CS 211: High Performance Computing Project 1

Performance Optimization via Register Reuse

Due date: 11:59pm, Oct 3rd, 2016

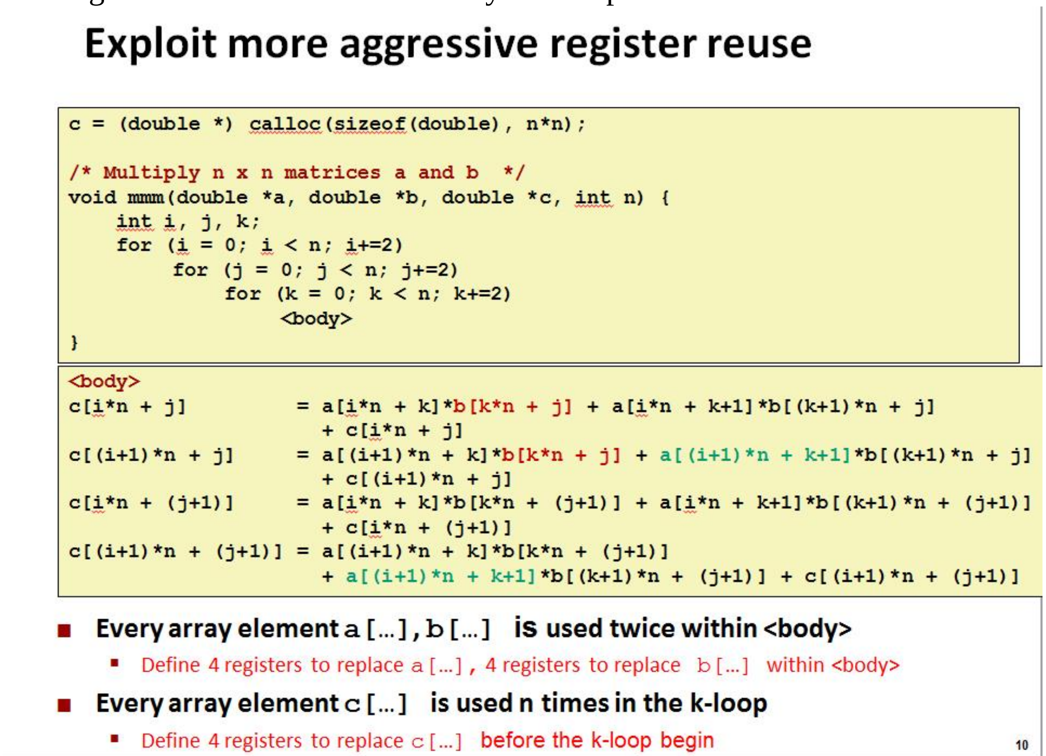
Note: You need to upload a pdf report for the project into the iLearn system. Please also upload all your source codes and a makefile as a tar file into iLearn system so that our TA can verify what you achieved in your report.

Part #1. (50 points) Assume your computer is able to complete 4 double floating-point operations per cycle when operands are in registers and it takes an additional delay of 100 cycles to access any operands that are not in registers. The clock frequency of your computer is 2 Ghz. How long it will take for your computer to finish the following algorithm ***dgemm0*** and ***dgemm1*** respectively for n= 1000? How much time is wasted on accessing operands that are not in registers? Implement the algorithm ***dgemm0*** and ***dgemm1*** and test them on TARDIS with n= 64, 128, 256, 512, 1024, 2048. Measure the time spend in the triple loop for each algorithm. Calculate the performance (in Gflops) of each algorithm. Performance is often defined as the number of floating-point operations performed per second. A performance of 1 Gflops means 1 billion of floating-point operations per second. You must use the system default compiler to compile your program. Your test matrices have to be 64 bit double floating point random numbers. Report the maximum difference of all matrix elements between the two results obtained from the two algorithms. This maximum difference can be used as a way to check the correctness of your implementation.

```
/*dgemm0: simple ijk version triple loop algorithm*/
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        for (k=0; k<n; k++)
            c[i*n+j]  += a[i*n+k] * b[k*n+j];

/*dgemm1: simple ijk version triple loop algorithm with register reuse*/
for (i=0; i<n; i++)
    for (j=0; j<n; j++) {
         register double r = c[i*n+j] ;
         for (k=0; k<n; k++)
            r += a[i*n+k] * b[k*n+j];
        c[i*n+j] = r;
    }
```

Part #2. (40 points) Let's use ***dgemm2*** to denote the algorithm in the following ppt slide from our class. Implement ***dgemm2*** and test it on TARDIS with n= 64, 128, 256, 512, 1024, 2048. Measure the time spend in the algorithm. Calculate the performance (in Gflops) of the algorithm. You must use the system default compiler to compile your program. Your test matrices have to be 64 bit double floating point random numbers. Do not forget to check the correctness of your computation results.



Part #3 (10 points). Assume you have 16 registers to use, please maximize the register reuse in your code (call this version code **dgemm3**) and compare your performance with ***dgemm0, dgemm1, and dgemm2***.

CS 211: High Performance Computing Project 1
Performance Optimization via Register Reuse
Shahriyar Valielahi Roshan – 861241935 – svali003@ucr.edu

Part #1.

$Floating - point\ operation\ time$	$\frac{T}{4}$
$Memory\ Access\ Time$	$100T$
Cycle Time	T
Frequency	2×10^9
N	1000

```
/*dgemm0: simple ijk version triple loop
algorithm*/
for (i=0; i<n; i++)
for (j=0; j<n; j++)
for (k=0; k<n; k++)
      c[i*n+j] += a[i*n+k] * b[k*n+j];
```

First I rewrite the operation inside the inner loop:

$$\boldsymbol{C = C + A \times B}$$

As you can see, First we should load A, B, and C into the registers

$$\boldsymbol{loadTime = 3 \times Memory\ Access\ Time = 300T}$$

Then we should do two floating-point computations (* then +)

$$\boldsymbol{computationTime = 2 \times Floating - point\ operation\ time = \frac{T}{2}}$$

Finally, we should store the result from C to memory, and we repeat it N^3 Times.

$$\boldsymbol{storeTime = Memory\ Access\ Time = 100T}$$

$$\boldsymbol{TotalRunTime = N^3 \times \left[300T + \frac{T}{2} + 100T\right] = N^3 \times [400.5T] = 200.25s}$$

$$\boldsymbol{Wasted\ Time = LoadTime + StoreTime = N^3 \times 400T = 200s}$$

$$\boldsymbol{percentage\ of\ Wasted\ Time = \frac{totalTime - [Computation\ Time]}{totalTime} = \frac{200s}{200.25} = 0.9988}$$

```
 /*dgemm1: simple ijk version triple loop
algorithm with register reuse*/
for (i=0; i<n; i++)
for (j=0; j<n; j++) {
register double r = c[i*n+j];
for (k=0; k<n; k++)
r += a[i*n+k] * b[k*n+j];
c[i*n+j] = r;
}
```

Initialize register r with the content of C for N^2 Times.

$$\boldsymbol{initializationTime = N^2 \times 100T}$$

Floating Point operation and Loading Operands A and B into registers for N^3 Time

$$\boldsymbol{LoadTime = N^3 \times 200T}$$

$$\boldsymbol{computationTime = N^3 \times \frac{T}{2}}$$

Final Memory update to store result in the register r to the memory for N^2 Times

$$\boldsymbol{storeTime = N^2 \times 100T}$$

$$\boldsymbol{totalRunTime = N^2 \times 200T + N^3 \times \left[\frac{T}{2} + 200T\right] = N^2 \times 200T + N^3 \times [200.5T] = 100.35s}$$

$$\boldsymbol{Wasted\ Time = initializationTime + StoreTime + LoadTime = N^2 \times 200T + N^3 * 200T = 100.1s}$$

$$\boldsymbol{percentage\ of\ Wasted\ Time = \frac{totalTime - ComputationTime}{totalTime} = \frac{100.1s}{100.35s} = 0.9975}$$

Part #2.

