A Gut-Feel, Informal

Introduction to Parallel Computing

Doug James
djames@tacc.utexas.edu
Sep 2013

© The University of Texas at Austin, 2013
Please see the final slide for copyright and licensing information.
Overview

Architectures and Programming Models
Levels of Parallelism
Practical and Theoretical Performance Limits
Other Issues and Challenges
Summary
Architectures
and Programming Models
What is Parallel Programming?

More than one paint brush!

Paint the fence faster...

...or paint a bigger fence

Paint brushes = cores
What is Parallel Programming?

More than one mower!

Mow the lawn faster...

...or mow a bigger lawn

Lawn mowers = cores

Brett Chisum 2012 (Augusta National)
Wikipedia Commons
http://www.flickr.com/photos/brettchisum/7051114207
Shared Memory

- All cores share a common pool of memory (RAM)
- The programming challenge is coordination: how to avoid competing for access to the same puzzle pieces (memory)
- Principal programming model: OpenMP
- A **single executable** spawns independent threads and manages threads' access to data
Distributed Memory

- Each core* has its own memory (RAM), inaccessible to other cores
- The programming challenge is communication: how to share puzzle pieces (data)
- Principal programming model is MPI (Message Passing Interface)
- Every assigned core runs a separate copy of the same executable -- a “rank aware” task

*we’ll modify this in a few slides
Hybrid Architecture

• Most large clusters are hybrids of these models
  – Each node (blade) is a multi-core shared memory computer running its own (Linux) operating system
  – Many such nodes connected in distributed configuration
  – Each core sees only the memory on its own node!
Programming Hybrid Architectures

- Programming models vary
  - Pure MPI: ignore shared memory
  - Hybrid: mix MPI and OpenMP
  - Pure OpenMP: and confine yourself to one node
Hybrid Architecture with MICs

Stampede’s Xeon Phi MICs present additional programming models

- Native: MIC as stand-alone shared memory computer (OpenMP, MPI)
- Symmetric: MICs running MPI tasks with other MICs and Sandy Bridge hosts
- Offload: MIC as servant (coprocessor) to the Sandy Bridge E5 host -- like General Purpose Graphical Processing Units (GPUs)
Levels of Parallelism
Needle(s) in the Haystack(s)

• First approach: think top-down and coarse-grained
• Partition the work into essentially independent tasks
Coarse-Grained Parallelism

- Assign tasks to processors (nodes, cores, ...)
- Also called task-based parallelism
Single Program Multiple Data (SPMD)

- The same code operates on different data
- Logic within the program may differ across processors
- How much communication, coordination, synchronization?
Massive (Embarrassing) Parallelism

- High degree of independence
- Little to no coordination, communication
Massive (Embarrassing) Parallelism

• Important example: parameter sweeps
• We have tools that support this: launcher, pylauncher
Domain Decomposition

Key issues

- Dependencies across ghost (halo/transition/boundary) regions
- Communication
- Load balancing
- Bookkeeping (code complexity)
Fine-Grained Parallelism: Vectorization

One combine, multiple rows of wheat

C. Holmes 2009
Wikipedia Commons
http://www.flickr.com/photos/inventorchris2/7723117886/
Fine-Grained Parallelism: Vectorization

One core, multiple calculations

C. Holmes 2009
Wikipedia Commons
http://www.flickr.com/photos/inventorchris2/7723117886/
Fine-Grained Parallelism: Vectorization

Think tight, long inner loops with a few familiar array calculations:

/* C-style loop */

for ( int i=0; i<n; i++ )
    c[i] = a[i] + b[i];

------

c = a + b  !Fortran arrays
Vectorization: Intel Xeon E5 Host

Each core has two 256 bit (32 Byte) vector units
- One supports floating point addition, the other multiplication
- Each can produce up to 4 double precision results/cycle
- Together they can produce up to 8 DP (16 SP) results/cycle

array a: \( \cdots 13 \quad 14 \quad 15 \quad 16 \quad 17 \quad 18 \quad 19 \quad 20 \quad 21 \quad 22 \quad 23 \quad 24 \quad \cdots \)
array b: \( \cdots 13 \quad 14 \quad 15 \quad 16 \quad 17 \quad 18 \quad 19 \quad 20 \quad 21 \quad 22 \quad 23 \quad 24 \quad \cdots \)

\( b[i] = a[i] + b[i] \)

depicted indices assumes 0-based aligned arrays
Vectorization: Xeon Phi MIC

Each core has 512 bit (64 Byte) vector unit
- up to 8 double precision (16 single precision) results per cycle
- supports Fused Multiply-Add (FMA3), so 16 DP ops/cycle

\[ y[i] = a[i] \times x[i] + y[i] \]

depicted indices assumes 0-based aligned arrays
Vectorization: Issues

- Helping the compiler do its job
- Latency and locality
- Alignment

array $a$: $\cdots$ 13 14 15 16 17 18 19 20 21 22 23 24 $\cdots$

array $x$: $\cdots$ 13 14 15 16 17 18 19 20 21 22 23 24 $\cdots$

array $y$: $\cdots$ 13 14 15 16 17 18 19 20 21 22 23 24 $\cdots$

$$a[i] = a[i]*x[i] + y[i]$$

depicted indices assumes 0-based aligned arrays
Medium-Grained Parallelism

Your algorithm undoubtedly consists of a number of steps...

