Introduction to Parallelism

Todd C. Mowry & Dave Eckhardt

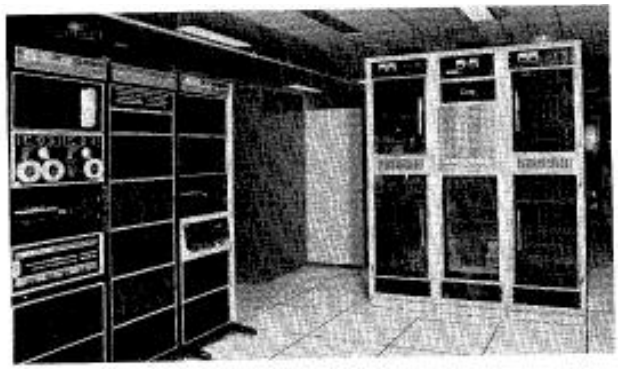
I. Why is parallel processing so important now?

II. What's new and different with parallelism?

III. Parallel Programming Models

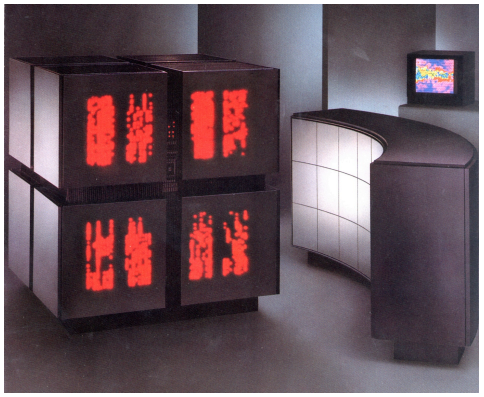
A Brief History of Parallel Processing

- Initial Focus (starting in 1970's): "Supercomputers" for Scientific Computing



C.mmp at CMU (1971)
16 PDP-11 processors

Cray XMP (circa 1984)
4 vector processors



Thinking Machines CM-2 (circa 1987)
65,536 1-bit processors +
2048 floating-point co-processors

SGI UV 1000cc-NUMA (today)
4096 processor cores

← Blacklight at the Pittsburgh Supercomputer Center

A Brief History of Parallel Processing

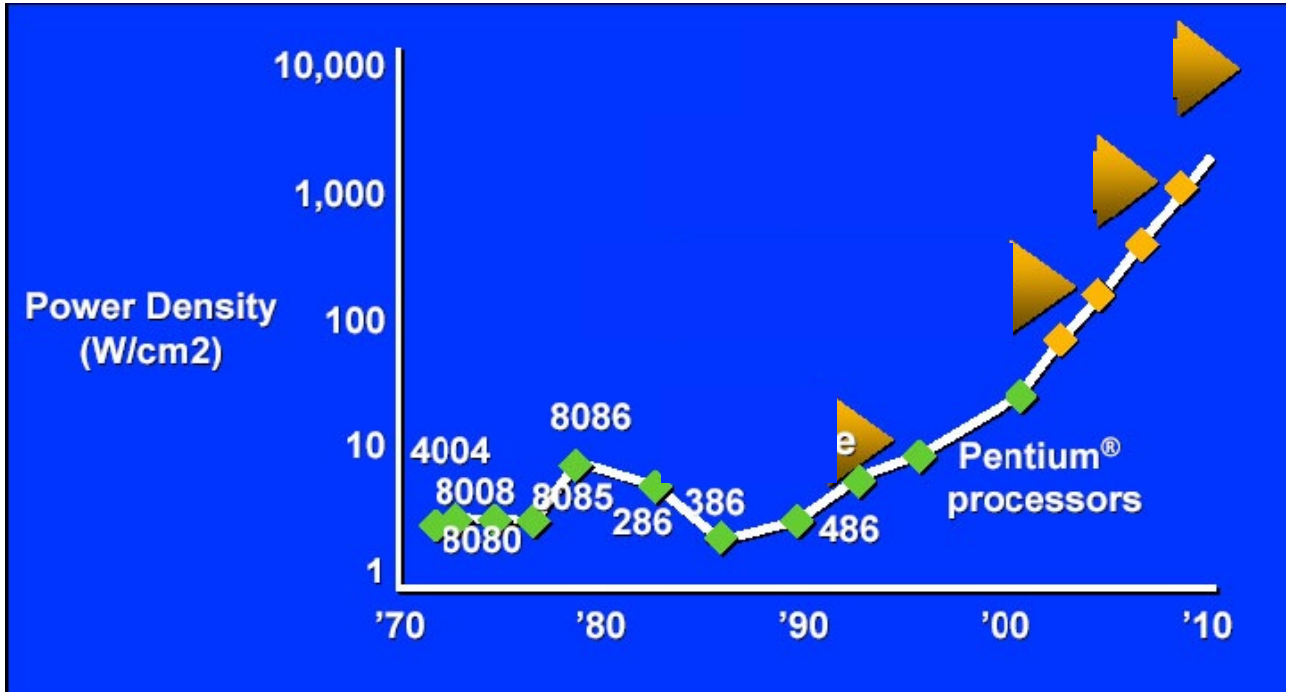
- Initial Focus (starting in 1970's): "Supercomputers" for Scientific Computing
- Another Driving Application (starting in early 90's): Databases

Sun Enterprise 10000 (circa 1997)
64 UltraSPARC-II processors

Oracle SuperCluster M6-32 (today)
32 SPARC M2 processors

A Brief History of Parallel Processing

- Initial Focus (starting in 1970's): "Supercomputers" for Scientific Computing
- Another Driving Application (starting in early 90's): Databases
- Inflection point in 2004: Intel hits the Power Density Wall



Pat Gelsinger, ISSCC 2001

From the New York Times

Intel's Big Shift After Hitting Technical Wall

The warning came first from a group of hobbyists that tests the speeds of computer chips. This year, the group discovered that the Intel Corporation's newest microprocessor was running slower and hotter than its predecessor.

