# Parallel Computer Architecture

A Hardware / Software Approach

by Culler and Singh, Morgan Kaufmann/Elsevier, 1998

## Chapter2

## Parallel Programs

The University of
British Columbia

# Overview

Last lecture set…

- Optimizing your sequential program
  - Compiler optimizations
  - Measuring performance of a single program
  - Summarizing performance – **distill** set of performance results into 1 magic number

This set…

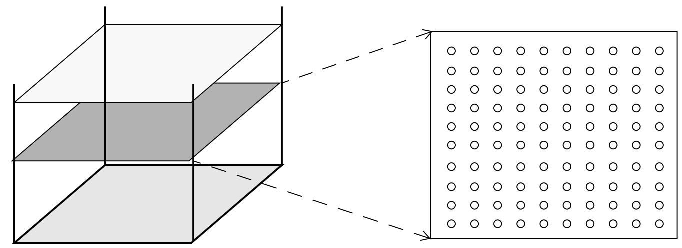
- Parallel programming
  - A general introduction
  - Readings – Culler text, Chapters 2 & 3

# Parallel Programming

Culler, Chapter 2

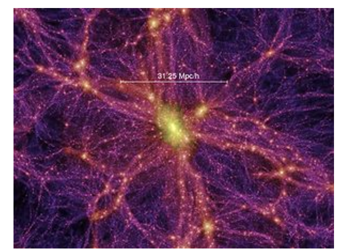
# Case Studies

- Simulating Ocean Currents
  (Regular grid)

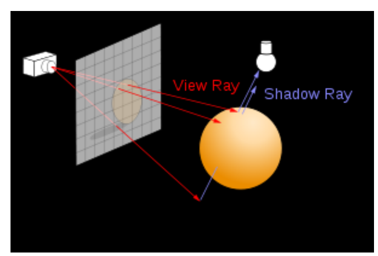


(a) Cross-sections (b) Spatial discretization of a cross-section

- Galaxy Simulation
  (Irregular)



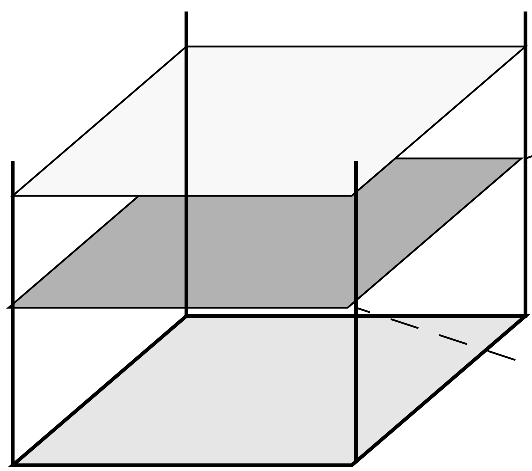
- Ray Tracing
  (individual beams)



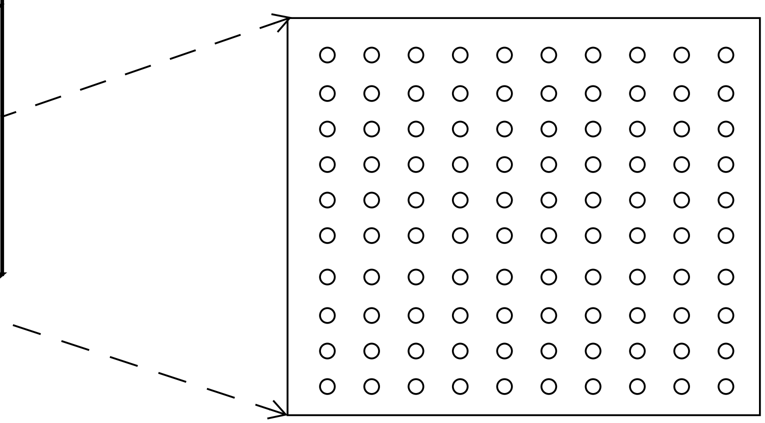

- Data Mining
  (dependencies among tables)

# Parallelizable Applications

- Ocean (similar to weather prediction)
  - Model current in ocean
  - Multi-dimensional grid points at each time stamp



