CSEP 524: Parallel Computation
(week 6)

Brad Chamberlain
Tuesdays 6:30 – 9:20
MGH 231
# Adding OpenMP to Our Categorization (part 1)

<table>
<thead>
<tr>
<th></th>
<th>C+Pthreads</th>
<th>Chapel</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>degree of voodoo</td>
<td>less voodoo</td>
<td>more voodoo</td>
<td>moderate-to-more voodoo</td>
</tr>
<tr>
<td>level of abstraction</td>
<td>more HW-oriented</td>
<td>more problem-oriented</td>
<td>in the middle</td>
</tr>
<tr>
<td>verbosity</td>
<td>more verbose</td>
<td>less verbose</td>
<td>in between</td>
</tr>
<tr>
<td>control of memory</td>
<td>more control due to C</td>
<td>less control (today)</td>
<td>same as C+Pthreads</td>
</tr>
<tr>
<td>(alignment/padding)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HW independenc e</td>
<td>less abstracted from HW</td>
<td>more abstracted...</td>
<td>more abstracted...</td>
</tr>
<tr>
<td>portability</td>
<td>quite good</td>
<td>potentially more portable</td>
<td>as portable as C, Fortran, C++</td>
</tr>
</tbody>
</table>
## Adding OpenMP to Our Categorization (part 2)

<table>
<thead>
<tr>
<th></th>
<th>C+Pthreads</th>
<th>Chapel</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>libraries</td>
<td>lots of existing library support</td>
<td>very little currently*</td>
<td>can call sequential C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* = extern support for C</td>
<td></td>
</tr>
<tr>
<td>opportunities</td>
<td>more opportunities due to C and details of sync</td>
<td>less so</td>
<td>fragility w.r.t. mistyped pragma prefixes (use – Wall); ability to</td>
</tr>
<tr>
<td>for error</td>
<td>primitives</td>
<td></td>
<td>break seq case (reduce/SPMD)</td>
</tr>
<tr>
<td>notation</td>
<td>library</td>
<td>language</td>
<td>pragnas</td>
</tr>
<tr>
<td>maturity</td>
<td>very mature</td>
<td>much less so</td>
<td>mature, but evolving</td>
</tr>
<tr>
<td>“classic” concepts</td>
<td>the set of classic concepts</td>
<td>pretty significant departure</td>
<td>lower-level (locks), and higher (critical sections, barriers,</td>
</tr>
<tr>
<td>(mutex, condvar, ...)</td>
<td></td>
<td></td>
<td>reductions, data parallelism)</td>
</tr>
<tr>
<td>completeness</td>
<td>confidence that it’s complete</td>
<td>unclear</td>
<td>reasonably complete (no must parallelism)</td>
</tr>
</tbody>
</table>

---

*CSEP 524: Parallel Computation  Winter 2013: Chamberlain*
# Categorizing Based on Features/Capabilities

<table>
<thead>
<tr>
<th></th>
<th>C+Pthreads</th>
<th>Chapel</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>data parallelism</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><em>may</em> tasks</td>
<td>yes? (no implicit support)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td><em>must</em> tasks</td>
<td>yes</td>
<td>yes</td>
<td>not well</td>
</tr>
<tr>
<td>barriers</td>
<td>no</td>
<td>no (not yet)</td>
<td>yes</td>
</tr>
<tr>
<td>reductions</td>
<td>no</td>
<td>built-in + user-defined</td>
<td>built-in</td>
</tr>
<tr>
<td>scans</td>
<td>no</td>
<td>built-in + user-defined</td>
<td>no?</td>
</tr>
<tr>
<td>locks</td>
<td>yes</td>
<td>sync vars</td>
<td>yes (library)</td>
</tr>
<tr>
<td>incremental parallelism</td>
<td>so-so</td>
<td>so-so –to- yes</td>
<td>yes</td>
</tr>
<tr>
<td>scalability to dist. mem/locality</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>
Shared Memory Summary

*shared memory:* A system in which memory can be accessed via simple load/store instructions
  – *example:* your multicore laptop/desktop
  – typically corresponds to executing a single OS image

*shared memory programming models:*
  – parallelism/tasks typically implemented via system threads
    • or user threads running on top of system threads
  – any task can access any variable
e.g., OpenMP, Pthreads

+ support dynamic, fine-grain parallelism
+ considered simpler, more like traditional programming
  • “if you want to access something, simply name it”
– no support for expressing locality/affinity; limits scalability
– bugs can be subtle, difficult to track down (race conditions)
– tend to require complex memory consistency models
Large-Scale Shared Memory?

