

CS3210 Lab 2

Parallel Programming with OpenMP
& Performance Instrumentation

Calibration / Improvements

After previous tutorial :)

How fast should I speak? (As a multiplier on my current speed)

0

0.5x

0%

0.75x

0%

all good

0%

1.25x

0%

1.5x

0%

Lab 1 Feedback

hopefully all the submissions are marked by this lab session

Lab 1 Feedback

- Everyone gets passing marks :)
 - How we grade in this course might or might not be similar to others
 - Still read the comments in the pdf
- Some comments:
 - Keep critical section small
 - Main culprit: print statements
 - shmget copy-pasting
 - Try to minimize; put into structs if possible for better locality

Feedback 1: Keeping critical sections small

Do:

```
LOCK(m)
```

```
    Minimal work...
```

```
UNLOCK(m)
```

Printing stuff (especially in the critical section) can waste cycles.

Minimize the amount of IO in your program before benchmarking!

Do not:

```
LOCK(m)
```

```
    rand()/printf() ...
```

```
    other things that costs cycles unnecessarily
```

```
UNLOCK(m)
```

Tip: -DDEBUG flag

In your code:

```
#ifdef DEBUG  
printf("This program is compiled with -DDEBUG!");  
#endif  
printf("Hello world!");
```

Specify in your compilation command/Makefile to compile with DEBUG flag:

```
g++ -DDEBUG my_code.cpp -o my_program
```



Will print both
statements!

Common Issue 2: Shmget copy paste

Do:

```
// I want to store a and b in shared memory
struct shm_data { int a; int b; }
data = shmget... (sizeof(struct shm_data))
```

Fewer shm regions, better locality / caching, etc.

Do not:

```
// I want to store a and b
int* a = shmget...
int* b = shmget...
// or, manually index into shm:
shm = shmget...; shm[0] = 5...
```


"How to write report"

- Understand the skills we are trying to teach
 - Start with benchmarking
 - ==> Getting aggregated results
 - ==> Having a hypothesis on results
 - ==> Testing the given hypothesis
 - ==> Improving your code
 - ==> Back to benchmarking

"How to write report"

- Understand the skills we are trying to teach
 - Start with benchmarking
 - ==> Getting aggregated results
 - ==> Having a hypothesis on results
 - ==> Testing the given hypothesis
 - ==> Improving your code
 - ==> Back to benchmarking

Lab 1

"How to write report"

- Understand the skills we are trying to teach

- Start with benchmarking

- ==> Getting aggregated results

- ==> Having a hypothesis on results

- ==> Testing the given hypothesis

- ==> Improving your code

- ==> Back to benchmarking

- This is the basis of your performance analysis.
- Use whichever tools you want to use that might help – be it time, perf, hyperfine, etc.
- In this step, we need to get raw data.
Might or might not necessarily need to see individual data points in the final report
 - Put raw data in appendix
 - Put in a easily-digestible form

"How to write report"

- Understand the skills we are trying to teach

- Start with benchmarking

- ==> Getting aggregated results

- ==> Having a hypothesis on results

- ==> Testing the given hypothesis

- ==> Improving your code

- ==> Back to benchmarking

Data from my A1 in 2022
Results averaged from 10 runs

Machine	Program Type	Thread	Average time, s	Context-switch	Page-fault
xs-4114	Pre-optimization	4	117.094927332	2856	69382
xs-4114	Pre-optimization	8	99.167989729	3145	69428
xs-4114	Pre-optimization	16	70.265504577	3668	69483
xs-4114	Pre-optimization	32	224.247536403	11534962	69448
xs-4114	Pre-optimization	64	276.155826111	24591641	59981

"How to write report"

- Understand the skills we are trying to teach

- Start with benchmarking

- ==> **Getting aggregated results**

- ==> Having a hypothesis on results

- ==> Testing the given hypothesis

- ==> Improving your code

- ==> Back to benchmarking

Note:

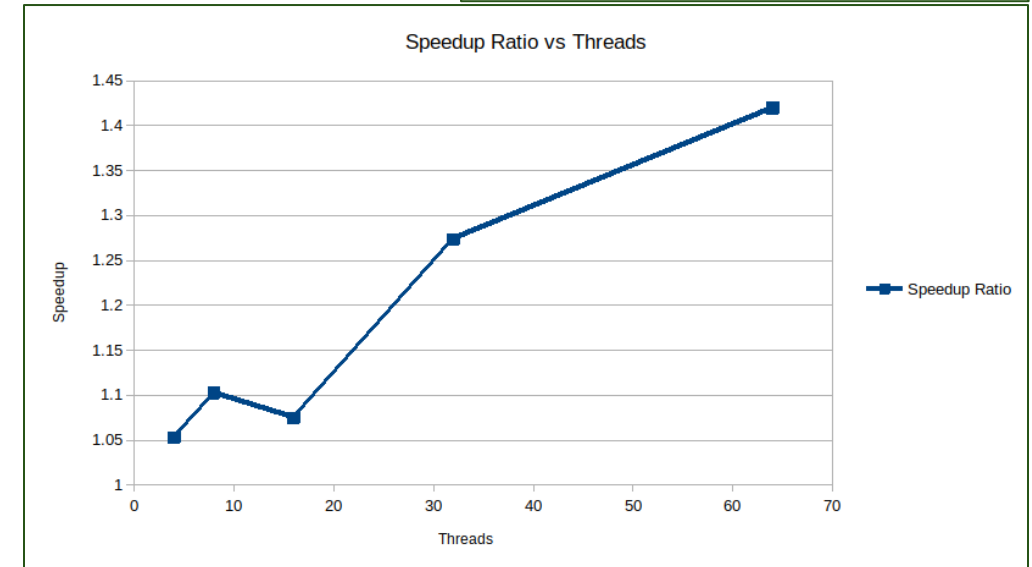
Don't worry about advanced statistical methods in this module.

- This is the part where you show the results of the data in an aggregated manner.
- The laziest is to put screenshots of perf/time/whatever in your report
 - Please don't :((((
- Use graphs to your advantage here.
 - Usually you want to see trends; how does your program benchmark...
 - across different machines?
 - across different input sizes?
 - across different algorithms?

"How to write report"

- Understand the skills we are trying to teach
 - Start with benchmarking
 - ==> Getting aggregated results
 - ==> Having a hypothesis on results
 - ==> Testing the given hypothesis
 - ==> Improving your code
 - ==> Back to benchmarking

Graph from previous data



Note: this graph is alright, but there are still quite some improvements which can be made:

- Error bars / standard deviation
- Scale of axis; log scale might be better for x-axis

"How to write report"

- Understand the skills we are trying to teach

- Start with benchmarking
 - ==> Getting aggregated results
 - ==> Having a hypothesis on results
 - ==> Testing the given hypothesis
 - ==> Improving your code
 - ==> Back to benchmarking

- Make it clear that you're putting forth your hypothesis
- Hypothesis needs to be plausible!
- Make sure you remove common issues before you even get to this point
 - sleep, print statements, etc.

"How to write report"

- Understand the skills we are trying to teach

- Start with benchmarking
 - ==> Getting aggregated results
 - ==> Having a hypothesis on results
 - ==> Testing the given hypothesis
 - ==> Improving your code
 - ==> Back to benchmarking

- A good hypothesis doesn't come out of thin air, but backed up by data
- Common points: perf, flamegraphs

"How to write report"

- Understand the skills we are trying to teach

- Start with benchmarking

- ==> Getting aggregated results

- ==> Having a hypothesis on results

- ==> Testing the given hypothesis

- ==> Improving your code

- ==> Back to benchmarking

(modified) excerpt from prev A1

In every single test on the Xeon machine, with an increasing number of threads, there is a speedup - up until a certain point where it suddenly has a large slowdown.

This might happen due to contention on the processing unit which happening due to having too many threads.

Figure A6 sheds some insight into this - once the number of threads is larger than 16, there is a huge increase in the number of context-switches, which is a good indicator of contention and would incur a large penalty in time taken.

"How to write report"