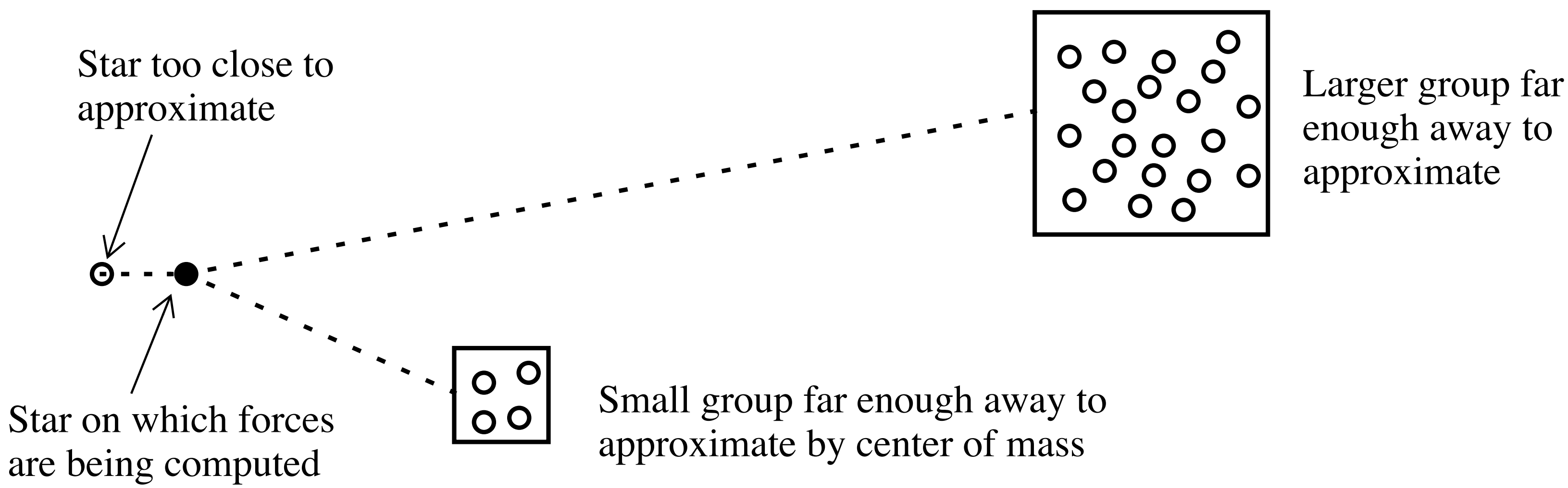
(a) Cross-sections



(b) Spatial discretization of a cross-section

# Parallelizable Applications

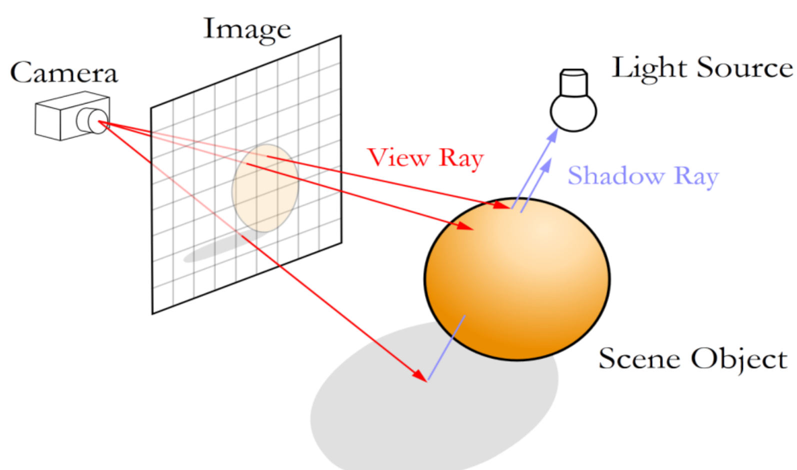
- Galaxies (N-body simulation)
  - Model force interactions among stars
  - Barnes-Hut:
    - Speed-up through approximation: group nearby stars together



# Raytrace Simulation

Rendering scenes by tracing:

- Produces high quality images but very expensive in computations
- Represent image as two-dimensional array of pixels
- Variables calculated for each pixel: color, opacity and brightness
- Shoot rays into scene through image plane, they strike and bounce
- Compute reflection, refraction and lighting interactions
- Concurrency is found for rays shot through different pixels




Source: Wikipedia

# Parallelizable Applications

- Data-Mining
  - Analyze correlations of attributes in a database
  - What is probability of:
    - buying item T2
    - given customer has bought item T1 ?
  - Determine itemset:
    - Set of k elements purchased together
  - Given that customer:
    - has set of items S1 in cart
    - what is probability of buying items in set S2 ?

# Parallelization Process

- At a high level, parallelization involves
  - Identifying what work can be done in parallel,
  - Determining how to distribute the work and perhaps the data among the processing nodes, and
  - Managing the necessary data access, communication and synchronization
- The of parallelization may be performed
  - Programmer
  - Compiler
  - Runtime system
  - Operating system

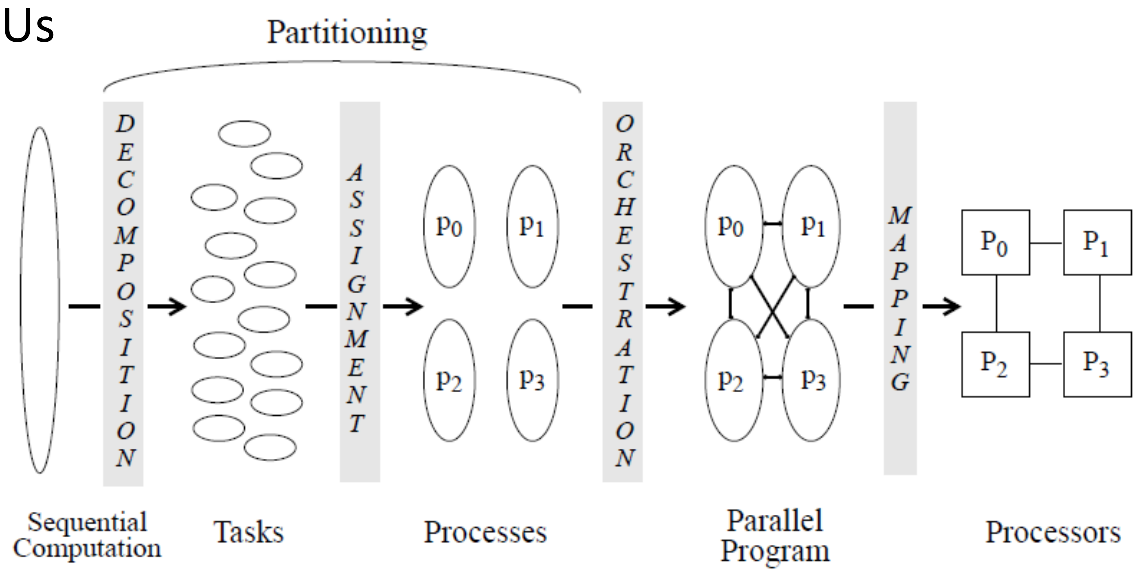
# Parallelization Process

- **Important concepts**
  - Task
    (Smallest unit of concurrency)
  - Process
    (An abstract entity that performs a subset of tasks)
  - Processor
    (Physical resource executing processes)

# Parallelization Process

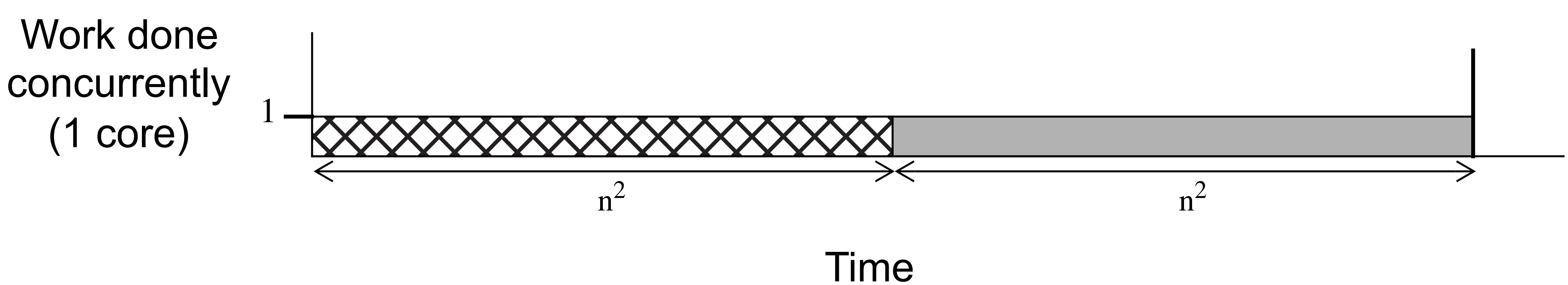
**Steps of Parallelization**

- Decomposition (into tasks)
- Assignment (group of tasks to processes)
- Orchestration (arrange communication between processes)
  - Select programming model: message passing and/or shared memory
- Mapping (assign processes to physical resources
  - Memory and CPUs