Q: We’ve focused on desktop-/latop-scale systems, but could these same principles and programming models be used with large-scale machines?

A: Yes and no (depends on your definition of large)
   – shared- vs. distributed-memory was a major topic of debate in parallel computing during the 1980’s-1990’s
     • which is easier to build?
     • which is easier to program?
ccNUMA Architectures

**ccNUMA:** Cache Coherent Non-Uniform Memory Access

– or, simply NUMA for short
  - (non-cache coherent is too confusing to be very useful)
– essentially, shared memory in which...
  ...all memory is capable of being accessed via loads/stores
  ...but not at uniform cost
ccNUMA: Cache Coherent Non-Uniform Memory Access

– or, simply NUMA for short
  • (non-cache coherent is too confusing to be very useful)
– essentially, shared memory in which...
  ...
  ...all memory is capable of being accessed via loads/stores
  ...
  ...but not at uniform cost
ccNUMA: Cache Coherent Non-Uniform Memory Access

– or, simply NUMA for short
  • (non-cache coherent is too confusing to be very useful)
– essentially, shared memory in which...
  ...all memory is capable of being accessed via loads/stores
  ...but not at uniform cost
ccNUMA Architectures

ccNUMA: Cache Coherent Non-Uniform Memory Access
– or, simply NUMA for short
  • (non-cache coherent is too confusing to be very useful)
– essentially, shared memory in which...
  ...all memory is capable of being accessed via loads/stores
  ...but not at uniform cost

```
node
    memory
    (0x0000-0x7fff)
    socket
          core
          core
          core
          core
          L1
          L2
          L3
load
0x3240

node
    memory
    (0x8000-0xffff)
    socket
          core
          core
          core
          core
          L1
          L2
          L3
network
```
**ccNUMA Architectures**

**ccNUMA**: Cache Coherent Non-Uniform Memory Access

– or, simply NUMA for short
  - (non-cache coherent is too confusing to be very useful)
– essentially, shared memory in which...
  ...all memory is capable of being accessed via loads/stores
  ...but not at uniform cost
ccNUMA Architectures

ccNUMA: Cache Coherent Non-Uniform Memory Access

– or, simply NUMA for short
  • (non-cache coherent is too confusing to be very useful)
– essentially, shared memory in which...
  ...all memory is capable of being accessed via loads/stores
  ...but not at uniform cost
ccNUMA: Cache Coherent Non-Uniform Memory Access

– or, simply NUMA for short
  • (non-cache coherent is too confusing to be very useful)
– essentially, shared memory in which...
  ...all memory is capable of being accessed via loads/stores
  ...but not at uniform cost
ccNUMA: Scalability

- For small numbers of processors, this is manageable
- As the number grows, however...
  - the fraction of network traffic required to keep the caches coherent can become quite large
  - opportunities for traditional shared memory concerns like false sharing and race conditions can grow
    - for these reasons, users often program large-scale ccNUMA machines using distributed memory techniques anyway...
ccNUMA: How big?

• SGI UV
  – 256 Intel Xeon sockets x 8 cores
    == 2,048 cores
    • “only solution that uses Intel Xeon beyond 4 sockets”
  – 64 TB memory
  – Source of images/
    for more information:
How Big? ccNUMA vs. distributed memory

**SGI UV** (ccNUMA)
- 2,048 cores
- 64 TB memory

**Cray Titan** (dist. memory)
- 299,008 cores (+ 18,688 GPUs)
- 710 TB memory

~146x

~11x
How Big? ccNUMA vs. distributed memory

**SGI UV (ccNUMA)**
- 2,048 cores
- 64 TB memory

**IBM Sequoia (dist. memory)**
- 1,572,864 cores
- 1.6 PB memory

Source: [https://computing.llnl.gov/tutorials/bgq/](https://computing.llnl.gov/tutorials/bgq/)
Distributed Memory
Distributed Memory Summary