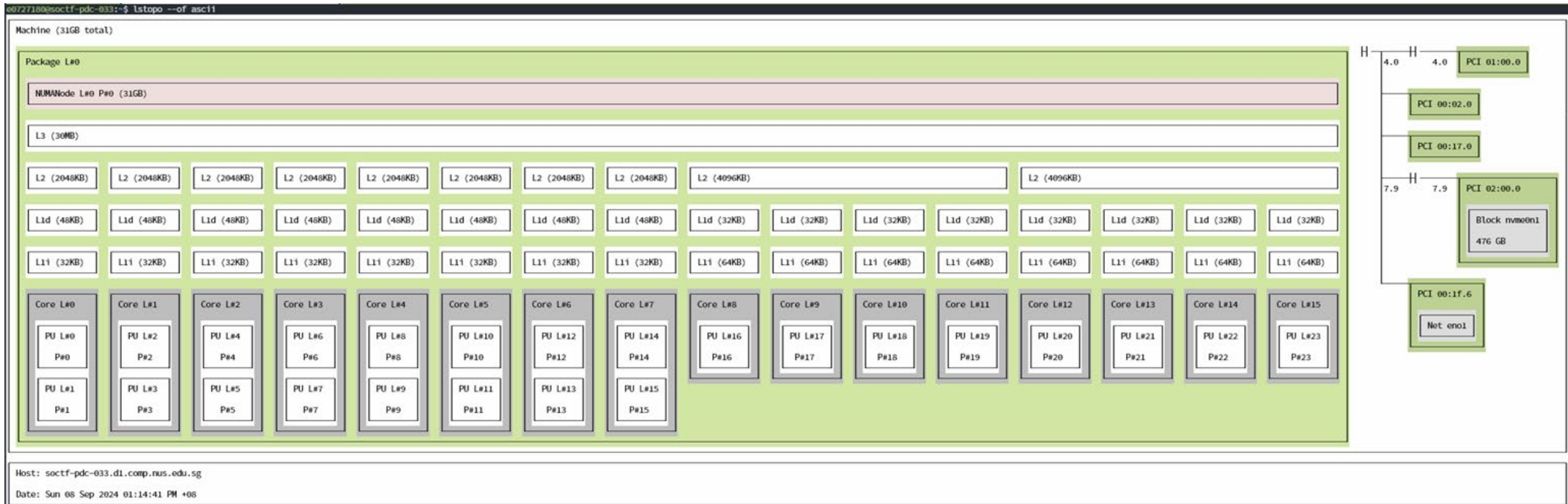
- Understand the skills we are trying to teach
 - Start with benchmarking
 - ==> Getting aggregated results
 - ==> Having a hypothesis on results
 - ==> Testing the given hypothesis
 - ==> Improving your code
 - ==> Back to benchmarking
 - This goes hand-in-hand with the previous point; if you suspect that something is causing your code to be slow, change it (if possible)
 - Now you have a new version of code that performs better 😊
 - Unlocks a new benchmarking data point
 - Also unlocks comparison across versions

Quick Recap + Revisiting

Previous tutorial, lectures

lstopo on heterogeneous processor (i7-13700)

Recall: i7-13700 has 8x P-cores and 8x E-cores

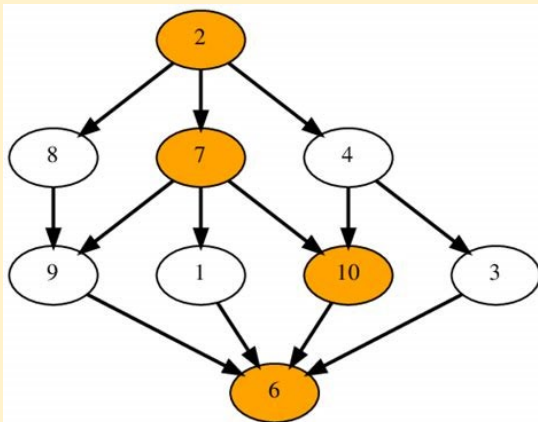


Recap

Parallel Programming Models I (Lec 4 & Lab 2)

Data vs Task Parallelism

Task Dependence Graphs



Coordination models

- Shared address space
- Data parallel
- Message passing

Foster's Design Methodology

Partitioning
↓
Communication
↓
Agglomeration
↓
Mapping

Parallel Patterns

- Fork-join
- Parbegin-Parend
- SIMD/SPMD
- Master-Worker
- Task Pools
- Producer-Consumer
- Pipelining

Prelude to Tutorial

I. OpenMP

OpenMP's parallel pattern

What kind of parallel pattern does OpenMP use? [p]

- Fork-Join
- Client-Server
- Task pool
- Producer-consumer
- Parbegin-Parend

What kind of parallel pattern does OpenMP use?

0

Fork-Join

0%

Client-Server

0%

Task pool

0%

Producer-Consumer

0%

Parbegin-Parend

0%

OpenMP's parallel pattern

What kind of parallel pattern does OpenMP use?

- Fork-Join (Mainly how OpenMP works in the backend)
- ~~Client-Server~~
- ~~Task pool~~
- ~~Producer-consumer~~
- Parbegin-Parend (Mainly what you, as a programmer, will do in your code)

OpenMP's parallel pattern

```
printf("program begin\n");  
N = 1000;
```

Serial

```
#pragma omp parallel for  
for (i=0; i<N; i++)  
    A[i] = B[i] + C[i];
```