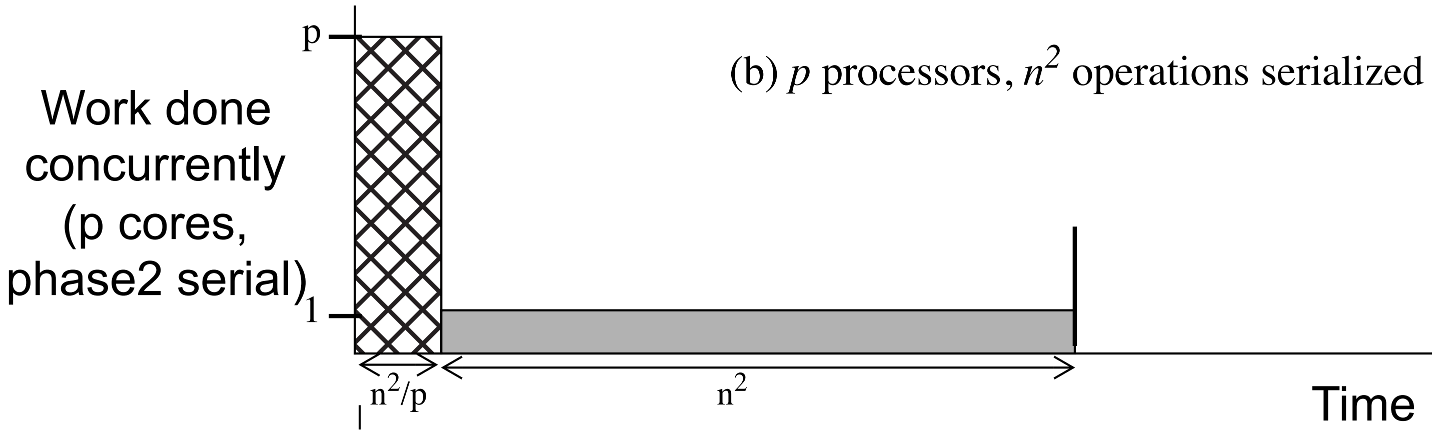


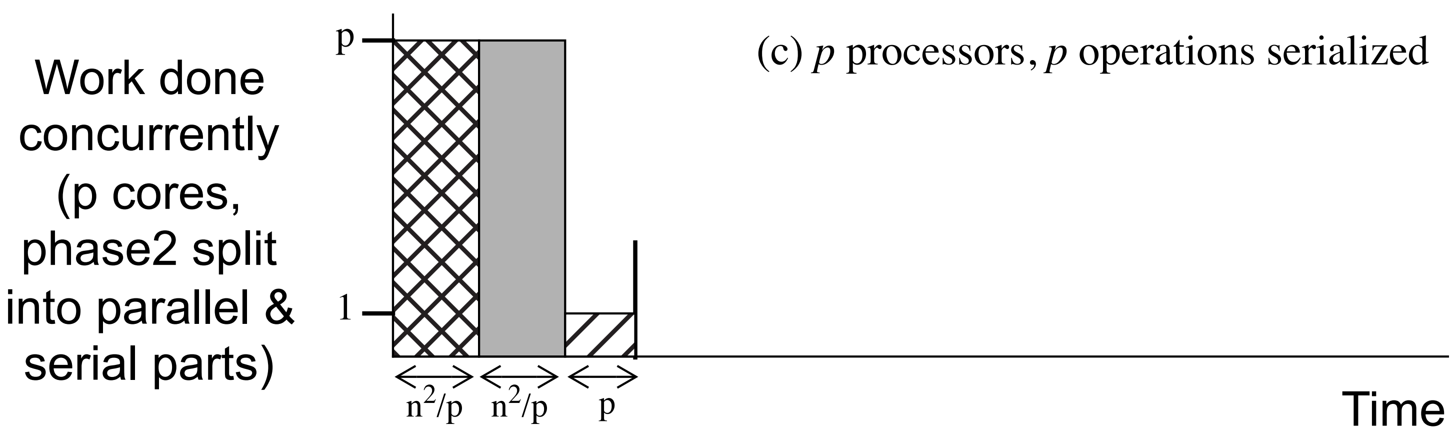
# Parallelization and Limits

- Example Problem
  - Two-phase algorithm
  - Phase 1: independently update $n^2$ data points
  - Phase 2: sum all $n^2$ data points

