **A; /*A is an (n + 2)-by-(n + 2) array*/
13. int i, j, done = 0;
14. float diff = 0, temp;
15. while (!done) do
16. diff = 0; /*outermost loop over sweeps*/
17. for i ← 1 to n do
18. for j ← 1 to n do /*sweep over nonborder points of grid*/
19. temp = A[i,j];
21. diff += abs(A[i,j] - temp);
22. end for
23. end for
24. if (diff/(n*n) < TOL) then done = 1;
25. end while
26. end procedure
Decomposition

- Simple way to identify concurrency is to look at loop iterations
  - *dependence analysis*; if not enough concurrency, then look further
- Not much concurrency here at this level (all loops *sequential*)
- Examine fundamental dependences, ignoring loop structure

- Concurrency $O(n)$ along anti-diagonals, serialization $O(n)$ along diag
- Retain loop structure, use pt-to-pt synch; Problem: too many synch ops
- Restructure loops, use global synch; imbalance and too much synch
Exploit Application Knowledge

- Reorder grid traversal: red-black ordering

- Different ordering of updates: may converge quicker or slower
- Red sweep and black sweep are each fully parallel
- Global synch between them (conservative but convenient)
- Ocean uses red-black; we use simpler, asynchronous one to illustrate
  - no red-black, simply ignore dependences within sweep
  - sequential order same as original, parallel program *nondeterministic*
Decomposition Only

15. while (!done) do  /* a sequential loop */
16.     diff = 0;
17.     for_all i ← 1 to n do  /* a parallel loop nest */
18.         for_all j ← 1 to n do
19.             temp = A[i,j];
22.             diff += abs(A[i,j] - temp);
23.         end for_all
24.     end for_all
25.     if (diff/(n*n) < TOL) then done = 1;
26. end while

- Decomposition into elements: degree of concurrency $n^2$
- To decompose into rows, make line 18 loop sequential; degree $n$
- for_all leaves assignment to the system
  - but implicit global synch. at end of for_all loop
Assignment

- **Static assignments** (given decomposition into rows)
  - **block** assignment of rows: Row $i$ is assigned to process $p_i$
  - **cyclic** assignment of rows: process $i$ is assigned rows $i, i+p, \ldots$, and so on

- **Dynamic assignment**
  - get a row index, work on the row, get a new row, and so on

- **Static assignment into rows reduces concurrency** (from $n$ to $p$)
  - block assignment reduces communication by keeping adjacent rows together

- Let’s dig into orchestration under three programming models
Data Parallel Solver

1. int n, nprocs; /*grid size (n + 2-by-n + 2) and number of processes*/
2. float **A, diff = 0;

3. main()
4. begin
5.  read(n); read(nprocs); /*read input grid size and number of processes*/
6.  A ← G_MALLOC (a 2-d array of size n+2 by n+2 doubles);
7.  initialize(A); /*initialize the matrix A somehow*/
8.  Solve (A); /*call the routine to solve equation*/
9.  end main

10. procedure Solve(A)
11.  float **A; /*solve the equation system*/
12.  begin
13.   int i, j, done = 0;
14.   float mydiff = 0, temp;
14a.  DECOMP A[BLOCK,*, nprocs]; /*A is an (n + 2-by-n + 2) array*/
15.   while (!done) do /*outermost loop over sweeps*/
16.      mydiff = 0; /*initialize maximum difference to 0*/
17.      for_all i ← 1 to n do /*sweep over non-border points of grid*/
18.         for_all j ← 1 to n do /*save old value of element*/
19.            temp = A[i,j];
22.            mydiff += abs(A[i,j] - temp);
23.         end for_all
24.      end for_all
24a.     REDUCE (mydiff, diff, ADD);
25.      if (diff/(n*n) < TOL) then done = 1;
26.    end while
27.  end procedure
Shared Address Space Solver

Single Program Multiple Data (SPMD)

- Assignment controlled by values of variables used as loop bounds
1. int n, nprocs; /*matrix dimension and number of processors to be used*/
2a. float **A, diff; /*A is global (shared) array representing the grid*/
2b. /*diff is global (shared) maximum difference in current
2c. sweep*/

2d. LOCKDEC(diff_lock); /*declaration of lock to enforce mutual exclusion*/
2e. BARDEC(bar1); /*barrier declaration for global synchronization between
2f. sweeps*/

3. main()
4. begin
5. read(n); read(nprocs); /*read input matrix size and number of processes*/
6. A ← G_MALLOC (a two-dimensional array of size n+2 by n+2 doubles);
7. initialize(A); /*initialize A in an unspecified way*/
8a. CREATE (nprocs-1, Solve, A);
8b. Solve(A); /*main process becomes a worker too*/
8c. WAIT_FOR_END (nprocs-1); /*wait for all child processes created to terminate*/
9. end main

