Optimization Techniques for Parallel Code

1. Parallel programming models

Sylvain Collange
Inria Rennes – Bretagne Atlantique

http://www.irisa.fr/alf/collange/
sylvain.collange@inria.fr

OPT - 2018
Goals of the course

- How to design parallel algorithms?
- How to implement them on parallel architectures (e.g. GPU)?
- How to get good performance?

- Will focus on NVIDIA CUDA environment
  - Similar concepts in OpenCL
- Non-goals : we will not cover
  - Graphics programming (OpenGL, Direct3D)
  - High-level languages that target GPUs (OpenACC, OpenMP 4…)
  - Parallel programming for multi-core and clusters (OpenMP, MPI…)
  - Programming for SIMD extensions (SSE, AVX…)

Why graphics processing unit (GPU)?

- Graphics rendering accelerator for computer games
  - Mass market: low unit price, amortized R&D
  - Increasing programmability and flexibility
- Inexpensive, high-performance parallel processor
  - GPUs are everywhere, from cell phones to supercomputers
GPUs in high-performance computing

- 2002: General-Purpose computation on GPU (GPGPU)
  - Scientists run non-graphics computations on GPUs
- 2010s: the fastest supercomputers use GPUs/accelerators

Today:
Heterogeneous multi-core processors influenced by GPUs

#1 Summit (USA)
4,608 \times (2 \text{ Power9 CPUs} + 6 \text{ Volta GPUs})

#2 Sunway TaihuLight (China)
40,960 \times \text{SW26010 (4 big + 256 small cores)}
Outline of the course

- Today: Introduction to parallel programming models
  - PRAM
  - BSP
  - Multi-BSP

- GPU / SIMD accelerator programming
  - The software side
  - Programming model

- Performance optimization
  - Memory access optimization
  - Compute optimization

- Advanced features
  - Warp-synchronous programming, thread groups
  - Unified memory
Introduction to parallel prog. models

Today's course

- **Goals:**
  - Be aware of the major parallel models and their characteristics
  - Write platform-independent parallel algorithms

- **Incomplete:** just an overview,
  not an actual programming model course
  - Just cover what we need for the rest of the course

- **Informal:** no formal definition, no proofs of theorems
  - Proofs by handwaving
Outline of the lecture

- Why programming models?
- PRAM
  - Model
  - Example
  - Brent's theorem
- BSP
  - Motivation
  - Description
  - Example
- Multi-BSP
- Algorithm example: parallel prefix
Why programming models?

- Abstract away the complexity of the machine
- Abstract away implementation differences (languages, architectures, runtime...)

- Focus on the algorithmic choices
  - First design the algorithm, then implement it
- Define the asymptotic algorithmic complexity of algorithms
  - Compare algorithms independently of the implementation / language / computer / operating system...
Serial model: RAM/Von Neumann/Turing

- Abstract machine

- One unlimited memory, constant access time
- Basis for classical algorithmic complexity theory
  - Complexity = number of instructions executed as a function of problem size
Serial algorithms

• Can you cite examples of serial algorithms and their complexity?
Serial algorithms

- Can you cite examples of serial algorithms and their complexity?
  - Binary search: $O(\log n)$
  - Quick sort: $O(n \log n)$
  - Factorization of n-digit number: $O(e^n)$
  - Multiplication of n-digit numbers:
    $O(n \log n) < O(n \log n \ 8^{\log^* n}) < O(n \log n \ 2^{O(\log^* n)})$

- Still an active area of research!
- But for parallel machines, we need parallel models
Outline of the lecture

- Why programming models?
- PRAM
  - Model
  - Example
  - Brent's theorem
- BSP
  - Motivation
  - Description
  - Example
- Multi-BSP
- Algorithm example: parallel prefix
Parallel model: PRAM

- 1 memory, constant access time
- N processors
- Synchronous steps
  - All processors run an instruction simultaneously
  - Usually the same instruction
- Each processors knows its own number

PRAM example: SAXPY

- Given vectors X and Y of size n, compute vector $R \leftarrow a \times X + Y$ with n processors