**distributed memory:** A system with multiple distinct memory segments that are not trivially accessible from one another

– **examples:** commodity clusters; workstations on a network; large Cray, IBM, HP, etc. systems

– typically a distinct OS image per memory segment
Distributed Memory Programming Models

**distributed memory programming models:**

– parallelism typically implemented via processes
  
  • typically much more static than what we’ve been studying

– processes can only access their own local memory directly
  
  • must use communication to coordinate with other processes
Distributed Memory Architectures

Distributed Memory Architectures:

– A number of compute nodes
  • Historically, many custom processor designs have been used
  • Today, virtually indistinguishable from your laptop/desktop

– Connected by a network
  • Network topologies and technologies vary greatly
  • What might they look like?
Bus-based Networks

- As with a memory bus, one node communicates at a time
  - **Example**: ethernet

+ Easy(-ish) to implement

- A bottleneck for communication-intensive apps
Ring-based Networks

- As with a memory bus, one node communicates at a time
  - Example: KSR (1990’s)
- Still Easy(-ish) to implement
- Supports multiple communications at once, unlike bus
  - $O(\text{numNodes})$ hops in worst-case route
Crossbar-based Networks

- Links between every pair of nodes
  + Contention-free O(1) communication
    - not a scalable design
      - (e.g., Titan would require 349,222,656 links)
Hypercube Networks

• Links between every pair of nodes with a 1-bit difference in ID
  – e.g., SGI Origin

+ Fixed number of steps to reach any node ($\log_2 \text{numNodes}$)

Hypercube Networks

• Links between every pair of nodes with a 1-bit difference in ID
  – e.g., SGI Origin
+ Fixed number of steps to reach any node ($\log_2 numNodes$)
– not scalable from network interface chip (NIC) perspective
  – maximum size of machine determined by # of output channels
    • contrast with bus-based network with 1 channel per NIC
  – smaller machines waste unused channels and HW area on NIC

Image source: http://www.ece.eng.wayne.edu/~czxu/ece7660_f05/network.pdf
Butterfly Networks

• Shuffle at each stage of network based on bits of node ID
  – e.g., Butterfly BBN
+ Fixed number of steps to reach any node \( (\log_2 \text{numNodes}) \)
  – requires \( N/2 \times \log N \) router nodes

Image source: [http://www.ece.eng.wayne.edu/~czxu/ece7660_f05/network.pdf](http://www.ece.eng.wayne.edu/~czxu/ece7660_f05/network.pdf)
Butterfly Networks

- Can also build using higher-radix building blocks

Fat-TREE Networks

• Tree w/ multiple roots, multiple parents per node
  – processors are at leaves; internal nodes are routers only
  – Why a “fat” tree? To reduce contention higher in the tree.
  – e.g., Connection Machine
Fat-Tree (Top-view)

Image source: [http://24-7-solutions.net/reviews/cluster-arch.html](http://24-7-solutions.net/reviews/cluster-arch.html)
Mesh-Based Networks

- Chips connected to nearest neighbors
  - Modest/Scalable chip design: \#channels = \#mesh neighbors
  - Some communications require more hops than others
Mesh-Based Networks

• Chips connected to nearest neighbors
  + Modest/Scalable chip design: \#channels = \#mesh neighbors
  - Some communications require more hops than others
    - variable time for a message to cross from source to destination
    - increased chances of collisions with other messages
      • (compared to crossbar, hypercube, butterfly)
Mesh-Based Networks w/ Toroidal Wraparound

- Similar to mesh
  - Major advantage: doesn’t make traffic as dependent on placement in the mesh
IBM BG/Q Network: a 5D Toroidal Mesh

compare to hypercube:

# hops:
- BG/Q: $8+6+8+8+1 = 31$ hops
- Hypercube: $\log_2 98,304 = 17$ hops

# channels:
- BG/Q: $5 \times 2 = 10$ channels
  - (and scales to larger sizes)
- Hypercube: 17 channels

Image source: [https://computing.llnl.gov/tutorials/bgg/](https://computing.llnl.gov/tutorials/bgg/)
Dragonfly Networks

- Developed jointly by Stanford and Cray
  - Network topology for Cray XC30
    - Cray’s current flagship architecture
  - Developed under DARPA HPCS
    - same program as Chapel
  - Name intended to be evocative of next-generation butterfly
- The topic of this week’s reading
Network Design: A Rich Field of Study

(but largely outside the scope of this course)

• Areas of concern:
  – topology
  – choice of route
  – determinism / message ordering
  – congestion avoidance
  – fault tolerance
    • to network failure ("a board and its links just went down!")
    • to data loss ("sorry, that message never arrived")
Network Metrics

Latency:

Bandwidth:
Network Metrics

*Latency*: How long it takes a message to reach its destination

*Bandwidth*: 
Network Metrics

**Latency:** How long it takes a message to reach its destination

As programmers, we have techniques available to *tolerate* latency
- i.e., don’t just sit around waiting
- do some other computation in this task
- switch to another task
- ...

**Bandwidth:** How much data/how many messages the network can handle simultaneously

By contrast, there’s not much that can be done to deal with bandwidth limitations
- “Don’t communicate as much data” is presumably something we’re already trying to do for latency reasons
Networks in a Nutshell

- Networks should only have a performance impact
  - not correctness
- For the past few generations of HPC machines, whether or not you access the network is far more important than...
  - where you have to go in it
  - the length of your message
    - \( \alpha + \beta \times \text{messageLength} \)
- Instead, cost of accessing the network dominates
  - working through software stack
  - copies/buffering at various levels
Network-Specific Computations

• Sensitivity to network depends a lot on algorithm
  – amount of communication, topology of communication, size of messages, etc.
  – In practice, most programmers don’t code to the network
    • has similar performance/portability tensions as coding to a CPU
    • this has been a significant change since the 80’s...
      – typical paper title then: “multiplying matrices on an xyz network”
HPC and Networks

• In HPC...
  – computations tend to be reasonably network-intensive
  – bandwidth tends to be the most precious/expensive commodity

• So why do we place so much value in the top500?
  – recall: a peak FLOPs/CPU-bound benchmark
  – alternatives have been proposed:
    • HPC Challenge
    • Graph 500
    • ...

...but so far, none have caught on as much (yet)
A Slight Aside About Execution Models
SIMD vs. MIMD

**SIMD:**

- Single Instruction, Multiple Data
- One instruction drives a bunch of similar operations
- A lightly-coupled style of execution
- e.g., vector processors or GPUs
  - e.g., “add these 1000 numbers to those 1000 numbers”

**MIMD:**

CSEP 524: Parallel Computation

Winter 2013: Chamberlain
**SIMD vs. MIMD**

**SIMD**: Single Instruction, Multiple Data
- one instruction/PC drives a bunch of similar operations
- a tightly-coupled style of execution
- e.g., vector processors or GPUs
- e.g., “add these 1000 numbers to those 1000 numbers”

**MIMD:**


**SIMD** vs. **MIMD**

**SIMD:** Single Instruction, Multiple Data
- one instruction/PC drives a bunch of similar operations
- a tightly-coupled style of execution
- e.g., vector processors or GPUs
- e.g., “add these 1000 numbers to those 1000 numbers”

**MIMD:** Multiple Instruction, Multiple Data
- distinct instructions/PCs drive (potentially) distinct operations
- more loosely-coupled, general
- e.g., most distributed memory programming
# Flynn’s Taxonomy

<table>
<thead>
<tr>
<th>Data</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>SISD</td>
</tr>
<tr>
<td>Multiple</td>
<td>SIMD</td>
</tr>
</tbody>
</table>

Essentially, sequential programming by a fancy name

Execute the same thing redundantly (for resilience?)
Distributed Memory Programming
SPMD Programming/Execution Models

**SPMD:** Single Program, Multiple Data

– not an actual member of Flynn’s taxonomy
– the dominant model for distributed memory programming
– Concept:
  • write one copy of a program
  • execute multiple copies of it simultaneously
    – various terms: *images, processes, PEs (Processing Elements), ranks*, ...
    – one per compute node? one per core?
  • in a pure SPMD model, this is the only source of parallelism
    – i.e., run $p$ copies of my program in parallel
    – our parallel tasks are essentially the program images
  • in practice, each program can also contain parallelism
    – typically achieved by mixing two notations (e.g., MPI + OpenMP)
How Do SPMD Program Images Interact?

