Crystin VanWagenen

STUDYING THE CACHE IMPACT OF MULTICORE PROCESSORS

GOAL

In this lab, you will compile and run the Matrix Multiplication programs on the PDS Lab's cluster and Intel Xeon to study about the cache effects on sequential and parallel programs running on multi-core processors. You also learn the skill of profiling a program for performance analysis.

1. Run the program with matrix sizes 500, 1000, 2000, 3000, 4000. Then fill in **Table 1** as well as plot the runtime for Heracles in a chart (x-axis for the matrix size, y-axis for the runtime). For every matrix size, you run the program at least 2 times and get their average runtime.

   The programs should be run on the following machines: Hydra, Heracles and Xeon.

Table 1 – Hydra results

Matrix size	Runtime of Version 1	Runtime of Version 2	Runtime of Version 3	Runtime of Version 4
500 x 500	0.85	0.38	0.14	0.09
1000 x 1000	8.25	3.6	1.26	0.73
2000 x 2000	68.27	29.2	7.73	6.53
3000 x 3000	226.07	96.46	29.63	21.77
4000 x 4000	661.13	235.77	88.75	50.65

Table 1.1 - Speedup on Hydra

Matrix size	Speedup of Version 2	Speedup of Version 3	Speedup of Version 4
500 x 500	2.2368	6.071428	9.444444
1000 x 1000	2.29166	6.547619	11.3013698
2000 x 2000	2.3380136	8.8318240	10.4548238

3000 x 3000	2.34366576	7.6297671	10.3844740
4000 x 4000	2.80413114475972	7.44935211267606	13.052912142152

Table 2 – Heracles results

Matrix size	Runtime of Version 1	Runtime of Version 2	Runtime of Version 3	Runtime of Version 4
500 x 500	0.42	0.18	0.05	0.03
1000 x 1000	3.2	1.23	0.16	0.14
2000 x 2000	29.9	9.66	1.8	0.84
3000 x 3000	112.44	34.31	5.89	2.73
4000 x 4000	266.36	88.27	15.82	6.5

Table 2.1 - Speedup on Heracles

Matrix size	Speedup of Version 2	Speedup of Version 3	Speedup of Version 4
500 x 500	2.3333333	8.4	14
1000 x 1000	2.601626	20	22.857142
2000 x 2000	3.09523809	16.611111	35.595238
3000 x 3000	3.277178	19.089983	41.1868131
4000 x 4000	3.0175597	16.836915	40.9784615

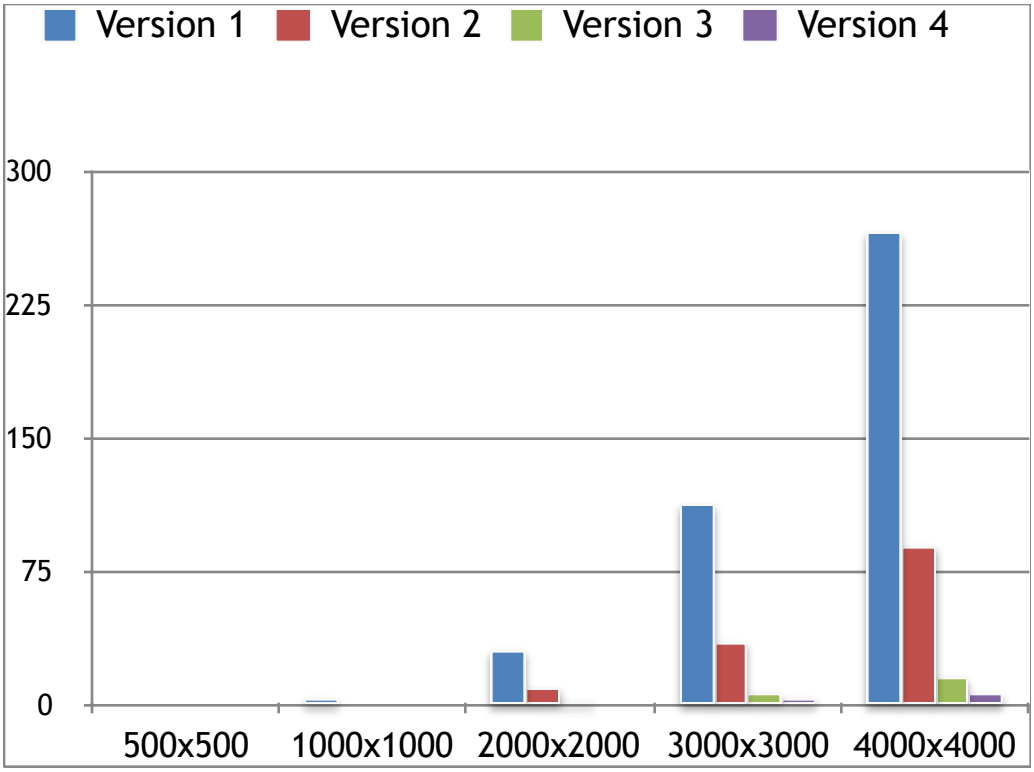
Compile the codes on Xeon using offload mode and fill the table 3 (see **compiling C/C++ programs on Xeon** section).

