Lecture 7:
Performance Optimization Part II:
Locality, Communication, and Contention
Paul Thorn
Help Me Out, Hook Me Up
(Ain’t Love Strange)

- Paul Thorn, after starting too late on Assignment 2
Today: more parallel program optimization

- Recall last lecture: distributing work to processors
  - Goal: achieving good workload balance while also minimizing overhead
  - Discussed static vs. dynamic assignment
  - Tip: keep it simple (implement, analyze, then tune/optimize if required)

- Today: minimizing communication and exploiting locality
Terminology

Latency
The amount of time needed for an operation to complete.
Example: A memory load that misses the cache has a latency of 200 cycles.
A packet takes 20 ms to be sent from my computer to Google.

Bandwidth
The rate at which operations are performed.
Example: Memory can provide data to the processor at 25 GB/sec.
A communication link can send 10 million messages per second.

“Cost”
The effect operations have on program execution time
(or some other metric, e.g., power...)
“My slow program sends most of its time waiting on memory.” (cost of latency)
“saxpy achieves low ALU utilization because it is bandwidth bound.” (cost of insufficient bandwidth)
But I’m going to start with the idea of: Pipelining
Doing your laundry

Operation: do your laundry

1. Wash clothes
2. Dry clothes
3. Fold clothes

Latency of completing 1 load of laundry = 2 hours
Increasing laundry throughput
Goal: maximize throughput of many loads of laundry

On approach: duplicate execution resources:
use two washers, two dryers, and call a friend

Latency of completing 2 loads of laundry = 2 hours
Throughput increases by 2x: 1 load/hour
Resources increased by 2x: two washers, two dryer
Pipelining

Goal: maximize throughput of many loads of laundry

Latency: 1 load takes 2 hours
Throughput: 1 load/hour
Resources: one washer, one dryer
Another example: an instruction pipeline

Break instruction execution down into many steps
Key to scaling processor clock frequency (each clock, a simple short operation is done by each unit)

Latency: 1 instruction takes 4 cycles
Throughput: 1 instruction per cycle
(Important: special care must be taken to ensure correctness in case of dependent instructions)

Modern Intel Core i7 pipeline is variable length (it depends on the instruction) ~15-20 stages
A very simple model of communication

\[ T(n) = T_0 + \frac{n}{B} \]

- \( T(n) \) = transfer time (overall latency of the operation)
- \( T_0 \) = start-up latency (e.g., time until first bit arrives)
- \( n \) = bytes transferred
- \( B \) = transfer rate (bandwidth of the link)

Assumption: processor does no other work while waiting for transfer to complete ...

Effective bandwidth = \( \frac{n}{T(n)} \)

Effective bandwidth depends on transfer size
A more general model of communication

Communication time = overhead + occupancy + network delay

Example: sending a message

- Send API call, copy message to network buffer
- Routing: determine address of destination
- Send data over slow link 1: $T_0 + n/B_{\text{small}}$
- Send data over fast link 2: $T_0 + n/B_{\text{large}}$
- Copy message to receiver's network buffer

- Overhead (time spent on the communication by a processor)
- Occupancy (time for data to pass through slowest component of system)
- Network delay (everything else)
Pipelined communication

- Overhead: time spent on communication by a processor
- Occupancy: time for data to pass through the slowest component of the system
- Network delay: everything else

Occupancy determines communication rate (effective bandwidth)

- Messages buffered while link is busy
- Buffer size = 2

Legend:
- Orange: Overhead (time spent on the communication by a processor)
- Blue: Occupancy (time for data to pass through the slowest component of the system)
- Gray: Network delay (everything else)
**Pipelined communication**

Occupancy determines communication rate
(in steady state: \(\text{msg/sec} = 1/\text{occupancy}\))

Processor sends burst of messages (faster than \(1/\text{occupancy}\))

Max buffer size = 2
Communication cost

Total communication cost = frequency $\times$ (communication time - overlap)

Overlap: portion of communication performed concurrently with other work
“Other work” can be computation or other communication (as in the previous example)

Remember, what really matters is not the absolute cost of communication, but its cost relative to the cost of the computation fundamental to the problem being solved.
Inherent communication

Communication that must occur in a parallel algorithm. Fundamental to the algorithm.
Reducing inherent communication

- Good assignment can reduce inherent communication (decrease a program’s communication-to-computation ratio)

### 1D blocked assignment

- Elements communicated $\approx 2PN$
- Elements computed $\approx N^2$
- $\frac{\text{elements communicated}}{\text{elements computed}} \propto \frac{P}{N}$

### 1D interleaved assignment

- Elements communicated $\approx 2P$
- Elements computed $= 2$

---

CMU 15-418, Spring 2014
Reducing inherent communication

2D blocked assignment

\[ \frac{N^2}{P} \]

\[ \frac{N}{\sqrt{P}} \]

\[ \frac{\sqrt{P}}{N} \]

Asymptotically better communication scaling than 1D blocked assignment
Communication costs increase sub-linearly with \( P \)
Assignment captures 2D locality of algorithm
Communication-to-computation ratio

\[
\frac{\text{amount of communication}}{\text{amount of computation}}
\]

- If denominator is execution time of computation, ratio gives average bandwidth requirements.

