CS6560: Parallel Computer Architecture

Basics of Parallel Programming

Madhu Mutyam
PACE Laboratory
Department of Computer Science and Engineering
Indian Institute of Technology Madras

Feb 5 - 12, 2018

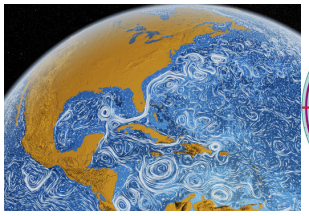
Understand Program Behavior

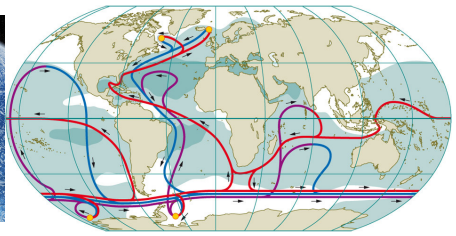
- Understand the behavior of programs that run on a computing system
  - Principle of locality – Design of memory hierarchy
  - Type of operations – Design of ISA
  - Frequency of various operations – Efficient design of common case
- Important for uniprocessor as well as multiprocessor systems
- Important for algorithm designers, programmers, and architects

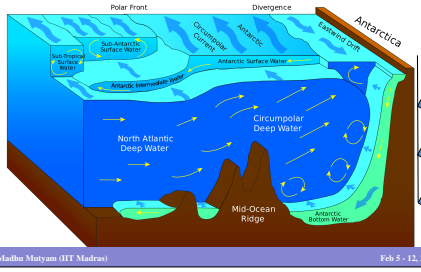
Parallel Application Case Studies

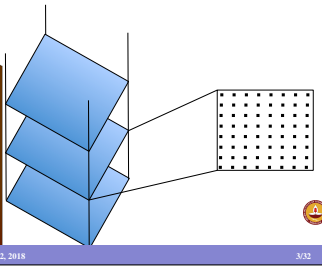
- Simulating Ocean Currents
- Simulating Evolution of Galaxies
- Visualizing Complex Scenes using Ray Tracing
- Mining Data for Associations