What they had stumbled upon was a major threat to Intel's longstanding approach to dominating the semiconductor industry - relentlessly raising the clock speed of its chips.

Then two weeks ago, Intel, the world's largest chip maker, publicly acknowledged that it had hit a "thermal wall" on its microprocessor line. As a result, the company is changing its product strategy and disbanding one of its most advanced design groups. Intel also said that it would abandon two advanced chip development projects, code-named Tejas and Jayhawk.

Now, Intel is embarked on a course already adopted by some of its major rivals: obtaining more computing power by stamping multiple processors on a single chip rather than straining to increase the speed of a single processor.

…

John Markoff, New York Times, May 17, 2004

Impact of the Power Density Wall

- The real "Moore's Law" continues
  - i.e. # of transistors per chip continues to increase exponentially
- But thermal limitations prevent us from scaling up clock rates
  - otherwise the chips would start to melt, given practical heat sink technology

- How can we deliver more performance to individual applications?
  → increasing numbers of cores per chip

- Caveat:
  - in order for a given application to run faster, it must exploit parallelism

Parallel Machines Today

Examples from Apple's product line:

Mac Pro
12 Intel Xeon E5 cores

iMac
4 Intel Core i5 cores

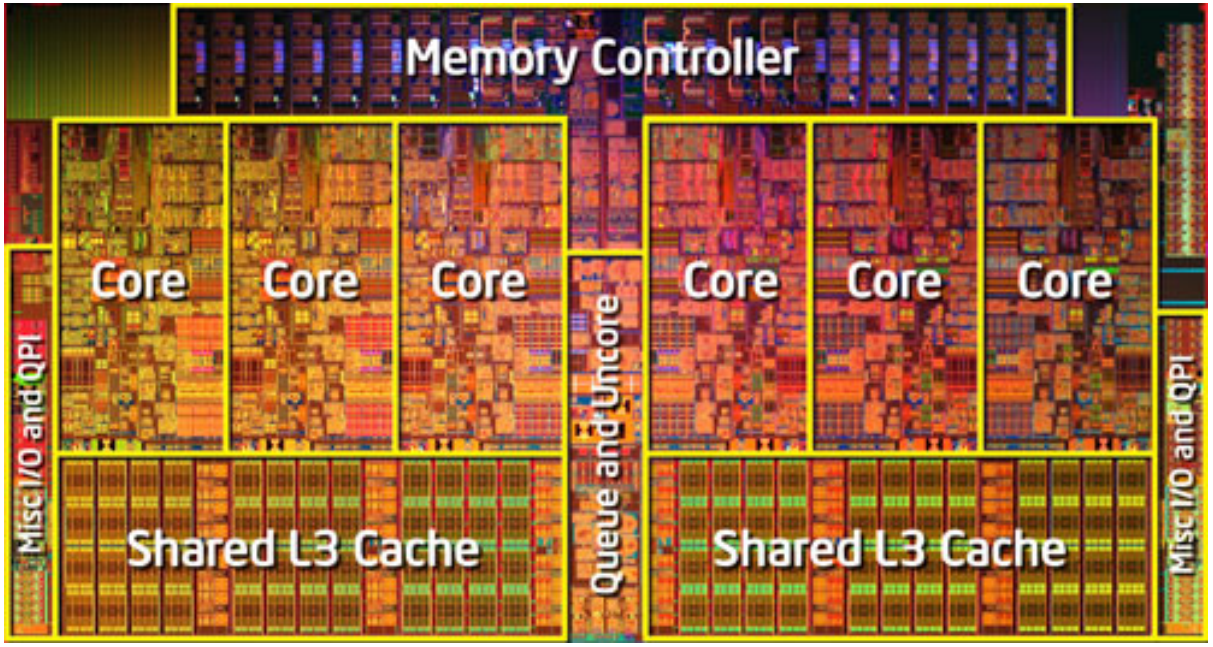
MacBook Pro Retina 15"
4 Intel Core i7 cores

iPad Retina
2 A6X cores
(+ 4 graphics cores)

iPhone 5s
2 A7 cores

(Images from apple.com)

