Introduction to Performance Optimization

the slides are kindly provided by Tim Mattson

Other than the Intel lab’s research processors. Die photos from UC Berkeley CS194 lecture notes

Third party names are the property of their owners
less than 0.9% of programmers care about the performance of the code

9% of Programmers care about the cleanliness of the code

90% of Programmers care about the correctness of the code
Outline

• Performance issues in computing
• Key concepts in parallel programming
Suppose $A$, $B$, and $C$ are $n \times n$ matrices.

What is the Big-Oh complexity or the number of scalar multiply-adds performed during the matrix-multiplication $C = A \times B$?

- We perform $n \times n$ dot products.
- Each dot product is between two length-$n$ vectors, and performs 1 multiply-add per element.

Who knows the answer?

$$(2n-1)n^2 \approx 2n^3 = O(n^3)$$
So - How long does $C = A \times B$ take?

Note: Both $x$ and $y$ axes are log-scale.
So - How long does $C = A \times B$ take?

Note: Both x and y axes are log-scale

131x using a single core
So - How long does $C = A \times B$ take?

Note: Both x and y axes are log-scale

3.2x using 4 4-SIMD CPU cores
So - How long does $C = A \times B$ take?

5.3x using 30 8-SIMD GPU cores

Note: Both x and y axes are log-scale
So - How long does C=A*B take?

2,300x Overall

Note: Both x and y axes are log-scale
So - How long does $C = A \times B$ take?

Note: Both x and y axes are log-scale
A typical microprocessor memory hierarchy

- Instruction cache and data cache pull data from a unified cache that maps onto RAM.
- TLB implements virtual memory and brings in pages to support large memory footprints.
Cache Blocking

\[ C_{i,j} = A_i \times B^j \]
\[ A_i \times B^j = \sum_k A_{i,k} \times B^{k,j} \]

- Break up the Large Matrix-Multiply into sums of smaller Matrix-Multiplies.
- Make sure that \( A_{i,k} \), \( B_{k,j} \), and \( C_{i,j} \) all fit into the cache: Blocks are reused and cache misses are avoided.

- Can block for multiple levels of cache / Memory Hierarchy:
  - Registers, L1, L2, L3, TLB, DRAM, Disk
Do you need to worry about the TLB?

Transpose: 2 threads on a Dual Proc Xeon

Ignore TLB issues (no tiling)

Tiled to optimize use of TLB

Source: M Frumkin, R. van de Wijngaart, T. G. Mattson, Intel
Blocking for TLB

- TLB caches the mapping from virtual addresses to physical addresses
  - If the entry is in the page table, the overhead of reading the entry and computed the physical address is needed
  - If the entry is not in the page table, a page fault exception will be raised, and OS needs to handle that, so the overhead is huge

```c
void Transpose(int** matrix, int N)
for i = 1 to N
  for j = 1 to N
    tmp = matrix[i][j];
    matrix[i][j] = matrix[j][i];
    matrix[j][i] = tmp;
```
NUMA* issues on a Multicore Machine

2-socket Clovertown Dell PE1950

A single quad-core chip is a NUMA machine!

2 threads, 2 cores, sharing a cache

2 threads, 2 cores, 1 socket, no shared cache

2 threads, 2 cores, 2 sockets

*NUMA == Non Uniform Memory architecture … memory is shared but access times vary.

Third party names are the property of their owners.
Example: Loop Unrolling

- Pack more code into a loop body, and expose more potential Instruction-parallelism between branches

```c
for (; l < k; l++)
    c[i + m*j] += a[i + l*m] * b[1 + j*k];
```
- Computes 2 indices, 2 memory accesses, and 1 mul-add for every branch instruction.

```c
for (; l+3 < k; l += 4){
    c[i + m*j] += a[i + l*m] * b[1 + j*k];
    c[i + m*j] += a[i + (l+1)*m] * b[1+1 + j*k];
    c[i + m*j] += a[i + (l+2)*m] * b[1+2 + j*k];
    c[i + m*j] += a[i + (l+3)*m] * b[1+3 + j*k];
}
```
- Computes 8 indices, 8 memory accesses, and 4 mul-adds for every branch instruction.
- Compilers can perform many of this type of transformation automatically, but frequently do so sub-optimally.
Outline

- Performance issues in computing
- Key concepts in parallel programming
Talking about performance

- **Speedup**: the increased performance from running on P processors.