... compute gradients
solve for momentum
compute fluxes
solve for pressure correction
update pressure field
...
Medium-Grained Parallelism

... compute gradients
solve for momentum
compute fluxes
solve for pressure correction
update pressure field
...

Some of those steps include loops that may be candidates for parallel execution...

for each cell $i$
  findCritPoints($i$)
next $i$

i=1  i=2  i=3  i=4
Medium-Grained Parallelism

One approach: assign OpenMP threads to loop iterations (sometimes as simple as adding a one-line directive)...

... compute gradients
solve for momentum
compute fluxes
solve for pressure correction
update pressure field
...

for each cell i
   findCritPoints(i)
next i

thread 0
i=0

thread 1
i=1

thread 2
i=2

thread 3
i=3

TACC
TEXAS ADVANCED COMPUTING CENTER
Medium-Grained Parallelism

Can be a great way to pursue low risk, high impact incremental parallelism, one loop at a time...

\[
\begin{align*}
\text{compute gradients} \\
\text{solve for momentum} \\
\text{compute fluxes} \\
\text{solve for pressure correction} \\
\text{update pressure field} \\
\end{align*}
\]

... thread 0
\[
\begin{align*}
i=0 \\
\text{for each cell } i \\
\text{findCritPoints}( i ) \\
\text{next } i \\
\end{align*}
\]

thread 1
\[
\begin{align*}
i=1 \\
\text{for each cell } i \\
\text{findCritPoints}( i ) \\
\text{next } i \\
\end{align*}
\]

thread 2
\[
\begin{align*}
i=2 \\
\text{for each cell } i \\
\text{findCritPoints}( i ) \\
\text{next } i \\
\end{align*}
\]

thread 3
\[
\begin{align*}
i=3 \\
\text{for each cell } i \\
\text{findCritPoints}( i ) \\
\text{next } i \\
\end{align*}
\]
Multi-Level Parallelism

• There’s no one way to do this
• Does not necessarily require hybrid MPI-OpenMP
• But one common hybrid approach looks like this:
  – Coarse-grained parallelism across nodes via MPI (tasks manage blocks, slabs, sub-domains)
  – Medium-grained parallelism on the nodes via OpenMP (threads manage middle loops over mesh points, slices, elements, particles)
  – Vectorization for the tight inner loops within each thread
Linear Algebra Kernels

Off-the-shelf optimization and parallelism from mature, robust libraries; e.g.

– Intel Math Kernel Library (MKL) -- robust support for MIC
– PETSc – dense and sparse object-oriented solvers
– Lots of others (I'm probably skipping your favorite...)

TACC
Practical and Theoretical Performance Limits
Measuring Scalability
Measuring Scalability

Two basic metrics

Speedup on $p$ processors*:

$$S_p = \frac{\text{time on one processor}}{\text{time on } p \text{ processors}}$$

($S_p = p$ is perfect)

*here, "processor" (or "process") could mean many things depending on context: OpenMP thread, MPI task, core, node…
Measuring Scalability

Two basic metrics

Speedup on $p$ processors*:

$$S_p = \frac{\text{time on one processor}}{\text{time on } p \text{ processors}}$$

($S_p = p$ is perfect)

Efficiency on $p$ processors:

$$E_p = \frac{S_p}{p}$$

($E_p = 1$ is perfect)
Measuring Scalability

- Reality sets in
  - Expect to lag behind perfection
  - Expect a sweet spot beyond which adding processors will make things worse

- There are several reasons
  - Various kinds of overhead
  - Communication costs
  - Load imbalances
  - But one particular issue that deserves its own discussion...
How fast can we bake a cake?

• Assume (with no claim of realism)...
  • Two hours to prepare the cake for baking
  • Another half hour to bake the cake

• So total time is...
  Prep: 120 minutes
  Bake: 30 minutes
  Total: 150 minutes
How fast can we bake a cake?

• But what if we have two cooks?
  • Assume they work perfectly together
  • No wasted time, no overhead

• Then total time is...

  Prep: \( \frac{120}{2} = 60 \) minutes
  Bake: 30 minutes
  Total: 90 minutes
How fast can we bake a cake?

<table>
<thead>
<tr>
<th>cooks</th>
<th>prep*</th>
<th>bake</th>
<th>total</th>
<th>speedup</th>
<th>efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>120/1 = 120.0</td>
<td>30.0</td>
<td>150.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>120/2 = 60.0</td>
<td>30.0</td>
<td>90.0</td>
<td>1.7</td>
<td>0.8</td>
</tr>
</tbody>
</table>

**Total Time (mins)**

- 0
- 5
- 10
- 15
- 20

**Speedup**

- 0
- 1

**Efficiency**

- 0
- 1

---

**Graphs:**

- **Total Time (mins)**
- **Speedup**
- **Efficiency**
How fast can we bake a cake?