10. procedure Solve(A)
11. float **A; /*A is entire n+2-by-n+2 shared array,
12. as in the sequential program*/
13. begin
14. int i, j, pid, done = 0;
15. float temp, mydiff = 0; /*private variables*/
16a. int mymin = 1 + (pid * n/nprocs); /*assume that n is exactly divisible by*/
16b. int mymax = mymin + n/nprocs - 1 /*nprocs for simplicity here*/
17. while (!done) do /*outer loop over all diagonal elements*/
18. mydiff = diff = 0; /*set global diff to 0 (okay for all to do it)*/
19a. BARRIER(bar1, nprocs); /*ensure all reach here before anyone modifies diff*/
19b. for i ← mymin to mymax do /*for each of my rows*/
20. for j ← 1 to n do /*for all nonborder elements in that row*/
21. temp = A[i,j];
24. mydiff += abs(A[i,j] - temp);
25a. LOCK(diff_lock); /*update global diff if necessary*/
25b. diff += mydiff;
25c. UNLOCK(diff_lock);
25d. BARRIER(bar1, nprocs); /*ensure all reach here before checking if done*/
25e. if (diff/(n*n) < TOL) then done = 1; /*check convergence; all get
25f. same answer*/
26. endwhile
27. end procedure
Notes on SAS Program

- **SPMD:** not lockstep or even necessarily same instructions
- Assignment controlled by values of variables used as loop bounds
  - Unique *pid* per process, used to control assignment
- “*Done*” condition evaluated redundantly by all
- Code that does the update identical to sequential program
  - Each process has private *mydiff* variable
- Most interesting special operations are for synchronization
  - Accumulations into shared *diff* have to be mutually exclusive
  - Why the need for all the barriers?
Need for Mutual Exclusion

- Code each process executes
  
  - load the value of `diff` into register `r1`
  
  - add the register `r2` to register `r1`
  
  - store the value of register `r1` into `diff`

- A possible interleaving

- Need the sets of operations to be atomic (mutually exclusive)
Mutual Exclusion

- Provided by LOCK-UNLOCK around *critical section*
  - Set of operations we want to execute atomically
  - Implementation of LOCK/UNLOCK must guarantee mutual exclusive

- Can lead to significant serialization if contended
  - Especially since expect non-local accesses in critical section
  - Another reason to use private *mydiff* for partial accumulation
Global Event Synchronization

- **BARRIER(nprocs):** wait here till $nprocs$ processes get here
  - Built using lower level primitives
  - Global sum example: wait for all to accumulate before using sum
  - Often used to separate phases of computation

<table>
<thead>
<tr>
<th>Process P_1</th>
<th>Process P_2</th>
<th>Process P_nprocs</th>
</tr>
</thead>
<tbody>
<tr>
<td>set up eqn system</td>
<td>set up eqn system</td>
<td>set up eqn system</td>
</tr>
<tr>
<td>Barrier (name, nprocs)</td>
<td>Barrier (name, nprocs)</td>
<td>Barrier (name, nprocs)</td>
</tr>
<tr>
<td>solve eqn system</td>
<td>solve eqn system</td>
<td>solve eqn system</td>
</tr>
<tr>
<td>Barrier (name, nprocs)</td>
<td>Barrier (name, nprocs)</td>
<td>Barrier (name, nprocs)</td>
</tr>
<tr>
<td>apply results</td>
<td>apply results</td>
<td>apply results</td>
</tr>
<tr>
<td>Barrier (name, nprocs)</td>
<td>Barrier (name, nprocs)</td>
<td>Barrier (name, nprocs)</td>
</tr>
</tbody>
</table>

- Conservative form of preserving dependences, but easy to use

- **WAIT_FOR_END (nprocs-1)**
Pt-to-pt Event Synch

- One process notifies another event so it can proceed
  - Common example: producer-consumer (bounded buffer)
  - Concurrent programming on uniprocessor: semaphores
  - Shared address space parallel programs: semaphores, or use ordinary variables as flags

```
P1
A = 1;
b: flag = 1;

a: while (flag is 0) do nothing;
print A;
```

- Busy-waiting or spinning
Group Event Synchronization

- Subset of processes involved
  - Can use flags or barriers (involving only the subset)
  - Concept of producers and consumers

- Major types
  - Single-producer, multiple-consumer
  - Multiple-producer, single-consumer
  - Multiple-producer, multiple-consumer
Message Passing Grid Solver

- Cannot declare $A$ to be shared array any more

- Need to compose it logically from per-process private arrays
  - Usually allocated in accordance with the assignment of work
  - Process assigned a set of rows allocates them locally