- **Perfect Linear Speedup**: happens when no parallel overhead and algorithm is 100% parallel.

- **Super-linear Speedup**: typically due to cache effects ... i.e. as P grows, aggregate cache size grows so more of the problem fits in cache

\[
S(P) = \frac{\text{Time}_{\text{seq}}(1)}{\text{Time}_{\text{par}}(P)}
\]

\[
S(P) = P
\]

\[
S(P) > P
\]
Efficiency measures how well the parallel system’s resources are being utilized.

\[
\varepsilon = \frac{Time_{seq}}{P \times Time_{par}(P)} = \frac{S(P)}{P}
\]

Where \( P \) is the number of nodes and \( T \) is the elapsed runtime.
Amdahl’s Law

- What is the maximum speedup you can expect from a parallel program?
- Approximate the runtime as a part that can be sped up with additional processors and a part that is fundamentally serial.

\[
Time_{par}(P) = (\text{serial \_ fraction} + \frac{\text{parallel \_ fraction}}{P}) \times Time_{seq}
\]

- If \( \text{serial\_fraction} \) is \( \alpha \) and \( \text{parallel\_fraction} \) is \( 1 - \alpha \) then the speedup is:

\[
S(P) = \frac{Time_{seq}(1)}{(\alpha + \frac{1-\alpha}{P}) \times Time_{seq}(1)} = \frac{1}{\alpha + \frac{1-\alpha}{P}}
\]

- If you had an unlimited number of processors: \( P \rightarrow \infty \)

- The maximum possible speedup is: \( S = \frac{1}{\alpha} \)
Consider benefits of adding processors to your parallel program for different serial fractions.

Note: getting a serial fraction under 10% is challenging for the typical application.
**Gustafson's Observation:** For many problems, as the size of the problem (N) grows, the serial fraction ($\alpha(N)$) decreases. What does this imply for the speedup ($S(P,N)$)?

$$S(P,N) = \frac{T_{seq}(1)}{(\alpha(N) + \frac{1-\alpha(N)}{P})*T_{seq}(1)}$$

$$\lim_{N \to N_{large}} \alpha(N) = 0$$

In other words … if parallelizable computations asymptotically dominate the runtime, then solving a larger problem will increase your Amdahl-limited speedup.

**Weak Scaling:** Performance of an application when the problem size increases with the number of processors (fixed size problem per node)
Consider a system where tasks arrive periodically. The system takes some finite amount of time to execute each job.

- Suppose that the system is in Equilibrium: the average rate at which tasks arrive is equal to the average rate at which they are completed. Then, the average over time:

\[
\text{# tasks in the system} = \text{response time} \times \text{arrival rate}
\]
Little's Law

# tasks in the system = response time * arrival rate

- Tells us the number of "in flight" tasks we must have to keep our system busy, once we know how long tasks take to execute and the rate at which we can execute them.
- Applies in many situations:
  - # Outstanding load instrs = DRAM latency * DRAM bandwidth
  - # Matrix-multiplies in flight = Runtime * FLOP Rate
  - Pipeline Depth = Instruction Latency * Pipeline Width
Two important definitions:

- **Concurrency**: A condition of a system in which multiple tasks are *logically* active at one time.
- **Parallelism**: A condition of a system in which multiple tasks are *actually* active at one time.

Figure from “An Introduction to Concurrency in Programming Languages” by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010
Concurrent vs. Parallelism

Two important definitions:

- **Concurrency**: A condition of a system in which multiple tasks are *logically* active at one time.
- **Parallelism**: A condition of a system in which multiple tasks are *actually* active at one time.

Figure from “An Introduction to Concurrency in Programming Languages” by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010
A **Web Server** is a Concurrent Application (the problem is fundamentally defined in terms of concurrent tasks):

- An arbitrary, large number of clients make requests which reference per-client persistent state

Consider an Image Server, which relieves load on primary web servers by storing, processing, and serving only images.
Parallel Program example

- The Mandelbrot set: An iterative map in the complex plane

\[ z_{n+1} = z_n^2 + C \quad z_0 = 0, \quad c \text{ is constant} \]

- Plot rate of divergence for different values of C.