I made your life easy, you should just run these commands to compile and run every tests at once, and also you can find run time, performance, and maximum Difference for verification in a single file called result.drsvr

Warning: It can take a while before you see results, feel free to run other simulations.

make clean

make [make file is included]

chmod +x HW1.job

./HW1.job [Creates Result.txt]

#chmod +x HW1_V2.job

#./HW1_V2.job [Shows results on screen]

$$GFlops = \frac{Number\ of\ Operations}{Running\ Time\ in\ ns}$$

Runtime s	**64**	**128**	**256**	**512**	**1024**	**2048**
Dgemm0	0.004190	0.034568	0.317202	3.483223	37.319869	517.113316
Dgemm1	0.002871	0.022525	0.180913	2.490731	26.768490	378.567966
Dgemm2	0.000988	0.009243	0.080418	1.033047	12.932202	157.930397
Dgemm3	0.000912	0.006349	0.052006	0.613017	8.305896	121.283358

# Operations	64	128	256	512	1024	2048
Dgemm0	524288	4194304	33554432	268435456	2147483648	17179869184
Dgemm1	524288	4194304	33554432	268435456	2147483648	17179869184
Dgemm2	524288	4194304	33554432	268435456	2147483648	17179869184
Dgemm3	524288	4194304	33554432	268435456	2147483648	17179869184

#Runtime ns	64	128	256	512	1024	2048
Dgemm0	4190032	34568204	317202307	3483223347	37319868945	517113316415
Dgemm1	2870851	22525039	180913184	2490730531	26768489564	378567966000
Dgemm2	987815	9242924	80417596	1033047086	12932202114	157930397237
Dgemm3	911929	6348894	52006222	613017039	8305896317	121283357800

# GFLOPS	64	128	256	512	1024	2048
Dgemm0	0.125127	0.034568	0.317202	0.077065	0.057543	0.033223
Dgemm1	0.182625	0.022525	0.180913	0.107774	0.080224	0.045381
Dgemm2	0.530755	0.009243	0.080418	0.259848	0.166057	0.108781
Dgemm3	0.574922	0.006349	0.052006	0.437892	0.258549	0.141651

#maxDiff	64	128	256	512	1024	2048
Dgemm0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
Dgemm1	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
Dgemm2	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
Dgemm3	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

Part #3.

In this approach I used 9 Registers to store C4 (Result) Matrix, and 6 Registers for each column calculation that get different values to calculate different columns in the innermost loop.

Notes:

1- Counter jumps 3 steps at once (i+=3)
2- Multiply 3 * 3 matrices using these equations.

C00 = A00*B00 + A01*B10 + A02*B20
C01 = A00*B01 + A01*B11 + A02*B21
C02 = A00*B02 + A01*B12 + A02*B22

C10 = A10*B00 + A11*B10 + A12*B20
C11 = A10*B01 + A11*B11 + A12*B21
C12 = A10*B02 + A11*B12 + A12*B22

C20 = A20*B00 + A21*B10 + A22*B20
C21 = A20*B01 + A21*B11 + A22*B21
C22 = A20*B02 + A21*B12 + A22*B22

3- Each color shows usage of 6 registers to calculate each row, I reuse these 6 registers 3 times to make calculations.

```
void dgemm3 (){

            for (int i = 0; i < N; i += 3) {
                for (int j = 0; j < N; j += 3) {

                    register int t = i*N+j; // COLUMN 0 ROW 0
                    register int tt = t+N;  // COLUMN 0 ROW 1
                    register int ttt = tt+N; // COLUMN 0 ROW 2

                    register double rc00 = C4[t];
                    register double rc01 = C4[t+1];
                    register double rc02 = C4[t+2];

                    register double rc10 = C4[tt];
                    register double rc11 = C4[tt+1];
                    register double rc12 = C4[tt+2];

                    register double rc20 = C4[ttt];
                    register double rc21 = C4[ttt+1];
                    register double rc22 = C4[ttt+2];

                    for (int k = 0; k < N; k += 3) {

                        register int ta = i*N+k;
                        register int tta = ta+N;
                        register int ttta = tta+N;

                        register int tb = k*N+j;
                        register int ttb = tb+N;
                        register int tttb = ttb+N;


                        /*
                         *  C00 = A00*B00 + A01*B10 + A02*B20
                         *  C01 = A00*B01 + A01*B11 + A02*B21
                         *  C02 = A00*B02 + A01*B12 + A02*B22
                         *
                         *  C10 = A10*B00 + A11*B10 + A12*B20;
                         *  C11 = A10*B01 + A11*B11 + A12*B21;
                         *  C12 = A10*B02 + A11*B12 + A12*B22;
                         *
                         *  C20 = A20*B00 + A21*B10 + A22*B20;
                         *  C21 = A20*B01 + A21*B11 + A22*B21;
                         *  C22 = A20*B02 + A21*B12 + A22*B22;
                         */



                        register double R1 = A[ta]; // ra00
                        register double R2 = A[tta]; // r10
                        register double R3 = A[ttta]; // 20

                        register double R4 = B[tb]; // rb00
                        register double R5 = B[tb+1]; // rb01
                        register double R6 = B[tb+2]; // rb02

                        rc00 += R1 * R4;
                        rc01 += R1 * R5;
                        rc02 += R1 * R6;
```

```
                        }
                    }

                    rc10 += R2 * R4;
                    rc11 += R2 * R5;
                    rc12 += R2 * R6;

                    rc20 += R3 * R4;
                    rc21 += R3 * R5;
                    rc22 += R3 * R6;

                    R1 = A[ta+1];
                    R2 = A[tta+1];
                    R3 = A[ttta+1];

                    R4 = B[ttb];
                    R5 = B[ttb+1];
                    R6 = B[ttb+2];

                    rc00 += R1 * R4;
                    rc01 += R1 * R5;
                    rc02 += R1 * R6;

                    rc10 += R2 * R4;
                    rc11 += R2 * R5;
                    rc12 += R2 * R6;

                    rc20 += R3 * R4;
                    rc21 += R3 * R5;
                    rc22 += R3 * R6;

                    R1 = A[ta+2];
                    R2 = A[tta+2];
                    R3 = A[ttta+2];

                    R4 = B[tttb];
                    R5 = B[tttb+1];
                    R6 = B[tttb+2];

                    rc00 += R1 * R4;
                    rc01 += R1 * R5;
                    rc02 += R1 * R6;

                    rc10 += R2 * R4;
                    rc11 += R2 * R5;
                    rc12 += R2 * R6;

                    rc20 += R3 * R4;
                    rc21 += R3 * R5;
                    rc22 += R3 * R6;

                }
                C4[t] = rc00;
                C4[t+1] = rc01;
                C4[t+2] = rc02;

                C4[tt] = rc10;
                C4[tt+1] = rc11;
                C4[tt+2] = rc12;

                C4[ttt] = rc20;
                C4[ttt+1] = rc21;
                C4[ttt+2] = rc22;
            }
        }
}
```

CS 211: High Performance Computing Project 2

Performance Optimization via Cache Reuse

Due date: 11:59pm, Oct 23th, 2015

Suppose your data cache has 60 lines and each line can hold 10 doubles. You are performing a matrix-matrix multiplication (**C=C+A*B**) with square matrices of size **10000X10000** and **10X10** respectively. Assume data caches are only used to cache matrix elements which are doubles. The cache replacement rule is ***least recently used first***. Assume no registers can be used to cache intermediate computing results. One-dimensional arrays are used to represent matrices with the row major order.

```
/* ijk – simple triple loop algorithm with simple single register reuse*/
for (i=0; i<n; i++)
    for (j=0; j<n; j++) {
        register double r=c[i*n+j];
        for (k=0; k<n; k++)
            r  += a[i*n+k] * b[k*n+j];
        c[i*n+j]=r;
    }
```

```
/* ijk – blocked version algorithm*/
for (i = 0; i < n; i+=B)
    for (j = 0; j < n; j+=B)
        for (k = 0; k < n; k+=B)
            /* B x B mini matrix multiplications */
            for (i1 = i; i1 < i+B; i1++)
                for (j1 = j; j1 < j+B; j1++) {
                    register double r=c[i1*n+j1];
                    for (k1 = k; k1 < k+B; k1++)
                        r  += a[i1*n + k1]*b[k1*n + j1];
                    c[i1*n+j1]=r;
                }
```

Part 1. (25 points) When matrix-matrix multiplication is performed using the *simple triple-loop* algorithm with single register reuse, there are 6 versions of the algorithm (ijk, ikj, jik, jki, kij, kji). Calculate the **number** of read cache misses for **each** element in **each** matrix for **each** version of the algorithm when the sizes of the matrices are **10000X10000** and **10X10** respectively. What is the percentage of read cache miss for each algorithm?