Simulating Ocean Currents

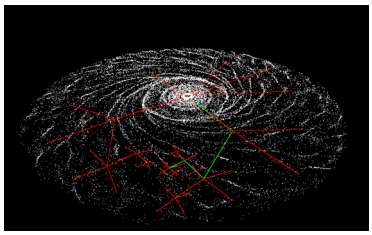




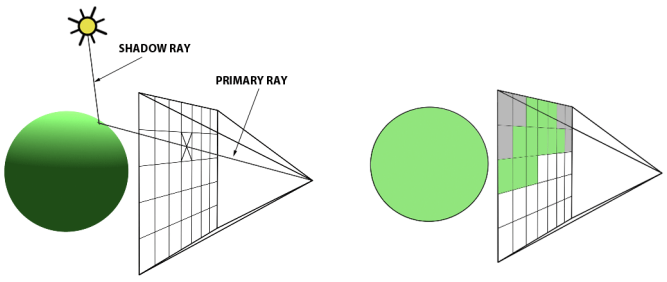




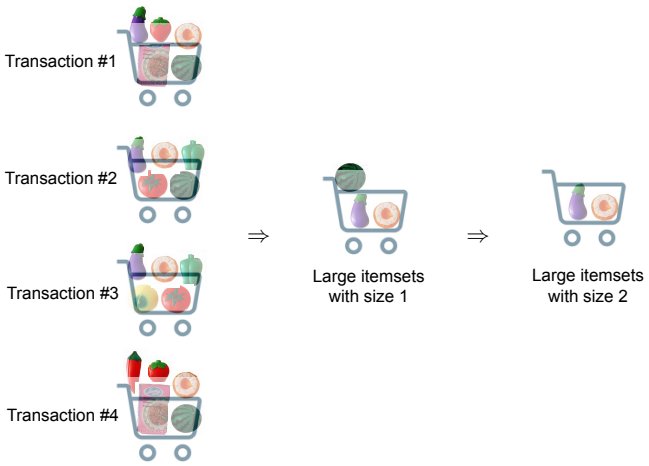
Simulating Evolution of Galaxies



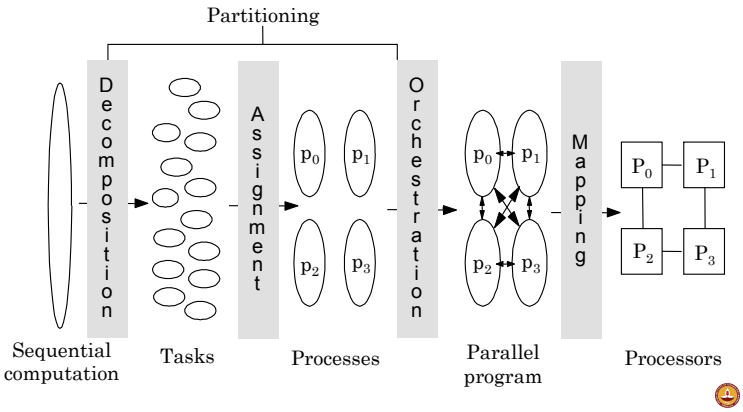
Visualizing Complex Scenes using Ray Tracing



Mining Data for Associations



The Parallelization Process



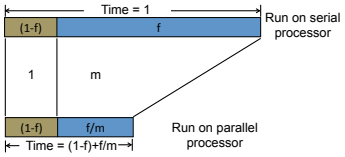
Decomposition

- Breaking the computation into a collection of tasks
- Provides an upper bound on the number of processes
- Expose enough concurrency without having too much overhead in managing the tasks

Amdahl's Law[a]

[a] Amdahl. Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities. AFIPS, 1967.

- Consider a workload w, where fraction f of work can be executed in parallel and the remaining in serial



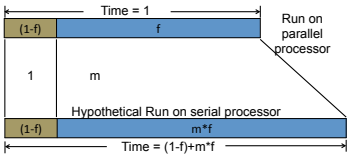
$$Speedup_{Amdahl} = \frac{w_{Sequential}}{w_{Parallel}} = \frac{1}{(1-f)+\frac{f}{m}}$$

- For small values of f, the enhancements will have little effect
- As $m \to \infty$, the speedup is bound by $\frac{1}{(1-f)}$
- Favours single processors
- Fixed-size speedup model

Gustafson's Law[a]

[a] Gustafson. Reevaluating Amdahl's Law. Comm. ACM, 1988.

- A machine with parallel computation ability lets more computations in the same amount of time
- w and $w' = ((1-f)+f*m)*w$ take the same amount of time executing with serial processing and parallel processing with m processors, resp.



$$Speedup_{Gustafson} = \frac{w'_{Sequential}}{w'_{Parallel}} = \frac{w'_{Sequential}}{w_{Sequential}}$$

$$Speedup_{Gustafson} = (1-f)+f \times m$$

- The speedup is a linear function of m if the workload is scaled up to maintain a fixed execution time
- Favours large-scale parallel system
- Fixed-time speedup model

Amdahl's Law in the Multicore Era[a]

[a] Hill and Marty. Amdahl's Law in the Multicore Era. IEEE Computer, 2008.

- Resources in a multicore system are bounded, say n
- Microarchitects can improve single-core performance using more of the bounded resources
- A **simple baseline core** consumes 1 Base Core Equivalent (BCE) resources and provides normalized performance of 1
- An **enhanced core** consumes r BCE resources and provides normalized performance of $perf(r)$
  - if $perf(r) > r$, speeds up for both sequential and parallel execution
  - if $perf(r) < r$, benefits sequential execution, but hurts parallel execution
  - $perf(r) = \sqrt{r}$ (Pollack's Law)

Symmetric Multicore Chips[a]

[a] Hill and Marty. Amdahl's Law in the Multicore Era,. IEEE Computer, 2008.

- The number of resources in a multicore chip is n
- The number of cores is $\frac{n}{r}$ with r BCE resources per core
- Serial fraction $(1-f)$ uses 1 core at rate $perf(r)$
- Serial time = $\frac{1-f}{perf(r)}$
- Parallel fraction (f) uses $\frac{n}{r}$ cores at rate $perf(r)$ each
- Parallel time = $\frac{f}{perf(r)\times\frac{n}{r}} = \frac{f\times r}{perf(r)\times n}$
- Speedup of symmetric multicore w.r.t. one base core:

$$Speedup_{Symmetric} = \frac{1}{\frac{(1-f)}{perf(r)} + \frac{f\times r}{perf(r)\times n}}$$

- If $m = \frac{n}{r}$,

$$Speedup_{Symmetric} = \frac{perf(r)}{(1-f) + \frac{f}{m}}$$

- Amdahl's law, fixed-size speedup model for multicore

Asymmetric Multicore Chips[a]

[a] Hill and Marty. Amdahl's Law in the Multicore Era. IEEE Computer, 2008.