<table>
<thead>
<tr>
<th>cooks</th>
<th>prep*</th>
<th>bake</th>
<th>total</th>
<th>speedup</th>
<th>efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>120/1 = 120.0</td>
<td>30.0</td>
<td>150.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>120/2 = 60.0</td>
<td>30.0</td>
<td>90.0</td>
<td>1.7</td>
<td>0.8</td>
</tr>
<tr>
<td>3</td>
<td>120/3 = 40.0</td>
<td>30.0</td>
<td>70.0</td>
<td>2.1</td>
<td>0.7</td>
</tr>
<tr>
<td>4</td>
<td>120/4 = 30.0</td>
<td>30.0</td>
<td>60.0</td>
<td>2.5</td>
<td>0.6</td>
</tr>
<tr>
<td>5</td>
<td>120/5 = 24.0</td>
<td>30.0</td>
<td>54.0</td>
<td>2.8</td>
<td>0.6</td>
</tr>
<tr>
<td>19</td>
<td>120/19 = 6.3</td>
<td>30.0</td>
<td>36.3</td>
<td>4.1</td>
<td>0.2</td>
</tr>
<tr>
<td>20</td>
<td>120/20 = 6.0</td>
<td>30.0</td>
<td>36.0</td>
<td>4.2</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Total Time (mins)

Speedup

Efficiency
Amdahl's Law: Strong Scaling

- Back-of-napkin scalability bounds

\[ S_p = \frac{T_1}{T_p} \leq \frac{t_s + t_p}{t_s + \frac{t_p}{p}} \rightarrow \frac{1}{\alpha} \]
Amdahl's Law: Strong Scaling

- Back-of-napkin scalability bounds

\[ S_p = \frac{T_1}{T_p} \leq \frac{t_s + t_p}{t_s + \frac{t_p}{p}} \rightarrow \frac{1}{\alpha} \]

- Actual speedup
- Perfect on parallel section
- Reciprocal of serial fraction
- Infinitely fast on parallel section

![Graphs showing Speedup and Efficiency](image)

The graphs illustrate the speedup and efficiency for different serial fractions as a function of number of processors (procs).
Amdahl's Law: Strong Scaling

• Back-of-napkin scalability bounds

\[ S_p = \frac{T_1}{T_p} \leq \frac{t_s + t_p}{t_s + \frac{t_p}{p}} \rightarrow \frac{1}{\alpha} \]

• Under reasonable assumptions...
  – Serial fraction is a severe constraint
  – As you add processors, you reach a point of diminishing returns
  – Can't run faster than serial time
  – Speedup bounded by the reciprocal of the serial fraction
Gustafson's Law: Weak Scaling

- Increase problem size in proportion to processors
- How big a problem can we solve in the same time as the serial problem?
- Generalizing meaning of speedup...

\[
S_p \quad \text{bounded by} \quad \approx (1 - \alpha)p
\]
\[
E_p \quad \text{bounded by} \quad \approx 1 - \alpha
\]
Gustafson's Law: Weak Scaling

- Picture is a bit more optimistic
- Serial fraction is still a constraint
- But incremental benefit of additional processors can continue indefinitely (at least in theory)

\[ S_p \text{ bounded by } \approx (1 - \alpha)p \]
\[ E_p \text{ bounded by } \approx 1 - \alpha \]
Correctness and Independence

• Your program will happily execute your incorrect code
• The game is to identify independence and manage dependencies

Fanghong 2005
Wikipedia Commons
http://commons.wikimedia.org/wiki/
File:GreatWall2.jpg
Correctness and Independence*

for ( int i=0; i<n; i++ ) {
    a[i] = b[i] + c[i];
}

Independent and safe

Unsafe: order matters!

for ( int i=0; i<n; i++ ) {
    temp = a[i];
    a[i] = b[i];
    b[i] = temp;
}

Potentially unsafe as written: is temp shared by all iterations?

*These examples reflect issues typically encountered when using OpenMP, but similar MPI examples are not hard to generate.
Correctness and Repeatability

• Repeatability: the great debugging challenge
  – Code may work sequentially and fail in parallel
  – Behavior may vary from one run to another
  – Problems may occur only at large scale

• No magic bullet, but general advice
  – Avoid temptation to blame the environment
  – Learn to use parallel debugging tools
  – Test serial vs parallel regularly on small test probs
Tools and Languages

• Most parallel work done in Fortran/C/C++
• Some maturing capabilities in other languages
  – MPI wrappers
  – Threading support “under the hood”
• Profilers and parallel debuggers
Summary

Shared vs Distributed Memory: OpenMP vs MPI

Limits of Scalability: expect a sweet spot

Some Work Ahead: but it's worth the effort!
Further reading

- General page: http://www.tacc.utexas.edu/~eijkhout/istc/istc.html
- Direct download: http://tinyurl.com/EijkhoutHPC
License

© The University of Texas at Austin, 2013

This work is licensed under the Creative Commons Attribution Non-Commercial 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc/3.0/

When attributing this work, please use the following text: “Introduction to Parallel Computing”, Texas Advanced Computing Center, 2013. Available under a Creative Commons Attribution Non-Commercial 3.0 Unported License”