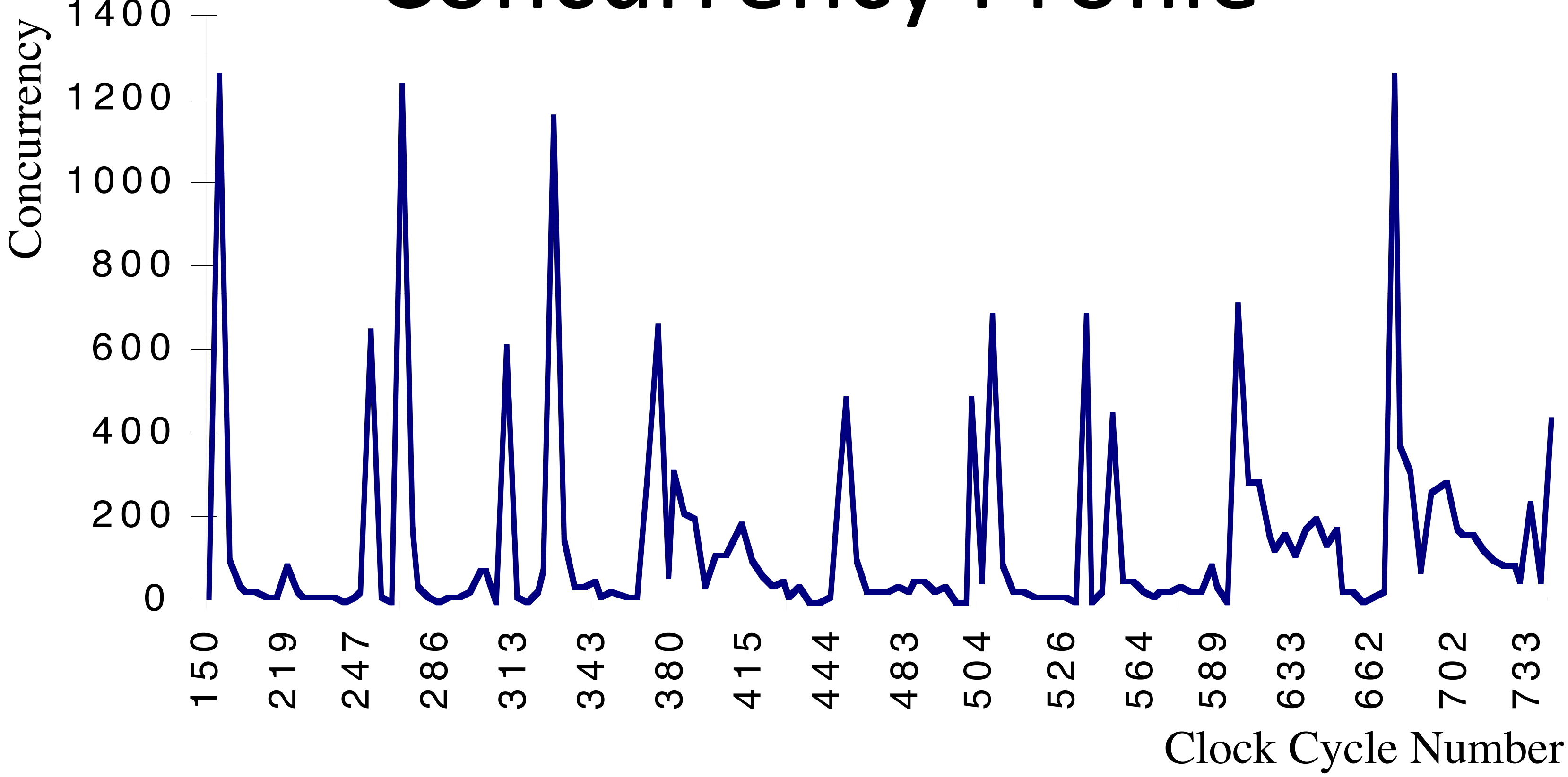


# Parallelization and Limits



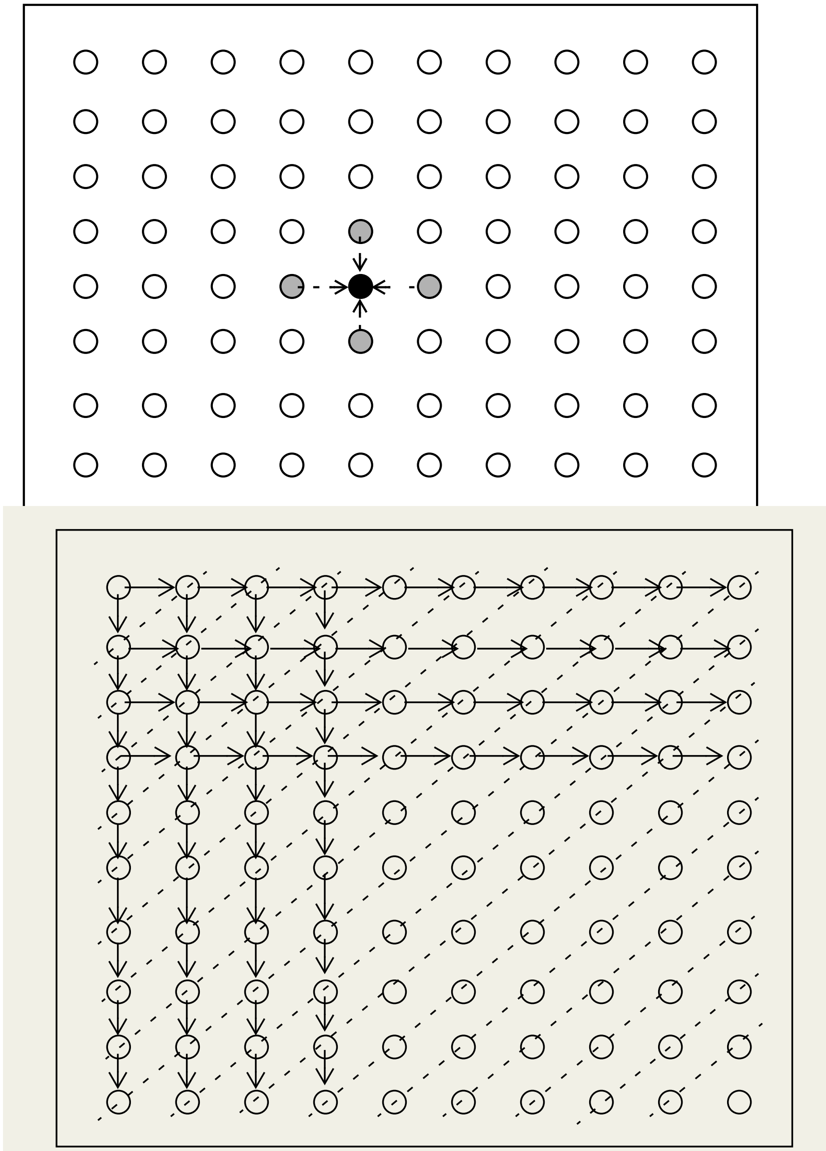



# Concurrency Profile




Concurrency profile for logic simulator
(Fig 2.5 in Culler)

# Case Study: Ocean

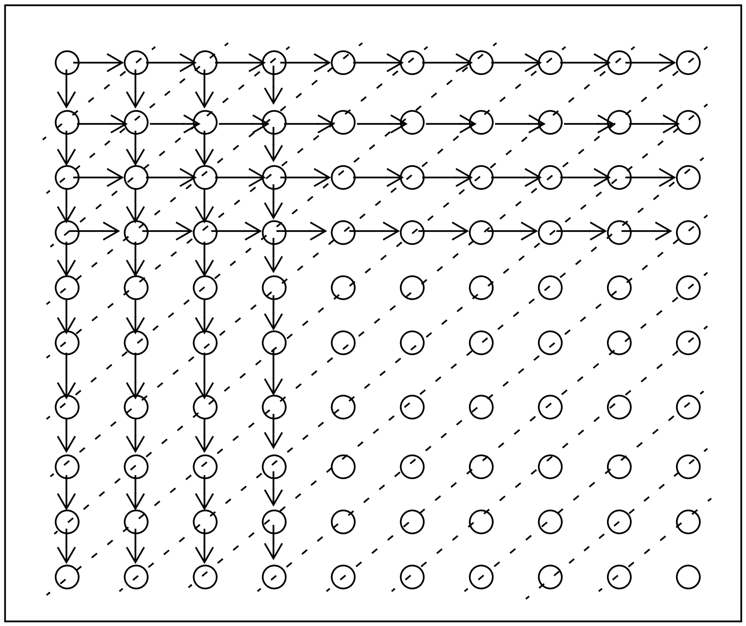


```
Do
   err=0;
   For each data point in grid
      temp = A[];
      A[] = 0.2 * ( … A[] … );
      err += abs(A[] – temp);
   End for
Until err < tolerance
```

In the sequential algorithm, we execute row-by-row, top-down

# Ocean: Decomposition

- Find the parallelism
- Can we divide grid points into groups
  - Compute each group on different processors?