Table 3 – Xeon results

Matrix size	Runtime of Version 1	Runtime of Version 2	Runtime of Version 3	Runtime of Version 4
500 x 500	0.25	0.11	0.09	0.02
1000 x 1000	1.43	0.57	0.12	0.07
2000 x 2000	25.88	6.37	1.43	0.56
3000 x 3000	73.1	27.45	5.3	1.55
4000 x 4000	221.08	45.45	14.43	3.25

Table 3.1 - Speedup on Xeon

Matrix size	Speedup of Version 2	Speedup of Version 3	Speedup of Version 4
500 x 500	2.272727	2.77777	12.5
1000 x 1000	2.508771	11.9166	20.42857
2000 x 2000	4.062794	18.0979020979021	46.2142857142857
3000 x 3000	2.66302367941712	13.7924528301887	47.1612903225806
4000 x 4000	4.86424642464246	15.3208593208593	68.0246153846154



Plot the runtime for heracles (Table 2) in a chart (x-axis for the matrix size, y-axis for the runtime)

3. **Answer the following questions based in your experiments in requirement 2.**

a. How many threads the parallel versions (version 3 and 4) are using in each machine (Heracles, Xeon and Hydra)? Explain it in terms of physical core, logical core and hyper threads.

For Hydra's version 3 and 4, there are 12 logical cores and 12 threads.

For Heracles's version 3 and 4, there are 48 logical cores and 48 threads.

For Xeon's version 3 and 4, there are 32 logical cores and 32 threads.

b. What is the main factor for the runtime difference between Version 1 and Version 2? Why?

The main factor for the runtime difference between Version 1 and 2 is the loop interchange switch made between j and k. Since b_{kj} will now be accessed in sequential order vs striding through memory every n times. This will help with spatial locality.

c. What is the main factor for the runtime difference between Version 1 and Version 3? Why?

The main factor for the runtime difference between Version 1 and 3 is that Version 3 is being ran in parallel. The job is divided into 12 sub parts (Hydra) or 16 parts (Xeon) each being ran simultaneously. So the answer is parallelism.

d. What is the ideal speedup when you run a program that uses N cores? Compare it to the speedup of Version 4 and compare the performance improvements obtained from combining the techniques used in Version 2 & 3 into Version 4 and provide the result of your comparisons.

The ideal speedup when you run a program that uses N cores is N. Ideally with each **i** core that is introduced it should speed up by **i**. Hydra uses 12 logical cores so we should see a speed up of 12 for version 3, for version 4 we should see up to 12 times the speed up of Version 2. Thus ideal speedup for Hydra in Version 4 should be roughly 24 times speedup, where it has an average of 9-13% speedup. Almost half. Using the same calculations for Heracles, we construe that Heracles is performing at half of its optimum speedup. Then for Xeon, we see little performance for Version 3, however Version 4 is closer to its optimal speedup from Version 2. Since version 2 is 2-4x speedup, version 4 ideal speedup should be 64-128x speedup. And from the results we see 46-68x speedup.

