Recap

- Two primary patterns of multicore architecture design
  - **Shared memory**
    - Ex: Intel Core 2 Duo/Quad
    - One copy of data shared among many cores
    - Atomicity, locking and synchronization essential for correctness
    - Many scalability issues
  - **Distributed memory**
    - Ex: Cell
    - Cores primarily access local memory
    - Explicit data exchange between cores
    - Data distribution and communication orchestration is essential for performance
Programming Shared Memory Processors

- Processor 1...n ask for X
- There is only one place to look
- Communication through shared variables
- Race conditions possible
  - Use synchronization to protect from conflicts
  - Change how data is stored to minimize synchronization
Example Parallelization

Data parallel
- Perform same computation but operate on different data

A single process can fork multiple concurrent threads
- Each thread encapsulates its own execution path
- Each thread has local state and shared resources
- Threads communicate through shared resources such as global memory

Java Code:
```java
for (i = 0; i < 12; i++)
    C[i] = A[i] + B[i];
```

Diagram:
- fork (threads)
- join (barrier)
  - i = 0
  - i = 1
  - i = 2
  - i = 3
  - i = 4
  - i = 5
  - i = 6
  - i = 7
  - i = 8
  - i = 9
  - i = 10
  - i = 11
Example Parallelization With Threads

```c

void add_arrays(int start)
{
    int i;
    for (i = start; i < start + 4; i++)
        C[i] = A[i] + B[i];
}

int main (int argc, char *argv[])
{
    pthread_t threads_ids[3];
    int rc, t;
    for(t = 0; t < 4; t++)
    {
        rc = pthread_create(&thread_ids[t],
                        NULL /* attributes */,
                        add_arrays /* function */,
                        t * 4 /* args to function */);
    }
    pthread_exit(NULL);
}
```

Dr. Rodric Rabbah, IBM.
Types of Parallelism

- **Data parallelism**
  - Perform same computation but operate on different data

- **Control parallelism**
  - Perform different functions

```c
pthread_create(/* thread id */,
               /* attributes */,
               /* any function */,
               /* args to function */);
```
Parallel Programming with OpenMP

- Start with a parallelizable algorithm
  - SPMD model (same program, multiple data)

- Annotate the code with parallelization and synchronization directives (pragmas)
  - Assumes programmers knows what they are doing
  - Code regions marked parallel are considered independent
  - Programmer is responsibility for protection against races

- Test and Debug
Simple OpenMP Example

- **(data) parallel pragma**
  execute as many as there are processors (threads)

- **for pragma**
  loop is parallel, can divide work (work-sharing)

```
#pragma omp parallel
#pragma omp for
  for(i = 0; i < 12; i++)
    C[i] = A[i] + B[i];
```
Programming Distributed Memory Processors

- Processors 1…n ask for X
- There are n places to look
  - Each processor’s memory has its own X
  - Xs may vary

- For Processor 1 to look at Processors 2’s X
  - Processor 1 has to request X from Processor 2
  - Processor 2 sends a copy of its own X to Processor 1
  - Processor 1 receives the copy
  - Processor 1 stores the copy in its own memory
**Message Passing**

- Architectures with distributed memories use explicit communication to exchange data
  - Data exchange requires synchronization (cooperation) between senders and receivers

![Diagram]

- How is “data” described
- How are processes identified
- Will receiver recognize or screen messages
- What does it mean for a send or receive to complete
Example Message Passing Program

- Calculate the distance from each point in $A[1..4]$ to every other point in $B[1..4]$ and store results to $C[1..4][1..4]$

```
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```

Dr. Rodric Rabbah, IBM.
Example Message Passing Program

- Calculate the distance from each point in \( A[1..4] \) to every other point in \( B[1..4] \) and store results to \( C[1..4][1..4] \)

\[
\text{for (i = 1 to 4)} \\
\quad \text{for (j = 1 to 4)} \\
\quad \quad C[i][j] = \text{distance}(A[i], B[j])
\]

Dr. Rodric Rabbah, IBM.
Example Message Passing Program

- Calculate the distance from each point in $A[1..4]$ to every other point in $B[1..4]$ and store results to $C[1..4][1..4]$.

- Can break up work between the two processors:
  - $P_1$ sends data to $P_2$