Part 2. (25 points) If matrices are partitioned into block matrices with each block being a 10 by 10 matrix, then the matrix-matrix multiplication can be performed using one of the 6 ***blocked version algorithms*** (ijk, ikj, jik, jki, kij, kji). Assume the multiplication of two blocks in the inner three loops uses the same loop order as the three outer loops in the blocked version algorithms. Calculate the **number** of read cache misses for **each** element in **each** matrix for **each** version of the blocked algorithm when the size of the matrices is **10000**. What is the percentage of read cache miss for each algorithm?

Part 3. (25 points) Implement the algorithms in part (1) and (2). Report your execution time on TARDIS cluster. Adjust the block size from 10 to other numbers to see what is the optimal block size on your computer. Compile your code using the default compiler on Tardis without optimization tag. Compare and analyze the performance of your codes for n=2048. Please always verify the correctness of your code.

Part 4. (25 points) Improve your implementation by using both cache blocking and register blocking at the same time. Optimize your block sizes. Compile your code using both the default compiler and gcc-4.7.2 with different optimization flags (-O0, -O1, -02, and -O3.) respectively. Compare and analyze the performance of your codes for n=2048. Highlight the best performance you achieved. Please always verify the correctness of your code.

CS 211: High Performance Computing Project 2
Performance Optimization via Cache Reuse
Shahriyar Valielahi Roshan – 861241935 – svali003@ucr.edu

Part #1.

