CME342 - Parallel Methods in Numerical Analysis

Parallel Performance Analysis

April 21, 2014
Lecture 6
Objectives of Parallel Computing

• Our objective in this course is to learn about the necessary parallel programming models to:

  ➜ Guide algorithmic software development by using predictive models of performance

  ➜ Determine, a priori, the best choice of algorithm to be executed on a parallel computer
Example

1. Building of the Hadrian’s wall in the border between England and Scotland. Commissioned by Emperor Hadrian of Rome in 122 A.D. (73.3 miles in length).
   • Perfect example of a parallel computing problem.
   • Large size, embarrassingly parallel, parallel I/O
Example

- A large number of legionnaires (processors) can be used at the same time with high efficiency.
- Global communication is not required: each legionnaire only needs to talk to the two legionnaires to his right and left.
- SPMD (Single Program Multiple Data) strategy is being used: legionnaires are replicated along the wall.
- Each legionnaire can retrieve his own mortar and stone (parallel I/O) assuming there is no bottleneck at the supply location.
Example

• Consider instead the building of the tower of Babel. Is there any parallelism to be found in this problem? Some, at each floor level, but this problem is inherently *sequential*. Similar observations can be made in the time-accurate integration of PDEs unless you can think cleverly about the problem and reduce it.

• Your job as a parallel scientist is to rediscover the way of building a tower such that additional areas of parallelism can be found and exploited.
Single-Processor Performance

• Mflop rating
• Vector vs. Cache architectures
• Memory hierarchy
• Microprocessor architectural issues (no. of ops/cycle, no. of load/store ops/cycle, pipelining ability, etc.)
• Careful decomposition and programming issues
Multiple-Processor Performance

- CPU scalability
- Memory scalability
- Interconnection network
- Bandwidth and latency issues
- Problem size and granularity
- *How many processors can we use efficiently?*
Mflop Rating

- 1 Mflop = $10^6$ Floating Point Operations/Sec
- Speed of computation: $N$ operations in $t$ microseconds

\[ r = \frac{N}{t} \text{ Mflops} \]

- Execution time:

\[ t = \frac{N}{r} \mu \text{ sec} \]
Amdahl’s Law - Single Processor

- Algorithm execution requires, \( N \) flops
- Fraction, \( f \), can be executed at speed \( V \) Mflops; the rest executes at \( S \) Mflops, and

\[
V \gg S
\]

- Total execution time:

\[
t = \frac{fN}{V} + \frac{(1-f)N}{S} = N\left(\frac{f}{V} + \frac{1-f}{S}\right)\mu \text{sec}
\]
Amdahl’s Law - Single Processor

• Computational performance:

\[
r = \frac{N}{t} = \frac{1}{f/V + (1 - f)/S} \text{ Mflops}
\]

• Therefore:

\[
t > \frac{(1 - f)N}{S} \mu \text{sec}
\]

• If \( V \gg S \), then

\[
r = \frac{1}{f/V + (1 - f)/S} \approx \frac{1}{(1 - f)S}
\]
Amdahl’s Law - Single Processor

- Relative gain in CPU time bounded by $\frac{1}{1-f}$
- $f$ needs to be quite large to obtain reasonable performance
Amdahl’s Law - Single Processor

• In serial programs, if even a small portion of your algorithm executes at a very slow speed, $S$, the performance of the complete program may be limited by these effects.

• Usual approach is to identify the areas of the program that consume the largest percentages of the execution time (main contribution to $f$), and make sure that they are coded in such a way that they can achieve execution speeds closer to $V$ than to $S$. 
The issues involved in optimizing single processor performance are many, and they are usually dependent on the architecture of the processor at hand.

Look at Dongarra’s book for some suggestions.

Consult computer manufacturer’s performance tuning guides (usually online) to discover the issues of highest relevance.

For modern (RISC), cache-based microprocessors a lot of these most important issues are common.