Parallel

```
M = 500;
```

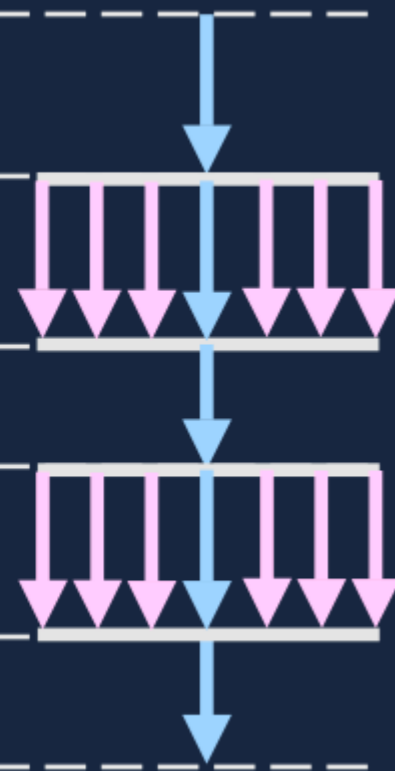
Serial

```
#pragma omp parallel for  
for (j=0; j<M; j++)  
    p[j] = q[j] - r[j];
```

Parallel

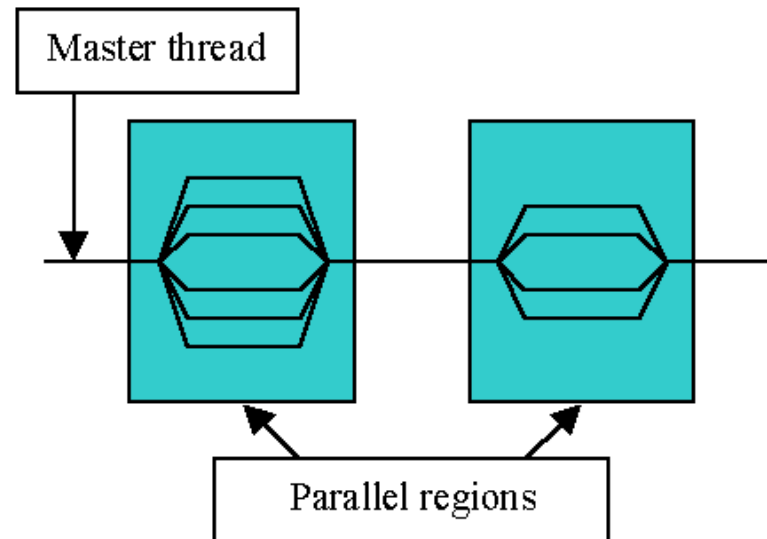
```
printf("program done\n");
```

Serial



What is OpenMP?

- Compiler directives that help you parallelize parts of your code easily!
- TL;DR: Add "`#pragma omp ...`" above your parallelizable code.



What is OpenMP?

Main usage: Parallelize for loops!

- Use `#pragma omp` to tell the compiler to generate parallel code for the code block below.
- All threads will wait for each other before continuing (implicit barrier).

```
void mm(matrix a, matrix b, matrix result)
{
    int i, j, k;

    // Do the multiplication
    for (i = 0; i < size; i++)
        for (j = 0; j < size; j++)
            for (k = 0; k < size; k++)
                result.element[i][j] += a.element[i][k] * b.element[k][j];
}
```

Sequential (mm-seq.cpp)

```
void mm(matrix a, matrix b, matrix result)
{
    int i, j, k;

    // Parallelize the multiplication
    // Iterations of the outer-most loop are divided
    // amongst the threads
    // Variables are shared among threads (a, b, result)
    // and each thread has its own private copy (i, j, k)
    #pragma omp parallel for shared(a, b, result) private(i, j, k)
    for (i = 0; i < size; i++)
        for (j = 0; j < size; j++)
            for (k = 0; k < size; k++)
                result.element[i][j] += a.element[i][k] * b.element[k][j];
}
```

OpenMP (mm-omp.cpp)

Warning: OpenMP Nesting

- Nesting might lead to unexpected results!
- New feature: OMP_NESTED environment variable as flag
 - Unfortunately, will not be enabled
- Workaround: use collapse(2)



Nesting Work-Sharing Constructs

Please note that OpenMP does not allow nesting of work-sharing constructs (including if the nested loop is called in a function). For example, the following code is not allowed:



```
#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < n; i++)
        #pragma omp for
        for (j = 0; j < m; j++)
            // can't nest the omp for directives
}
```

The compiler will likely throw an error, but even if it does not, the code will be silently serialized. Please watch out for this common mistake!. See [this link](#) for more info.

OpenMP: Controlling work per thread

We can control how loop iterations are given to threads.