To solve this problem, let's take a look at a cache behavior. Our cache has 60 lines; each line can hold 10 doubles, or 10 elements. So we fetch 10 elements at the same time on read. When we read elements on the same row, we will miss on the first elements, so we fetch the first element and 9 elements after it. So we get hit 9 times. But when we read data on different rows, we will miss after each read. Also we have 60 lines, so we can store 600 elements into the cache.

```
/* ijk */
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = c[i][j]
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

For ijk, jik [10X10] let's review the innermost loop:

for (k=0; k<n; k++)
r += a[i*n+k] * b[k*n+j];

A[0][0] Cold Miss, A[0][k] HIT $0 \leq k \leq 9$

B[k][0] Cold Miss

After first iteration [j loop], the whole B matrix will be in the cache since we have enough lines. [11 lines used. From the second iteration we only have miss for the first element of each row in A, and because all B elements are in the cache, we do not have any miss. We read the C matrix in the j loop, and we access elements in the same row consequently, so it looks like a Matrix A.

For jik, in the inner most there is no difference, but the for the C matrix in the j loop, it looks like a Matrix B because we access elements in consequent rows. But it does not change the number of misses

---

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

For jki, kji [10X10] let's review the innermost loop: It looks like the previous approach but we replace A with C, and also the access is as following: A [Element from each column], B[Element from the same Row], C [Elements from the same row]. Since the cache is big enough, for 10x10, the result is the same. [Rule of Thumb: When we enter a new row, you should bring data into the cache, if it's not already in the cache]

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

For ikj and jki, we access A before the inner most j loop in [elements in the same row], and in the inner most loop, we access both C and B in the worst possible way [Elements in different rows and the same column] , since our cache is big enough, after the first iteration C and B will be in the cache, and after 10 iteration, A will be in the cache. [In 10 x 10 case, when we access elements in the same column, it takes 10 iterations before we completely push everything into the cache, but when we access elements row by row, it takes longer [less miss] which is better when we do not have enough space in the cache]

Summery:

[10X10]

$$\mathit{Cache\ Misses}\ [10X10][ijk]\colon \begin{cases} A[i][0] & 0 \le i \le 9\ [10] \\ B[k][0] & 0 \le k \le 9\ [10] \\ C[i][0] & 0 \le i \le 9\ [10] \end{cases}$$

Note: Indexes can be different in different cases, but the rule is the same

General rule: When we access to a row for the first time, we will have a miss, and after each miss we bring 10 elements into the cache [10x10 case]

$$\text{missRate [10X10]} = \frac{n^2\times\left(\frac{1}{10}\right)+n^2\times\left(\frac{1}{10}\right)+n^2\times\left(\frac{1}{10}\right)}{2n^3+n^2} = \frac{10+10+10}{2000+100} = 0.0142$$

10 X 10				
Algorithm	A	B	C	totalRate
ijk	10	10	10	0.0142
jik	10	10	10	0.0142
kij	10	10	10	0.0142
ikj	10	10	10	0.0142
jki	10	10	10	0.0142
kji	10	10	10	0.0142

[10000X10000]

$$\text{Cache Misses [ijk]: }\begin{cases} C[i][j] = 1 \ when\ j\%10 = 0 \\ A[i][k] = 10000 \ when\ k\%10 = 0 \\ B[k][j] = 10000 \end{cases}$$

$$\text{Cache Misses [jik]: }\begin{cases} C[i][j] = 1 \\ A[i][k] = 10000 \ when\ k\%10 = 0 \\ B[k][j] = 10000 \end{cases}$$

$$\text{Cache Misses [kij]: }\begin{cases} A[i][k] = 10000 \ when\ k\%10 = 0 \\ B[k][j] = 10000 \\ C[i][j] = 10000 \ when\ j\%10 = 0 \end{cases}$$

$$\text{Cache Misses [ikj]: }\begin{cases} A[i][k] = 10000 \ when\ k\%10 = 0 \\ B[k][j] = 10000 \\ C[i][j] = 10000 \ when\ j\%10 = 0 \end{cases}$$

Algorithm	Order	Misses	Memory Requests
ijk	C, A, B	$n^2 \times \left[\frac{1}{10} + \frac{n}{10} + n\right]$	$2n^3 + n^2$
jik	C, A, B	$n^2 \times \left[1 + \frac{n}{10} + n\right]$	$2n^3 + n^2$
kij	A, C, B	$n^2 \times \left[\frac{1}{10} + \frac{n}{10} + \frac{n}{10}\right]$	$2n^3 + n^2$
ikj	A, C, B	$n^2 \times \left[1 + \frac{n}{10} + \frac{n}{10}\right]$	$2n^3 + n^2$
jki	B, C, A	$n^2 \times \left[\frac{1}{10} + n + n\right]$	$2n^3 + n^2$
kji	B, C, A	$n^2 \times [1 + n + n]$	$2n^3 + n^2$

$$\text{Miss Rate} = \frac{Misses}{\text{Memory Requests}}$$

10000 X 10000					
Algorithm	A	B	C	totalRate	
ijk	10^{11}	10^{12}	10^7	0.55	**55%**
jik	10^{11}	10^{12}	10^8	0.55	**55%**
kij	10^7	10^{11}	10^{11}	0.10	**10%**
ikj	10^8	10^{11}	10^{11}	0.10	**10%**
jki	10^{12}	10^7	10^{12}	1	**100%**
kji	10^{12}	10^8	10^{12}	1	**100%**

Part #2.

We divide the 10000X10000 matrix to 10X10 submatrices, or 1000 blocks.

The three inner for loops looks like the first part of Question 1.

$$\text{Cache Misses [ijk]: } \begin{cases} C[i][j] = 1 \; when \; j\%10 = 0 \\ A[i][k] = 1000 \; when \; k\%10 = 0 \\ B[k][j] = 1000 \; when \; j\%10 = 0 \end{cases}$$

$$\text{Cache Misses [ikj]: } \begin{cases} A[i][k] = 1 \; when \; k\%10 = 0 \\ C[i][j] = 1000 \; when \; j\%10 = 0 \\ B[k][j] = 1000 \; when \; j\%10 = 0 \end{cases}$$

$$\text{Cache Misses [jki]: } \begin{cases} B[k][j] = 1 \; when \; j\%10 = 0 \\ C[i][j] = 1000 \; when \; j\%10 = 0 \\ A[i][k] = 1000 \; when \; k\%10 = 0 \end{cases}$$

Algorithm	Order	Misses	Memory Requests
ijk, jik	C, A, B	$n^2 \times \left[\frac{1}{10} + \frac{n}{10B} + \frac{n}{10B}\right]$	$2n^3 + n^2$
ikj, kij	A, C, B	$n^2 \times \left[\frac{n}{10B} + \frac{1}{10} + \frac{n}{10B}\right]$	$2n^3 + n^2$
jki, kji	B, C, A	$n^2 \times \left[\frac{n}{10B} + \frac{n}{10B} + \frac{1}{10}\right]$	$2n^3 + n^2$

10000 X 10000					
Algorithm	A	B	C	totalRate	
ijk, jik	10^{10}	10^{10}	10^7	0.01	**1%**
kij, ikj	10^7	10^{10}	10^{10}	0.01	**1%**
jki, kji	10^{10}	10^7	10^{10}	0.01	**1%**

As you can see we reduced the miss rate by using blocking.

Part #3.

I have two versions from my algorithm, Simple version that do not use blocks, and Blocked version which uses blocks to calculate matrix multiplication.

Execution Time vs BlockSize							
Algorithm	Simple	16	32	64	128	256	512
ijk	320.9622	101.210450	94.711216	**87.671870**	89.006691	135.937647	163.330216
jik	388.7847	101.372003	94.456923	**89.937711**	92.065349	138.566756	147.763091
kij	119.6071	105.553809	98.242799	**85.802558**	**78.992826**	**92.916022**	100.252411
ikj	**116.0564**	**93.617857**	**90.663208**	88.773439	**79.900538**	98.504831	**87.094167**
jki	604.7584	139.701071	131.018723	128.479362	**120.625693**	200.837373	223.503842
kji	638.3867	138.953699	135.671798	120.441258	**118.435365**	190.319990	227.301362

Performance (GFlops) vs BlockSize							
Algorithm	Simple	16	32	64	128	256	512
ijk	0.0535	0.169744	**0.188094**	0.195956	0.193018	0.126381	0.105185
jik	0.0441	0.169474	0.180056	0.191020	0.186605	0.123983	0.116266
kij	0.1436	0.162759	0.167277	**0.200226**	**0.217486**	**0.184897**	0.171366
ikj	**0.148**	**0.183511**	0.156829	0.193525	0.215016	0.174406	**0.197256**
jki	0.0284	0.122976	0.141189	0.133717	0.142423	0.085541	0.076866
kji	0.0269	0.123637	0.142656	0.142641	0.145057	0.090268	0.075582

Maximum Difference vs BlockSize							
Algorithm	Simple	16	32	64	128	256	512
ijk	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
jik	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
kij	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
ikj	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
jki	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
kji	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

Analysis:

From algorithm perspective, type two algorithm (kij, ikj) has a better performance because it accesses elements in the same row. I supposed that kij has the best performance because it accesses the elements in the same row even for matrix A outside the inner loop, since matrix A has $O(n^2)$ the difference between kij and ikj is not considerable.

From BlockSize perspective, for type one algorithm (ijk, jik) BlockSize 64 has the best performance while for the other algorithms BlockSize 128 is the absolute winner. As you can see adding cache blocking improve performance, but somehow the numbers are close to each other and running a simulation several times can lead to different winner. The clear point is that using cache blocking increases performance.

Note: The performance on Tardis is not stable, for example for block size of 32 in my first Experiment, ijk got the best performance that was wired for me, so I ran simulation five more times, and surprisingly I noticed that it changes the winner between first four algorithms, maybe because the difference between them are a few seconds which is not noticeable, but it was interesting. So you may have other results!

You can see the order of algorithm for performance is kij>ikj>ijk>jik>jki> kji

Champions		
Algorithm	Optimal Timing	Optimal BlockSize
ijk	**87.671870**	64
jik	**89.937711**	64
kij	**78.992826**	128
ikj	**79.900538**	128
jki	**118.435365**	128
kji	**120.625693**	128

THE BEST PERFORMANCE WILL GOES TO KIJ ALGORTHIM WITH THE BLOCKSIZE OF 128! CONGRATULATIONS!

Part #4.

Execution Time vs Compiler for BlockSize 16								
Algorithm	O0 4.4	O1 4.4	O2 4.4	O3 4.4	O0 4.7	O1 4.7	O2 4.7	O3 4.7
ijk	37.893012	13.463167	14.470418	15.398180	42.009286	14.673069	**13.389047**	13.910281
jik	41.345496	**19.260286**	21.285943	20.827836	43.102472	20.495231	20.943850	21.653704
kij	32.722011	12.591931	11.056184	11.013003	32.831316	12.888266	**10.466415**	11.369516
ikj	30.553409	16.288962	12.676198	14.299871	31.175408	15.943236	**12.004149**	16.142780
jki	**41.476096**	94.299239	94.688433	101.736886	43.803887	98.114105	95.016552	95.983527
kji	**41.862634**	82.312419	80.837605	87.134141	45.845905	83.669785	85.618336	85.083224

Performance (GFlops) vs Compiler for BlockSize 16								
Algorithm	O0 4.4	O1 4.4	O2 4.4	O3 4.4	O0 4.7	O1 4.7	O2 4.7	O3 4.7
ijk	0.453378	1.276064	1.187241	1.115708	0.408954	1.170844	**1.283129**	1.235048
jik	0.415520	**0.891984**	0.807099	0.824851	0.398582	0.838237	0.820282	0.793392
kij	0.525025	1.364355	1.553870	1.559962	0.523277	1.332985	**1.641428**	1.511047
ikj	0.562290	1.054694	1.355286	1.201400	0.551071	1.077565	**1.431161**	1.064245
jki	**0.414211**	0.182185	0.181436	0.168866	0.392200	0.175101	0.180809	0.178988
kji	**0.410387**	0.208715	0.212523	0.197166	0.374731	0.205329	0.200656	0.201918

Maximum Difference vs Compiler for BlockSize 16								
Algorithm	O0 4.4	O1 4.4	O2 4.4	O3 4.4	O0 4.7	O1 4.7	O2 4.7	O3 4.7