Link from web site to SGI tuning guide. Very useful!!!
Single Processor Performance

- Things to worry about:
  - Use existing optimized code when possible (SGI `complib.sgimath` libraries or Intel Math Kernel)
  - Find out where to tune: profiling tools, performance analysis tools, etc. (prof, hardware counters, etc.)
  - Loop ordering: Fortran vs. C, column- and row-major order
  - Understand cache hierarchy:
    - 2-level cache hierarchies. Size? 32Kb for primary cache, 1-4 Mb for secondary cache
    - cache-block or cache line concept. Size? 32 bytes for primary cache, 128 bytes for secondary cache
Single Processor Performance

• **Things to worry about:**
  – **Understand cache hierarchy:**
    • Cache hit and cache miss (hiding memory latency)
    • Access times:
      – registers: 0 cycles
      – 1st level cache: 2-3 clock cycles
      – 2nd level cache: 8-10 clock cycles
      – main memory: 200-1100 nanosec (60-200 cycles!)
• Stride 1 accesses:

\[
\begin{align*}
    &\text{do } i=1,n \\
    &\quad \text{do } j=1, n \\
    &\quad \quad a(i,j) = b(i,j) \\
    &\quad \text{end do}
\end{align*}
\]

• If matrices are large, every element in the loop will have to be loaded multiple times from main memory!
Single Processor Performance

• Stride 1 accesses:

\[
\begin{align*}
\text{do } & j=1, n \\
\text{do } & i=1, n \\
\text{a}(i,j) & = b(i,j) \\
\text{end do}
\end{align*}
\]

• Operations are done one row at a time. Cache is reused efficiently.
Single Processor Performance

• Group together data used at the same time:

\[ D = 0.0 \]
\[ \text{do } i=1,n \]
\[ \quad j = \text{ind}(i) \]
\[ \quad d = d + \sqrt{x(j)\times x(j) + y(j)\times y(j) + z(j)\times z(j)} \]
\[ \text{end do} \]

• 3 cache lines need to be loaded every time, since x, y, and z, are likely to be stored in different areas of memory.
Single Processor Performance

• Instead:

\[ D = 0.0 \]
\[ \text{do } i=1,n \]
\[ \quad j = \text{ind}(i) \]
\[ \quad d = d + \sqrt{r(1,j)*r(1,j) + r(2,j)*r(2,j) + r(3,j)*r(3,j)} \]
\[ \text{end do} \]

• Data for x, y, z are now stored in a contiguous array. When loading the cache line for r(1,j), it is likely that the other elements will be loaded at the same time.
Single Processor Performance

• Cache thrasing (cache associativeness):

```
parameter (max=1024*1024)
dimension a(max), b(max), c(max), d(max)
do i=1,max
   a(i) = b(i) + c(i)*d(i)
end do
```

• Data is stored in cache according to the lower n bits of the memory address (2-way, 4-way, etc)

• Arrays are allocated sequentially, therefore, they will be contiguous in memory
Single Processor Performance

• Since their size is 4Mb (single precision), they will share the lowest 22 bits of the memory addresses and the 4 vectors will map to the same cache location.
• Cache is constantly loading and unloading because of this reason, even though accesses are stride-1, and the performance degrades significantly.
• You can re-dimension the arrays so that they do not all map to the same cache location. Good rule of thumb, do not use powers of two.
• Introduce padding variables in the declaration.
Amdahl’s Law - Parallel Processing

• In an ideal world, if computation can be carried out in $p$ equal parts, the total execution time will be nearly $1/p$ of the time required by a single processor.

• Suppose $t_j$ denotes the wall clock time required to execute a task with $j$ processors.