- Static: Array is split into equal chunks, each thread is pre-allocated a fixed chunk of the array in round-robin fashion
 - Less overhead, great if work size is approximately equal
- Dynamic: Fixed size chunks are dynamically allocated to idle threads during runtime
 - Good for unpredictable work times, but more overhead

STATIC




DYNAMIC



OpenMP constructs


Useful constructs for your exploration here
& in Assignment 1:

- **Sections:** The programmer manually defines some code blocks that will be assigned to any available thread, one at a time. Example:




```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            work1();
        }
        #pragma omp section
        {
            work2();
        }
        #pragma omp section
        {
            work3();
        }
    }
}
```

- **barrier directive:** synchronizes all threads (threads wait until all threads arrive at the barrier).




```
#pragma omp barrier
```

- **master directive:** Specifies a region that must be executed only by the master thread.




```
#pragma omp master
    structured_block
```

- **atomic directive:** Works like a mini-critical section; specifies that a specific memory location must be updated atomically.



```
#pragma omp atomic
    statement_expression
```

- **critical directive:** Specifies a critical region that must be executed only by one thread at a time (example on following page)



```
#include <omp.h>

int main(int argc, char *argv[])
{
    int x;
    x = 0;

    #pragma omp parallel shared(x)
    {

        #pragma omp critical
        x = x + 1;

    } /* end of parallel region */
}
```

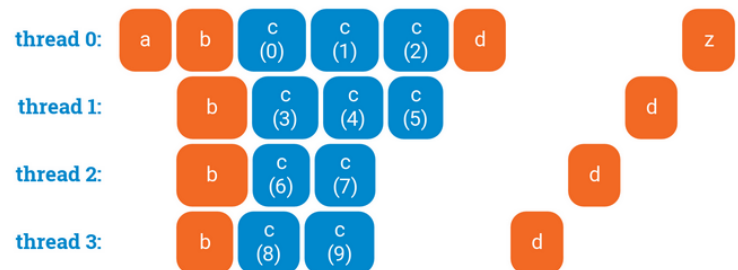
OpenMP: visual guides

<https://ppc.cs.aalto.fi/ch3/> (link should be in your lab sheet)

Interaction with critical sections

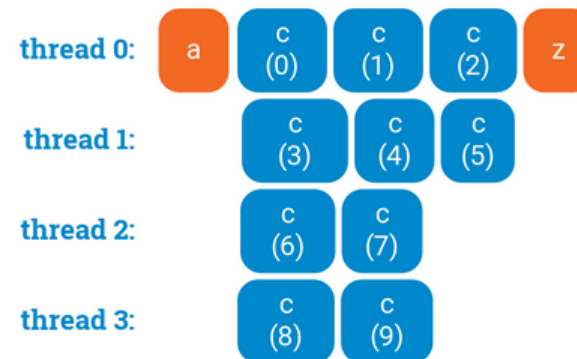
If you need a critical section after a loop, note that normally OpenMP will first wait for all threads to finish their loop iterations before letting any of the threads to enter a critical section:

```
a();
#pragma omp parallel
{
    b();
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    #pragma omp critical
    {
        d();
    }
}
z();
```



With OpenMP parallel for loops, we can easily parallelize it so that we are making a much better use of the computer. Note that OpenMP automatically waits for all threads to finish their calculations before continuing with the part that comes after the parallel for loop:

```
a();
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    c(i);
}
z();
```



Prelude to Tutorial

ll. perf

What is perf?

Unified interface to collect performance statistics from Linux from 2 sources:

- **Hardware counters** (from performance measurement unit [PMU])
 - e.g. cache misses, branch misses, instructions, ...
- **Software counters** (provided by Linux kernel)
 - e.g. page faults, cpu-migrations, ...

perf events

Get list of events that perf can measure via **perf list**

- Differs for Linux kernels + Hardware types
- Example usage: `perf stat -- ls`

Do not be too alarmed by the huge list of events - the lab sheet will guide you to measure events that are more important for this course.

List of pre-defined events (to be used in -e):

