Operating Systems EE431L

Complex Engineering Problem - Report

Parallelized implementation of Gauss-Seidel method using OpenMP and POSIX Threads

Submitted by:

2016-EE-189 Muhammad Kamil

Submitted to:

Mr. M. Usama Zubair

CEP Report
Parallelized implementation of Gauss-Seidel method using OpenMP and POSIX Threads

Muhammad Kamil *2016-EE-189* *kamiljaved98@gmail.com*

December 2020

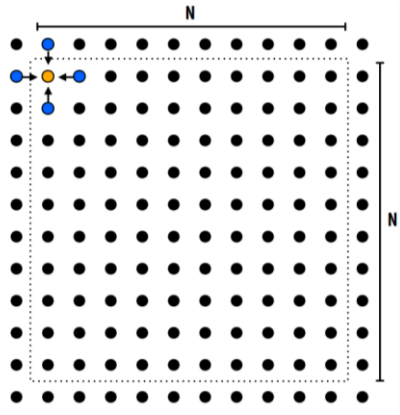
Department of Electrical Engineering
University of Engineering & Technology, Lahore

Table of Contents

Section I
Problem Statement

The task is to solve a partial differential equation on (N+2) × (N+2) grid (2D), in a parallel fashion, that is, to perform Gauss-Seidel sweeps over the grid until convergence. For a cell at point [i, j], the new value is calculated using the formula:

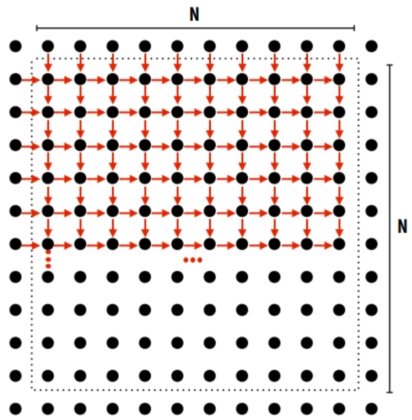
$$\mathrm{A}[\mathrm{i},\mathrm{j}] \;=\; 0.2 \;\times (\,\mathrm{A}[\mathrm{i},\mathrm{j}] +\; \mathrm{A}[\mathrm{i},\mathrm{j}-1] +\; \mathrm{A}[\mathrm{i}-1,\mathrm{j}] +\; \mathrm{A}[\mathrm{i},\mathrm{j}+1] +\; \mathrm{A}[\mathrm{i}+1,\mathrm{j}]\,)$$



The simplest method to do a sweep would be to start at the first cell (indicated in yellow above), and move through the row, and then onto the next row. Repeat the process until convergence occurs.

To implement this in a parallelized method, some dependencies (per iteration over entire gird) must be taken care of, enumerated below.

1. Each row element depends on element to left.
2. Each row depends on previous row.



In this report, two approaches, namely (1) Red-Black Cells Approach, and (2) Anti-Diagonals Approach, are discussed, and the results of their implementation have been shared.

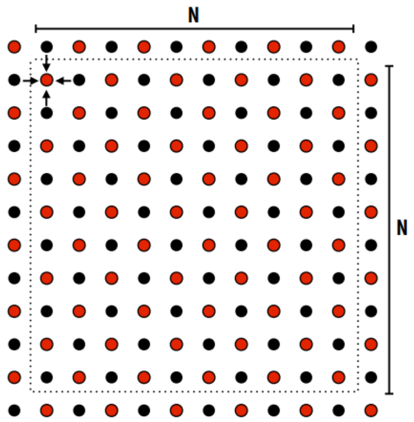
Section II.1

RED-BLACK CELLS APPROACH

Sequential Implementation

Methodology

In this approach, the alternate grid cells are bunched into red and black groups, as follows:

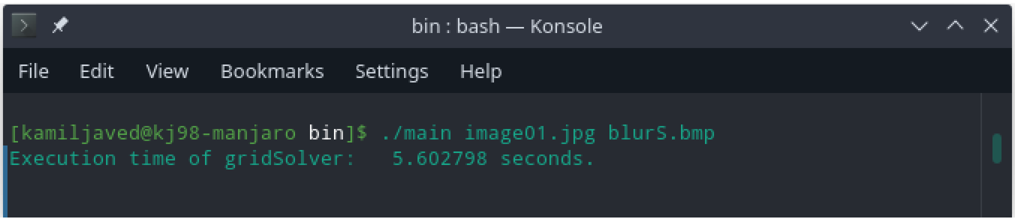


First, update all red cells. When done updating red cells, update all black cells (respect dependency on red cells). Repeat until convergence.