- The number of resources in a multicore chip is n
- Enhance some (but not all) cores
- One enhanced core (r-*BCE*) and $(n-r)$ number of 1-*BCE* cores
- The number of cores is $1+n-r$
- Serial fraction $(1-f)$ uses 1 enhanced core at rate $perf(r)$
- Serial time = $\frac{1-f}{perf(r)}$
- Parallel fraction (f) uses one core at $perf(r)$ and $(n-r)$ cores at rate 1
- Parallel time = $\frac{f}{perf(r)+n-r}$
- Speedup of asymmetric multicore w.r.t. one base core:

$$Speedup_{Asymmetric} = \frac{1}{\frac{(1-f)}{perf(r)} + \frac{f}{perf(r)+n-r}}$$

Fixed-Time Speedup Model for Symmetric Multicore[a]

[a] Sun and Chen. Reevaluating Amdahl's Law in the Multicore Era. JPDC, 2010.

- From the fixed-size speedup model for symmetric multicore, the speedup achieved with $\frac{n}{r}$ cores, each having r *BCE* resources, is $\frac{1}{\frac{(1-f)}{perf(r)} + \frac{f\times r}{perf(r)\times n}}$
- If $n = r$, the speedup achieved with one core is $perf(r)$
- Time (t) to execute work w using one core at $perf(r)$ is $\frac{(1-f)\times w}{perf(r)} + \frac{f\times w}{perf(r)}$
- w' is the amount of work carried out by a multicore system with m cores such that $t = \frac{(1-f)\times w}{perf(r)} + \frac{f\times w'}{perf(r)\times m}$
- Thus, $w' = w \times m$
- The scaled up speedup, w.r.t. to single core system, is

$$\begin{aligned} Speedup_{FT\_Symmetric} &= \frac{w'_{Sequential}}{w_{Sequential}} = \frac{\frac{(1-f)\times w}{perf(r)} + \frac{f\times w'}{perf(r)}}{\frac{w}{perf(r)}} \\ &= (1-f) + m\times f \end{aligned}$$

Assignment

- Mechanism by which tasks will be distributed among processes
- Inspection of the code or high-level understanding of the application can help in appropriate assignment
- Static vs dynamic assignment
- Performance goals:
  - Workload balancing
  - Reducing interprocess communication
  - Reducing run-time overhead of managing assignment
- Decomposition and assignment are the major algorithmic steps in parallelization
  - Independent of the underlying architecture and programming model

Orchestration

- Deals with naming, organizing data structures, scheduling tasks, communicating, synchronizing
- Performance goals:
  - Reducing the cost of communication and synchronization
  - Preserving data locality
  - Scheduling tasks to satisfy dependences early
  - Reducing the overhead of parallelism management
- Architecture, programming model, and programming language play a big role

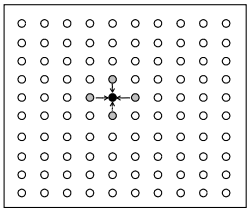
Mapping

- Deals with which process to run on which processor
- Specific to the system or programming environment
- Processes can be pinned to processors or controlled by OS dynamically
- Performance goals:
  - Keeping related processes in the same processor, if necessary
  - Exploiting locality in network topology

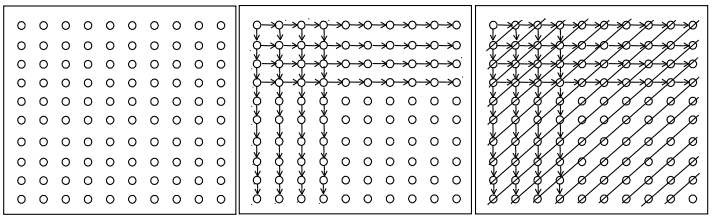
The Gauss-Seidel Equation Solver Kernel

```
1.  int n;
2.  float **A, diff=0;
3.  main()
4.  begin
5.    read(n);
6.    A← malloc(a 2-d array of size n+2-by-n+2 doubles);
7.    Initialize (A);
8.    Solve (A);
9.  end main

10. procedure Solve (A)
11. float **A;
12. begin
13.   int i, j, done=0;
14.   float diff=0, temp;
15.   while (!done) do
16.     diff = 0;
17.     for i← 1 to n do
18.       for j ←1 to n do
19.         temp = A[i,j];
20.         A[i,j]← 0.2*(A[i,j]+A[i,j-1]+A[i-1,j]+A[i,j+1]+A[i+1,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
```