ijk	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
jik	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
kij	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
ikj	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
jki	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
kji	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

Execution Time vs Compiler for BlockSize 32								
Algorithm	O0 4.4	O1 4.4	O2 4.4	O3 4.4	O0 4.7	O1 4.7	O2 4.7	O3 4.7
ijk	38.794821	**10.139402**	10.205301	10.562660	37.738560	11.026609	10.812506	11.206263
jik	39.731048	11.503693	**11.317565**	12.049233	36.436390	11.901268	13.605493	12.717188
kij	28.476009	9.975536	9.208324	**9.208047**	28.103408	9.415405	10.882588	10.423749
ikj	29.442511	15.590896	14.553845	14.797097	30.891497	16.869331	**14.023568**	17.169997
jki	**38.270154**	70.407767	82.201893	91.837362	38.979559	79.006377	89.833188	94.343839
kji	38.492346	76.235141	82.453689	95.768057	**38.250230**	83.186663	94.169920	96.355575

Performance (GFlops) vs Compiler for BlockSize 32								
Algorithm	O0 4.4	O1 4.4	O2 4.4	O3 4.4	O0 4.7	O1 4.7	O2 4.7	O3 4.7
ijk	0.442839	**1.694367**	1.683426	1.626472	0.455234	1.558037	1.588889	1.533060
jik	0.432404	1.493422	**1.517983**	1.425806	0.471503	1.443533	1.262716	1.350917
kij	0.603310	1.722200	1.865689	**1.865745**	0.611309	1.824655	1.578657	1.648147
ikj	**0.583506**	1.101917	1.180435	1.161030	0.556136	1.018408	**1.225071**	1.000575
jki	0.448910	0.244005	0.208996	0.187068	0.440740	0.217449	0.191242	0.182098
kji	0.446319	0.225354	0.208358	0.179390	**0.449144**	0.206522	0.182435	0.178297

Maximum Difference vs Compiler for BlockSize 32								
Algorithm	O0 4.4	O1 4.4	O2 4.4	O3 4.4	O0 4.7	O1 4.7	O2 4.7	O3 4.7
ijk	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
jik	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
kij	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
ikj	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

jki	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
kji	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

Execution Time vs Compiler for BlockSize 64

Algorithm	O0 4.4	O1 4.4	O2 4.4	O3 4.4	O0 4.7	O1 4.7	O2 4.7	O3 4.7
ijk	34.171971	8.762472	9.047565	9.087642	34.817642	9.417123	**8.618811**	9.515787
jik	35.747219	9.702485	**9.574083**	10.229009	34.029380	10.560879	10.865458	11.046892
kij	29.179270	9.625773	8.512562	8.205840	26.260760	**8.496770**	9.431076	10.016275
ikj	28.777594	15.803121	15.250358	**15.106993**	28.581303	16.672180	15.235426	15.896279
jki	36.480017	80.279461	90.107550	100.484677	**34.237677**	84.282590	95.628289	94.133947
kji	35.460803	76.598780	87.104136	96.269823	**34.585463**	83.464123	96.814175	91.962138

Performance (GFlops) vs Compiler for BlockSize 64

Algorithm	O0 4.4	O1 4.4	O2 4.4	O3 4.4	O0 4.7	O1 4.7	O2 4.7	O3 4.7
ijk	0.502747	1.960619	1.898839	1.890465	0.493424	1.824323	**1.993299**	1.805407
jik	0.480593	1.770667	**1.794414**	1.679524	0.504854	1.626746	1.581145	1.555177
kij	0.588770	1.784778	2.018178	2.093615	0.654203	**2.021929**	1.821623	1.715195
ikj	0.596988	1.087119	1.126522	**1.137213**	0.601088	1.030451	1.127626	1.080748
jki	0.470939	0.214001	0.190660	0.170970	**0.501783**	0.203837	0.179653	0.182505
kji	0.484475	0.224284	0.197234	0.178455	**0.496737**	0.205835	0.177452	0.186815

Maximum Difference vs Compiler for BlockSize 64

Algorithm	O0 4.4	O1 4.4	O2 4.4	O3 4.4	O0 4.7	O1 4.7	O2 4.7	O3 4.7
ijk	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
jik	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
kij	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
ikj	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
jki	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
kji	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

Execution Time vs Compiler for BlockSize 128

Algorithm	O0 4.4	O1 4.4	O2 4.4	O3 4.4	O0 4.7	O1 4.7	O2 4.7	O3 4.7
ijk	33.139180	**9.413595**	10.651993	10.573076	36.157414	10.154625	9.848832	11.005016
jik	34.903567	**10.565572**	11.437628	11.030151	38.286021	11.587819	10.644882	12.455841
kij	27.785718	**8.195379**	8.696238	7.915331	27.503487	8.348419	8.266873	8.336197
ikj	28.781579	15.405212	14.420867	**13.498125**	29.717416	14.528673	13.529626	12.870839
jki	**34.674965**	70.462689	82.766986	87.653281	37.761199	76.515961	80.483129	83.462515
kji	**34.085939**	77.561173	81.897450	80.844475	36.520133	74.393477	79.765624	77.754173

Performance (GFlops) vs Compiler for BlockSize 128

Algorithm	O0 4.4	O1 4.4	O2 4.4	O3 4.4	O0 4.7	O1 4.7	O2 4.7	O3 4.7
ijk	0.518416	**1.825006**	1.612831	1.624869	0.475141	1.691827	1.744356	1.561094
jik	0.492210	**1.626024**	1.502048	1.557537	0.448724	1.482580	1.613909	1.379262
kij	0.618299	**2.096287**	1.975552	2.170455	0.624643	2.057859	2.078158	2.060876
ikj	0.596905	1.115198	1.191320	**1.272760**	0.578108	1.182480	1.269796	1.334790
jki	**0.495455**	0.243815	0.207569	0.195998	0.454961	0.224527	0.213459	0.205839
kji	**0.504016**	0.221501	0.209773	0.212505	0.470422	0.230932	0.215379	0.220951

Maximum Difference vs Compiler for BlockSize 128

Algorithm	O0 4.4	O1 4.4	O2 4.4	O3 4.4	O0 4.7	O1 4.7	O2 4.7	O3 4.7
ijk	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
jik	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
kij	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
ikj	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
jki	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

kji	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

Execution Time vs Compiler for BlockSize 256

Algorithm	O0 4.4	O1 4.4	O2 4.4	O3 4.4	O0 4.7	O1 4.7	O2 4.7	O3 4.7
ijk	52.314855	25.592310	26.046878	27.735898	55.278113	**24.853323**	26.214148	27.888186
jik	54.035648	28.544143	27.372949	28.982880	53.230221	**26.723237**	27.884524	29.821319
kij	28.822770	8.760009	**8.088056**	8.391567	26.049203	8.650840	8.279317	9.706191
ikj	28.791199	16.256293	13.539854	14.904799	29.010043	14.633374	**12.888690**	13.741572
jki	65.014522	76.896825	79.012638	83.452252	**64.519370**	74.324769	81.537351	87.787638
kji	67.731844	78.631314	76.135712	77.768469	**66.826269**	76.438582	79.303590	84.196502

Performance (GFlops) vs Compiler for BlockSize 256

Algorithm	O0 4.4	O1 4.4	O2 4.4	O3 4.4	O0 4.7	O1 4.7	O2 4.7	O3 4.7
ijk	0.328394	0.671290	0.659575	0.619409	0.310790	**0.691250**	0.655366	0.616027
jik	0.317936	0.601870	0.627622	0.592759	0.322747	**0.642881**	0.616108	0.576094
kij	0.596052	1.961170	**2.124104**	2.047278	0.659516	1.985919	2.075034	1.769991
ikj	0.596706	1.056813	1.268837	1.152640	0.592204	1.174020	**1.332941**	1.250211
jki	0.264247	0.223415	0.217432	0.205865	**0.266275**	0.231146	0.210699	0.195698
kji	0.253645	0.218486	0.225648	0.220910	**0.257083**	0.224754	0.216634	0.204045

Maximum Difference vs Compiler for BlockSize 128

Algorithm	O0 4.4	O1 4.4	O2 4.4	O3 4.4	O0 4.7	O1 4.7	O2 4.7	O3 4.7
ijk	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
jik	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
kij	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
ikj	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
jki	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
kji	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

Execution Time vs Compiler for BlockSize 512

Algorithm	O0 4.4	O1 4.4	O2 4.4	O3 4.4	O0 4.7	O1 4.7	O2 4.7	O3 4.7
ijk	66.182422	**31.562257**	37.261185	32.546454	56.730174	33.965052	35.289226	35.019352
jik	57.489144	30.686346	**30.683797**	31.162392	57.998867	34.046616	32.149038	35.273889
kij	31.423632	10.426891	**9.602553**	9.957016	28.431979	11.247988	10.016257	10.677739
ikj	32.337379	15.376492	**12.942619**	14.019735	32.212238	16.862499	13.365487	13.694497
jki	84.389684	84.112159	**78.497251**	80.107508	78.800603	94.996964	85.783229	100.926886
kji	90.650917	81.115537	86.469938	85.121484	**71.832878**	85.077794	83.494800	97.402381

Performance (GFlops) vs Compiler for BlockSize 512

Algorithm	O0 4.4	O1 4.4	O2 4.4	O3 4.4	O0 4.7	O1 4.7	O2 4.7	O3 4.7
ijk	0.259584	**0.544317**	0.461066	0.527857	0.302835	0.505810	0.486830	0.490582
jik	0.298837	0.559854	**0.559900**	0.551301	0.296210	0.504598	0.534382	0.487042
kij	0.546718	1.647650	**1.789094**	1.725403	0.604245	1.527373	1.715199	1.608943
ikj	0.531270	1.117281	**1.327387**	1.225406	0.533334	1.018821	1.285390	1.254509
jki	0.203578	0.204250	**0.218860**	0.214460	0.218017	0.180847	0.200271	0.170221
kji	0.189517	0.211795	0.198680	0.201828	**0.239164**	0.201931	0.205760	0.176380

Maximum Difference vs Compiler for BlockSize 128

Algorithm	O0 4.4	O1 4.4	O2 4.4	O3 4.4	O0 4.7	O1 4.7	O2 4.7	O3 4.7
ijk	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
jik	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
kij	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
ikj	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
jki	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
kji	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

Analysis:

Champions				
BlockSize	Algorithm	Optimal Timing	Compiler	flag
16	kij	10.466415	GCC 4.7.2	-O2
32	kij	9.208047	GCC 4.4.7	-O2
64	kij	8.496770	GCC 4.7.2	-O1
128	kij	8.195379	GCC 4.4.7	-O1
256	**kij**	**8.088056**	**GCC 4.4.7**	**-O2**
512	kij	9.602553	GCC 4.4.7	-O2

THE BEST PERFORMANCE WILL GOES TO KIJ ALGORTHIM WITH THE BLOCKSIZE OF 256 WITH REGISTER BLOCKING [2x2] COMPILED WITH GCC 4.4.7 WITH -O2 FLAG!
CONGRATULATIONS

The result in this part was really interesting for me, Register Blocking is awesome, it improved the performance almost 80 times. The worst performance was for kji without blocking, but by adding both cache blocking, register blocking [block size = 256 and Registers 2X2], we reduced the computation time from 638.3867 to 8.08. Incredible!

Using higher version of compiler and optimization flags can helps performance, but I think in my case it the default version works better, maybe because I wrote my code efficiently, so compiler optimization just made things worse, or maybe because of TARDIS, or even the value of Matrices' elements I do not get the best performance from GCC 4.7.2 -O4. [I do not know why?!!!] but I reached to an Interesting conclusion: **Forget about compiler Optimization, take HPC course and do your homework to get the best performance.**

Note: For register blocking I used [2X2] instead of [3X3] with had better performance by virtue of two reason:

- [3X3] has more terms, longer, takes longer time from me and is error prone [Can be headache to debug]
- [3X3] is not aligned with 2048, I had to add more elements to increase the dimension which could lead to a different matrix size.

How to Compile

To change the compiler:

```
module purge
module load gcc-4.7.2
gcc -O4 multBlockedV2.c -lrt -lm  -o  multBlockedV4_O
```

For default compile without flags:

```
make clean
make
```

How to run?

```
./multSimple 2048
./multBlocked 2048 128
./multBlockedV2 2048 128
./multBlockedV4_O4 2048 128
```

Stop Typing! Do not run code on the head node.

```
qsub multSimple.job
qsub multBlocked.job
qsub multBlockedV2.job
```

you can change the job with your desired input!

Sample Output:

```
BlockSize: 32
MatrixSize: 2048

*******************************************
TEST ijkBlocked: runTime = 94.999899 s    94999898532 ns    number of FloatingPoint Operations = 17179869184    GFLOPS = 0.180841
TEST jikBlocked: runTime = 95.536401 s    95536401069 ns    number of FloatingPoint Operations = 17179869184    GFLOPS = 0.179825
TEST ikjBlocked: runTime = 98.438184 s    98438184039 ns    number of FloatingPoint Operations = 17179869184    GFLOPS = 0.174524
TEST kijBlocked: runTime = 93.083403 s    93083403087 ns    number of FloatingPoint Operations = 17179869184    GFLOPS = 0.184564
TEST jkiBlocked: runTime = 133.196697 s   133196696914 ns   number of FloatingPoint Operations = 17179869184    GFLOPS = 0.128981
TEST kjiBlocked: runTime = 138.142716 s   138142716344 ns   number of FloatingPoint Operations = 17179869184    GFLOPS = 0.124363

*******************************************
Maximum Difference 2 for Experiment jikBlocked : 0.000000
Maximum Difference 2 for Experiment ikjBlocked : 0.000000
Maximum Difference 2 for Experiment kijBlocked : 0.000000
Maximum Difference 2 for Experiment jkiBlocked : 0.000000
Maximum Difference 2 for Experiment kjiBlocked : 0.000000