Processor $P_i$:
- $x_i \leftarrow X[i]$
- $y_i \leftarrow Y[i]$
- $r_i \leftarrow a \times x_i + y_i$
- $R[i] \leftarrow r_i$
**PRAM example: SAXPY**

Given vectors $X$ and $Y$ of size $n$, compute vector $R ← a×X+Y$ with $n$ processors

Processor $Pi$:

- $x_i ← X[i]$
- $y_i ← Y[i]$
- $r_i ← a × x_i + y_i$
- $R[i] ← r_i$
PRAM example

- Given vectors X and Y of size n, compute vector $R \leftarrow a \times X + Y$ with n processors

Processor $P_i$:
- $x_i \leftarrow X[i]$
- $y_i \leftarrow Y[i]$
- $r_i \leftarrow a \times x_i + y_i$
- $R[i] \leftarrow r_i$
PRAM example: SAXPY

- Given vectors X and Y of size n, compute vector $R \leftarrow a \times X + Y$ with n processors

Processor $P_i$:
- $x_i \leftarrow X[i]$
- $y_i \leftarrow Y[i]$
- $r_i \leftarrow a \times x_i + y_i$
- $R[i] \leftarrow r_i$

- Complexity: $O(1)$ with n processors
  $O(n)$ with 1 processor
Access conflicts: PRAM machine types

- Can multiple processors read the same memory location?
  - **Exclusive** Read: any 2 processors have to read different locations
  - **Concurrent** Read: 2 processors can access the same location

- Can multiple processors write the same location?
  - **Exclusive** Write: any 2 processors have to write different locations
  - **Concurrent** Write: Allow reduction operators: And, Or, Sum...

- Types of PRAM machines: EREW, CREW, CRCW...

- We assume CREW in this lecture
Reduction example

- Given a vector \( a \) of size \( n \), compute the sum of elements with \( n/2 \) processors

- **Sequential algorithm**

  ```cpp
  for i = 0 to n-1
    sum ← sum + a[i]
  ```

  - Complexity: \( O(n) \) with 1 processor

- All operations are dependent: no parallelism
  - Need to use associativity to reorder the sum
Parallel reduction: dependency graph

We compute \( r = ((a_0 \oplus a_1) \oplus (a_2 \oplus a_3)) \oplus \ldots \oplus a_{n-1}) \ldots \)
Parallel reduction: PRAM algorithm

- First step

Processor $P_i$:
$$a[i] = a[2i] + a[2i+1]$$
Parallel reduction: PRAM algorithm

- Second step
  - Reduction on n/2 processors
  - Other processors stay idle

Processor Pi:

\[
n' = \frac{n}{2}
\]

\[
\text{if } i < n' \text{ then}
\]

\[
a[i] = a[2i] + a[2i+1]
\]

\[
\text{end if}
\]
Parallel reduction: PRAM algorithm

- Complete loop

Processor Pi:
while n > 1 do
  if i < n then
    a[i] = a[2*i] + a[2*i+1]
  end if
  n = n / 2;
end while

- Complexity: O(log(n)) with O(n) processors
Less than n processors

- A computer that grows with problem size is not very realistic
- Assume \( p \) processors for \( n \) elements, \( p < n \)
- Back to \( R \leftarrow a \times X + Y \) example
  - Solution 1: group by blocks
    assign ranges to each processor

Processor \( P_i \):
  for \( j = i \times \lceil \frac{n}{p} \rceil \) to \( (i+1) \times \lceil \frac{n}{p} \rceil - 1 \)
  \( x_j \leftarrow X[j] \)
  \( y_j \leftarrow Y[j] \)
  \( r_j \leftarrow a \times x_j + y_j \)
  \( R[j] \leftarrow r_j \)
Less than n processors

- A computer that grows with problem size is not very realistic
- Assume $p$ processors for $n$ elements, $p < n$
- Back to $R \leftarrow a \times X + Y$ example
  - Solution 1: group by blocks
    assign ranges to each processor