• Message Passing (this week):
  – “messages”: essentially buffers of data
  – primitive message passing ops: send/receive
    • also, typically, collective operations (reductions, barriers, bcasts, ...)
  – primary example: MPI
    • (historically, PVM, NX, and a host of others...)

• Other alternatives (topics for future weeks):
  – Single-Sided Communication
  – Partitioned Global Address Spaces
  – Active Messages
  – ...

CSEP 524: Parallel Computation
Winter 2013: Chamberlain
Message Passing: The Curse and the Blessing

• Using message passing...
  – In contrast to shared memory programming, we can no longer simply refer to other tasks’ variables
    • Instead, tasks need to explicitly communicate
  + Happily, this means a bunch of problematic issues go away
    • false sharing
    • RRWW errors
    • race conditions
    • memory consistency models
  – But of course message passing has its own problems
    • Parallel programming still isn’t easy...
In pictures: “Apply a 3-Point Stencil to a vector”

**Global-View**

\[
\left( \begin{array}{cccc}
\text{pink} & \text{pink} & \text{pink} & \text{pink} \\
\text{pink} & \text{pink} & \text{pink} & \text{pink}
\end{array} \right) / 2
\]

\[
+ \left( \begin{array}{cccc}
\text{pink} & \text{pink} & \text{pink} & \text{pink} \\
\text{pink} & \text{pink} & \text{pink} & \text{pink}
\end{array} \right) / 2
\]

\[
= \left( \begin{array}{cccc}
\text{yellow} & \text{yellow} & \text{yellow} & \text{yellow} \\
\text{yellow} & \text{yellow} & \text{yellow} & \text{yellow}
\end{array} \right)
\]

**Local-View**

\[
\left( \begin{array}{cccc}
\text{pink} & \text{pink} & \text{pink} & \text{pink} \\
\text{pink} & \text{pink} & \text{pink} & \text{pink}
\end{array} \right)
\]

\[
+ \left( \begin{array}{cccc}
\text{pink} & \text{pink} & \text{pink} & \text{pink} \\
\text{pink} & \text{pink} & \text{pink} & \text{pink}
\end{array} \right) / 2
\]

\[
= \left( \begin{array}{cccc}
\text{yellow} & \text{yellow} & \text{yellow} & \text{yellow} \\
\text{yellow} & \text{yellow} & \text{yellow} & \text{yellow}
\end{array} \right)
\]

\[
\left( \begin{array}{cccc}
\text{pink} & \text{pink} & \text{pink} & \text{pink} \\
\text{pink} & \text{pink} & \text{pink} & \text{pink}
\end{array} \right) / 2
\]

\[
+ \left( \begin{array}{cccc}
\text{pink} & \text{pink} & \text{pink} & \text{pink} \\
\text{pink} & \text{pink} & \text{pink} & \text{pink}
\end{array} \right) / 2
\]

\[
= \left( \begin{array}{cccc}
\text{yellow} & \text{yellow} & \text{yellow} & \text{yellow} \\
\text{yellow} & \text{yellow} & \text{yellow} & \text{yellow}
\end{array} \right)
\]

\[
\left( \begin{array}{cccc}
\text{pink} & \text{pink} & \text{pink} & \text{pink} \\
\text{pink} & \text{pink} & \text{pink} & \text{pink}
\end{array} \right)
\]

\[
+ \left( \begin{array}{cccc}
\text{pink} & \text{pink} & \text{pink} & \text{pink} \\
\text{pink} & \text{pink} & \text{pink} & \text{pink}
\end{array} \right) / 2
\]

\[
= \left( \begin{array}{cccc}
\text{yellow} & \text{yellow} & \text{yellow} & \text{yellow} \\
\text{yellow} & \text{yellow} & \text{yellow} & \text{yellow}
\end{array} \right)
\]
In code: “Apply a 3-Point Stencil to a vector”