- “Arithmetic intensity” = 1 / communication-to-computation ratio
  - I personally find arithmetic intensity a more intuitive quantity, since higher is better.

- High arithmetic intensity is needed on modern parallel processors since the ratio of compute capability to available bandwidth is very high (recall saxpy)
Parallel system as an extended memory hierarchy

- Up until now: I’ve described data “partitioned among processors”, and that “non-local data” required communication.
- In reality, parallel system is multi-memory, multi-cache system. Characteristics of data access latencies and bandwidths can have large effect on performance.

View from one processor

- Accesses not satisfied in local memory cause communication with next level.
- Managing locality is important at all levels (achieving high arithmetic intensity).

Lower latency, higher bandwidth, smaller capacity

Higher latency, lower bandwidth, larger capacity
Artifactual communication

- Inherent communication: assumes unlimited capacity, small transfers, perfect knowledge of what is needed to communicate

- Artifactual communication is everything else (depends on interaction of application and system)
  - System might have a minimum granularity of transfer (result: system must send more data than what is needed: e.g., program reads one word but entire cache line must be loaded from memory)
  - Poor allocation of data among distributed memories (data doesn’t reside near processor that accesses it most)
  - Finite replication capacity (same data communicated to processor multiple times because tier of memory hierarchy is too small to have kept it)
Review of the three (now four) Cs

- **Cold miss**
  First time data touched. Unavoidable in a sequential program.

- **Capacity miss**
  Working set larger than cache. Can be decreased by larger caches.

- **Conflict miss**
  Miss induced by cache management policy. Can reduce by changing cache associativity, or data access pattern in application.

- **Communication miss (new)**
  Due to inherent or artifactual communication in parallel system.
Communication: working set perspective

This diagram holds true at any level of the memory hierarchy in a parallel system.

Question: how much capacity should an architect build for this workload?
Reducing amount of communication
Improve temporal locality

- “Blocking”: reorder computation to make working sets map well to system’s memory hierarchy

Consider order in which elements are updated in solver example from previous class(es).

Common technique in sequential programs (recall matrix transpose assignment in 15-213)

Main idea: replicate block of data from in local memories (cache)

Process block of data in its entirety (accessing it many times) prior to moving only next block (goal: reduce capacity misses)
Improve temporal locality

- Exploit sharing: co-locate tasks that operate on the same data
  - Schedule threads working on the same data structure at the same time on the same processor
  - Reduces inherent communication

- Example: CUDA thread block
  - Abstraction to localize related processing in the machine
  - Threads in block often cooperate to perform an operation
  - Leverage fast access to / synchronization via CUDA shared memory
Exploiting spatial locality

- Granularities can be very important
  - Granularity of allocation
  - Granularity of communication / data transfer
  - Granularity of coherence (future lecture)
Artifactual communication due to comm. granularity

Shared memory system, cache line communication granularity
(assume line contains four elements)

2D blocked partitioning of data

- Good spatial locality for non-local accesses to top-bottom rows
- Poor spatial locality for non-local accesses to left-right columns

Need one element from line. Must communicate four.

Implication: 1D blocked layout may perform better despite higher inherent communication-to-computation ratio.

○ = required elements assigned to other processors
Artifactual communication
(due to communication/coherence granularity)

Data partitioned among memories local to processors

Processors access their assigned elements (no inherent communication)

But data access on real machine triggers communication (artifactual)

Also, writes by different processors require cache coherence **

** Implementing cache coherence will be a topic of future lectures.
Reducing artifactual comm: better layout

Memory layout: arrows designate contiguous addresses

- Memory page straddles partition boundary
- Cache line straddles partition boundary
- Page contained within partition
- Cache line within partition

2D, row-major array layout

4D array layout (block-major)
Structuring communication to reduce cost

Total communication cost = frequency \times (communication time - overlap)

Total communication cost = frequency \times (overhead + occupancy + network delay - overlap)

Total communication cost = frequency \times (overhead + (n/B + contention) + network delay - overlap)

Occupancy primarily consists of the time it takes to transfer a message (n bytes) over slowest link + delays due to contention for link
Demo: contention
Contestion

- All resources have non-zero occupancy
  - Memory, communication links, comm. controller, etc.
  - Each has fixed number of transactions per unit time
- Contention occurs when many requests to a resource are made within a small window of time
  - Resource is a “hot spot”

Example: updating a shared variable

Flat communication: potential for high contention (but low latency if no contention)

Tree structured communication: reduces contention (but higher latency under no contention)
NVIDIA GTX 480 contention example

- Shared memory implementation
  - 48 KB on-chip indexable storage, divided into 32 banks
  - Address X is stored in bank b, where b = X % 32
  - Each bank can be accessed in parallel (one word per bank per clock)
  - Special broadcast mode if all addresses are the same (example 1)

- Figure shows addresses requested from each bank as a result of shared memory load instruction from 32 threads in a WARP

```c
__shared__ float A[512];
int index = threadIdx.x;
float x1 = A[0]; // single cycle
float x2 = A[index]; // single cycle
float x3 = A[3*index]; // single cycle
float x4 = A[16 * index]; // 16 cycles
```
Another contention example