Dependences and concurrency in the Gauss-Seidel equation solver computation.

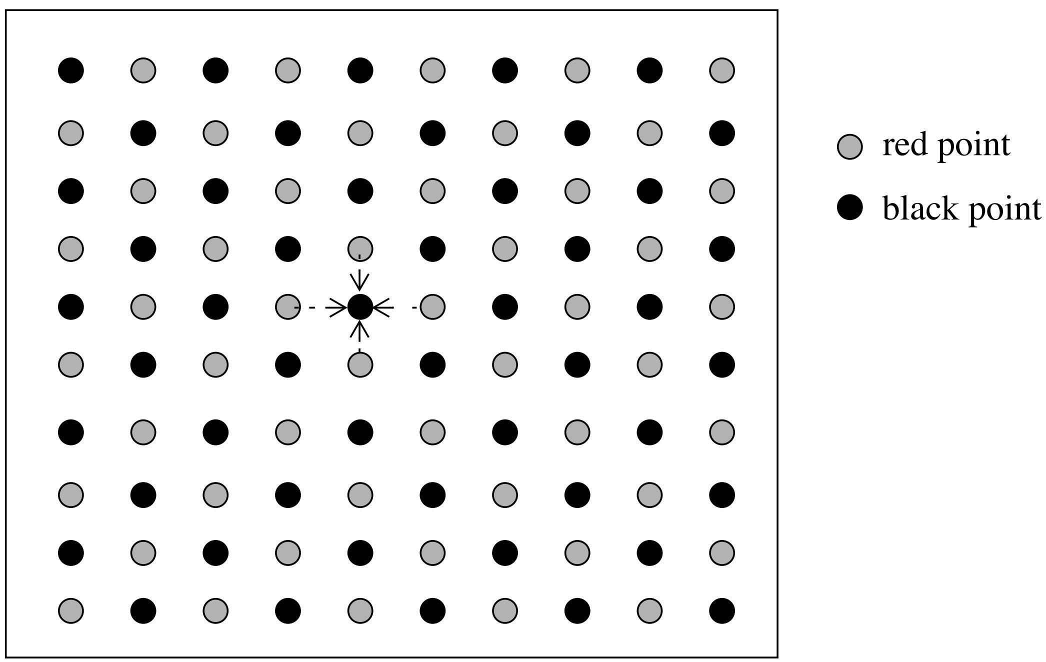
- Obstacles:
  - Each point has a data dependency on its neighbors
  - The final summation of data at each point

# Solution 1

- Analyze the dependence pattern
- Identify: no dependence along diagonal line
  - Each diagonal computed in sequence
  - Within each diagonal, points can be processed in parallel
- Problems
  - Load imbalance
    - Diagonal changes length each iteration !
  - Average parallelism N/2
  - Synchronization overhead

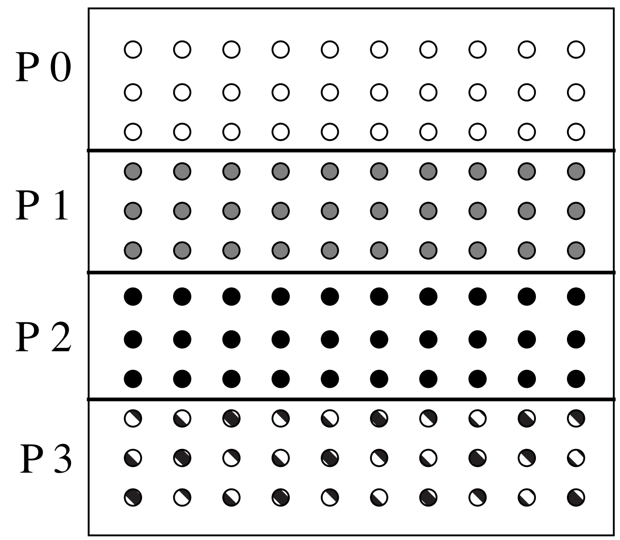
# Solution 2

- Divide grid points to dark and light colors in alternating order
  - Points of the same color are independent of each other
  - Average parallelism is $N^2/2$
- NB: This is a different algorithm, produces a different result!
  - Computation may be different, but still solves the problem (!)
  - More parallelism, but requires more iterations to converge



# Solution 3

- Each process assigned contiguous, equal number of rows

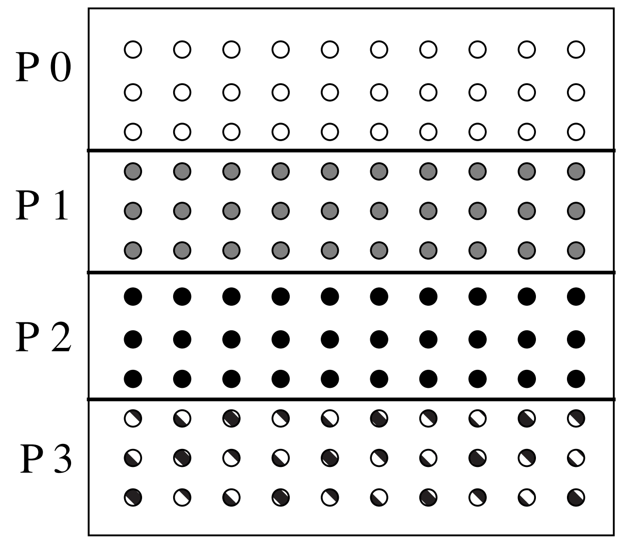



  - Note: this will not deliver the same result that we get from the sequential implementation!
- Communication required for adjacent rows
  - May need "ghost rows": a copy of adjacent rows to use locally
- Let's use this version going forward…

# Shared Memory

# Solution 3

- Each process assigned contiguous, equal number of rows




- **Communication required for rows at the partition boundaries**

# Ocean: Orchestration in Shared Memory

- Shared memory
  - Each process can access all data (any partition)

- *Barrier* is used to ensure all processes proceed in lock-step to the next sweep (iteration #)

- Requires *mutual exclusion* to modify any shared (global) variables
  - Eg, the final summation