Use Hydra in order to profile the programs with matrix size 1000x1000 as the instructions of the **Profiling C/C++ programs** section and report the results in Table 4.

For parallel versions (Version 3 and Version 4), their computation is distributed to 12 cores of a compute node. Each core will run 1 thread (there is no hyper threads on Hydra). The profiling outputs are the total results of 12 cores. So, you report these outputs in the Total columns and divide **Total** by 12 to get the average results for the **Per Core** columns.

Table 4 – Profile code results (Hydra).

Profiling outputs	Version 1	Version 2	Version 3 (Total)	Version 4 (Total)	Version 3 (Per Core)	Version 4 (Per Core)
CPU clocks (msec)	8312.142805	3644.579190	11833.698555	6113.268027	986.14154625	509.43900225

CR accesses	2,055,659,222	3,038,311,019	3,067,819,618	3,043,640,137	255,651,634.833	253,636,678.083
CM misses	1,310,766	618,690	842,014	535,199	70,167.83333333	44,599.91666666
DTLB misses	38,074	17,263	19,097,554	23,841	1,591,462.83333	1,986.75
Branch exec	1,033,270,803	1,022,872,235	1,047,404,187	1,038,836,430	87,283,682.25	86,569,702.5
Inst exec	11,179,363,357	11,106,653,298	16,165,040,541	16,119,246,295	1,347,086,711.75	1,343,270,524.58
misp branch	1,857,819	1,387,661	1,565,110	1,374,245	130,425.8333333	114,520.4166666

Where:

- *CPU clocks*: CPU clocks not halted event - `cpu-clock`
- CR accesses: number of cache references - `cache-references`
- CM : cache misses - `cache-misses`
- DTLB misses: data translation look aside buffer L1 misses & L2 misses - `dTLB-load-misses`
- Branch exec: the number of branches executed - `branch-instructions`
- Inst exec: the number of instructions executed - `instructions`
- misp branch: the number of mispredict branches - `branch-misses`

Hint: You may provide your answers based on the profiling results of requirement 4 and your own understanding about computer architecture to support your answers.

a. Write pseudo-code of a matrix multiplication program that would have the worst execution time and explain why your program would run the slowest. Use the work you've done so far on previous questions to help design your code.

The worst execution time would be:

Pseudo Code

```
for j = 1 to n
    for k = 1 to n
        for i = 1 to n
            cij = cij + aik*bkj
```

This would run the slowest, because each loop would have to jump through memory. The only sequential element is i=1 to n, which carries the row of C.

b. If you parallelize your program in the item (**e)** using the same technique as Version 3 and 4 above (i.e. using only the directive #pragma omp parallel for), will your program always produce correct outputs? Why?

If I could parallelize it, it would not produce correct outputs, because each partition would end up with a different c_{ij}.

c. When the matrix size increases, what are the factors resulting in the speed up of the Version 4?

The factors which contribute to the speedup of Version 4 is the fact that the code will have better spatial locality due to loop interchange, and then from there could use parallelism to speed up the program.

THE PROCESSOR ARCHITECTURE OF A COMPUTE NODE OF HERACLES

Visit our webclass to check Hydra architecture

http://pds.ucdenver.edu/webclass/Hydra%20Architecture.html

THE PROCESSOR ARCHITECTURE OF THE INTEL XEON PROCESSOR

Visit our webclass to check Xeon architecture http://pds.ucdenver.edu/webclass/Xeon%20and%20Phi%20Architecture.html
Access this link about Intel Xeon Processor Architecture
https://software.intel.com/en-us/articles/intel-xeon-processor-e5-26004600-product-family-technical-overview
Visit this site in order to get more information about Hyper-threading technology.
http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html

THE PROCESSOR ARCHITECTURE OF A COMPUTE NODE OF HERACLES

Visit our webclass to check Heracles architecture

http://pds.ucdenver.edu/webclass/Heracles_Architecture.html

SOURCE CODE

The Matrix Multiplication was implemented in four different versions. Each version is a function in the source code file **mm1.cpp**.