**Global-View**

```plaintext
proc main() {
    var n = 1000;
    var A, B: [1..n] real;
    forall i in 2..n-1 do
        B[i] = (A[i-1] + A[i+1])/2;
}
```

**Local-View (SPMD)**

```plaintext
proc main() {
    var n = 1000;
    var p = numProcs(),
        me = myProc(),
        myN = n/p,
        myLo = 1,
        myHi = myN;
    var A, B: [0..myN+1] real;
    if (me < p-1) {
        send(me+1, A[myN]);
        recv(me+1, A[myN+1]);
    } else
        myHi = myN-1;
    if (me > 0) {
        send(me-1, A[1]);
        recv(me-1, A[0]);
    } else
        myLo = 2;
    forall i in myLo..myHi do
        B[i] = (A[i-1] + A[i+1])/2;
}
```
Problem: “Apply 3-pt stencil to vector”

SPMD (pseudo-Chapel + MPI)

```chapel
var n: int = 1000;
var p, me: int;
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &me);
var locN: int = n/p;
var a, b: [0..locN+1] real;
var innerLo: int = 1, innerHi: int = locN;
var status: MPI_Status;
var retval: int;
if (me < numProcs-1) {
    retval = MPI_Send(&a[locN]), 1, MPI_FLOAT, me+1, 0, MPI_COMM_WORLD);
    if (retval != MPI_SUCCESS) { handleErr(retval); }
    retval = MPI_Recv(&a[locN+1]), 1, MPI_FLOAT, me+1, 1, MPI_COMM_WORLD, &status);
    if (retval != MPI_SUCCESS) { handleErrWithStatus(retval, status); }
} else
    innerHi = locN-1;
if (me > 0) {
    retval = MPI_Send(&a[1]), 1, MPI_FLOAT, me-1, 1, MPI_COMM_WORLD);
    if (retval != MPI_SUCCESS) { handleErr(retval); }
   retval = MPI_Recv(&a[locN+1]), 1, MPI_FLOAT, me-1, 0, MPI_COMM_WORLD, &status);
    if (retval != MPI_SUCCESS) { handleErrWithStatus(retval, status); }
} else
    innerLo = 2;
forall i in (innerLo..innerHi) {
    b(i) = (a(i-1) + a(i+1))/2;
}
```
Introduction to MPI
MPI: Message Passing Interface

– a standard HPC library for communicating between cooperating processes
  • the *de facto* standard for scalable HPC programming

– IMO, more than simply “a library” due to its impact on the user’s programming/execution models
  • i.e., most libraries don’t change the way you run your program, think of main(), etc.
  • this is as much an effect of the SPMD programming model as anything related to MPI
Primary MPI Concepts

**Communicators:** groups of program images (processes)

**Sends/Receives:** primary building block for communication

**Collectives:** routines for working as a group
(switch to Rajeev’s Slides here)
Message Passing Hazards

• Main issues you’re likely to run into:
  – mismatch between sends/receives
    • e.g., send doesn’t have a matching receive or vice-versa
    • e.g., send and receive don’t name right tag, source/destination
  – collectives in which participants are missing
    • e.g., a process never calls into a barrier or reduction
  – issues related to resource constraints/timing
    • e.g., insufficient memory to buffer things
    • (not likely to hit this in this class)

• These tend to manifest themselves like deadlocks
  – or as “out-of-resource” msg, degraded performance,
Prior to computing a stencil, communication is typically required to refresh the ghost cells.

Notes:
- Lots of optimization opportunities
- Have to eventually start skipping processors for coarser levels
This Week’s Homework

• Finish atomic increment + mod if you haven’t
• Translate manual reduction to MPI
• Translate 9-point stencil to MPI
  (in both cases, starting from scratch may be best)