Dependences and Concurrency in the Gauss-Seidel Equation Solver Kernel



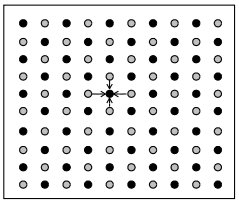
- Out of $O(n^2)$ work involved in each sweep, there exist $O(n)$ concurrency and $O(n)$ sequential dependence along anti-diagonals

Exploiting Concurrency in the Gauss-Seidel Equation Solver Kernel

- Assume the computation is decomposed at the grid-point granularity
- Retain the loop structure and insert point-to-point synchronization
  - the new value for a grid point has to be produced in the current sweep before it is used by the points below and to it right
  - Synchronization overhead is too high
- Restructure the loops and use global synchronization between iterations
  - The outer *for* loop can be over anti-diagonals and the inner *for* loop can be over elements within an anti-diagonal
  - $O(n)$ synchronizations and load imbalance

Exploiting Knowledge of Problem for Efficient Solution

- As Gauss-Seidel method is not an exact solution method, we can update grid points in *red-black* ordering



- Compute all $(\frac{n^2}{2})$ red points in parallel $\Rightarrow$ Global synchronization $\Rightarrow$ compute all $(\frac{n^2}{2})$ black points in parallel
- Produced results are independent of the number of processors

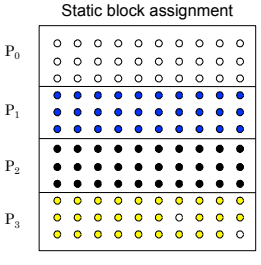
Decomposition

- Decompose into individual inner loop iterations
  - Degree of concurrency is n^2

```
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];
20.       A[i,j] ← 0.2*(A[i,j]+A[i,j-1]+A[i-1,j]+A[i,j+1]+A[i+1,j])
21.       diff +=abs(A[i,j]-temp);
22.     end for_all
23.   end for_all
24.   if (diff/(n*n) < TOL) then done = 1;
25. end while
```


Assignment

- Assume row-based decomposition; n rows and p processes
- Static Assignment
  - Cyclic assignment: process i gets rows $i, i+p, \cdots$
  - Block assignment: each process gets a group of rows



- $\frac{Communication}{Computation} = O(\frac{p}{n})$
- Dynamic Assignment
  - Each process gets a next available row

Orchestration Under the Data Parallel Model

```
1.  int n, nprocs;
2.  float **A, diff=0;
3.  main()
4.  begin
5.     read(n); read(nprocs);
6.     A ← G_MALLOC(2-d array of
           size(n+2)x(n+2)doubles);
7.     initialize(A);
8.     Solve(A);
9.  end main
```

```
10. procedure Solve(A)
11.   float **A;
12. begin
13.   int i, j, done = 0;
14.   float temp, mydiff = 0;
14a.  DECOMP A[BLOCK, *, nprocs];
15.   while (!done) do
16.     mydiff = 0;
17.     for_all i ← 1 to n do
18.       for_all j ← 1 to n do
19.         temp = A[i, j];
20.         A[i,j] = 0.2*(A[i,j] + A[i,j-1] + A[i-1,j]
                          + A[i,j+1] + A[i+1,j]);
21.         mydiff += abs(A[i,j]-temp);
22.       end for_all
23.     end for_all
24a.    REDUCE(mydiff, diff, ADD);
25.     if(diff/(n*n)<TOL) then done = 1;
26.   end while
27. end procedure
```


Orchestration Under the Shared Memory Model

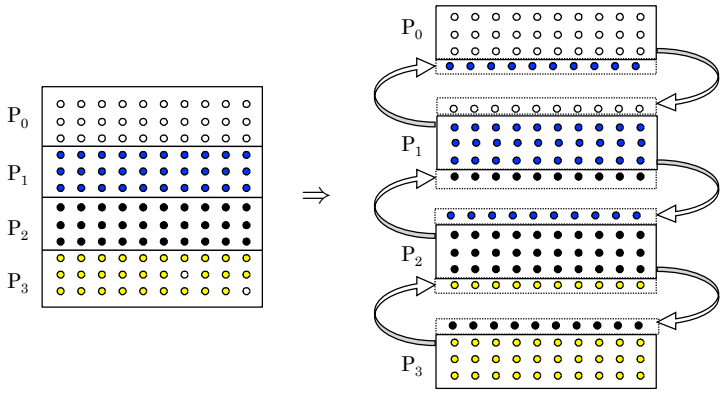
```
1.  int n, nprocs;
2a. float **A, diff;
2b. LOCKDEC(diff_lock);
2c. BARDEC (bar1);

3.  main()
4.  begin
5.     read(n); read(nprocs);
6.     A ← G_MALLOC(2-d array of size(n+2)x(n+2) doubles);
7.     initialize(A);
8a.    CREATE(nprocs-1, Solve, A);
8.     Solve(A);
8b.    WAIT_FOR_END(nprocs-1);
9.  end main
```