*******************************************
```

CS 211 High Performance Computing Project 3

Parallel Sieve of Eratosthenes for Finding All Prime Numbers within 10^{10}

Due at 11:59 PM on Nov 9th, 2015

Part 1: Modify the parallel Sieve of Eratosthenes program in class so that the program does **NOT** set aside memory for even integers.

Part 2: Modify the parallel Sieve of Eratosthenes program in **Part 1** so that each process of the program finds its own sieving primes via local computations instead of broadcasts.

Part 3: Modify the parallel Sieve of Eratosthenes program in **Part2** so that the program can have a more effective use of caches.

Use your program to find all prime numbers within 10^{10}. **Output the total number of prime numbers within 10^{10} and the program execution time (i.e., maximum time of all processes used in the MPI program).** Benchmark your program on TARDIS with 32 (1 node), 64(2 nodes), 128(4 nodes), and 256 (8 node) cores to see whether your execution time is reduced by half or not when double the number of computing cores. Compare the execution time of each version of your program to see how different designs affect the execution time of your program. Note that, in syllabus, we emphasize for **ALL** homework assignments: "Please make sure that your programs are properly documented and indented. Provide instructions on how to run your programs, give example runs, **and analyze your results."**

CS 211: High Performance Computing Project 3
Parallel Sieve of Eratosthenes
Shahriyar Valielahi Roshan – 861241935 – svali003@ucr.edu

Introduction

Part #0.

Before modifying the Sieve algorithm, I ran the default code that you give us. You can find the results in the PART0 folder. In the simple form, we divide the large range into few processes, so we can make calculation in parallel, and also because of memory limitation, we cannot fit all number in the range, in memory, so dividing them into subranges can benefit us.

Part #1.

Each process is in charge of finding prime numbers in a specific subrange between $[2,2^{10}]$. As you know even numbers except 2 are not primes, so simply we remove them, so we do not have to dedicate extra memory to even numbers. We simply change the low_val and high_val to new odd values starting from 3, and the size for each process is half. And, I find the mapping formula to calculate local and global indexes, and mark prime numbers.

Part #2.

In this part, we removed the broadcast function, broadcasting means communication between processes that can run on different cores that cause delays which is relatively large. The main idea behind this part is that calculating primes between 3 to sqrt(n) is faster than broadcasting. So we introduce a new array called the primearray_marked with size of sqrt(n), and we calculate prime numbers in each process using a series of calculation! [Really hard to debug!]

Part #3.

In this part we want to increase cache hit rate, and improve performance by using cache efficiently. Our goal is to divide subranges for each process into blocks with the size of our cache. We calculate prime numbers in each block one by one to get the best performance. An important point in this part is that we should find the best cache size. I have tried cache sizes between 2 to 2^{30} and find out the cache size of 512k and 1024k results the best performance. Actually, 1024k is a little bit faster, but the difference is really neglectable.

Results

Parallel Sieve of Eratosthenes				
Program	32 (1 * 32)	64 (2 * 32)	128 (4 * 32)	256 (8 * 32)
Sieve0	32.796226	17.291934	14.116973	7.1752947
Sieve1	16.595538	8.9473116	7.0491083	3.597747
Sieve2	15.702535	7.8506176	6.8844335	3.3548285
Sieve3	6.2862165	3.1522029	1.584485	**0.8087525**

Table 1 – Running time for each algorithm

Parallel Sieve of Eratosthenes				
Program	32 (1 * 32)	64 (2 * 32)	128 (4 * 32)	256 (8 * 32)
Sieve0	1	1.89662	2.323177	4.570715