```plaintext
for (i = 1 to 4)
  for (j = 1 to 4)
    C[i][j] = distance(A[i], B[j])
```

Dr. Rodric Rabbah, IBM.
Example Message Passing Program

- Calculate the distance from each point in $\mathbf{A}[1..4]$ to every other point in $\mathbf{B}[1..4]$ and store results to $\mathbf{C}[1..4][1..4]$

- Can break up work between the two processors
  - $P_1$ sends data to $P_2$
  - $P_1$ and $P_2$ compute

```plaintext
for (i = 1 to 4)
    for (j = 1 to 4)
        C[i][j] = distance(A[i], B[j])
```
Example Message Passing Program

- Calculate the distance from each point in $A[1..4]$ to every other point in $B[1..4]$ and store results to $C[1..4][1..4]$

- Can break up work between the two processors
  - $P_1$ sends data to $P_2$
  - $P_1$ and $P_2$ compute
  - $P_2$ sends output to $P_1$

```plaintext
for (i = 1 to 4)
  for (j = 1 to 4)
    $C[i][j] = \text{distance}(A[i], B[j])$
```
**Example Message Passing Program**

processor 1

\[
\text{for } (i = 1 \text{ to } 4) \\
\quad \text{for } (j = 1 \text{ to } 4) \\
\quad \quad C[i][j] = \text{distance}(A[i], B[j])
\]

parallel with messages

sequential

processor 1

\[
A[n] = \{\ldots\}  \\
B[n] = \{\ldots\}
\]

Send \((A[n/2+1..n], B[1..n])\)

for \((i = 1 \text{ to } n/2)\)

\[
\quad \text{for } (j = 1 \text{ to } n) \\
\quad \quad C[i][j] = \text{distance}(A[i], B[j])
\]

Receive \((C[n/2+1..n][1..n])\)

processor 2

\[
A[n] = \{\ldots\}  \\
B[n] = \{\ldots\}
\]

Receive \((A[n/2+1..n], B[1..n])\)

for \((i = n/2+1 \text{ to } n)\)

\[
\quad \text{for } (j = 1 \text{ to } n) \\
\quad \quad C[i][j] = \text{distance}(A[i], B[j])
\]

Send \((C[n/2+1..n][1..n])\)
Performance Analysis

- Distance calculations between points are independent of each other
  - Dividing the work between two processors → 2x speedup
  - Dividing the work between four processors → 4x speedup

- Communication
  - 1 copy of $B[\cdot]$ sent to each processor
  - 1 copy of subset of $A[\cdot]$ to each processor

- Granularity of $A[\cdot]$ subsets directly impact communication costs
  - Communication is not free
Understanding Performance

- What factors affect performance of parallel programs?
- **Coverage** or extent of parallelism in algorithm
- **Granularity** of partitioning among processors
- **Locality** of computation and communication
Rendering Scenes by Ray Tracing

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

Dr. Rodric Rabbah, IBM.
Limits to Performance Scalability

- Not all programs are "embarrassingly" parallel
- Programs have sequential parts and parallel parts

