Identification of Parallelism

• We can run computations in parallel if they not share data
• Refinement:
  – Both read: OK
  – Write and (read or write): NOK
• The order becomes important: data races
• Formalization [Bernstein-1960]: Two computations C1 and C2 can be done in parallel if none of the following conditions hold:
  1. \textit{C1 writes into a location that is later read by C2}:
     a read-after-write (RAW) race.
  2. \textit{C1 reads from a location that is later written into by C2}:
     a write-after-read (WAR) race.
  3. \textit{C1 writes into a location that is later overwritten by C2}:
     a write-after-write (WAW) race.
Parallel Loop Programming

- Different iterations to different processors
- On shared memory systems we can do a PARALLEL DO loop
  - The loop must be examined to find dependencies

Example

- Loop with NO data sharing:
  DO I = 1, N
  A(I) = A(I) + C
  END DO
- Loop with possible WAR race
  DO I = 1, N
  A(I) = A(I+1) + C
  END DO
- In some cases it is possible to achieve significant parallelism in the presence of races:
  SUM = 0.0
  DO I = 1, N
  R = F(B(I),C(I)) ! an expensive computation
  SUM = SUM + R
  END DO
- There is a race involving SUM, but if that floating-point addition is commutative and associative the order in which the results are added to SUM does not matter
- If F is expensive, we can gain by computing the values of F in parallel and then update the SUM in the order in which computations finish
Example

- However, to make this work, SUM updates must be in a critical region
  
  \[
  \text{SUM} = 0.0
  \]
  
  PARALLEL DO I = 1, N
  
  \[
  R = F(B(I), C(I)) \quad \text{! an expensive computation}
  \]
  
  BEGIN CRITICAL REGION
  
  \[
  \text{SUM} = \text{SUM} + R
  \]
  
  END CRITICAL REGION
  
  END DO

SPMD Programming

- To implement the SUM reduction on a message passing system the SPMD can be used
  - Scalar variables replicated
  - Explicit communication
  
  ! This code is executed by all processors
  
  ! MYSUM, MYFIRST, MYLAST, R, and I are private local variables
  
  ! MYFIRST and MYLAST are computed separately on each processor
  
  ! to point to nonintersecting sections of B and C
  
  ! GLOBALSUM is a global collective communication primitive
  
  \[
  \text{MYSUM} = 0.0
  \]
  
  DO I = MYFIRST, MYLAST
  
  \[
  R = F(B(I), C(I)) \quad \text{! an expensive computation}
  \]
  
  \[
  \text{MYSUM} = \text{MYSUM} + R
  \]
  
  ENDDO
  
  \[
  \text{SUM} = \text{GLOBALSUM(MYSUM)}
  \]

- Here the communication is built into the function GLOBALSUM, which takes one value of its input parameter from each processor and computes the sum of all those inputs, storing the result into a variable that is replicated on each processor
Finite Difference Calculation Example

- Take a simple Fortran code that computes a new average value for each data point in array A using a two-point stencil and stores the average into array ANEW
- Parallel version on a shared memory machine with four processors, using a parallel-loop dialect of Fortran
- This code may not have sufficient granularity to compensate for the overhead of dispatching parallel threads

```fortran
REAL A(100), ANEW(100)

DO I = 2, 99
    ANEW(I) = (A(I-1) + A(I+1)) / 0.5
ENDDO

REAL A(100), ANEW(100)

PARALLEL DO I = 2, 99
    ANEW(I) = (A(I-1) + A(I+1)) / 0.5
ENDO
```

Version with Higher Granularity

- Each processor gets ¼ of the work
- The PRIVATE statement specifies that each iteration of the IB-loop has its own private value of each variable in the list
- This permits each instance of the inner loop to execute independently without simultaneous updates of the variables that control the inner loop iteration

```fortran
REAL A(100), ANEW(100)

PARALLEL DO IB = 1, 100, 25
PRIVATE I, myFirst, myLast
    myFirst = MAX(IB, 2)
    myLast = MIN(IB * 24, 99)
    DO I = myFirst, myLast
        ANEW(I) = (A(I-1) + A(I+1)) / 0.5
    ENDDO
ENDO
```
Message Passing Version

- This code is written in SPMD style so that the scalar variables myP, myFirst, and myLast are all automatically replicated on each processor—the equivalent of PRIVATE variables in shared memory
- Each global array is replaced by a collection of local arrays in each memory
- Two extra storage locations on each processor—A(0) and A(26)—are used to hold values communicated from neighboring processors. These cells are often referred to as ghost cells, halo cells, or overlap areas.