# Shared Memory Parallel Programming Primitives

Table 2-2 Key Shared Address Space Primitives

| Name | Syntax | Function |
|---|---|---|
| CREATE | `CREATE(p,proc,args)` | Create `p` processes that start executing at procedure `proc` with arguments `args`. |
| G_MALLOC | `G_MALLOC(size)` | Allocate shared data of `size` bytes |
| LOCK | `LOCK(name)` | Acquire mutually exclusive access |
| UNLOCK | `UNLOCK(name)` | Release mutually exclusive access |
| BARRIER | `BARRIER(name, number)` | Global synchronization among `number` processes: None gets past BARRIER until `number` have arrived |
| WAIT_FOR_END | `WAIT_FOR_END(number)` | Wait for `number` processes to terminate |
| wait for flag | `while (!flag);` or `WAIT(flag)` | Wait for `flag` to be set (spin or block); for point-to-point event synchronization. |
| set flag | `flag = 1;` or `SIGNAL(flag)` | Set `flag`; wakes up process spinning or blocked on `flag`, if any |

# Orchestration – Shared memory

```
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 */
                                   /* diff is global (shared) maximum difference in current sweep */
2b.  LOCKDEC(diff_lock);           /* declaration of lock to enforce mutual exclusion */
2c.  BARDEC (bar1);                /* barrier declaration for global synchronization between 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);
8       Solve(A);                       /* main process becomes a worker too*/
8b.     WAIT_FOR_END;                   /* 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, as in the sequential program */
12.  begin
13.     int i,j, pid, done = 0;
14.     float temp, mydiff = 0;                   /* private variables/
14a.    int mymin ← 1 + (pid * n/nprocs);          /* assume that n is exactly divisible by  */
14b.    int mymax ← mymin + n/nprocs - 1;          /* nprocs for simplicity here*/

15.  while (!done) do                             /* outer loop over all diagonal elements */
16.     mydiff = diff = 0;                        /* set global diff to 0 (okay for all to do it) */
17.     for i ← mymin to mymax do                 /* for each of my rows */
18.           for j ← 1 to n do                   /* for all elements in that row */
19.                 temp = A[i,j];
20.                 A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                       A[i,j+1] + A[i+1,j]);
22.                 mydiff += abs(A[i,j] - temp);
23.           endfor
24.     endfor
25a.    LOCK(diff_lock);                    /* update global diff if necessary */
25b.          diff += mydiff;
25c.    UNLOCK(diff_lock);
25d.    BARRIER(bar1, nprocs);                    /* ensure all have got here before checking if done*/

25e.    if (diff/(n*n) < TOL) then done = 1;  /* check convergence; all get same answer*/
25f.    BARRIER(bar1, nprocs);              /* see Exercise c */
26.  endwhile
27.end procedure
```

# Orchestration – Shared memory

```
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 */
                             /* diff is global (shared) maximum difference in current sweep */
2b. LOCKDEC(diff_lock);      /* declaration of lock to enforce mutual exclusion */
2c. BARDEC (bar1);           /* barrier declaration for global synchronization between sweeps*/
```

What is a lock? (mutual exclusion)

- Only one process can ”acquire” lock at a time
- Other processes wait until lock is available

What is a barrier? (wait for rendezvous)

- Each process stops & waits
- After all processes reach barrier, they can proceed

```
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*/
```

G_MALLOC: allocate from shared memory region

# Orchestration – Shared memory

```
    CREATE (nprocs-1, Solve, A);
    Solve(A);                    /* main process becomes a worker too*/
    WAIT_FOR_END;                /* wait for all child processes created to terminate */
end main
```

- CREATE(p,proc,args)
  - Create p processes
- WAIT_FOR_END(number)
  - Wait for number processes to terminate

- Pseudo code is implementable in Pthread library in C:

  CREATE

  *int* **pthread_create**(*pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void *), void *restrict arg*);

  WAIT_FOR_END

  *int* **pthread_join**(*pthread_t thread, void **value_ptr*);

# Orchestration – Shared memory

```
while (!done) do                              /* outer loop over all diagonal elements */
  mydiff = diff = 0;                          /* set global diff to 0 (okay for all to do it) */
  for i ← mymin to mymax do                   /* for each of my rows */
        for j ← 1 to n do                     /* for all elements in that row */
              temp = A[i,j];
              A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
                    A[i,j+1] + A[i+1,j]);
              mydiff += abs(A[i,j] - temp);
        endfor
  endfor
```

Main algorithm

- Similar to serial code
- Each thread computes one part of the array
- Each thread computes the diff of its part (local variable “mydiff”)

# Orchestration – Shared memory

- Serial part – how to add all mydiffs – needs a LOCK

```
25a.  LOCK(diff_lock);              /* update global diff if necessary */
25b.        diff += mydiff;
25c.  UNLOCK(diff_lock);
```

- Why do we need a lock?

| P1 | P2 | |
|---|---|---|
| `r1 ← diff` | | *{P1 gets 0 in its r1}* |
| | `r1 ← diff` | *{P2 also gets 0}* |
| `r1 ← r1+r2` | | *{P1 sets its r1 to 1}* |
| | `r1 ← r1+r2` | *{P2 sets its r1 to 1}* |
| `diff ← r1` | | *{P1 sets cell_cost to 1}* |
| | `diff ← r1` | *{P2 also sets cell_cost to 1}* |

  - each CPU needs to execute this read/add/write atomically

# Orchestration – Shared memory

Process/Thread Synchronization:

```
25d.  BARRIER(bar1, nprocs);            /* ensure all have got here before checking if done*/

25e.  if (diff/(n*n) < TOL) then done = 1; /* check convergence; all get same answer*/
25f.  BARRIER(bar1, nprocs);      /* see Exercise c */
```