- Problem: place 100K point particles in a 16-cell uniform grid
  - Parallel data structure manipulation problem: build a 2D array of lists
- Recall: 15 cores, up to 1024 threads per core on GTX 480 GPU (and only 16 lists)

<table>
<thead>
<tr>
<th>Cell id</th>
<th>Count</th>
<th>Particle id</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3, 5</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>1, 2, 4</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Solution 1: parallelize over cells

- One answer: partition work by cells: for each cell, independently compute contained particles (no synchronization required)
  - 16 parallel tasks (insufficient parallelism: need thousands of independent tasks for GPU)
  - Also: performs 16 times more particle-in-cell computations than sequential algorithm (15 parallel units, but 16 times more work! Ug!)

```c
list cell_lists[16]; // 2D array of lists

for each cell c // in parallel
    for each particle p // sequentially
        if (p is within c)
            append p to cell_lists[c]
```
Solution 2: parallelize over particles

  - Massive contention: thousands of threads contending for access to update data structure

```c
list cell_list[16]; // 2D array of lists
lock cell_list_lock;

for each particle p // in parallel
    c = compute cell containing p
    lock(cell_list_lock)
    append p to cell_list[c]
    unlock(cell_list_lock)
```
Solution 3: use finer-granularity locks

- Alleviate contention by using per-cell locks
  - Assuming uniform distribution of particles... ~16x less contention

```c
list cell_list[16]; // 2D array of lists
lock cell_list_lock[16];

for each particle p // in parallel
  c = compute cell containing p
  lock(cell_list_lock[c])
  append p to cell_list[c]
  unlock(cell_list_lock[c])
```
Solution 4: compute partial results + merge

Yet another answer: generate N “partial” grids in parallel, then combine

- Example: create 15 grids on GTX 480 (one per core)
- All threads in thread block update same grid
  - Faster synchronization: contention reduced by factor of N and synchronization performed on local variables
- Extra work: merging the grids at the end of the computation
- Extra memory footprint: Store 15 grids, rather than 1
**Solution 5: data-parallel approach**

**Step 1:** compute cell containing each particle

<table>
<thead>
<tr>
<th>Array Index: 0 1 2 3 4 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>result:</td>
</tr>
<tr>
<td>9 6 6 4 6 4</td>
</tr>
</tbody>
</table>

**Step 2:** sort results by cell

<table>
<thead>
<tr>
<th>Array Index: 3 5 1 2 4 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>result:</td>
</tr>
<tr>
<td>4 4 6 6 6 9</td>
</tr>
</tbody>
</table>

**Step 3:** find start/end of each cell

```cpp
cell = result[index]
if (index == 0 || cell != result[index-1]) {
    cell_starts[cell] = index;
    if (index > 0) // special case for first cell
        cell_ends[result[index-1]] = index;
}
if (index == numParticles-1) // special case for last cell
    cell_ends[cell] = index+1;
```

**Removes need for fine-grained synchronization... at cost of sort and extra passes over the data (extra BW)**

<table>
<thead>
<tr>
<th>cell_starts</th>
<th>0xff</th>
<th>0xff</th>
<th>0xff</th>
<th>0xff</th>
<th>0</th>
<th>0xff</th>
<th>2</th>
<th>0xff</th>
<th>0xff</th>
<th>5</th>
<th>0xff</th>
<th>⋮</th>
</tr>
</thead>
<tbody>
<tr>
<td>cell_ends</td>
<td>0xff</td>
<td>0xff</td>
<td>0xff</td>
<td>0xff</td>
<td>2</td>
<td>0xff</td>
<td>5</td>
<td>0xff</td>
<td>0xff</td>
<td>6</td>
<td>0xff</td>
<td>⋮</td>
</tr>
</tbody>
</table>

This code is run for each element of ‘result’
Common use: N-body problems

- A common operation is to compute interactions with neighboring particles
- Example: find all particles within radius R
  - Create grid with cells of size R
  - Only need to inspect particles in surrounding grid cells
Reducing communication costs

- **Reduce overhead to sender/receiver**
  - Send fewer messages, make messages larger
  - Coalesce many small messages into large ones

- **Reduce delay**
  - HW implementor: improve communication architecture
  - Application writer: restructure code to exploit locality

- **Reduce contention**
  - Stagger access to contended resources
  - Replicate contended resources (local copies, fine-grained locks)

- **Increase overlap**
  - HW implementation: pipelining, multi-threading, pre-fetching, out-of-order exec
  - Application: asynchronous communication (e.g., async messages)
  - Requires additional concurrency in application (more concurrency than number of processors)
Summary: optimizing communication

- Inherent vs. artifactual communication
  - Artifactual communication depends on the machine
  - Often as important to performance as inherent communication

- Improving program performance:
  - Identify and exploit locality: communicate less
    - Increase arithmetic intensity
  - Reduce overhead (few, large messages)
  - Reduce contention
  - Maximize overlap (hide latency so as to not incur cost)