```plaintext
a = b + c;
d = a + 1;
e = d + a;
for (i=0; i < e; i++)
    M[i] = 1;
```

Sequential part (data dependence)

Parallel part (no data dependence)
Coverage

- **Amdahl's Law**: *The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.*
- Demonstration of the law of diminishing returns
Amdahl’s Law

- Potential program speedup is defined by the fraction of code that can be parallelized.

Use 5 processors for parallel work:

<table>
<thead>
<tr>
<th>Time</th>
<th>Sequential</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>25s</td>
<td>25s</td>
<td>10s</td>
</tr>
<tr>
<td>50s</td>
<td></td>
<td>10s</td>
</tr>
<tr>
<td>25s</td>
<td></td>
<td>25s</td>
</tr>
<tr>
<td>100s</td>
<td></td>
<td>60s</td>
</tr>
</tbody>
</table>

Dr. Rodric Rabbah, IBM.
Amdahl’s Law

- Speedup = old running time / new running time
- = 100 seconds / 60 seconds
- = 1.67
- (parallel version is 1.67 times faster)

Dr. Rodric Rabbah, IBM.
Amdahl’s Law

- \( p \) = fraction of work that can be parallelized
- \( n \) = the number of processor

\[
speedup = \frac{\text{old running time}}{\text{new running time}} = \frac{1}{(1 - p) + \frac{p}{n}}
\]

- fraction of time to complete sequential work
- fraction of time to complete parallel work
Implications of Amdahl’s Law

- Speedup tends to \( \frac{1}{1-p} \) as number of processors tends to infinity

- Parallel programming is worthwhile when programs have a lot of work that is parallel in nature
Performance Scalability

Super linear speedups are possible due to registers and caches.

Typical speedup is less than linear.
Understanding Performance

- **Coverage** or extent of parallelism in algorithm
- **Granularity** of partitioning among processors
- **Locality** of computation and communication
Granularity

- Granularity is a qualitative measure of the ratio of computation to communication
- Computation stages are typically separated from periods of communication by synchronization events
Fine vs. Coarse Granularity

- **Fine-grain Parallelism**
  - Low computation to communication ratio
  - Small amounts of computational work between communication stages
  - Less opportunity for performance enhancement
  - High communication overhead

- **Coarse-grain Parallelism**
  - High computation to communication ratio
  - Large amounts of computational work between communication events
  - More opportunity for performance increase
  - Harder to load balance efficiently
The Load Balancing Problem

- Processors that finish early have to wait for the processor with the largest amount of work to complete
  - Leads to idle time, lowers utilization

```c
// PPU tells all SPEs to start
for (int i = 0; i < n; i++) {
    spe_write_in_mbox(id[i], <message>);
}

// PPU waits for SPEs to send completion message
for (int i = 0; i < n; i++) {
    while (spe_stat_out_mbox(id[i]) == 0);
    spe_read_out_mbox(id[i]);
}
```
Static Load Balancing

- Programmer make decisions and assigns a fixed amount of work to each processing core a priori

- Works well for homogeneous multicores
  - All core are the same
  - Each core has an equal amount of work

- Not so well for heterogeneous multicores
  - Some cores may be faster than others
  - Work distribution is uneven
Dynamic Load Balancing

- When one core finishes its allocated work, it takes on work from core with the heaviest workload.
- Ideal for codes where work is uneven, and in heterogeneous multicore.
Granularity and Performance Tradeoffs

1. Load balancing
   ■ How well is work distributed among cores?

2. Synchronization
   ■ Are there ordering constraints on execution?
Data Dependence Graph

A 2 + B 3 + C[0] + C[1] +
    * +
    -
Dependence and Synchronization

Synchronisation Points
Synchronization Removal

Synchronisation Points

Dr. Rodric Rabbah, IBM.
Granularity and Performance Tradeoffs

1. Load balancing
   ■ How well is work distributed among cores?

2. Synchronization
   ■ Are there ordering constraints on execution?

3. Communication
   ■ Communication is not cheap!
Communication Cost Model

\[ C = f \times (o + l + \frac{n}{m} B + t - \text{overlap}) \]

- **total data sent**
- **number of messages**
  - **frequency of messages**
  - **overhead per message (at both ends)**
- **bandwidth along path (determined by network)**
- **network delay per message**
- **cost induced by contention per message**
  - **cost induced by contention per message**
- **amount of latency hidden by concurrency with computation**
Types of Communication

- Cores exchange data or control messages
  - Cell examples: DMA vs. Mailbox

- Control messages are often short

- Data messages are relatively much larger
Overlapping Messages and Computation

- Computation and communication concurrency can be achieved with pipelining
  - Think instruction pipelining in superscalars
Overlapping Messages and Computation

- Computation and communication concurrency can be achieved with pipelining
  - Think instruction pipelining in superscalars
- Essential for performance on Cell and similar distributed memory multicores