Example "Multicore" Processor: Intel Core i7

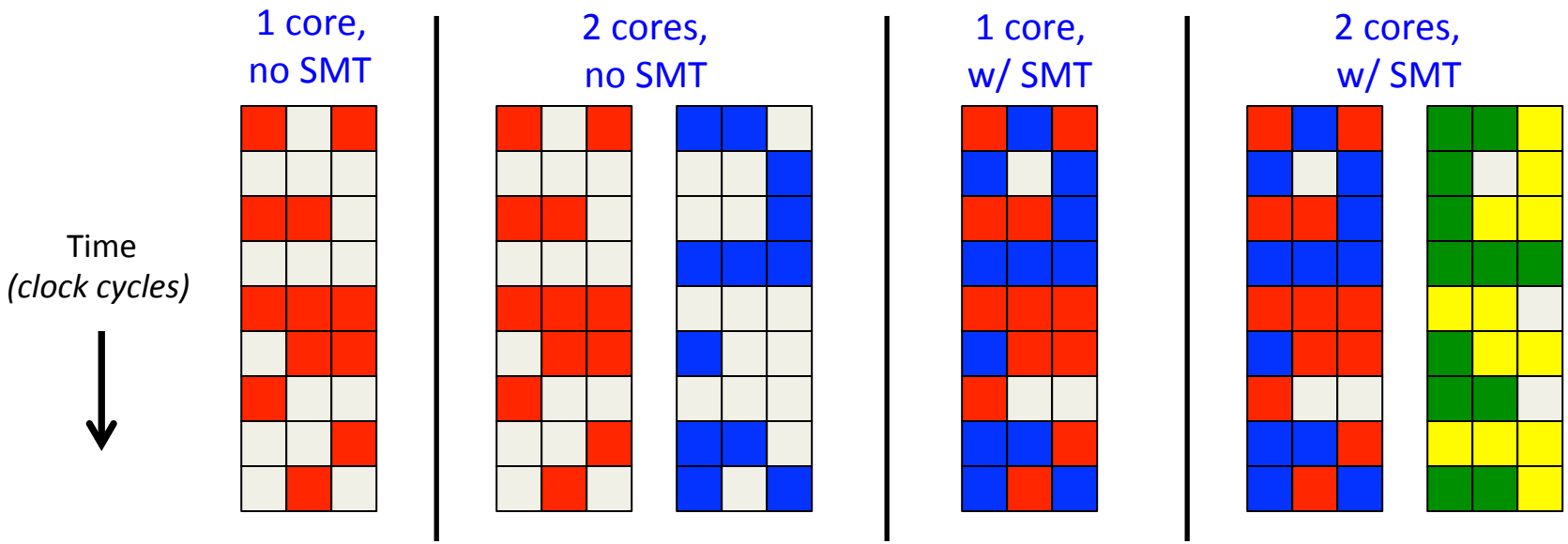


Intel Core i7-980X (6 cores)

- Cores: six 3.33 GHz Nahelem processors (with 2-way "Hyper-Threading")
- Caches: 64KB L1 (private), 256KB L2 (private), 12MB L3 (shared)

Even More Parallelism: Multiple Physical Threads per Core

- What this feature is called:
  - "Simultaneous Multithreading" (SMT) in original research papers
  - "Hyper-Threading" in Intel products
- Basic idea:
  - multiple (e.g., 2) hardware threads can execute *simultaneously* on same core
  - BIOS can enable/disable this feature



(Each box is a functional unit.)

Counting Hardware Threads: Some Terminology

In the Old Days (pre 2004) :

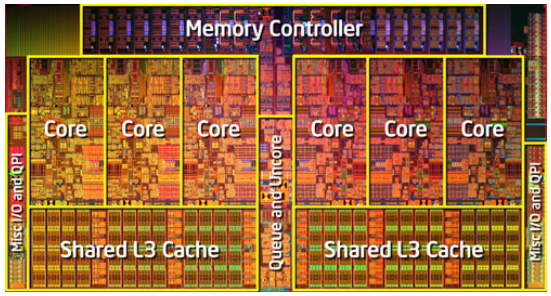
- 1 processor (aka "CPU", "core") per chip
  - each chip lives in a "socket"

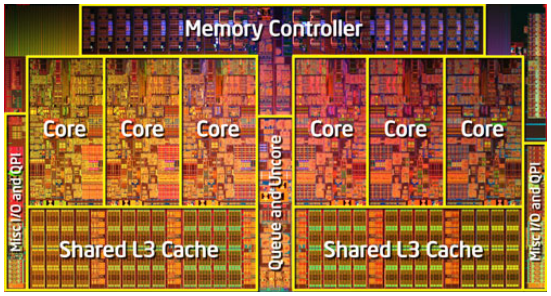
Today:

- X processor chips (each in a socket)
- Y "cores" per processor chip
- Z hardware threads per core (>1 if Hyper-Threading enabled)
- total OS-visible hardware threads = X x Y x Z

Example: dual-socket 6-core Intel Core i7's with 2-way Hyper-Threading

→ 2 x 6 x 2 = 24 hardware threads





What's Fundamentally New for the Programmer?

- Programs now contain *multiple threads* that must
  (i) *communicate* (i.e. share data) and
  (ii) *coordinate* (i.e. synchronize)
  with each other *efficiently* for the sake of improving overall performance.

- While these concepts may seem familiar, the tradeoffs are a bit different when:
  – there is physical parallelism, and
  – performance is paramount.