The following pseudocode sums up the process (for sequential approach):

```
convert image to grayscale

begin loop1

  diff = 0

  for each red cell 'redc'
    tmp = redc
    redc = 0.2 * (redc + top_cell + bottom_cell + left_cell + right_cell)
    diff = diff + absolute(redc - tmp)
  endfor

  for each black cell 'blackc'
    tmp = blackc
    blackc = 0.2 * (blackc + top_cell + bottom_cell + left_cell + right_cell)
    diff = diff + absolute(blackc - tmp)
  endfor

if (diff/num_cells < threshold): end loop1
```

Results



$$\text{Execution Time} = 5.6 \text{ seconds}$$

Original Image

Resultant Image

Section II.2.a RED-BLACK CELLS APPROACH

Parallel Implementation with OpenMP using “pragma omp critical”

Parallelization of Red-Black Cells Approach

This approach can be parallelized by updating the red cells in parallel and the black cells in parallel. First, update all red cells in parallel. When done updating red cells, update all black cells in parallel (respect dependency on red cells). Repeat until convergence.

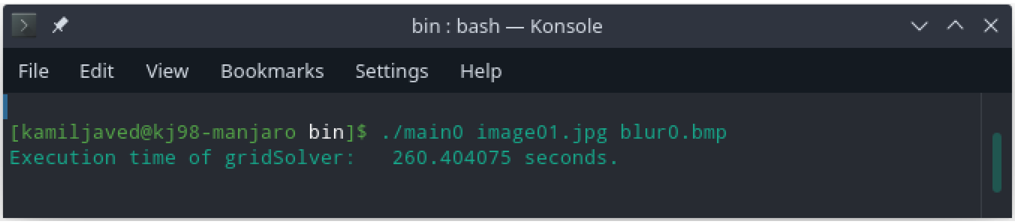
Methodology

The OpenMP library is used to update cells of one color in parallel in threads, using interleaved assignment (starting at thread_num for a thread, rows after step of thread_count assigned to that thread). Thread count was set to 4.

To avoid race condition on updating of the variable ‘diff’, the line(s) updating the ‘diff’ variable were wrapped inside a critical block. The following pseudocode sums up the (paralellized) process:

```
begin loop1

  diff = 0

  #pragma omp parallel for
  for each row
    for each red cell 'redc' in row
      tmp = redc
      redc = 0.2 * (redc + top_cell + bottom_cell + left_cell + right_cell)
      #pragma omp critical {  diff = diff + absolute(redc - tmp)  }
    endfor
  endfor

  #pragma omp barrier

  #pragma omp parallel for
  for each row
    for each black cell 'blackc'
      tmp = blackc
      blackc = 0.2 * (blackc + top_cell + bottom_cell + left_cell + right_cell)
      #pragma omp critical {  diff = diff + absolute(blackc - tmp)  }
    endfor
  endfor

if (diff/num_cells < threshold): end loop1
```

Results



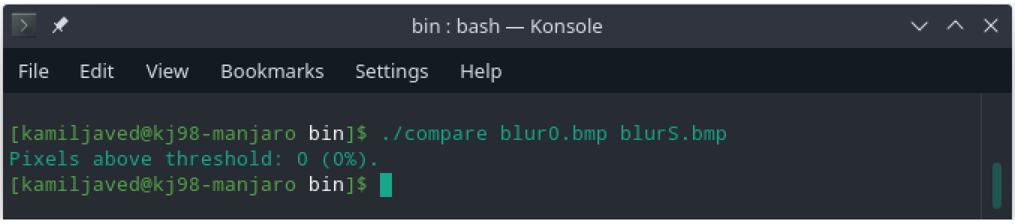
$$\text{Execution Time} = 260.4 \text{ seconds}$$

Speedups

$$\text{Speedup (compared to sequential method)} = \frac{5.6}{260.4} = 0.02$$

There is no speedup, in fact the method took longer time than sequential approach. This is because each thread must wait in a queue (in each iteration) to be able to update the ‘diff’ variable (as that is a critical section), i.e. the behavior effectively becomes sequential. And hence, the synchronization cost outweighs any benefit achieved by parallelizing here (using “pragma omp critical”).