Sieve1	1.976207	3.665484	4.652535	9.115768
Sieve2	2.088594	4.177535	4.763823	9.775828
Sieve3	5.217165	10.40422	20.69835	**40.55162**

Table 2 – Speed Up for each algorithm

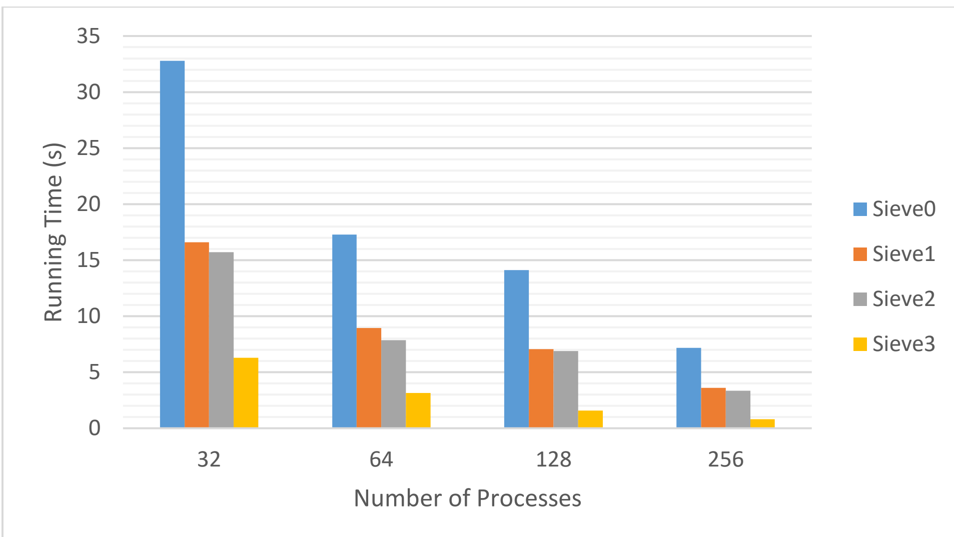


Chart 1 – Running time for each program

As you can see in the table 1 and chart 1, we made reduced the computation time from 32.79s to 0.80s which means 40x faster! Good job: D

You can see the speed up from the base program [sieve 0 with 32 processes] in the table 2. As you can see in chart 2 to 4, we have the speed up for doubling number of processes, but actually our improvement is less that predicted. We expected to get 2x speed up by doubling number of cores, but the reality is that we cannot achieve to this goal, because of communication between cores and poor caching except in sieve3. As you can see we almost got to the goal in the sieve 3 by removing extra communication between cores, and using cache efficiently.

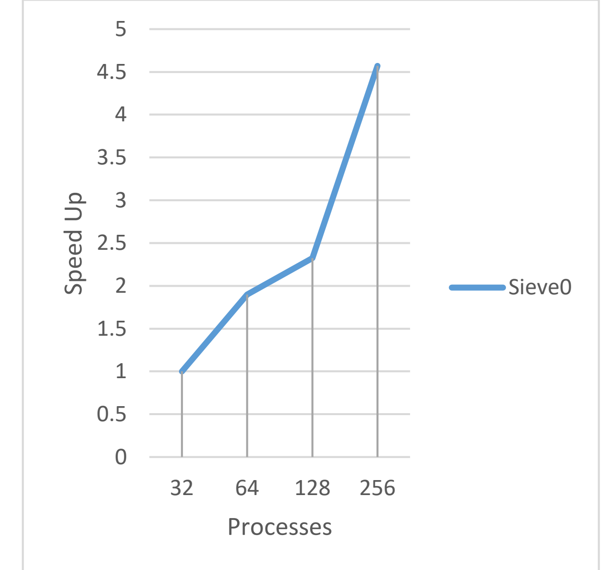


Chart 1 – Speedup
Doubling number of processes

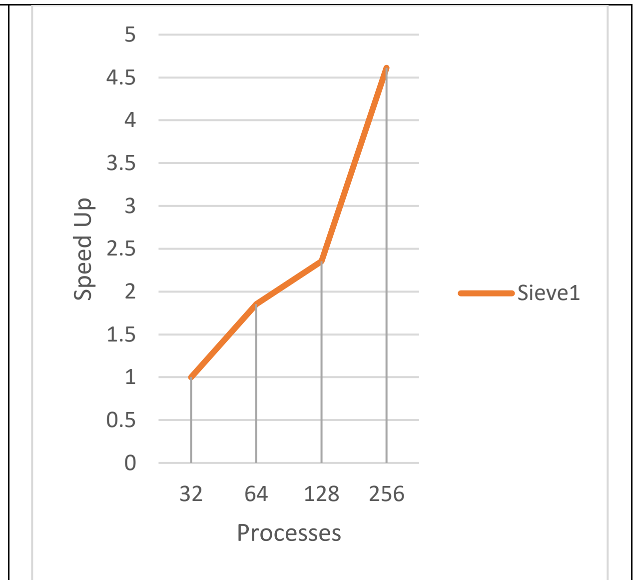


Chart 2 – Speedup
Doubling number of processes

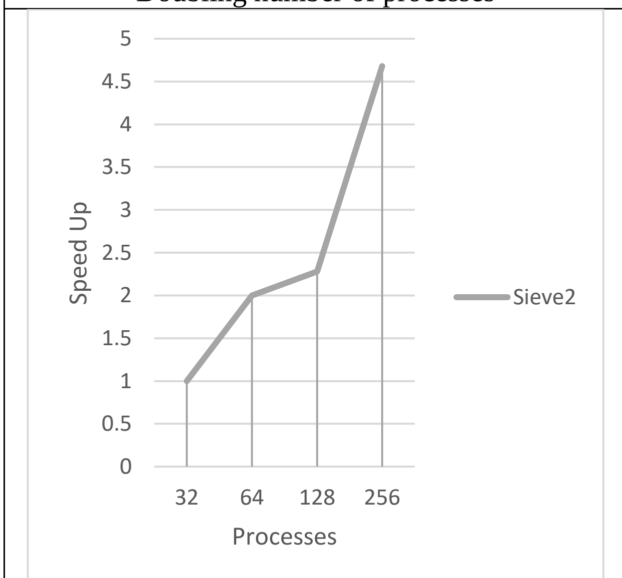


Chart 3 – Speedup
Doubling number of processes

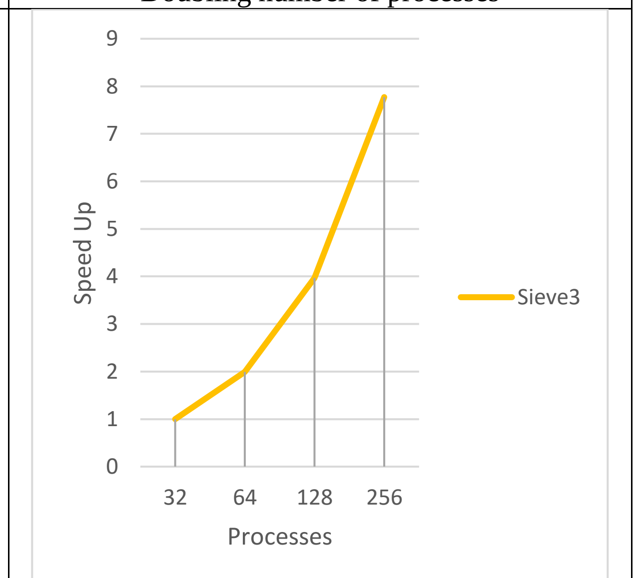


Chart 4 – Speedup
Doubling number of processes

You can see the speed up from the base program [sieve 0 with 32 processes] in the table 2. As you can see in chart 2 to 4, we have the speed up for doubling number of processes, but actually our improvement is less that predicted. We expected to get 2x speed up by doubling number of cores, but the reality is that we cannot achieve to this goal, because of communication between cores and poor caching except in sieve3. As you can see we almost got to the goal in the sieve 3 by removing extra communication between cores, and using cache efficiently. In the chart 5, and 6 you can find the speed up and Normalized running time for each program as we increase number of processes and cores.

Using blocks for best cache efficiency has the most speed-up effect, after that elimination of even numbers has the most positive effect. Removing broadcasting it-self has a minor effect on performance without proper cache utilization, but as we improve cache utilization, we get a great performance.

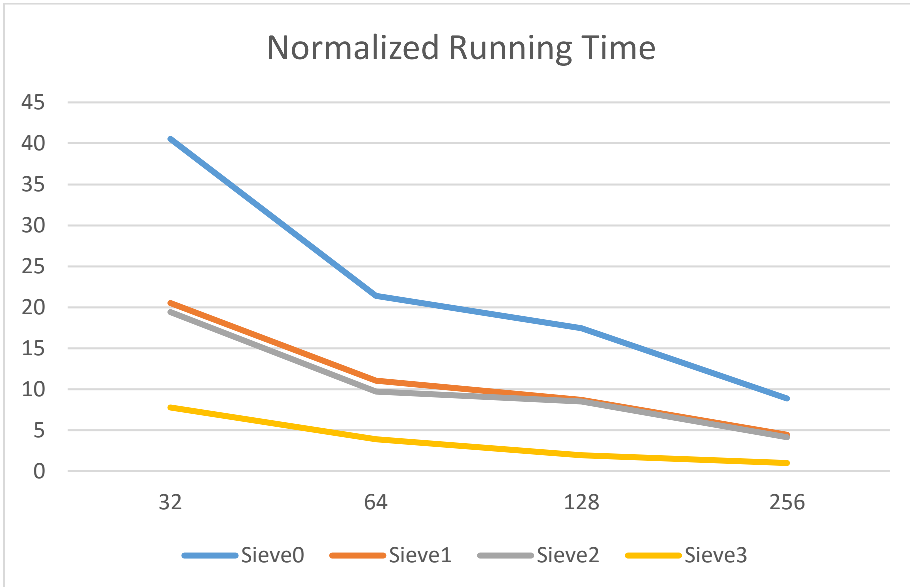


Chart 5 – Normalized Running Time

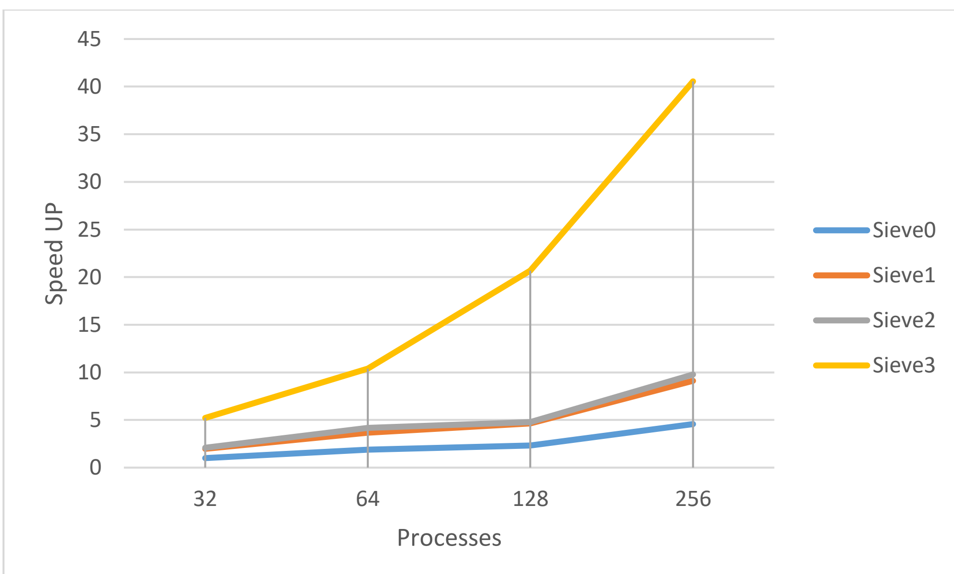


Chart 6 – Speedup

In the program sieve3, to use cache efficiently, we should find the best cache Size [block size], I have tried block sizes from 2^{11} to 2^{30}, as you can see in Chart 7 and chart 8, the best cache size is 1024k, albeit 512k has close running time.

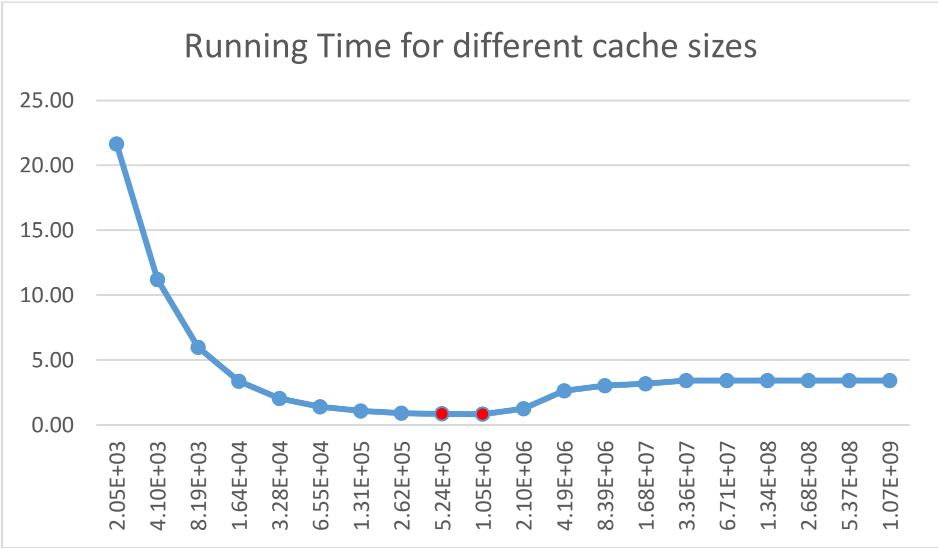


Chart 7 – Running time for different cache sizes

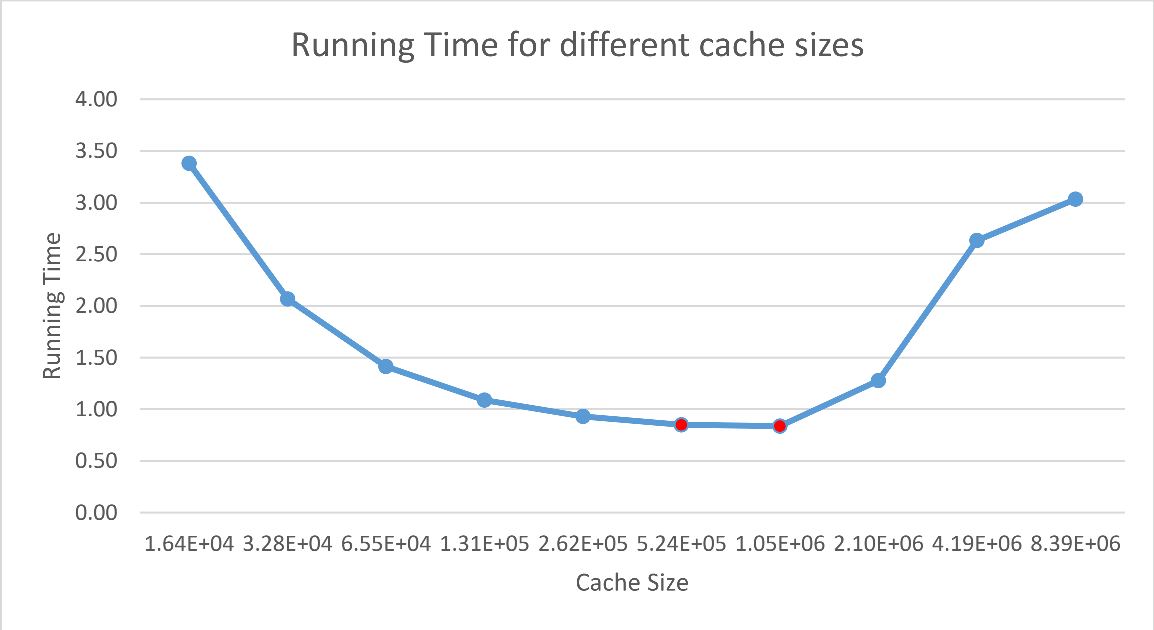


Chart 8 – Running time for different cache sizes [better zoom]

WE HAVE 455052511 PRIME NUMBER BETWEEN 2 to 10^{10}

How to Compile

```
module purge
module load mvapich2-1.9a2/gnu-4.6.2
module load gcc-4.6.2

mpicc sieve3.c -o sieve3
```

How to Run

qsub X.job

Replace [0, 1, 2, 3] in X

sieveX_128.job
sieveX_256.job
sieveX_32.job
sieveX_64.job

You can find the job content here:

```
#!/bin/sh
#PBS -l nodes=1:ppn=32,walltime=100:00:00
#PBS -N PART0_10p10_32

module purge
module load mvapich2-1.9a2/gnu-4.6.2
module load gcc-4.6.2

cd $PBS_O_WORKDIR

mpirun ./sieve0 10000000000 >> PART0_10p10_32.drsvr
mpirun ./sieve0 10000000000 >> PART0_10p10_32.drsvr
mpirun ./sieve0 10000000000 >> PART0_10p10_32.drsvr
mpirun ./sieve0 10000000000 >> PART0_10p10_32.drsvr
mpirun ./sieve0 10000000000 >> PART0_10p10_32.drsvr
```

Please note that you can open *_10p10_*.drsvr with gedit to see results for running the simulation 5 times. You can remove extra runs if you want!

Personally, I ran each simulation 10 times to get the most reliable result from the average of 10 runs.

Thank you