Processor $P_i$:
for $j = i \times [n/p]$ to $(i+1) \times [n/p] - 1$
  $x_j \leftarrow X[j]$
  $y_j \leftarrow Y[j]$
  $r_j \leftarrow a \times x_j + y_j$
  $R[j] \leftarrow r_j$
Solution 2: slice and interleave

Processor Pi:
for j = i to n step p
    xj ← X[j]
    yj ← Y[j]
    rj ← a × xj + yj
    R[j] ← rj
Solution 2: slice and interleave

Processor $P_i$:

for $j = i$ to $n$ step $p$

$x_j \leftarrow X[j]$

$y_j \leftarrow Y[j]$

$r_j \leftarrow a \times x_j + y_j$

$R[j] \leftarrow r_j$

• Equivalent in the PRAM model
  • But not in actual programming!
Brent's theorem

- Generalization for any PRAM algorithm of complexity $C(n)$ with $n$ processors
- Derive an algorithm for $n$ elements, with $p$ processors only
- Idea: each processor emulates $n/p$ virtual processors
  - Each emulated step takes $n/p$ actual steps
- Complexity: $O(n/p \times C(n))$
Interpretation of Brent's theorem

- We can design algorithms for a variable (unlimited) number of processors
  - They can run on any machine with fewer processors with known complexity
  - There may exist a more efficient algorithm with \( p \) processors only
PRAM: wrap-up

- Unlimited parallelism
- Unlimited shared memory
- Only count operations: zero communication cost

Pros

- Simple model: high-level abstraction
  - can model many processors or even hardware

Cons

- Simple model: far from implementation

Outline of the lecture

- Why programming models?
- PRAM
  - Model
  - Example
  - Brent’s theorem
- BSP
  - Motivation
  - Description
  - Example
- Multi-BSP
- Algorithm example: parallel prefix
PRAM limitations

- PRAM model proposed in 1978
  - Inspired by SIMD machines of the time

- Assumptions
  - All processors synchronized every instruction
  - Negligible communication latency

- Useful as a theoretical model, but far from modern computers
Modern architectures

- Modern supercomputers are clusters of computers
  - Global synchronization costs millions of cycles
  - Memory is distributed
- Inside each node
  - Multi-core CPUs, GPUs
  - Non-uniform memory access (NUMA) memory
- **Synchronization** cost at all levels

Mare Nostrum, a modern distributed memory machine
**Bulk-Synchronous Parallel (BSP) model**

- Assumes distributed memory
  - But also works with shared memory
  - Good fit for GPUs too, with a few adaptations
- Processors execute instructions independently
- Communications between processors are **explicit**
- Processors need to **synchronize** with each other

---

Superstep

- A program is a sequence of supersteps
- Superstep: each processor
  - Computes
  - Sends result
  - Receive data
- Barrier: wait until all processors have finished their superstep
- Next superstep: can use data received in previous step
Example: reduction in BSP

- Start from dependency graph again

```
\begin{align*}
    & a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 \\
    & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
    & \oplus & \oplus & \oplus & \oplus \\
    & \downarrow & \downarrow & \downarrow & \downarrow \\
    & \oplus & \oplus \\
    & \downarrow & \downarrow \\
    & \oplus \\
    & \downarrow \\
    & r
\end{align*}
```
Reduction: BSP

- Adding barriers

\[ a_0 \oplus a_1 \oplus a_2 \oplus a_3 \quad \quad a_i \leftarrow a_{2i} + a_{2i+1} \]

\[ \oplus \quad \oplus \quad \oplus \quad \oplus \quad \quad a_i \leftarrow a_{2i} + a_{2i+1} \]

\[ \oplus \quad \oplus \quad \quad a_i \leftarrow a_{2i} + a_{2i+1} \]

\[ \oplus \quad \quad a_i \leftarrow a_{2i} + a_{2i+1} \]

Communication

Barrier
Reducing communication

- Data placement matters in BSP
- Optimization: keep left-side operand local
Wrapup

- BSP takes synchronization cost into account
- Explicit communications
- Homogeneous communication cost: does not depend on processors
Outline of the lecture