Parallel Programming Models

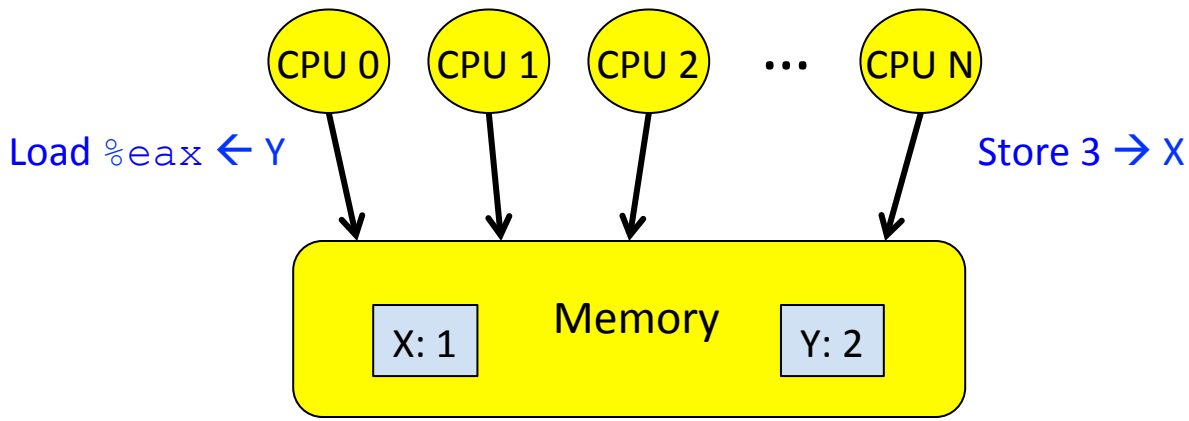
- What is the programmer's abstraction for how inter-thread communication and coordination takes place?

Three major parallel programming models today:

- Shared Address Space (aka "Shared Memory")
- Message Passing
- Data Parallel

Shared Address Space Model

- Analogy: like cooperating by reading and writing on a shared chalk board
- Mechanism:
  - threads share a common virtual address space in memory
  - can communicate through normal memory reads and writes
- Cartoon view:



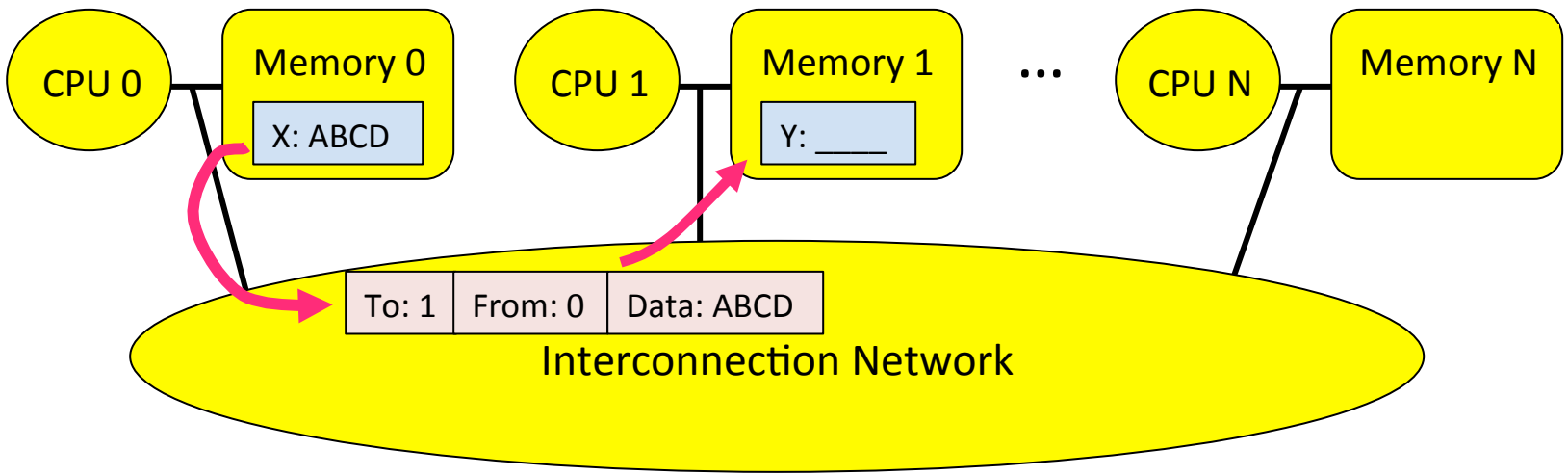
- Examples: OpenMP (compiler pragmas; `gcc -fopenmp`), Intel's TBB (C++ library)

Message Passing Model

- Analogy: like communicating by sending email
- Mechanism:
  - send() and receive() primitives support inter-thread messaging
  - each thread's virtual address space is private to that thread
- Cartoon view:

send(target_thread,&X,length,…);

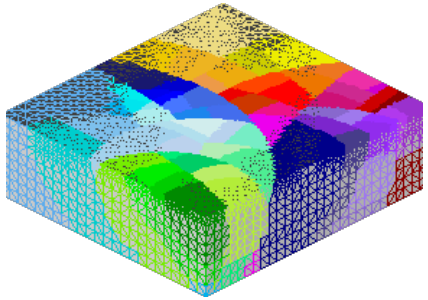
receive(&Y,&from, …);