Correctness



Section II.2.b RED-BLACK CELLS APPROACH

Parallel Implementation with OpenMP using per-thread 'diff' (no padding)

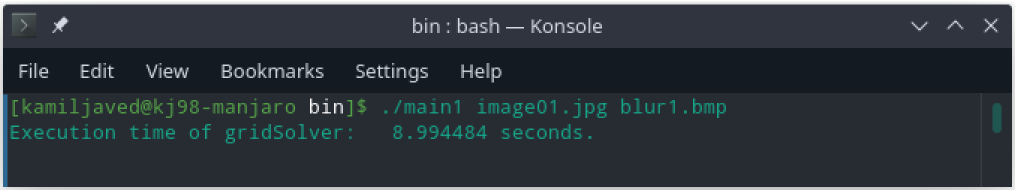
Methodology

To avoid the blockage occurring at identified critical section i.e. updating 'diff', each thread is provided with a variable that it can use to store its personal diff-sum. Hence, an array diffs[thread_count] is kept, and the actual diff value is computed at the end of entire-grid iteration by simply summing all the per-thread diffs in the diffs[] array.

The following pseudocode sums up the process:

```
begin loop1

   diffs[thread_count]

   initialize all elements of diffs[] to 0

   #pragma omp parallel for
   for each row
      for each red cell 'redc'
         tmp = redc
         redc = 0.2 * (redc + top_cell + bottom_cell + left_cell + right_cell)
         diffs[thread_num] = diffs[thread_num] + absolute(redc - tmp)
      endfor
   endfor

   #pragma omp barrier

   #pragma omp parallel for
   for each row
      for each black cell 'blackc'
         tmp = blackc
         blackc = 0.2 * (blackc + top_cell + bottom_cell + left_cell + right_cell)
         diffs[thread_num] = diffs[thread_num] + absolute(blackc - tmp)
      endfor
   endfor

   diff = sum of all elements in diffs[]

if (diff/num_cells < threshold): end loop1
```

Results



$$\text{Execution Time} = 8.99 \text{ seconds}$$

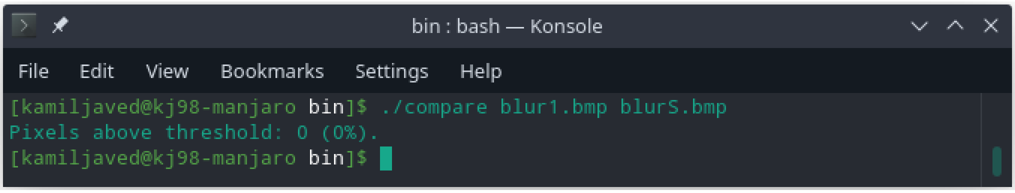
Speedups

$$\text{Speedup (compared to sequential method)} = \frac{5.6}{8.99} = 0.62$$

There is no speedup, in fact the method still took longer time than sequential approach. The most probable cause of the delays is false-sharing here. False sharing is a term which applies when threads unwittingly impact the performance of each other while modifying independent variables sharing the same cache line.

False sharing occurs when another thread has written to an address within the cache line which causes the current thread's cache line to be flushed, and that other thread has written to a different address from the one that the current thread is referencing. Cache misses due to false sharing occur because the cache hardware operates at a cache-line granularity, not a data-word granularity.

Correctness



Section II.2.c RED-BLACK CELLS APPROACH

Parallel Implementation with OpenMP using per-thread 'diff' (with padding)

Methodology

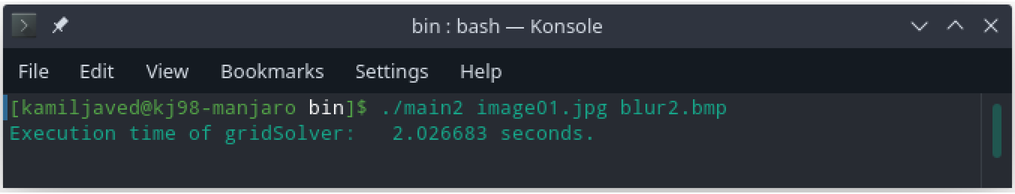
The false-sharing issue occurring due to using single array for per-thread diff without padding (would be contained within single cache line) can be avoided by changing the relevant data structures so that the different addresses referenced in each thread are on different cache lines.

The cache line-size for the CPU used was determined to be 64 Bytes. Hence, a padding of 60 Bytes (64 - sizeof(float) i.e. 4 = 60) was added along-with each element of the diffs array.