$$C = A * B = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} * \begin{vmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \dots & \dots & \dots & \dots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{vmatrix} = \begin{vmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \dots & \dots & \dots & \dots \\ c_{n1} & c_{n2} & \dots & c_{nn} \end{vmatrix}$$

- **Version1**: The function ***SequentialMatrixMultiplication_Version1*** in source code file

  Pseudo Code

  ```
  for i = 1 to n
      for j = 1 to n
          for k = 1 to n
                  cij = cij + aik*bkj
  ```

- **Version 2**: The function ***SequentialMatrixMultiplication_Version2*** in source code file

  Pseudo Code

  ```
          for i = 1 to n
          for k = 1 to n
                  for j = 1 to n
                      cij = cij + aik*bkj
  ```

- **Version 3:** The function ***ParallelMatrixMultiplication_Version3*** in source code file
  - This function is a parallel version of the version 1. The **#pragma omp parallel for** directive indicates that the following loop is executed in parallel.
  - If you run the program on one compute node of the cluster (Hydra), it will utilize 12 cores for its computation by dividing the workload into 12 parts or and assigning each core with one part.
  - If you run the program on Xeon processor it will utilize 8 cores for its computation by dividing the workload into 16 parts or and assigning each core with 2 parts.

- **Version 4:** The function ***ParallelMatrixMultiplication_Version3*** in source code file
  - This function is a parallel version of the version 2. The **#pragma omp parallel for** directive indicates that the following loop is executed in parallel.
  - If you run the program on one compute node of the cluster (Hydra), it will utilize 12 cores for its computation by dividing the workload into 12 parts and assigning each core with one part.

- If you run the program on Xeon processor it will utilize 8 cores for its computation by dividing the workload into 16 parts or and assigning each core with 2 parts.

PROFILING C/C++ PROGRAMS BY USING PERF

Profiling is a method used to analyze the execution of a program by measuring, for example, the usage of memory, the usage of particular instructions, or frequency and duration of function calls, cache miss/hit rate, branch prediction, etc. The profiling result is a helpful resource to aid program optimization.

In this Lab, you will use **Perf** profiler to measure information of CPU clocks, caches misses, branch prediction, etc. . Below are instructions to profile a C/C++ program on the Hydra cluster and Xeon. More information of **Perf** can be found at
http://www.brendangregg.com/perf.html
and
https://perf.wiki.kernel.org/index.php/Main_Page

 # Listing all currently known events:

perf list

 # The perf command alone will list the subcommands;

perf

PERF on Hydra

- **bpsh <#n>** perf stat -e <event1> -e <event2> -e<event n> <path/>Myprogram arg1 arg2 argn
  - Perf: profile tool that will monitor your program.
  - event: predefined events displayed in perf list command.
  - bpsh : execute your program on a compute node.
  - #n: is the node number (0, 1, 2, 3, ...,n) that you'd like to run your program on.
- Example: The following command will trace the number of page faults that occurred during the application execution on node 2.
  - **bpsh 2** perf stat -e cpu-clock -e instructions -e cache-references /myPath/mm 4 2 1

PERF on Heracles

- **ssh node<#n>** perf stat -e <event1> -e <event2> -e <event n> <path/>Myprogram arg1 arg2 argn
  - Perf: profile tool that will monitor your program.

- event: predefined events displayed in perf list command.
- bpsh : execute your program on a compute node.
- #n: is the node number (0, 1, 2, 3, ..., n) that you'd like to run your program on.

- Example: The following command will trace the number of page faults that occurred during the application execution on node 2.
  - ssh node2 perf stat -e cpu-clock -e instructions -e cache-references /myPath/mm 4 2 1

PERF on Xeon

Profile your program using perf on Xeon

- perf stat –e page-faults –e cpu-clock /myPath/mm1 4 2 1

Presents the number of page faults occurred during the application execution.