- Transfers of entire rows between traversals

- Structurally similar to SAS (e.g. SPMD), but orchestration different
  - Data structures and data access/naming
  - Communication
  - Synchronization
1. int pid, n, b; /* process id, matrix dimension and number of processors to be used */
2. float **myA;
3. main()
4. begin
5. read(n); read(nprocs); /* read input matrix size and number of processes */
8a. CREATE (nprocs-1, Solve); /* main process becomes a worker too */
8b. Solve();
8c. WAIT_FOR_END (nprocs-1); /* wait for all child processes created to terminate */
9. end main
10. procedure Solve()
11. begin
13. int i, j, pid, n' = n/nprocs, done = 0;
14. float temp, tempdiff, mydiff = 0; /* private variables */
6. myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2); /* my assigned rows of A */
7. initialize(myA); /* initialize my rows of A, in an unspecified way */
15. while (!done) do
16. mydiff = 0; /* set local diff to 0 */
16a. if (pid != 0) then SEND(&myA[1,0], n*sizeof(float), pid-1, ROW);
16b. if (pid != nprocs-1) then SEND(&myA[n',0], n*sizeof(float), pid+1, ROW);
16c. if (pid != 0) then RECEIVE(&myA[0,0], n*sizeof(float), pid-1, ROW);
16d. if (pid != nprocs-1) then RECEIVE(&myA[n'+1,0], n*sizeof(float), pid+1, ROW); /* border rows of neighbors have now been copied into myA[0,] and myA[n'+1,] */
17. for i ← 1 to n' do
18. for j ← 1 to n do /* for each of my (nghost) rows */
19. temp = myA[i,j];
21. mydiff += abs(myA[i,j] - temp);
22. endfor
23. endfor
24. /* communicate local diff values and determine if done; can be replaced by reduction and broadcast */
25a. if (pid != 0) then /* process 0 holds global total diff */
25b. SEND(mydiff, sizeof(float), 0, DIFF);
25c. RECEIVE(done, sizeof(int), 0, DONE);
25d. else /* pid 0 does this */
25e. for i ← 1 to nprocs-1 do /* for each other process */
25f. RECEIVE(tempdiff, sizeof(float), *, DIFF);
25g. mydiff += tempdiff; /* accumulate into total */
25h. endfor
25i. if (mydiff/(n*n) < TOL) then done = 1;
25j. for i ← 1 to nprocs-1 do /* for each other process */
25k. SEND(done, sizeof(int), i, DONE);
25l. endfor
25m. endif
26. endwhile
27. end procedure
Notes on Message Passing Program

- Use of ghost rows
- Receive does not transfer data, send does
  - Unlike SAS which is usually receiver-initiated (load fetches data)
- Communication done at beginning of iteration, so no asynchrony
- Communication in whole rows, not element at a time
- Core similar, but indices/bounds in local rather than global space
- Synchronization through sends and receives
  - Update of global diff and event synch for done condition
  - Could implement locks and barriers with messages
- Can use REDUCE and BROADCAST library calls to simplify code

```c
/* communicate local diff values and determine if done, using reduction and broadcast*/
25b.   REDUCE(0, mydiff, sizeof(float), ADD);
25c.   if (pid == 0) then
25i.    if (mydiff/(n*n) < TOL) then done = 1;
25k.    endif
25m.    BROADCAST(0, done, sizeof(int), DONE);
```
Ghost Points and Ghost Row
Send and Receive Alternatives

- Can extend functionality: stride, scatter-gather, groups
- Semantic flavors: based on when control is returned
  Affect when data structures or buffers can be reused at either end

  - Send/Receive
    - Synchronous
    - Asynchronous
      - Blocking asych.
      - Nonblocking asych.

  - Affect event synch (mutual exclusive: only one process touches data)
  - Affect ease of programming and performance

- Synchronous messages provide built-in synchronous through match
  - Separate event synchronization needed with asynchronous messages
- With synchronous messages, our code is deadlocked
Orchestration: Comparison

- **Shared address space**
  - Shared and private data explicitly separate
  - Communication implicit in access patterns
  - No correctness need for data distribution
  - Synchronization via atomic operations on shared data
  - Synchronization explicit and distinct from data communication

- **Message passing**
  - Data distribution among local address spaces needed
  - No explicit shared structures (implicit in communication patterns)
  - Communication is explicit
  - Synchronization implicit in communication (at least in synchronous case)
Summary in Grid Solver Program

- Decomposition and assignment similar in SAS and message-passing
- Orchestration is different
  - Data structures, data access/naming, communication, synchronization

<table>
<thead>
<tr>
<th></th>
<th>SAS</th>
<th>Msg-Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit global data structure?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Assignment independent of data layout?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Communication</td>
<td>Implicit</td>
<td>Explicit</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Explicit</td>
<td>Implicit</td>
</tr>
<tr>
<td>Explicit replication of border rows?</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
The content expressed in this chapter comes from