branch-instructions OR branches	[Hardware event]
branch-misses	[Hardware event]
bus-cycles	[Hardware event]
cache-misses	[Hardware event]
cache-references	[Hardware event]
cpu-cycles OR cycles	[Hardware event]
instructions	[Hardware event]
ref-cycles	[Hardware event]
alignment-faults	[Software event]
bpf-output	[Software event]
cgroup-switches	[Software event]
context-switches OR cs	[Software event]
cpu-clock	[Software event]
cpu-migrations OR migrations	[Software event]
dummy	[Software event]
emulation-faults	[Software event]
major-faults	[Software event]
minor-faults	[Software event]
page-faults OR faults	[Software event]
task-clock	[Software event]
duration_time	[Tool event]
L1-dcache-load-misses	[Hardware cache event]
L1-dcache-loads	[Hardware cache event]
L1-dcache-stores	[Hardware cache event]
L1-icache-load-misses	[Hardware cache event]
LLC-load-misses	[Hardware cache event]
LLC-loads	[Hardware cache event]
LLC-store-misses	[Hardware cache event]
LLC-stores	[Hardware cache event]
branch-load-misses	[Hardware cache event]
branch-loads	[Hardware cache event]
dTLB-load-misses	[Hardware cache event]
dTLB-loads	[Hardware cache event]
dTLB-store-misses	[Hardware cache event]
dTLB-stores	[Hardware cache event]
iTLB-load-misses	[Hardware cache event]
iTLB-loads	[Hardware cache event]
node-load-misses	[Hardware cache event]
node-loads	[Hardware cache event]
node-store-misses	[Hardware cache event]
node-stores	[Hardware cache event]
branch-instructions OR cpu/branch-instructions/	[Kernel PMU event]
branch-misses OR cpu/branch-misses/	[Kernel PMU event]
bus-cycles OR cpu/bus-cycles/	[Kernel PMU event]
cache-misses OR cpu/cache-misses/	[Kernel PMU event]
cache-references OR cpu/cache-references/	[Kernel PMU event]
cpu-cycles OR cpu/cpu-cycles/	[Kernel PMU event]
instructions OR cpu/instructions/	[Kernel PMU event]
mem-loads OR cpu/mem-loads/	[Kernel PMU event]
mem-stores OR cpu/mem-stores/	[Kernel PMU event]
ref-cycles OR cpu/ref-cycles/	[Kernel PMU event]
topdown-fetch-bubbles OR cpu/topdown-fetch-bubbles/	[Kernel PMU event]
topdown-recovery-bubbles OR cpu/topdown-recovery-bubbles/	[Kernel PMU event]
topdown-slots-issued OR cpu/topdown-slots-issued/	[Kernel PMU event]
topdown-slots-retired OR cpu/topdown-slots-retired/	[Kernel PMU event]
topdown-total-slots OR cpu/topdown-total-slots/	[Kernel PMU event]

Live Demo: Tut 1's pthread_addsub results

4C/8T i7-7700 CPU @ 3.60GHz*: 5.565s

8C/8T i7-9700 CPU @ 3.00GHz*: 6.374s

10C/20T Xeon Silver 4114 @ 2.20GHz*: 6.8194s

8C/16T Xeon W-2245 @ 3.90GHz*: 5.21152s

16C/24T i7-13700 @ 2.10GHz* (P-core), 1.50GHz* (E-core): 1.3826s

For me:
`cd ~/tut1`
`tail -n +1 *.out`

*Base clock speed, different from tutorial 1 slides

Rebench w/ perf instead of hyperfine.

Why does i7-7700 run faster than i7-9700 & Xeon 4114?

Why is i7-13700 the fastest here?

Live Demo: Tut 1's pthread_addsub results

Commands to replicate results:

```
g++ -o pthread_addsub pthread_addsub.cpp
```

```
srun -p i7-7700 perf stat -r 3 ./pthread_addsub > /dev/null 2> i7-7700.out
```

```
srun -p i7-9700 perf stat -r 3 ./pthread_addsub > /dev/null 2> i7-9700.out
```

```
srun -p xs-4114 perf stat -r 3 ./pthread_addsub > /dev/null 2> xs4114.out
```

```
srun -p xw-2245 perf stat -r 3 ./pthread_addsub > /dev/null 2> xw-2245.out
```

```
srun -p i7-13700 perf stat -r 3 ./pthread_addsub > /dev/null 2> i7-13700.out
```

Live Demo: Tut 1's pthread_addsub results

```
==> i7-7700.out <==
```

Performance counter stats for './pthread_addsub' (3 runs):

4,227.75 msec	task-clock	#	0.991 CPUs utilized
4,027	context-switches	#	961.066 /sec
48	cpu-migrations	#	11.455 /sec
114	page-faults	#	27.207 /sec
11,232,933,309	cycles	#	2.681 GHz
6,042,462,730	instructions	#	0.54 insn per cycle
2,008,435,891	branches	#	479.325 M/sec
934,139	branch-misses	#	0.05% of all branches

4.266 +- 0.640 seconds time elapsed (+- 15.00%)

i7-7700 @ 3.60GHz

```
==> xs4114.out <==
```

Performance counter stats for './pthread_addsub' (3 runs):