COMPILING AND RUNNING PROGRAMS ON HYDRA

REQUIREMENT

- Logon the Hydra server (see Lab 1)
- Make a folder named csc5593/lab3 in your home directory using the **mkdir** command.
- Copy the source code files to csc5551 or csc7551. If you are using MAC or Linux, you can copy these files using **scp** or **sftp** command. If you are using Windows, you can use WinSCP or SSH Secure Shell to login and copy files from your PC to the cluster. For more information on downloading WinSCP and how to use this software, visit:

http://pds.ucdenver.edu/document.php?type=software&name=winscp

- Hardware Information about Hydra:

http://pds.ucdenver.edu/webclass/Hydra%20Architecture.html

- Compiling C++/openMP programs on Hydra

http://pds.ucdenver.edu/webclass/Compiling%20openMP%20programs.html

- Running program on Hydra

  The first step when running your program on a compute node on hydra is to find an open node to run on. Visit for http://pds.ucdenver.edu/webclass/Monitoring%20the%20Cluster.html for how to monitor the compute nodes on Hydra. Try to select a node that shows the lowest usage.

  Afterwards, you can choose to execute your program immediately or with the batching system.

  Run a program immediately on Hydra:

  http://pds.ucdenver.edu/webclass/Running%20programs%20on%20Hydra.html

  Run a program on hydra by scheduling jobs with <at>:

  http://pds.ucdenver.edu/webclass/at%20-%20Command%20for%20scheduling%20programs.html

COMPILING AND RUNNING THE PROGRAMS ON HERACLES

REQUIREMENT

- Logon the Heracles server (see Lab 1)
- Make a folder named csc5593/lab3 in your home directory using the **mkdir** command.
- Copy the source code files to csc5551 or csc7551. If you are using MAC or Linux, you can copy these files using **scp** or **sftp** command. If you are using Windows, you can use WinSCP or SSH Secure Shell to login and copy files from your PC to the cluster. For more information on downloading WinSCP and how to use this software, visit:

http://pds.ucdenver.edu/document.php?type=software&name=winscp

- Hardware Information about Heracles:

http://pds.ucdenver.edu/webclass/Heracles_Architecture.html

- Compiling C++/openMP programs on Heracles

http://pds.ucdenver.edu/webclass/Compiling%20openMP%20programs.html

- Running program on Heracles

  The first step when running your program on a compute node on Heracles is to find an open node to run on. Visit https://heracles.ucdenver.pvt/mcms/ for monitoring the compute nodes on Heracles. Try to select a node that shows the lowest usage.

  http://pds.ucdenver.edu/webclass/Heracles-RunningPrograms.html

  Scheduling jobs on Heracles by using SLURM

  http://pds.ucdenver.edu/webclass/Heracles-RunningPrograms%20Slurm.html

COMPILING AND RUNNING PROGRAMS ON XEON

REQUIREMENT

- Logon the Xeon server (see Lab 1)
- Make a folder named csc5593/lab3 in your home directory using the **mkdir** command.
- Copy the source code files to csc5551 or csc7551. If you are using MAC or Linux, you can copy these files using **scp** or **sftp** command. If you are using Windows, you can use WinSCP or SSH Secure Shell to login and copy files from your PC to the cluster. For more information on downloading WinSCP and how to use this software, visit:
  - http://pds.ucdenver.edu/document.php?type=software&name=winscp
- In order to compile and execute an offload code on Xeon, you have to stay in the Xeon processor and execute the commands bellow:

```
icpc prog-off.cpp -o prog-off.out

./prog-off.out
```

Hardware Information about Xeon:

http://pds.ucdenver.edu/webclass/Xeon%20and%20Phi%20Architecture.html

Intel Xeon and Phi Programming Modes

http://pds.ucdenver.edu/webclass/Programming%20Models%20for%20Xeon%20Phi.html

Compiling C++/openMP programs on Xeon

http://pds.ucdenver.edu/webclass/Compiling%20openMP%20programs%20by%20using%20Intel%20compilers.html

Running offload program on Xeon/Phi

http://pds.ucdenver.edu/webclass/running%20Offload%20programs%20on%20Xeon-Phi.html

Scheduling jobs on xeon by using <at>

http://pds.ucdenver.edu/webclass/at%20-%20Command%20for%20scheduling%20programs.html