- Why programming models?
- PRAM
  - Model
  - Example
  - Brent's theorem
- BSP
  - Motivation
  - Description
  - Example
- Multi-BSP
- Algorithm examples
The world is not flat
It is hierarchical!
Multi-BSP model

- Multi-BSP: BSP generalization with groups of processors in multiple nested levels

- Higher level: more expensive synchronization
- Arbitrary number of levels

Multi-BSP and multi-core

- Minimize communication cost on hierarchical platforms
  - Make parallel program hierarchical too
  - Take thread *affinity* into account

- On clusters (MPI): add more levels **up**
- On GPUs (CUDA): add more levels **down**
Reduction: multi-BSP

- Break into 2 levels
Recap

- PRAM
  - Single shared memory
  - Many processors in lockstep

- BSP
  - Distributed memory, message passing
  - Synchronization with barriers

- Multi-BSP
  - BSP with multiple scales
Outline of the lecture

- Why programming models?
- PRAM
  - Model
  - Example
  - Brent's theorem
- BSP
  - Motivation
  - Description
  - Example
- Multi-BSP
- Algorithm examples
  - Parallel prefix
  - Stencil
Work efficiency

- Work: number of operations
- Work efficiency: ratio between work of sequential algorithm and work of parallel algorithm
- Example: reduction
  - Sequential sum: n-1 operations
  - Parallel reduction: n-1 operations
  - Work efficiency: 1
    This is the best case!
Parallel prefix

- Problem: on a given vector \( a \), for each element, compute the sum \( s_i \) of all preceding elements

\[
s_i = \sum_{k=0}^{i} a_k
\]

- Result: vector \( s = (s_0, s_1, \ldots, s_{n-1}) \)

- "Sum" can be any associative operator

- Applications: stream compaction, graph algorithms, addition in hardware…

Mark Harris, Shubhabrata Sengupta, and John D. Owens. "Parallel prefix sum (scan) with CUDA." GPU gems 3, 2007
Sequential algorithm

\[
\text{sum} \leftarrow 0 \\
\text{for } i = 0 \text{ to } n \text{ do} \\
\hspace{1em} s[i] \leftarrow \text{sum} \\
\hspace{1em} \text{sum} \leftarrow \text{sum} + a[i] \\
\text{end for}
\]

Can we parallelize it?
Sequential prefix: graph

- **Sequential algorithm**

  \[
  \text{sum} \leftarrow 0 \\
  \text{for } i = 0 \text{ to } n \text{ do} \\
  \quad \text{s}[i] \leftarrow \text{sum} \\
  \quad \text{sum} \leftarrow \text{sum} + a[i] \\
  \text{end for}
  \]

- **Can we parallelize it?**
  - Yes
  - Add more computations to reduce the depth of the graph
Parallel prefix

- Optimize for graph depth
  - Idea:
    Make a reduction tree for each output

- Reuse common intermediate results
A simple parallel prefix algorithm

Also known as Kogge-Stone in hardware

\[
\begin{array}{cccccccc}
P0 & P1 & P2 & P3 & P4 & P5 & P6 & P7 \\
a0 & a1 & a2 & a3 & a4 & a5 & a6 & a7 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{Step 0} & \oplus & \oplus & \oplus & \oplus & \oplus & \oplus & \oplus \\
a0 & \sum_{0-1} & \sum_{1-2} & \sum_{2-3} & \sum_{3-4} & \sum_{4-5} & \sum_{5-6} & \sum_{6-7} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{Step 1} & \oplus & \oplus & \oplus & \oplus & \oplus & \oplus & \oplus \\
a0 & \sum_{0-1} & \sum_{0-2} & \sum_{1-3} & \sum_{2-4} & \sum_{3-5} & \sum_{4-6} & \sum_{5-7} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{Step 2} & \oplus & \oplus & \oplus & \oplus & \oplus & \oplus & \oplus \\
a0 & \sum_{0-1} & \sum_{0-2} & \sum_{0-3} & \sum_{0-4} & \sum_{0-5} & \sum_{0-6} & \sum_{0-7} \\
\end{array}
\]

\[
\Sigma_{i-j} \text{ is the sum } \sum_{k=i}^{j} a_k
\]

\[
\begin{align*}
\text{Step } d & : \text{ if } i \geq 2^d \text{ then } \\
& \quad s[i] \leftarrow s[i-2^d] + s[i]
\end{align*}
\]