- First barrier
  - Ensures all threads update diff

- Second barrier
  - Get ready for next iteration

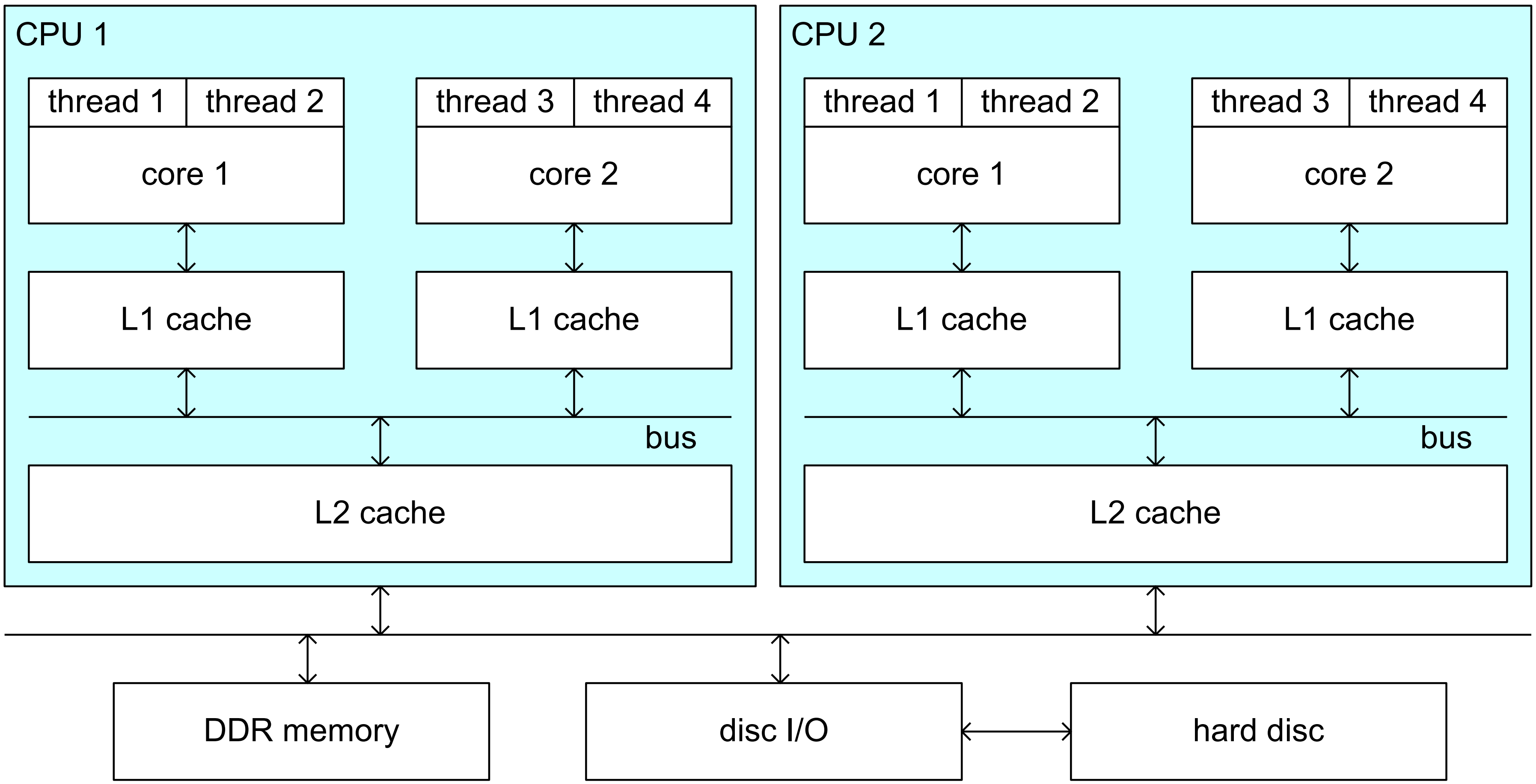
# Main Part + Lock + Barriers

```
while (!done) do                                  /* outer loop over all diagonal elements */
  mydiff = diff = 0;                              /* set global diff to 0 (okay for all to do it) */
  for i ← mymin to mymax do                       /* for each of my rows */
        for j ← 1 to n do                         /* for all elements in that row */
              temp = A[i,j];
              A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
                    A[i,j+1] + A[i+1,j]);
              mydiff += abs(A[i,j] - temp);
        endfor
  endfor
  LOCK(diff_lock);                          /* update global diff if necessary */
        diff += mydiff;
  UNLOCK(diff_lock);
  BARRIER(bar1, nprocs);                          /* ensure all have got here before checking if done*/

  if (diff/(n*n) < TOL) then done = 1;  /* check convergence; all get same answer*/
  BARRIER(bar1, nprocs);                /* see Exercise c */
endwhile
end procedure
```

# Orchestration – Shared memory

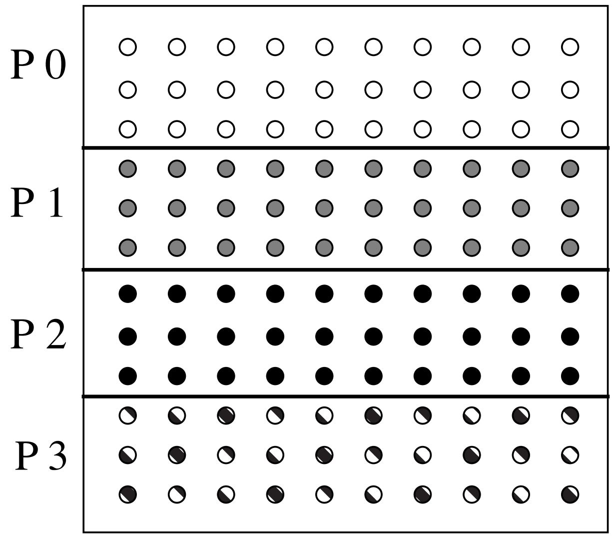
- Note that you have a memory hierachy: data locality?
  Commonly: Non-Uniform Memory Access (NUMA)



# Message Passing

# Solution 3

- Each process assigned contiguous, equal number of rows




- Communication required for rows at the partition boundaries
  - Must send messages containing "row updates" to neighbours

# Ocean: Orchestration in Message Passing

- Message passing
  - Each partition of points is stored on separate computer (private address space)
- No global variables / no shared memory
- Border rows need to be copied among neighbors
  - "Ghost rows"
- One process is responsible for global summation
  - Lock is implied, not explicit

# Orchestration – Message passing

```
while (!done) do
  mydiff = 0;                          /*set local diff to 0*/
  if (pid != 0) then SEND(&myA[1,0],n*sizeof(float),pid-1,ROW);
  if (pid != nprocs-1) then
     SEND(&myA[n',0],n*sizeof(float),pid+1,ROW);
  if (pid != 0) then RECEIVE(&myA[0,0],n*sizeof(float),pid-1,ROW);
  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,*]*/
  for i ← 1 to n' do                   /*for each of my (nonghost) rows*/
    for j ← 1 to n do                  /*for all nonborder elements in that row*/
      temp = myA[i,j];
      myA[i,j] = 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
        myA[i,j+1] + myA[i+1,j]);
      mydiff += abs(myA[i,j] - temp);
    endfor
  endfor
                                       /*communicate local diff values and determine if
                                       done; can be replaced by reduction and broadcast*/
  if (pid != 0) then                   /*process 0 holds global total diff*/
    SEND(mydiff,sizeof(float),0,DIFF);
    RECEIVE(done,sizeof(int),0,DONE);
  else                                 /*pid 0 does this*/
    for i ← 1 to nprocs-1 do           /*for each other process*/
      RECEIVE(tempdiff,sizeof(float),*,DIFF);
      mydiff += tempdiff;              /*accumulate into total*/
    endfor
    if (mydiff/(n*n) < TOL) then   done = 1;
    for i ← 1 to nprocs-1 do           /*for each other process*/
      SEND(done,sizeof(int),i,DONE);
    endfor
  endif
endwhile
end procedure
```