Orchestration Under the Shared Memory Model (Contd)

```
10. procedure Solve(A)
11.   float **A;
12. begin
13.   int i, j, pid, done = 0;
14.   float temp, mydiff = 0;
14a.  int mymin = 1 + (pid * n/nprocs);
14b.  int mymax = mymin + n/nprocs - 1;
15.   while (!done) do
16.     mydiff = diff = 0;
16a.    BARRIER(bar1, nprocs);
17.     for i ← mymin to mymax do
18.       for j ← 1 to n do
19.         temp = A[i, j];
20.         A[i,j] = 0.2*(A[i,j] + A[i,j-1] + A[i-1,j]
                          + A[i,j+1] + A[i+1,j]);
21.         mydiff += abs(A[i,j]-temp);
22.       end for
23.     end for
24a.    LOCK(diff_lock);
24b.    diff += mydiff;
24c.    UNLOCK(diff_lock);
24d.    BARRIER(bar1, nprocs);
24e.    if (diff/(n*n) < TOL) then done = 1;
24f.    BARRIER(bar1, nprocs);
25.   end while
26. end procedure
```


Data Layout Under the Message Passing Model



Orchestration Under the Message Passing Model

```
1. int n, pid, nprocs;
2. float **myA;
3. main()
4. begin
5.    read(n); read(nprocs);
8a.   CREATE(nprocs-1, Solve);
8b.   Solve();
8c.   WAIT_FOR_END(nprocs-1);
9. end main
```

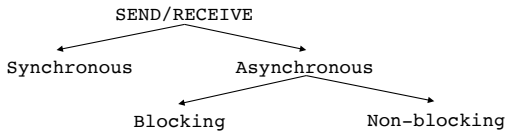
```
10.  procedure Solve()
11.  begin
12.    int i, j, pid, n'=n/nprocs, done=0;
13.    float temp, tempdiff, mydiff=0;,
 6.    myA←malloc(2-d array of size(n/nprocs+2)x(n+2));
 7.    initialize(myA);
15.    while(!done) do
16.      mydiff = 0;
16a.     if(pid!=0) SEND(&myA[1,0], n*sizeof(float), pid-1, ROW);
16b.     if(pid!=nprocs-1) SEND(&myA[n',0], n*sizeof(float),
                                pid+1, ROW);
16c.     if(pid!=0) RECEIVE(&myA[0,0], n*sizeof(float),
                                pid-1, ROW);
16d.     if(pid!=nprocs-1) RECEIVE(&myA[n'+1,0], n*sizeof(float),
                                pid+1, ROW);
```


Orchestration Under the Message Passing Model (Contd)

```
17.     for i←1 to n' do
18.       for j←1 to n do
19.         temp = myA[i,j];
20.         myA[i,j]=0.2*(myA[i,j] + myA[i,j-1] +
                    myA[i-1,j] + myA[i,j+1] + myA[i+1,j]);
21.         mydiff += abs(A[i,j]-temp);
22.       end for
23.     end for
24a.    if(pid!=0) then
24b.      SEND(mydiff, sizeof(float), 0, DIFF);
24c.      RECEIVE(done, sizeof(int), 0, DONE);
24d.    else
24e.      for i←1 to nprocs-1 do
24f.        RECEIVE(tempdiff, sizeof(float), *, DIFF);
24g.        mydiff += tempdiff;
24h.      end for
24i.      if(mydiff/(n*n)<TOL) then done = 1;
24j.      for i←1 to nprocs-1 do
24k.        SEND(done, sizeof(int), i, DONE);
24l.      end for
24m.    end if
25.   end while
26. end procedure
```

SEND/RECEIVE Alternatives

- Differ in their completion semantics – based on when they return control back to issuing process
  - Affect when the data structures/buffers can be reused without violating correctness



- Synchronous messages provide built-in synchronization through match
  - Separate event synchronizations are required for asynchronous messages

Summary

- Parallelizing a sequential application:
  - Depose the work into tasks
  - Assign the tasks to processes
  - Orchestrate data accesses, communication, and synchronization among processes
  - Map processes to processors
- Decomposition and assignment are independent of programming models and underlying hardware
- Communication is implicit in shared memory (SM) model while it is explicit in message passing (MP) model
- Synchronization primitives are required in SM but not in MP

Thank You