5,887.32 msec	task-clock	#	0.971 CPUs utilized
4,000	context-switches	#	666.662 /sec
110	cpu-migrations	#	18.333 /sec
110	page-faults	#	18.333 /sec
11,179,151,606	cycles	#	1.863 GHz
6,044,606,909	instructions	#	0.54 insn per cycle
2,008,806,711	branches	#	334.799 M/sec
582,475	branch-misses	#	0.03% of all branches

6.0608 +- 0.0579 seconds time elapsed (+- 0.95%)

xs-4114 @ 2.20GHz

i7-7700 wins due to higher clock speed
(& thus having higher single-threaded
performance)

Live Demo: Tut 1's pthread_addsub results

```
==> i7-7700.out <==
```

Performance counter stats for './pthread_addsub' (3 runs):

4,227.75 msec	task-clock	#	0.991 CPUs utilized
4,027	context-switches	#	961.066 /sec
48	cpu-migrations	#	11.455 /sec
114	page-faults	#	27.207 /sec
11,232,933,309	cycles	#	2.681 GHz
6,042,462,730	instructions	#	0.54 insn per cycle
2,008,435,891	branches	#	479.325 M/sec
934,139	branch-misses	#	0.05% of all branches

4.266 +- 0.640 seconds time elapsed (+- 15.00%)

i7-7700

```
==> i7-13700.out <==
```

Performance counter stats for './pthread_addsub' (3 runs):

1,496.08 msec	task-clock	#	1.054 CPUs utilized
4,016	context-switches	#	2.938 K/sec
92	cpu-migrations	#	67.303 /sec
114	page-faults	#	83.397 /sec
2,587,184,267	cycles	#	1.893 GHz
6,036,552,992	instructions	#	2.34 insn per cycle
2,007,460,407	branches	#	1.469 G/sec
297,733	branch-misses	#	0.01% of all branches

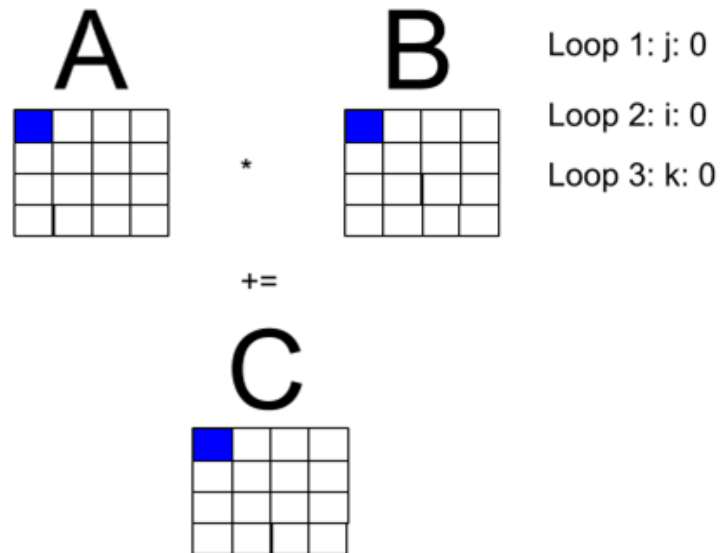
1.4197 +- 0.0797 seconds time elapsed (+- 5.62%)

i7-13700

i7-13700 wins due to its superior IPC &
branch prediction!
(despite having slower average clock speed)

Today's Problem: Matrix Multiplication

Can you see how it can be parallelized?



Getting started

- `wget` the lab from pdc (link in PDF)
- Please use `srun` or `sbatch` to run your programs from Ex3 and onwards.

Lab time!

- **Ex1 + Ex2:** Matrix Mult w/ OpenMP
- **Ex3:** System differences w/ OpenMP
- **Ex4:** Static vs Dynamic schedules
- **Ex5:** OpenMP sections
- **Ex6:** Synchronization in OpenMP
- **Ex7:** Profiling w/ perf
- **Ex8 + Ex9:** perf w/ Slurm
- **Ex10:** IPC & MFLOPS vs no. of threads
- **Ex11:** Matrix Mult optimization
- **Ex12:** Benchmarking Ex11

Quick taste of OpenMP

OpenMP exercises

perf exercises
(Some tidbits provided later)

**For submission
(by next Wed 2pm)**

Extra Content

perf subsampling

Live Demo: perf -M vs perf -e

- perf -M

- `perf stat -M GFLOPs python3 -c 'print(2.0 / 5.0)'`
- `srun -- hostname; perf stat -M GFLOPs python3 -c 'print(2.0 / 5.0)'`
- Run multiple times

- perf -e

- `perf stat -e fp_arith_inst_retired.scalar_double python3 -c 'print(2.0 / 5.0)'`
- `srun -- hostname; perf stat -e fp_arith_inst_retired.scalar_double python3 -c 'print(2.0 / 5.0)'`
- Run multiple times

- What is causing this inconsistency?