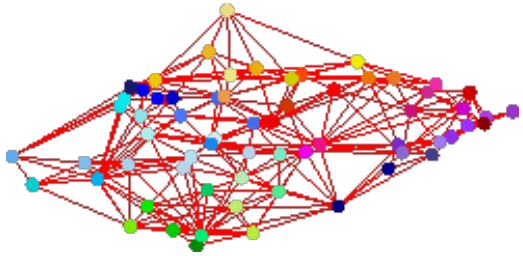


- Example: Message Passing Interface (MPI), www.open-mpi.org

Data Parallel Model

- Target applications:
  - have non-trivially large data sets (matrices, graphs, etc.)
  - visit each element, performing similar computations on the local data
  - results from one phase can be used in the next phase
- Mechanism:
  - programmer specifies local computation and partitioning strategy
  - system executes these in parallel across the data

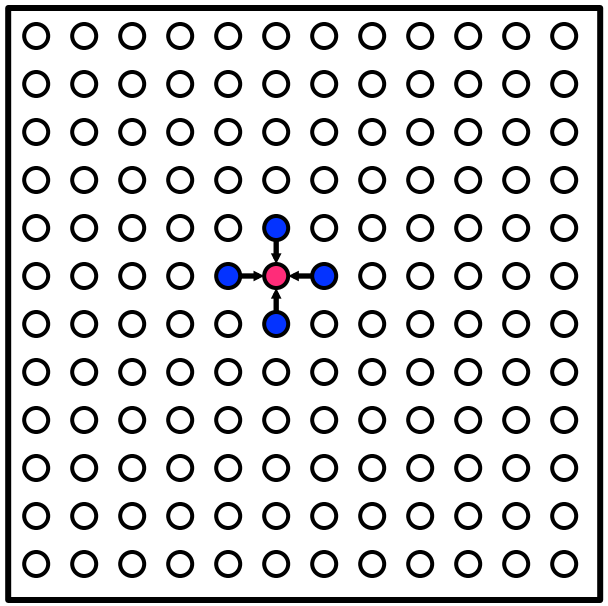




Images: CMU Quake Project

- Examples: MapReduce, OpenCL, CUDA

Example Program: Grid Solver



Updating interior points (per time step):

```
A[i,j] = 0.2 x (A[i,j]+A[i,j-1]+A[i-1,j]+
                A[i,j+1]+A[i+1,j])
```

- Simplified version of a physical simulation (using Guass-Seidel method)
- On each time step:
  - calculate new value for interior points, as shown above
  - add up differences between new and old values over all points
  - if overall difference < threshold, terminate (otherwise, on to next time step)

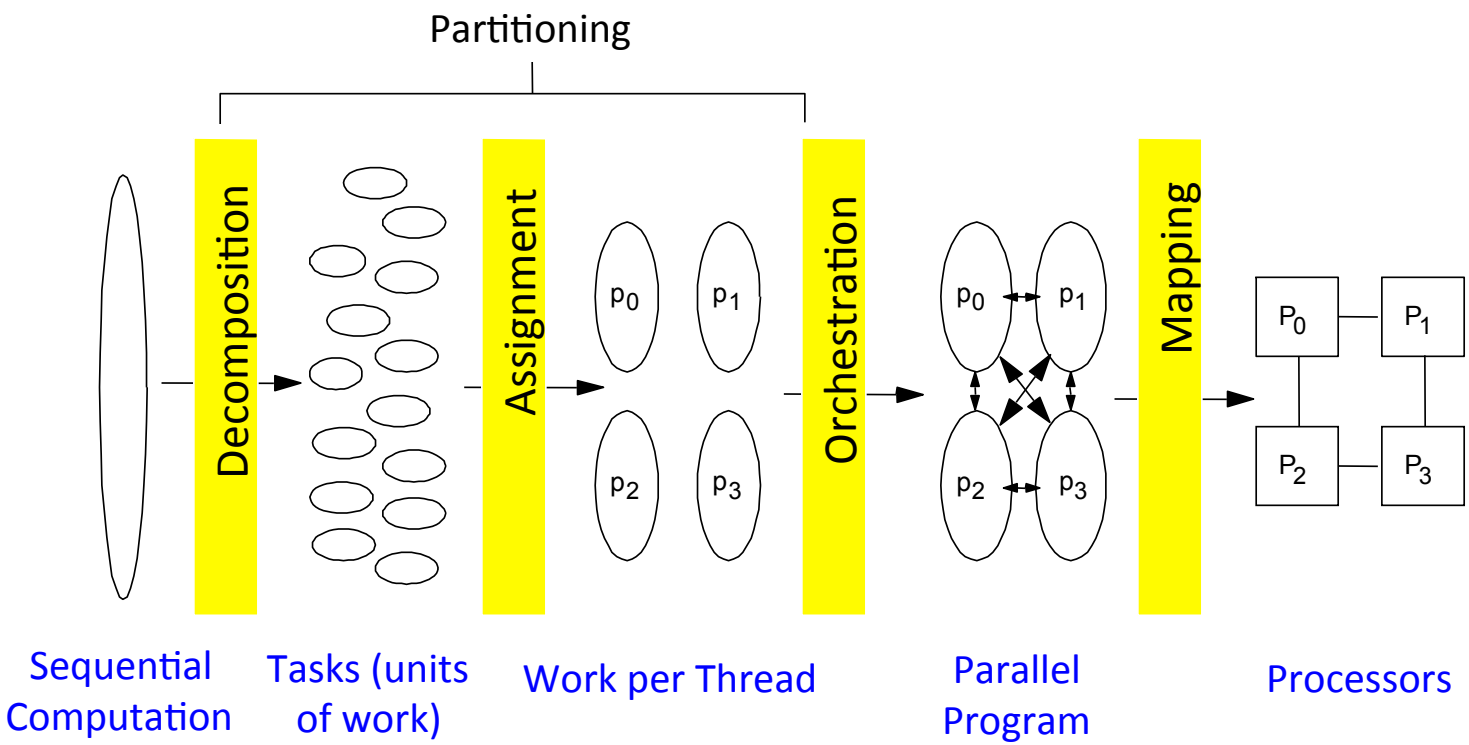
Sequential Version of Grid Solver

```
int n;                                  // size of matrix: (n + 2-by-n + 2) elements
float **A, diff = 0;

main()
begin
   read(n);                             // read input parameter: matrix size
   A ← malloc (a 2-d array of size n + 2 by n + 2 doubles);
   initialize(A);                       // initialize the matrix A somehow
   Solve(A);                            // call the routine to solve equation
end main

procedure Solve(A)                      // solve the equation system
   float **A;                           // A is an (n + 2)-by-(n + 2) array
begin
   int i, j, done = 0;
   float diff = 0, temp;
   while (!done) do                     // outermost loop over sweeps
      diff = 0;                         // initialize maximum difference to 0
      for i ← 1 to n do                 // sweep over interior points of grid
         for j ← 1 to n do
            temp = A[i,j];              // save old value of element
            A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
                                 A[i,j+1] + A[i+1,j]);  // compute average
            diff += abs(A[i,j] - temp);
         end for
      end for
      if (diff/(n*n) < TOL) then done = 1;
   end while
end procedure
```