# Message passing – update ghost rows

```
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,*]*/
```

# Message passing – main algorithm

```
                                     /*border rows of neighbors have now been copied
                                     into myA[0,*] and myA[n'+1,*]*/
for i ← 1 to n' do                   /*for each of my (nonghost) rows*/
  for j ← 1 to n do                  /*for all nonborder elements in that row*/
    temp = myA[i,j];
    myA[i,j] = 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
      myA[i,j+1] + myA[i+1,j]);
    mydiff += abs(myA[i,j] - temp);
  endfor
endfor
                                     /*communicate local diff values and determine if
                                     done; can be replaced by reduction and broadcast*/
```

# Message passing – add “mydiff” errors

```
if (pid != 0) then                                /*process 0 holds global total diff*/
   SEND(mydiff,sizeof(float),0,DIFF);
   RECEIVE(done,sizeof(int),0,DONE);
else                                              /*pid 0 does this*/
   for i ← 1 to nprocs-1 do                       /*for each other process*/
      RECEIVE(tempdiff,sizeof(float),*,DIFF);
      mydiff += tempdiff;                         /*accumulate into total*/
   endfor
```

# Orchestration – are we done yet?

```
if (pid != 0) then                              /*process 0 holds global total diff*/
   SEND(mydiff,sizeof(float),0,DIFF);
   RECEIVE(done,sizeof(int),0,DONE);
else                                            /*pid 0 does this*/
   for i ← 1 to nprocs-1 do                     /*for each other process*/
      RECEIVE(tempdiff,sizeof(float),*,DIFF);
      mydiff += tempdiff;                       /*accumulate into total*/
   endfor
   if (mydiff/(n*n) < TOL) then   done = 1;
   for i ← 1 to nprocs-1 do                     /*for each other process*/
      SEND(done,sizeof(int),i,DONE);
   endfor
endif
endwhile
end procedure
```

# Ocean: Orchestration in Message Passing

- Message is
  - Sent by calling "Send"
  - Received by calling "Receive"

- Ideally, the sender will wait (block) until the receiver has received the data
  - Thus, communication also establishes synchronization

- There are several "send" and "receive" modes which allow non-blocking behaviour

# Orchestration – Message passing

- Blocking Vs non-blocking
  - **Blocking send**
    - Returns when sending buffer is available for reuse
    - MPI_Send
  - **Blocking receive**
    - Waits until data has arrived in receiving buffer
    - MPI_Recv

  - **Non-blocking send and receive**
    - "start sending" and "start receiving"
    - Routine returns immediately, can do other work
    - Should be followed by MPI_Wait
      - MPI_Send = MPI_Isend + MPI_Wait
      - MPI_Recv = MPI_Ireceive + MPI_Wait

# Type of Communication

- Send Modes (MPI Standard, version 3)
  - Standard:
    - **either** sender blocks until matching receive,
    - **or** sender returns right away after copying data into a buffer (safe to re-use original send buffer)
  - Buffered (Bsend):
    - sender returns right away (data copied to new buffer)
  - Synchronous (Ssend):
    - send completes only when matching receive is posted
  - Ready (Rsend):
    - send executes only if the matching receive has already been executed and is ready to receive; otherwise undefined

# Type of Communication

- Receive Modes
  - Only one type of receive
  - Blocking: MPI_Recv()
  - Non-blocking: MPI_Irecv()
    - see also MPI_Isend()
    - The “I” means “immediate”, i.e. the system may start to receive data, but call will return before receive is finished allowing concurrent communication

# Type of Communication

- Completion
  - MPI_Wait()
  - MPI_Test()

- Probe
  - Check for incoming messages, without actually receiving
  - Determine message size + allocate buffer before receive
  - MPI_Probe() is blocking
  - MPI_IProbe() is non-blocking

# Orchestration – Message passing

- **Synchronous vs Asynchronous**
  - Synchronous mode ensures that destination process has started to receive the message
  - Asynchronous doesn't know anything about destination process state

# MPI Details

- Read the “MPI Standard”
  - http://www.mpi-forum.org/docs/
  - Latest is Version 3.1, June 2015
    - http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
  - Previous version 3.0, Sept 2012

- Version used on hydra2 is “OpenMPI”
  - Run “ompi_info” on command line
  - Installed version is OpenMPI 1.6.5, June 2013
    - (This is not the same as MPI standard version 1.6!!)
    - Implements MPI Standard 2.1

# Implementation

**WARNING: OpenMP != OpenMPI**

- OpenMP is totally different
- OpenMP: shared-memory OpenMPI: message-passing
- OpenMP: you add pragmas to C code (parallelism hints or directives)
- MPI is a general interface for communication between processors that be on entirely different computers (even heterogeneous!)

- **Shared memory programming APIs:**
  - **OpenMP (Open Multi-Processing, we will not use this)**
  - **Pthreads (API for creating and manipulating threads)**
- **Message passing programming APIs:**
  - **OpenMPI (Open Message Passing Interface)**

# Assignment 1

- Assignment 1
  - Only do Gaussian Elimination algorithm
  - Do only forward-elimination steps
    - **Parallelize and measure run-time using OpenMPI**

- Read
  - Culler text, Chapter 3
  - Will go over:
    - Message-passing code
    - Shared-memory code

# Parallel Programming

Culler, Chapter 3

# Culler, Chapter 3 Programming for Performance

- 3.1 Reduce sources of waste
  - Load balancing
  - Reduce communication
  - Reduce serial work (make parallel)
  - Reduce extra work
- Skip 3.2, 3.3, 3.4
- 3.5 Case studies (Ocean, Barnes-Hut, etc)
- Skip 3.6

On your own…

# Hennessy & Patterson, Ch. 4
## Multiprocessors and Thread-level Parallelism

- Skip 4.1
- 4.2 Symmetric Shared-Memory Architectures (SMPs)
  - Cache coherence
- 4.3 Performance Study of SMPs
- 4.4 Directory-based Coherence
- 4.5 Synchronization (Locks, Barriers)
- Skip 4.6, 4.7, 4.8

# Key takeaway

- Two separate issues
  - Cache coherence
  - Memory consistency
- Write a definition for yourself!
  - What is the difference?