Introduction to parallel computing

3. Parallel Software

Zhiao Shi (modifications by Will French)
Advanced Computing Center for Education & Research
Vanderbilt University
Last time

• Parallel hardware
  • Multi-core processors and hyperthreading
  • Memory hierarchy
  • von Neumann architecture
  • Flynn’s taxonomy
  • Shared memory vs. distributed memory systems
Process

• An instance of a computer program that is being executed.

• Components of a process:
  • The executable machine language program.
  • A block of memory.
  • Descriptors of resources the OS has allocated to the process.
  • Security information.
  • Information about the state of the process.
Threads

• Threads are contained within processes.
• A thread generally shares its process’s instructions, variables, and some memory, but can also be given its own local memory/variables/data to work independently from other threads.
• Multiple threads will execute across multiple CPU cores if they are available, otherwise the OS interleaves their execution.
• Allow programmers to divide their programs into (more or less) independent tasks.
Single and Multithreaded Processes

single-threaded process

multithreaded process
The burden is on software

• In shared memory programs:
  • Start a single process and spawn threads.
  • Threads carry out computations in parallel and update shared memory space.

• In distributed memory programs:
  • Start multiple processes.
  • Processes carry out computations in parallel and send/receive updated data via “messages” from other processes.
The main program `myprog` is scheduled to run by the OS.

`myprog` performs serial work and then spawns threads.

Each thread has local data, also share all the resources of `myprog`.

Threads communicate with each other implicitly through global memory.
Distributed Memory Programming Model

• A set of tasks (i.e. processes) that use their own local memory during computation.

• Tasks exchange data through communications by sending and receiving messages.

• Data transfer usually requires cooperative operations to be performed by each process.
Writing Parallel Programs: Rules of Thumb

(1) Divide the work among the processes/threads so each process/thread gets roughly the same amount of work and communication is minimized.

(2) Arrange for the processes/threads to synchronize.

(3) Arrange for communication among processes/threads.
Performance of a parallel system

• A sequential algorithm is evaluated by its run time.

• The parallel run time of a program depends not only on the input size, but also the number of processors, and the communication parameters of the machine.

• A parallel algorithm must therefore be analyzed in the context of the underlying platform.

• A parallel system is a combination of a parallel algorithm and an underlying parallel platform.
Performance Metrics for Parallel Systems

- Metrics that can be used to measure the performance of parallel systems:
  - Execution time
  - Speedup
  - Efficiency
  - Overhead
Speedup

- Speedup = $S$
- Number of cores = $p$
- Serial run-time = $T_{\text{serial}}$
- Parallel run-time = $T_{\text{parallel}}$

\[
S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}
\]

linear speedup

\[
T_{\text{parallel}} = \frac{T_{\text{serial}}}{p}
\]
Speedup

• For a given problem, there might be many serial algorithms available. These algorithms may have different asymptotic runtimes and may be parallelizable to different degrees.

• For the purpose of computing speedup, we always consider the best sequential program as the baseline.
Amdahl’s Law

• The possible speedup of a program is limited by the fraction that cannot be parallelized, regardless of the number of cores available.
  • For example, if 50% of a program is serial, the max speedup is 2.
• \( p \) is the number of CPU cores
• \( T \) is the execution time
• \( B \) is the serial fraction of the program

\[
T(p) = T(1) \left( B + \frac{1}{p} (1 - B) \right)
\]
Example: Amdahl’s Law

• We can parallelize 90% of a serial program.
• Parallelization is “perfect” regardless of the number of cores $p$ we use.
• $T_{\text{serial}} = 20$ seconds
• Runtime of parallelizable part is:

\[ T(p) = (20 \text{ sec})(0.10 + (1/p)(0.90)) \]
\[ T(p) = (2 + 18/p) \text{ sec} \]
\[ S(p) = 20 / (2 + 18/p) \]
SC290 Repo Example

• amdahl.py
Question

What are the assumptions made by Amdahl’s Law?

\[ T(p) = T(1) \left( B + \frac{1}{p} (1 - B) \right) \]
Sources of Overhead in Parallel Programs

• If I use two processors, shouldn’t my program run twice as fast?

• No - a number of overheads, including communication, idling, excess computation, and contention cause degradation in performance.

![Graph showing execution time for different processors](image-url)
Sources of Overhead in Parallel Programs

• Interprocess interactions: Processors working on any non-trivial parallel problem will need to talk to each other.
  • Generally the largest source of overhead.
  • Programs with no interprocess (or inter-thread) communication or synchronization are called **embarrassingly parallel**.

• Idling: Processes may idle because of load imbalance, synchronization, or serial components.

• Excess Computation: This is computation not performed by the serial version. This might be because the serial algorithm is difficult to parallelize, or that some computations are repeated across processors to minimize communication.

• Goal is to minimize these!
Overhead

\[ T_{\text{parallel}} = T_{\text{serial}} / p + T_{\text{overhead}} \]
Efficiency of a parallel program

\[ E = \frac{S}{p} = \left( \frac{T_{\text{serial}}}{T_{\text{parallel}}} \right) = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}} \]
Speedups and efficiencies of a parallel program

<table>
<thead>
<tr>
<th>( p )</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S  )</td>
<td>1.0</td>
<td>1.9</td>
<td>3.6</td>
<td>6.5</td>
<td>10.8</td>
</tr>
<tr>
<td>( E = S/p )</td>
<td>1.0</td>
<td>0.95</td>
<td>0.90</td>
<td>0.81</td>
<td>0.68</td>
</tr>
</tbody>
</table>
### Speedups and efficiencies of parallel program on different problem sizes

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half</td>
<td>S</td>
<td>1.0</td>
<td>1.9</td>
<td>3.1</td>
<td>4.8</td>
<td>6.2</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>1.0</td>
<td>0.95</td>
<td>0.78</td>
<td>0.60</td>
<td>0.39</td>
</tr>
<tr>
<td>Original</td>
<td>S</td>
<td>1.0</td>
<td>1.9</td>
<td>3.6</td>
<td>6.5</td>
<td>10.8</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>1.0</td>
<td>0.95</td>
<td>0.90</td>
<td>0.81</td>
<td>0.68</td>
</tr>
<tr>
<td>Double</td>
<td>S</td>
<td>1.0</td>
<td>1.9</td>
<td>3.9</td>
<td>7.5</td>
<td>14.2</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>1.0</td>
<td>0.95</td>
<td>0.98</td>
<td>0.94</td>
<td>0.89</td>
</tr>
</tbody>
</table>
Speedup

![Graph showing speedup with different process sizes.](Graph.png)
Efficiency
Scalability

• In general, a program is **scalable** if it can run efficiently as the number of processes/threads is increased.

• If we increase the number of processes/threads and keep the efficiency fixed without increasing problem size, the problem is **strongly scalable**.

• If the efficiency drops as we increase the number of processes/threads, the problem is **weakly scalable**.