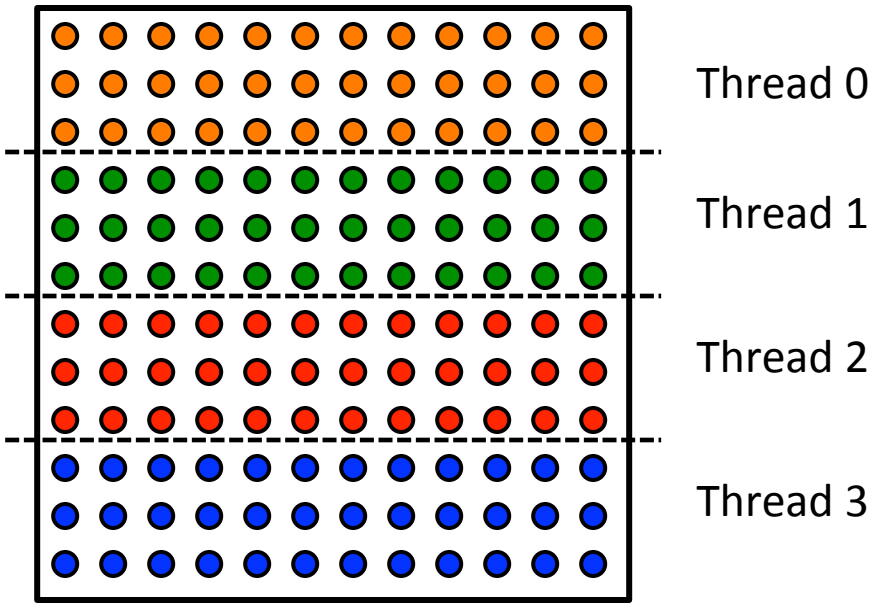
Steps in Creating a Parallel Program



- Partitioning (Decomposition + Assignment):
  - identify units of work, and group together per thread (balancing workload)
- Orchestration: add communication and synchronization primitives
- Mapping (optional): assign threads to physical processors

