Parallel Programming Concepts
What is concurrency?

- What is a sequential program?
  - A single thread of control that executes one instruction and when it is finished execute the next logical instruction

- What is a concurrent program?
  - A collection of autonomous sequential threads, executing (logically) in parallel

- The implementation (i.e. execution) of a collection of threads can be:
  - Multiprogramming
    - Threads multiplex their executions on a single processor.
  - Multiprocessing
    - Threads multiplex their executions on a multiprocessor or a multicore system
  - Distributed Processing
    - Processes multiplex their executions on several different machines
Concurrency and Parallelism

- Concurrency is not (only) parallelism

- Interleaved Concurrency
  - Logically simultaneous processing
  - Interleaved execution on a single processor

- Parallelism
  - Physically simultaneous processing
  - Requires a multiprocessor or a multicore system
Example Parallelization

for (i = 0; i < 12; i++)
   C[i] = A[i] + B[i];

- **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
Types of Parallelism

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

- **Control (task) parallelism**
  - Perform different functions

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

Sequential part
(data dependence)

Parallel part
(no data dependence)

\[
\begin{align*}
a &= b + c; \\
d &= a + 1; \\
e &= d + a; \\
\text{for } (i=0; i < e; i++) \\
&\quad M[i] = 1;
\end{align*}
\]
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:

- 25 seconds + 50 seconds + 25 seconds = 100 seconds
- Sequential + Parallel + Sequential = 60 seconds

Sequential

Sequential

Parallel

Sequential

Sequential
Amdahl’s Law

- **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)
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

Typical speedup is less than linear

speedup

number of processors

linear speedup (100% efficiency)
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
Granularity

- Parallel loops/regions have overhead
  - Invoking the parallel loops
  - Executing the barriers
  - Cache and synchronization effects
  - Thread management

- If the coverage is perfect, but the program invokes a very large number of very small parallel loops, then performance might be limited by granularity
Granularity

- Time in cycles for empty parallel do

- One should not parallelize a loop or region unless it takes significantly more time to execute than the parallel overhead

<table>
<thead>
<tr>
<th>Processors</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1800</td>
</tr>
<tr>
<td>2</td>
<td>2400</td>
</tr>
<tr>
<td>4</td>
<td>2900</td>
</tr>
<tr>
<td>8</td>
<td>4000</td>
</tr>
<tr>
<td>16</td>
<td>8000</td>
</tr>
</tbody>
</table>
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

Dependence and Synchronization

Synchronisation Points

P1 → P2 → P3

P1 → P2 → P3

P1 → P2 → P3

P1 → P2 → P3
Synchronization Removal

Synchronisation Points
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!
Understanding Performance

- **Coverage** or extent of parallelism in algorithm
- **Granularity** of data partitioning among processors
- **Locality** of computation and communication
for (i = 0; i < 16; i++)
C[i] = A[i] + ...;
Locality of Memory Accesses (Shared Memory)

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

- 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] + ...;

fork (threads)

join (barrier)

memory banks

A[0]
A[8]
A[12]
A[1]
A[5]
A[9]
A[13]
A[2]
A[10]
A[14]
A[3]
A[7]
A[15]
Locality of Memory Accesses
(Shared Memory)

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

● Distribute data to relieve contention and increase effective bandwidth
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?