• Speedup, $S_p$, for $p$ processors is defined as

$$S_p = \frac{t_1}{t_p}$$
Amdahl’s Law - Parallel Processing

- Speedup, $S_p$, for $p$ processors is defined as

\[ S_p = \frac{t_1}{t_p} \]

- Where $t_1$ is the time required for the most-efficient sequential algorithm to complete the calculation, and $t_p$ is the time required for the most efficient parallel implementation of the same algorithm, from beginning to end, using $p$ processors.
• The computational efficiency using \( p \) processors is defined as

\[
E_p = \frac{S_p}{p}, 0 \leq E_p \leq 1
\]

• Then, if \( f \) is the portion of the program that is parallelizable, the total execution time using \( p \) processors is given by

\[
t_p = \frac{ft_1}{p} + (1 - f)t_1 = \frac{t_1(f + (1 - f)p)}{p} \geq (1 - f)t_1
\]
Amdahl’s Law - Parallel Processing

• The speedup on \( p \) processors is then

\[
S_p = \frac{p}{(f + (1-f)p)} \leq \frac{1}{1-f}
\]

• This is often called Ware’s Law

• This shows that the speedup is considerable reduced even for pretty large values of \( f \) (close to 95%)
Amdahl’s Law - Parallel Processing

- Parallel overhead is the additional amount of work that is required on the parallel implementation of a sequential algorithm arising from the use of a parallel computer:
  - Interprocessor communication
  - Load imbalance
  - Additional computation resulting from the algorithm that is being parallelized not being as efficient as the most efficient serial algorithm.
Amdahl’s Law - Parallel Processing

f values:
0.7217
0.9226
0.9785
0.9940
0.9983
0.9995
0.9999
0.99999
1.0000
Amdahl’s Law - Parallel Processing

• Amdahl’s law is a simplistic, yet powerful way of looking at the problem of scalability.

• In a naïve way, it points out that a large number of processors cannot be used on any computational task, since f needs to be very close to 1. For example, f=0.999 would allow the use of a maximum number of processors equal to 1000. Does your problem have a 0.999 portion of parallelism?
Example - Adding on a Hypercube

• Consider adding $n$ numbers using $n$ processors of a hypercube ($n$ is a power of two)

• Initially, each processor gets one number, and at the end, one processor has the sum of all of them

• Addition and communication take each 1 unit of time
Example - Adding on a Hypercube

Step 1

Step 2

Step 3

Step 4

\[ T_p = \Theta(\log n) \quad S = \Theta\left(\frac{n}{\log n}\right) \quad E = \frac{S}{n} = \Theta\left(\frac{1}{\log n}\right) \]
Example - Adding on a Hypercube

\[ T_p = \Theta(\log n) \quad S = \Theta\left(\frac{n}{\log n}\right) \quad E = \frac{S}{n} = \Theta\left(\frac{1}{\log n}\right) \]

- Performance not impressive: for 32 numbers and 32 processors, E = 20% !!!!

- Maybe we are using too many processors for this problem? Maybe the problem size is too small?

- Let’s try again.
Example - Adding on a Hypercube

- Assign larger pieces of data to each processor (more than one number per processor)
- Consider adding \( n \) numbers using \( p \) processors of a hypercube, \( p < n \), both \( p \) and \( n \) are powers of two.
- Initially, each processor gets \( n/p \) numbers, and at the end, one processor has the sum of all of them
- Addition and communication take each 1 unit of time
Example - Adding on a Hypercube

<table>
<thead>
<tr>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Σ

<table>
<thead>
<tr>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Σ</td>
<td>Σ</td>
<td>Σ</td>
<td>Σ</td>
</tr>
</tbody>
</table>

Σ

<table>
<thead>
<tr>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Σ</td>
<td>Σ</td>
<td>Σ</td>
<td>Σ</td>
</tr>
</tbody>
</table>

Σ

<table>
<thead>
<tr>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Σ</td>
<td>Σ</td>
<td>Σ</td>
<td>Σ</td>
</tr>
</tbody>
</table>

Σ
Example - Adding on a Hypercube

Step 2
Example - Adding on a Hypercube

Step 3

- \( \sum \sum \)
- \( \sum \sum \)

- \( \sum \)
- \( \sum \)

0 1 2 3
0 1 2 3
Example - Adding on a Hypercube

Step 4

- First $\log p$ steps (until only one processor left) of the $\log n$ steps are done in
  - $(n/p) \log p$ steps on $p$ processors

- Remaining steps require no communication, therefore, adding $n/p$ numbers takes $n/p$ time
Example - Adding on a Hypercube