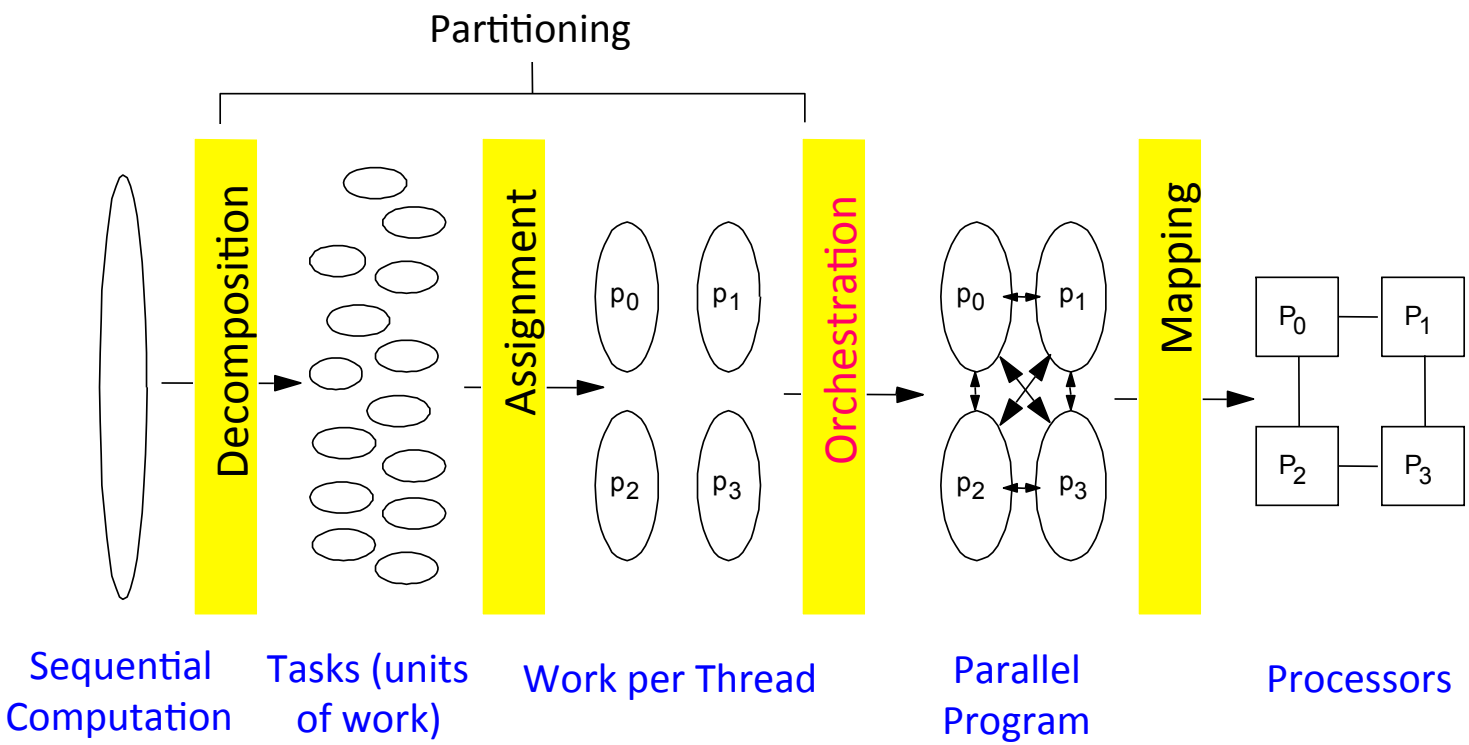
Partitioning the Grid Solver

- One approach:
  - assign contiguous sets of rows to threads



- Helps minimize communication (assuming row-major memory layout)
  - Why?

Next Step in Creating a Parallel Program: Orchestration



- Partitioning steps: independent of parallel programming model
- Orchestration: very much dependent on parallel programming model

Data Parallel Version of Grid Solver

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

main()
begin
   read(n); read(nprocs);               // read input grid size and number of threads
   A ← G_MALLOC(a 2-d array of size n + 2 by n + 2 doubles);
   initialize(A);                       // initialize the matrix A somehow
   Solve(A);                            // call the routine to solve equation
end main

procedure Solve(A)                      // solve the equation system
   float **A;                           // A is an (n + 2)-by-(n + 2) array
begin
   int i, j, done = 0;
   float mydiff = 0, temp;
   DECOMP A[BLOCK, *, nprocs];
   while (!done) do                     // outermost loop over sweeps
      mydiff = 0;                       // initialize maximum difference to 0
      for_all i ← 1 to n do             // sweep over interior points of grid
         for_all j ← 1 to n do
            temp = A[i,j];              // save old value of element
            A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
                                 A[i,j+1] + A[i+1,j]);  // compute average
            mydiff += abs(A[i,j] - temp);
         end for_all
      end for_all
      REDUCE(mydiff, diff, ADD);
      if (diff/(n*n) < TOL) then done = 1;
   end while
end procedure
```

System software does much of the heavy lifting for the programmer

Shared Address Space Version of Grid Solver (Setup)

```
1.  int n, nprocs;            // grid size (n + 2-by-n + 2) and number of threads
2a. float **A, diff;          // is global (shared) array representing the grid
                              // diff is global(shared) max diff in current sweep
2b. LOCKDEC(diff_lock);       // declaration of lock to enforce mutual exclusion
2c. BARDEC (bar1);            // barrier declaration for global sync between sweeps

3. main()
4. begin
5.    read(n); read(nprocs);            // read input grid size and number of threads
6.    A ← G_MALLOC(a 2-d array of size n + 2 by n + 2 doubles);
7.   initialize(A);                     // initialize the matrix A somehow
8a.   CREATE(nprocs-1, Solve, A);
8.    Solve(A);                         // call the routine to solve equation
8b.   WAIT_FOR_END(nprocs-1);           // wait for all child threads to terminate
9. end main

10. procedure Solve(A)                  // solve the equation system
    …
27. end procedure
```

Shared Address Space Version of Grid Solver (Solver)

```
    …
2b. LOCKDEC(diff_lock);          // declaration of lock to enforce mutual exclusion
2c. BARDEC (bar1);               // barrier declaration for global sync between sweeps
    …
```