```
// Start transfer for first buffer
id = 0;
mfc_get(buf[id], addr, BUFFER_SIZE, id, 0, 0);
id ^= 1;
while (!done) {
  // Start transfer for next buffer
  addr += BUFFER_SIZE;
mfc_get(buf[id], addr, BUFFER_SIZE, id, 0, 0);
  // Wait until previous DMA request finishes
  id ^= 1;
mfc_write_tag_mask(1 << id);
mfc_read_tag_status_all();
  // Process buffer from previous iteration
  process_data(buf[id]);
}
```
Communication Patterns

- With message passing, programmer has to understand the computation and orchestrate the communication accordingly
  - Point to Point
  - Broadcast (one to all) and Reduce (all to one)
  - All to All (each processor sends its data to all others)
  - Scatter (one to several) and Gather (several to one)
A Message Passing Library Specification

- **MPI**: specification
  - Not a language or compiler specification
  - Not a specific implementation or product
  - SPMD model (same program, multiple data)

- For parallel computers, clusters, and heterogeneous networks, multicore

- Full-featured

- Multiple communication modes allow precise buffer management

- Extensive collective operations for scalable global communication
Where Did MPI Come From?

- Early vendor systems (Intel’s NX, IBM’s EUI, TMC’s CMMD) were not portable (or very capable)

- Early portable systems (PVM, p4, TCGMSG, Chameleon) were mainly research efforts
  - Did not address the full spectrum of issues
  - Lacked vendor support
  - Were not implemented at the most efficient level

- The MPI Forum organized in 1992 with broad participation
  - Vendors: IBM, Intel, TMC, SGI, Convex, Meiko
  - Portability library writers: PVM, p4
  - Users: application scientists and library writers
  - Finished in 18 months
Point-to-Point

- Basic method of communication between two processors
  - Originating processor "sends" message to destination processor
  - Destination processor then "receives" the message

- The message commonly includes
  - Data or other information
  - Length of the message
  - Destination address and possibly a tag

Cell “send” and “receive” commands

```
mfc_get(destination LS addr, source memory addr, # bytes, tag, <...>)
mfc_put(source LS addr, destination memory addr, # bytes, tag, <...>)
```
Synchronous vs. Asynchronous Messages

- **Synchronous send**
  - Sender notified when message is received

- **Asynchronous send**
  - Sender only knows that message is sent
Blocking vs. Non-Blocking Messages

- **Blocking messages**
  - Sender waits until message is transmitted: 
    - buffer is empty
  - Receiver waits until message is received: 
    - buffer is full
  - Potential for deadlock

- **Non-blocking**
  - Processing continues even if message hasn't been transmitted
  - Avoid idle time and deadlocks

Cell blocking mailbox “send”

```c
// SPE does some work...
// SPE notifies PPU that task has completed
spu_write_out_mbox(<message>);

// SPE does some more work...
// SPE notifies PPU that task has completed
spu_write_out_mbox(<message>);
```

Cell non-blocking data “send” and “wait”

```c
// DMA back results
mfc_put(data, cb.data_addr, data_size, ...);

// Wait for DMA completion
mfc_read_tag_status_all();

// SPE does some work...
// SPE notifies PPU that task has completed
spu_write_out_mbox(<message>);
```

Dr. Rodric Rabbah, IBM.
Sources of Deadlocks

● If there is insufficient buffer capacity, sender waits until additional storage is available

● What happens with this code?

   \[P_1 \text{ Send}(\ldots) \quad P_2 \text{ Send}(\ldots)\]
   \[P_1 \text{ Recv}(\ldots) \quad P_2 \text{ Recv}(\ldots)\]

● Depends on length of message and available buffer
Solutions

- Increasing local or network buffering
- Order the sends and receives more carefully

![Diagram showing matching send-receive and receive-send pairs]

- Write message to buffer and block until message is transmitted (buffer becomes empty)
- Blocked since buffer is full (no progress until message can be sent)

Matching send-receive pair:
- \(P_1\): Send(...) Recv(...)
- \(P_2\): Send(...) Recv(...)

Matching receive-send pair:
- \(P_1\): Recv(...) Send(...)
- \(P_2\): Send(...) Recv(...)
Broadcast

- One processor sends the same information to many other processors
  - **MPI_BCAST**