- Expected computational time (from n-hypercube):
  \[ T_p = \Theta((n/p) \log n) \]

- However previous algorithm shows (for p-hypercube and n-vector):
  \[ T_p = \Theta((n/p) \log p) \]
Example - Adding on a Hypercube

• Total computation time, assuming it takes one unit of time to add two numbers and one unit of time to transfer a number between neighboring processors

\[ T_p = \left( \frac{n}{p} - 1 \right) + 2\log p \approx \frac{n}{p} + 2\log p \]

• Speedup

\[ S = \frac{n}{n + 2\log p} = \frac{np}{n + 2p\log p} \]
Example - Adding on a Hypercube

• Efficiency

\[ E = \frac{S}{p} = \frac{n}{n + 2p \log p} \]

• Note that:
  – Speedup does not increase linearly with the number of processors
  – Speedup increases with larger vector sizes (for fixed \( p \))
  – If we scale the problem at the same time we increase the number of processors, and efficiency remains constant, we have a scalable system and algorithm
Example - Adding on a Hypercube
Example - Adding on a Hypercube
Realistic Test Case

- FLO107-MB, multiblock flow solver, precursor to ASCI TFLO. Not very different!!!
Stanford University PSAAP Center

Brief experiences using hyperion.llnl.gov

November 14, 2008
Summary of Initial Experience

• Initial experiences based on running the compressible flow solver JOE on two test cases:
  – HyShot II Scramjet Engine (2\textsuperscript{nd} order discretization)
  – Convergent nozzle jet flow (Discontinuous Galerkin discretization)

• The new machine appears to be stable, well configured, and shows excellent scalability. Parallel I/O performance is much lower than expected (perhaps because the parallel file system is not full configured/optimized yet?)
Some Details of the Solver, JOE

• Fully unstructured, arbitrary polyhedral, compressible Unsteady Reynolds-Averaged Navier-Stokes and Large-Eddy Simulation solver

• Parallelization achieved via MPI using domain decomposition (ParMETIS)

• Time accurate simulation of the flow as an initial disturbance (heat release in the combustor) is introduced
HyShot II Test Case
Unsteady Unstart Simulation of a Scramjet

Stanford PSAAP Center – Initial Experiences on hyperion.llnl.gov
HyShot II Test Case
Unsteady Unstart Simulation
of a Scramjet
Scalability of JOE on hyperion.llnl.gov
Granularity

• Often, the concept of granularity is what matters most.

• Granularity: Ratio of amount of computation that a processor performs to the amount of communication that it requires

• Coarse grain parallelism: typical MPI applications. Decrease communication overhead. Most suitable for low performance networks (also for high performance, of course).
Granularity

- **Fine grain** parallelism: most typical in OpenMP or CUDA applications where small portions of computation (such as operations inside a loop) are divided among multiple processors.

- Fine grain parallelism requires high performance network/memory subsystems.
Granularity

• Note the inherent benefit in scaling up the problem size for a block of typical dimension, $L$:
  – Amount of communication is proportional to the surface area of the block $\approx L^2$
  – Amount of computation is proportional to the volume of the block $\approx L^3$

• Therefore, as the problem gets larger, the granularity increases $\approx L$ and ratio of comm vs computation goes down as $\approx \frac{1}{L}$
Bandwidth

• Bandwidth is the rate of information transfer that a communication subsystem can maintain (MB/sec)
  – Bandwidth that matters is software bandwidth (using MPI, for example)
  – More is better (… and much more expensive)
  – To minimize communication overhead, send only necessary information
Latency

- Latency is defined as the time it takes to send a zero-length message from one processor to another (measured in microseconds, typically)
  - Less is better (… and more expensive)
  - Software latency (MPI, for example) matters
  - Impacts short messages mostly (and algorithms that rely heavily on short messages, i.e. multigrid)
  - Try to agglomerate messages if possible to decrease communication cost.
  - Typical number: 15 microseconds for Origin2000 (2000), 0.5-1 microsecond Infiniband (2010)