- An embarrassingly parallel problem since each point is computed independently.
Mandelbrot set: OpenMP solution

The following is simplified code for the serial Mandelbrot program.

Using OpenMP, how would you parallelize this?

```c
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        complex c = get_const_at_pixel(i, j);
        complex image[i][j] = mandel(c);
    }
}
```
OpenMP uses a loop level parallelism approach.
- Create large number of loop iterations by collapsing the loops
- Use dynamic schedule since effort per point varies so much
- Chunk size >> 1 to reduce scheduling overhead

```c
#pragma omp parallel for collapse(2) private (j, c) schedule (dynamic, 10)
for (i=0; i<N; i++){
    for (j=0; j<N; j++) {
        complex c = get_const_at_pixel(i,j);
        complex image[i][j] = mandel( c);
    }
}
```
Concurrent vs. Parallelism: wrap up

Key points:

- A web server had concurrency in its problem definition ... it doesn’t make sense to even think of writing a “serial web server”.
- The Mandelbrot program didn’t have concurrency in its problem definition. It would take a long time, but it could be serial.

Both cases use concurrency:

- A concurrent application is concurrent by definition.
- A parallel program (1) finds concurrency in the problem, (2) exposes the concurrency, and (3) exploits the exposed concurrency to complete a job in less time.

Figure from “An Introduction to Concurrency in Programming Languages” by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010
Parallelization Overheads

- There are many potential sources of overhead:
  - **Synchronization**: Independent threads of execution must periodically "agree" that they are at the same stage of a computation.
  - **Communication**: Independent threads must exchange data.
  - **Work**: In most cases, a parallelized algorithm performs asymptotically (Big Oh) more work than a serial algorithm. Hence why its “poor form” to define $S(P)$ as ...

\[
S(P) \neq \frac{Time_{par}(1)}{Time_{par}(P)}
\]

Note: An *Embarrassingly parallel* program is a program where tasks run independently … they do not communicate or synchronize … parallel overheads are usually minimal. That’s why we like them so much.
Example: Synchronization via Barriers

- All processors perform some of their share of a computation, and write their partial results out to shared memory.
- Before reading the results of the previous phase of the computation, all processors must know that all other processors have written their results to shared memory.
The time spent executing the Barrier is a parallelization overhead: it would not need to be done in a serial implementation.

Additionally, if some processes finish their share of the computation early, the time spent waiting for other processors is wasted.

- This is an example of *Load Imbalance* - more in a moment.
Communication

- Unless the program is Embarassingly Parallel, processors will need to read the results computed by other processors.
- In many shared-memory programs, this boils down to synchronization and a memory fence:
  1) All processors **write results** out to shared memory,
  2) Execute a Barrier/fence
  3) All processors **read the results** written by other processors
- Time spent writing/reading intermediate values are communication overhead: wouldn't always be done in a sequential implementation

![Diagram](image-url)
On distributed-memory machines (e.g. a cluster), communication can only occur by sending discrete messages over a network.

- The sending processor **marshals the shared data** from the application's data structures into a message buffer.
- The receiving processor must **wait for the message** to arrive ...
- ... and **un-pack the data** back into data structures.
On distributed-memory machines (e.g. a cluster), communication can only occur by sending discrete messages over a network

- The sending processor **marshals the shared data** from the application's data structures into a message buffer
- The receiving processor must **wait for the message** to arrive ...
- ... and **un-pack the data** back into data structures

If the communication protocol is **synchronous**, then the sending processor must **wait for acknowledgement** that the message was received
Granularity is the ratio of compute time to communication time

- Hardware: compute rate vs. communication rate … also expressed as flops relative to memory latency
- Software: How much computation you need to compensate for parallel overhead.

Key rule: Granularity demanded by software must be met or bettered by hardware. Fine grained applications do not run well on coarse grained systems.
Recall our example earlier of Load-Imbalance in a barrier-synchronized computation

- Our parallel decomposition assigned tasks to processors unevenly
  - Some Threads' tasks contained more work than others'
- This is a performance problem because some processors are idle, waiting for others to finish