Live Demo: perf -M vs perf -e

What if we try replicating perf -M with perf -e?

- `perf stat -e
fp_arith_inst_retired.scalar_single,fp_arith_inst_retired.scalar_
double,fp_arith_inst_retired.128b_packed_double,fp_arith_inst_ret
ired.128b_packed_single,fp_arith_inst_retired.256b_packed_double
python3 -c 'print(2.0 / 5.0)'`

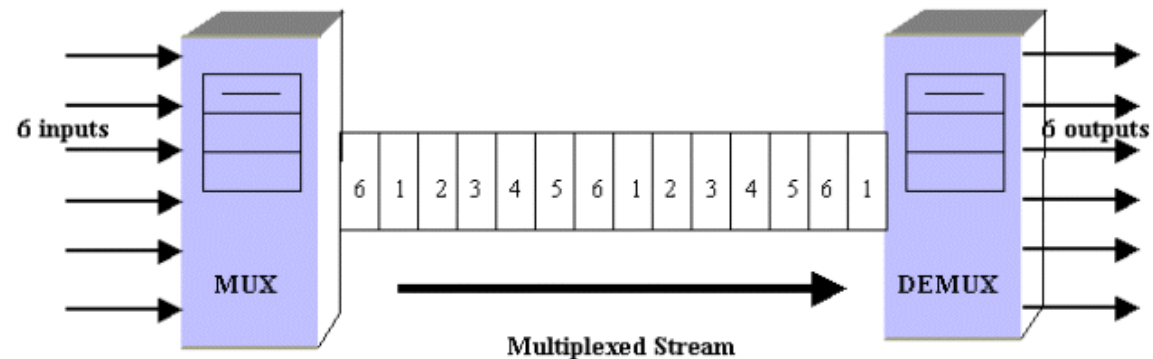
What happens?

Live Demo: perf -M vs perf -e

If there are not enough hardware counters: perf *multiplexes the events* (i.e. **subsampling**) and estimates the counts **based on execution time!**

multiplexing and scaling events

If there are more events than counters, the kernel uses time multiplexing (switch frequency = HZ, generally 100 or 1000) to give each event a chance to access the monitoring hardware. Multiplexing only applies to PMU events. With multiplexing, an event is **not** measured all the time. At the end of the run, the tool **scales** the count based on total time enabled vs time running.



% sign in perf readings

- Usually means that the reading is **subsampling**.
- Signifies % of time spent measuring the specific event.
 - <https://stackoverflow.com/questions/33679408/perf-what-do-n-percent-records-mean-in-perf-stat-output>

```
e0727180@soctf-pdc-009:/nfs/home/e0727180$ perf stat -M GFLOPs python3 -c 'print(2.0 / 5.0)'
0.4

Performance counter stats for 'python3 -c print(2.0 / 5.0)':

          0      fp_arith_inst_retired.256b_packed_single #      0.00 GFLOPs      (41.91%)
        447      fp_arith_inst_retired.scalar_double           (51.44%)
          0      fp_arith_inst_retired.128b_packed_single      (80.49%)
          0      fp_arith_inst_retired.scalar_single
          0      fp_arith_inst_retired.256b_packed_double      (77.60%)
          0      fp_arith_inst_retired.128b_packed_double      (48.56%)
    14,275,575 ns      duration_time

0.014275575 seconds time elapsed

0.014283000 seconds user
0.000000000 seconds sys
```

perf subsampling

Basically a warning to prevent what happened previously



Perf Event Sampling

If you ask `perf` to sample too many events (i.e., too many arguments after `-e`), it will not give you an *exact* count for all of those events. Instead, it will *sample* the events and give you an estimate. This is because the hardware event counters are limited in number, so the hardware cannot count all events at the same time. The number of available counters depends on the CPU being used.

Disclaimers for Ex 11: Optimizing matrix multiplication

- We are not concerned about total correctness.
 - The values for the matrices are generated **randomly**.
 - There is **NO NEED** to transpose the matrices! (you can assume a matrix is already transposed)
- There are multiple solutions possible.
 - As long as the order of access fulfills the criteria of row-wise access.

Summary

- OpenMP
- perf usage
- Using Slurm with OpenMP and perf

End of lab 2

- Slides uploaded!
- Feedback: bit.ly/feedback-theodore or scan below
- Email: theo@comp.nus.edu.sg