Improved Version

- Insertion of local computation between the sends and receives
- Communication is overlapped with the local computation to achieve better overall parallelism
Pipeline Parallelism

• Some parallelism may be achievable by staggering initiation of tasks and synchronizing them so that subsections with no interdependencies are run at the same time
• Although neither of the loops can be run in parallel, there is some parallelism
• All of the values on the shaded diagonal can be computed in parallel because there are no dependences between any of these elements

\[
\begin{align*}
\text{DO } J = 2, N:\text{1} \\
\text{DO } I = 2, N:\text{1} \\
A(I,J) = \{A(I-1,J) + A(I-1,J) + A(I,J-1) + A(I,J+1)\} \times 0.25 \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

Pipeline Parallelism

• If we compute all the elements in any column on the same processor, so that \(A(*,J)\) would be computed on the same processor for all values of \(J\)
• If we compute the elements in any column in sequence, all of the dependences along that column are satisfied. However, we must still be concerned about the rows.
• To get the correct result, we must delay the computation on each row by enough to ensure that the corresponding array element on the previous row is completed before the element on the current row is computed.
• This strategy can be implemented via the use of events

EVENT READY(N,M) : 1 initialized to false
PARALLEL ED I = 1, N
POST(READY(1,1))
ENDDO
PARALLEL ED J = 2, M, 1
ED I = 2, N, 1
WAIT(READY(I,1,1))
A(I,J) = \{A(I-1,J) + A(I+1,J) + A(I,J-1) + A(I,J+1)\} \times 0.25
POST(READY(I,J))
ENDDO
ENDDO
Pipeline Parallelism

- Initially all the events are false—a wait on a false event will suspend the executing thread until a post for the event is executed.
- All of the READY events for the first column are then posted, so the computation can begin.
- The computation for the first computed column, A(*,2), begins immediately.
- As each of the elements is computed, its READY event is posted so that the next column can begin computation of the corresponding element.

Example Parallelization with Threads

- 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.

```plaintext
for (i = 0; i < 12; i++)
  C[i] = A[i] + B[i];
```
Example Code 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(&threads_ids[t],
                            NULL /* attributes */ ,
                            add_arrays /* function */ ,
                            t /* arg to function */ );
    }
    pthread_exit(NULL); }
```

Functional and Domain Decomposition with Threads

- **Functional decomposition or control parallelism**
  - Each thread performs a different function

- **Domain decomposition**
  - Several threads perform same computation but operate on different data
Performance Evaluation

• Example
  – 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]

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

Performance Evaluation

• Example
  – 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]

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

- Example
  - 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)
  \(C[i][j] = \text{distance}(A[i], B[j])\)
```

Performance Evaluation

- Example
  - 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

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

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B[n] = (...)</td>
<td>B[n] = (...)</td>
</tr>
<tr>
<td>for (i = 1 to n/2)</td>
<td>for (i = n/2+1 to n)</td>
</tr>
<tr>
<td>for (j = 1 to n)</td>
<td>for (j = 1 to n)</td>
</tr>
<tr>
<td>C[i][j] = distance(A[i], B[j])</td>
<td>C[i][j] = distance(A[i], B[j])</td>
</tr>
<tr>
<td>receive (C[n/2+1...n][1..n])</td>
<td>send (C[n/2+1...n][1..n])</td>
</tr>
</tbody>
</table>

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[] sent to each processor
  - 1 copy of subset of A[] to each processor
- Granularity of A[] 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

Limits to Performance Scalability

• Not all programs are “embarrassingly” parallel
• Programs have sequential parts and parallel parts

```
a = b + c;
d = a + 1;
e = d / a;
for (i=0; i < e; i++)
    M[i] = 1;
```
Amdhal’s Law

- **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.