```
10. procedure Solve(A)                          // solve the equation system
11.    float **A;                               // A is an entire (n + 2)-by-(n + 2) array
12. begin
13.    int i, j, pid, done = 0;                 // assume pid is set to my thread ID somehow
14.    float mydiff = 0, temp;                  // private variables
14a.   int mymin = 1 + (pid * n/nprocs);        // for simplicity, assume nprocs evenly divides n
14b.   int mymax = mymin + n/nprocs - 1;
15.    while (!done) do                         // outermost loop over sweeps
16.       mydiff = diff = 0;                    // set global diff to 0
16a.      BARRIER(bar1, nprocs);                // ensure all reach here before anyone modifies diff
17.       for i ← my_min to my_max do           // for each of my rows
18.          for j ← 1 to n do
19.             temp = A[i,j];                  // save old value of element
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]);  // compute average
22.             mydiff += abs(A[i,j] - temp);
23.          end for
24.       end for
25a.      LOCK(diff_lock);                      // update global diff
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;
25f.      BARRIER(bar1, nprocs);
26.    end while
27. end procedure
```

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, but orchestration different
  - data structures and data access/naming
  - communication
  - synchronization

Message Passing Version of Grid Solver (Setup)

```
1. int pid, n, nprocs;          // thread ID, matrix size, num of threads to use
2. float **myA;                 // private portion of array

3. main()
4. begin
5. read(n); read(nprocs);                  // read input grid size and number of threads
8a.    CREATE(nprocs-1, Solve);
8.     Solve(A);                           // call the routine to solve equation
8b.    WAIT_FOR_END(nprocs-1);             // wait for all child threads to terminate
9. end main
```

```
10. procedure Solve(A)                     // solve the equation system
    …
27. end procedure
```

Message Passing Version of Grid Solver (Solver – Part 1)

```
10. procedure Solve(A)
11.    float **A;
12. begin
13.    int i, j, pid, n' = n/nprocs, done = 0;
14.    float mydiff = 0, temp, tempdiff;    // private variables
6.     myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2 doubles);  // my rows of A
7.     initialize(myA);                     // initialize my rows of A somehow

15.    while (!done) do                     // outermost loop over sweeps
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                // for each of my (non-ghost) rows
18.          for j ← 1 to n do              // for all non-border elements in that row
19.             temp = myA[i,j];
20.             myA[i,j] = 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
21.                                          myA[i,j+1] + myA[i+1,j]);
22.             mydiff += abs(myA[i,j] - temp);
23.          endfor
24.       endfor

                    // communicate local diff values and determine if done

26.    end while
27. end procedure
```

Message Passing Version of Grid Solver (Solver – Part 2)

```
10. procedure Solve(A)
11.    float **A;
12. begin
13.    int i, j, pid, n' = n/nprocs, done = 0;
14.    float mydiff = 0, temp, tempdiff;    // private variables
6.     myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2 doubles);  // my rows of A
7.     initialize(myA);                     // initialize my rows of A somehow

15.    while (!done) do                     // outermost loop over sweeps
```

```
// communicate border rows, calculate updated values for my data
```

```
                                  // communicate local diff values and determine if done
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.    end while
27. end procedure
```

Communication and Synchronization

Shared Address Space Model:

- Communication:
  - performed *implicitly* via memory loads and stores
- Synchronization: performed *explicitly* by programmer
  - mutual exclusion: locks
  - event ordering: barriers

Message Passing Model:

- Communication:
  - performed *explicitly* through sends and receives
- Synchronization: performed *implicitly*
  - mutual exclusion: nothing needed, since virtual address space is private
  - event ordering:
    - produce-consumer: implicit in receiving a message
    - phase ordering: via coordinated message passing (see example)

Achieving High Performance: Keys for Programmers

- Spend as little time in serial execution as possible
  - Amdahl's Law:
    - if S = fraction of time executing serially, then upper bound on speedup is 1/S
  - unlike time-slicing threads on a single CPU, where serial execution is the norm
- Evenly balance the work across all of the threads
  - overall execution time dictated by when the last straggler finishes
  - adjust dynamically? tradeoffs versus static allocation?
- Minimize communication between threads
  - otherwise, communication bandwidth can become a bottleneck
  - (but isn't this at odds with evenly balancing the work?)
- Minimize time spent stalled waiting for synchronization
  - finer-grained locking, shorter critical sections, less frequent barriers, etc.
- Minimize extra work needed to manage the parallelism
  - e.g., clever run-time scheduling algorithms might not pay off
- Take 15-418!

Delivering High Performance: Keys for System Design

- Replication of shared data is critical to avoiding bottlenecks
  - keeping replicas coherent (given updates) is a challenge
- Need fast communication mechanisms
  - low latency, high bandwidth
  - makes it easier to overcome other parallelism challenges

Summary

- Why parallelism appears to be here to stay
  - power density wall
- What parallel programs look like
  - Shared Address Space, Message Passing, Data Parallel programming models
- Keys to achieving high performance with parallelism

Looking ahead:

- What OS developers need to know about parallel memory systems
  - VM protection, cache coherence, memory consistency models, etc.