Analysis

- Complexity
  - $\log_2(n)$ steps with $n$ processors

- Work efficiency
  - $O(n \log_2(n))$ operations
  - $1/\log_2(n)$ work efficiency

- All communications are global
  - Bad fit for Multi-BSP model
Another parallel prefix algorithm

- Two passes
- First, upsweep: parallel reduction
  - Compute partial sums
  - Keep “intermediate” (non-root) results

Second pass: downsweep

- Set root to zero
- Go down the tree
  - Left child: propagate parent
  - Right child: compute parent + left child

```
for d = log₂(n)-1 to 0 step -1
  if i mod 2^{d+1} = 2^{d+1}-1 then
    x ← s[i]
    s[i] ← s[i-2^d] + x
    s[i-2^d] ← x
  end if
end for
```

In parallel

Hopefully nobody will notice the off-by-one difference...
Analysis

- Complexity
  - \(2 \log_2(n)\) steps: \(O(\log(n))\)

- Work efficiency
  - Performs \(2n\) operations
  - Work efficiency = 1/2: \(O(1)\)

- Communications are local away from the root
  - Efficient algorithm in Multi-BSP model
Another variation: Brent-Kung

- Same reduction in upsweep phase, simpler computation in downsweep phase
- Many different topologies: Kogge-Stone, Ladner-Fischer, Han-Carlson...
  - Different tradeoffs, mostly relevant for hardware implementation

```plaintext
if i mod 2^{d+1} = 2^{d+1} - 2^d then
  s[i] ← s[i-2^d] + s[i]
end if
```
Outline of the lecture

- Why programming models?
- PRAM
  - Model
  - Example
  - Brent's theorem
- BSP
  - Motivation
  - Description
  - Example
- Multi-BSP
- Algorithm examples
  - Parallel prefix
  - Stencil
Given a n-D grid, compute each element as a function of its neighbors

- All updates done in parallel: no dependence
- Commonly, repeat the stencil operation in a loop
- Applications: image processing, fluid flow simulation…
Stencil: naïve parallel algorithm

- 1-D stencil of size \(n\), nearest neighbors
- \(m\) successive steps

Processor \(i\):
for \(t = 0\) to \(m\)
    \[s[i] = f(s[i-1], s[i], s[i+1])\]
end for

Needs global synchronization at each stage

Can you do better?
Ghost zone optimization

- Idea: trade more computation for less global communication
- Affect tiles to groups of processors with overlap

### Processors

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>P8</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0</td>
<td>s1</td>
<td>s2</td>
<td>s3</td>
<td>s4</td>
<td>s5</td>
<td>s6</td>
<td>s7</td>
<td>s8</td>
</tr>
</tbody>
</table>

### Data

<table>
<thead>
<tr>
<th>P9</th>
<th>P10</th>
<th>P11</th>
<th>P12</th>
<th>P13</th>
<th>P14</th>
<th>P15</th>
<th>P16</th>
<th>P17</th>
</tr>
</thead>
<tbody>
<tr>
<td>s5</td>
<td>s6</td>
<td>s7</td>
<td>s8</td>
<td>s9</td>
<td>s10</td>
<td>s11</td>
<td>s12</td>
<td>s13</td>
</tr>
</tbody>
</table>

Overlap: Ghost zone

Memory for group P0-P8

Memory for group P9-P17
Ghost zone optimization

- Initialize all elements including ghost zones
- Perform local stencil computations
- Global synchronization only needed periodically
Wrapup

- Computation is cheap
  - Focus on communication and global memory access
  - Between communication/synchronization steps, this is familiar sequential programming

- Local communication is cheap
  - Focus on global communication

- Next time: CUDA programming