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

\[
\text{Speedup} = \frac{\text{old running time}}{\text{new running time}}
\]

\[
= \frac{100 \text{ seconds}}{60 \text{ seconds}} = 1.67
\]

(parallel version is 1.67 times faster)
Implications of Amdhal’s Law

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

\[
\text{speedup} = \frac{\text{old running time}}{\text{new running time}} = 1 - \frac{\frac{\text{fraction of time to complete sequential work}}{1}}{\text{fraction of time to complete parallel work}} + \frac{1}{n}
\]

- Speedup tends to 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
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 Grain Parallelism

- 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 stages
  - More opportunity for performance enhancement
  - Harder to balance efficiently
Communication Cost Model

\[ C = f \cdot \left( o + l + \frac{n}{m} \cdot \frac{1}{B} + t \cdot \text{overlap} \right) \]

- frequency of messages
- overhead per message (at oom ends)
- network delay per message
- number of messages
- cost induced by contention per message
- amount of latency hidden by concurrency with computation
- bandwidth along path (determined by network)

Overlapping Messages and Computation

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

```c
// Start transfers for first buffer
id = 0;
sfu_get(Sbuf[0], &sh, BUFFERSIZE, &id, 0, 0);
while (id) {
    // Start transfers for next buffer
    id = 0;
sfu_get(Sbuf[0], &sh, BUFFERSIZE, &id, 0, 0);
    // Wait until previous DMA request finishes
    id = 0;
sfu_get(Sbuf[0], &sh, BUFFERSIZE, &id, 0, 0);
sfu_set_tag_mask(id, 0);
sfu_set_tag_status_all(id);
    // Process buffer from previous iteration
    process_data(Sbuf[0]);
}```
Types of Messages

• Synchronous vs Asynchronous
• Blocking vs Non-blocking

SYNC
• Sender notified when message is received

BLOCKING
• Sender waits until message is transmitted: buffer is empty
• Receiver waits until message is received: buffer is full
• Potential for deadlock

ASYNC
• Sender only knows that message is sent

NON-BLOCKING
• Processing continues even if message hasn’t been transmitted
• Avoid idle time and deadlocks

Example

• Cell processor
  – SPE and PPU message passing

Cell blocking mailbox “send”

```c
// SPE does some work
...
// SPE notifies PPU that task has completed
cpu_write_out_mailbox(&message);
...
// SPE does some more work
...
// SPE notifies PPU that task has completed
cpu_write_out_mailbox(&message);
```

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

```c
// DMA task results
cpu_write_data(&data, &data_addr, &data_size, ...);
// Wait for DMA completion
cpu_read_tag_status_all();
```
Source of Deadlocks

- If there is insufficient buffer capacity, sender waits until additional storage is available
- What happens with the following code depends on length of message and available buffer

```
<table>
<thead>
<tr>
<th>P1</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send()</td>
<td>Send()</td>
</tr>
<tr>
<td>Recv()</td>
<td>Recv()</td>
</tr>
<tr>
<td>P1</td>
<td>P1</td>
</tr>
</tbody>
</table>
```

Solutions

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

```
<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send()</td>
<td>Send()</td>
</tr>
<tr>
<td>Recv()</td>
<td>Recv()</td>
</tr>
</tbody>
</table>
```

Blocked since buffer is full (no progress until message can be sent)
Broadcast

• One processor sends the same information to many other processors
  – MPI_BCAST (in MPI language)

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

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

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
  – 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
• BCAT/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)

• Which variables are shared?
  - step

• Which variables are private?
  - x

• Which variables does reduction apply to?
  - sum

```c
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);
}
```
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, sum, 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);
    }
    mpii = step * sum;
    MPI_BCAST(&mpi_i, 6pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0)
        printf("Pi = \%fn", pi);
    MPI_Finalize();
}
```

Importance of Locality

- Locality of memory accesses (shared memory)

```c
for (i = 0; i < 10; i++)
    C[i] = A[i] + ...;
```

Parallel computation is serialized due to memory contention and lack of bandwidth
Importance of Locality

- Locality of memory accesses (shared memory)

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

Distribute data to relieve contention and increase effective bandwidth.