```plaintext
for (i = 1 to n)
    for (j = 1 to n)
        C[i][j] = distance(A[i], B[j])

A[n] = {...}
B[n] = {...}

Broadcast(B[1..n])
for (i = 1 to n)
    // round robin distribute B
    // to m processors
    Send(A[i % m])
...
```
Reduction

- Example: every processor starts with a value and needs to know the sum of values stored on all processors
- A reduction combines data from all processors and returns it to a single process
  - \texttt{MPI\_REDUCE}
  - Can apply any associative operation on gathered data
    - ADD, OR, AND, MAX, MIN, etc.
  - No processor can finish reduction before each processor has contributed a value
- \texttt{BCAST\_REDUCE} can reduce programming complexity and may be more efficient in some programs
Example: Parallel Numerical Integration

\[ f(x) = \frac{4.0}{1 + x^2} \]

```c
static long num_steps = 100000;

void main()
{
    int i;
    double pi, x, step, sum = 0.0;

    step = 1.0 / (double) num_steps;
    for (i = 0; i < num_steps; i++) {
        x = (i + 0.5) * step;
        sum = sum + 4.0 / (1.0 + x * x);
    }

    pi = step * sum;
    printf("Pi = %f\n", pi);
}
```
Computing Pi With Integration (OpenMP)

static long num_steps = 100000;

void main()
{
    int i;
    double pi, x, step, sum = 0.0;
    step = 1.0 / (double) num_steps;

    #pragma omp parallel for \
    private(x) reduction(+:sum)
    for (i = 0; i < num_steps; i++){
        x = (i + 0.5) * step;
        sum = sum + 4.0 / (1.0 + x*x);
    }

    pi = step * sum;
    printf("Pi = %f\n", pi);
}

- Which variables are shared?
  - step

- Which variables are private?
  - x

- Which variables does reduction apply to?
  - sum
Computing Pi With Integration (MPI)

```c
static long num_steps = 100000;

void main(int argc, char* argv[])
{
    int i_start, i_end, i, myid, numprocs;
    double pi, mypi, x, step, sum = 0.0;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    MPI_BCAST(&num_steps, 1, MPI_INT, 0, MPI_COMM_WORLD);
    i_start = myid * (num_steps/numprocs);
    i_end = i_start + (num_steps/numprocs);
    step = 1.0 / (double) num_steps;
    for (i = i_start; i < i_end; i++) {
        x = (i + 0.5) * step
        sum = sum + 4.0 / (1.0 + x*x);
    }
    mypi = step * sum;

    MPI_REDUCE(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (myid == 0)
        printf("Pi = %.15f\n", pi);
    MPI_Finalize();
}
```

Dr. Rodric Rabbah, IBM.
Understanding Performance

- **Coverage** or extent of parallelism in algorithm
- **Granularity** of data partitioning among processors
- **Locality** of computation and communication
Locality in Communication (Message Passing)
Exploiting Communication Locality
Locality of Memory Accesses (Shared Memory)

```c
for (i = 0; i < 16; i++)
    C[i] = A[i] + ...;
```
Locality of Memory Accesses (Shared Memory)

\[
\text{for } (i = 0; i < 16; i++) \\
\quad C[i] = A[i] + \ldots;
\]

- Parallel computation is serialized due to memory contention and lack of bandwidth.
Locality of Memory Accesses
(Shared Memory)

for (i = 0; i < 16; i++)
C[i] = A[i] + ...;
Locality of Memory Accesses (Shared Memory)

```c
for (i = 0; i < 16; i++)
    C[i] = A[i] + ...;
```

- Distribute data to relieve contention and increase effective bandwidth
Memory Access Latency in Shared Memory Architectures

- **Uniform Memory Access (UMA)**
  - Centrally located memory
  - All processors are equidistant (access times)

- **Non-Uniform Access (NUMA)**
  - Physically partitioned but accessible by all
  - Processors have the same address space
  - Placement of data affects performance
Summary of Parallel Performance Factors

- Coverage or extent of parallelism in algorithm
- Granularity of data partitioning among processors
- Locality of computation and communication

... so how do I parallelize my program?