The following pseudocode sums up the process:

```
struct diffblk {
    float diff;
    char pad[60];
};

begin loop1

   struct diffblk diffs[thread_count]
   initialize all of diffs[].diff to 0

   #pragma omp parallel for
   for each row
      for each red cell 'redc'
         tmp = redc
         redc = 0.2 * (redc + top_cell + bottom_cell + left_cell + right_cell)
         diffs[thread_num].diff = diffs[thread_num].diff + absolute(redc - tmp)
      endfor
   endfor

   #pragma omp barrier

   #pragma omp parallel for
   for each row
      for each black cell 'blackc'
         tmp = blackc
         blackc = 0.2 * (blackc + top_cell + bottom_cell + left_cell + right_cell)
         diffs[thread_num].diff = diffs[thread_num].diff + absolute(blackc - tmp)
      endfor
   endfor

   diff = sum of all diffs[].diff

if (diff/num_cells < threshold): end loop1
```

Results



$$\text{Execution Time} = 2.03 \text{ seconds}$$

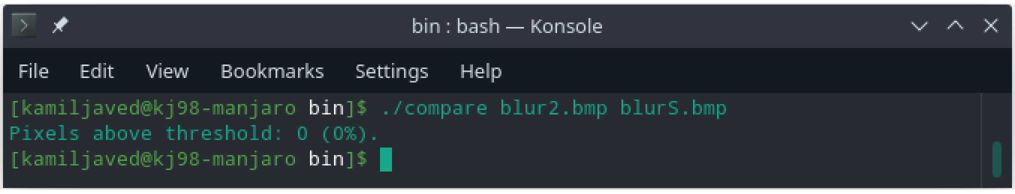
Speedups

$$\text{Speedup (compared to sequential method)} = \frac{5.6}{2.03} = 2.76$$

$$\text{Speedup (compared to per\_thread diff with no padding)} = \frac{8.99}{2.03} = 4.43$$

Speedup of 2.76 times is achieved compared to the sequential method. Considerable speedup is also achieved as compared to parallel approach using per-thread diff with no padding (false-sharing).

Correctness



Section II.3.a RED-BLACK CELLS APPROACH

Parallel Implementation with Pthreads using mutex lock

Methodology

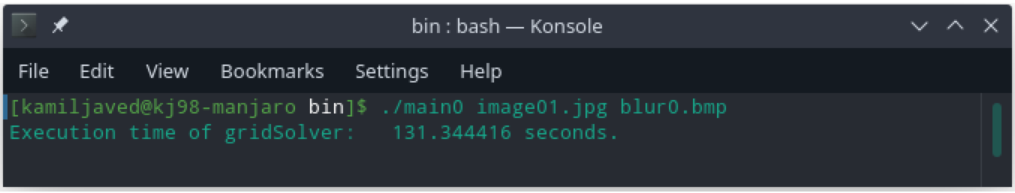
The Pthreads library is used to update cells of one color in parallel in threads, using interleaved assignment (starting at thread_num for a thread, rows after step of thread_count assigned to that thread). Thread count was set to 4.

To avoid race condition on updating of the variable 'diff', the line(s) updating the 'diff' variable were wrapped inside a critical block.

The following pseudocode sums up the process (for a thread):

```
diff = 0

begin loop1 in thread_num=i

   for each row, starting at i, step by thread_count
      for each red cell 'redc' in row
         tmp = redc
         redc = 0.2 * (redc + top_cell + bottom_cell + left_cell + right_cell)
         mutex_lock {  diff = diff + absolute(redc - tmp)  }
      endfor
   endfor

   barrier_wait()

   for each row, starting at i, step by thread_count
      for each black cell 'blackc' in row
         tmp = blackc
         blackc = 0.2 * (blackc + top_cell + bottom_cell + left_cell + right_cell)
         mutex_lock {  diff = diff + absolute(blackc - tmp)  }
      endfor
   endfor

  barrier_wait()

if i==1:
   (test) if (diff/num_cells < threshold): end loop1 (for all threads)
   else diff = 0
   barrier_wait()
else:
   barrier_wait()
```

Results



$$\text{Execution Time} = 131.3 \text{ seconds}$$

Speedups

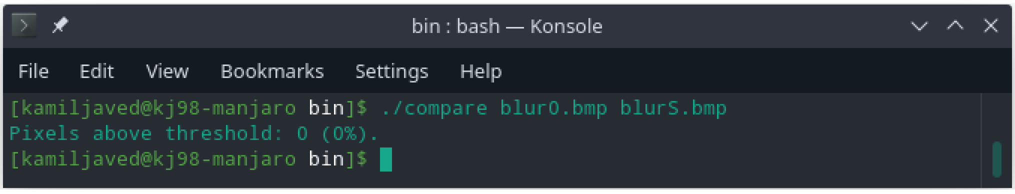
$$\text{Speedup (compared to sequential method)} = \frac{5.6}{131.3} = 0.04$$

$$\text{Speedup (compared to OpenMP pragma omp critical method)} = \frac{260.4}{131.3} = 1.98$$

There is no speedup compared to sequential method, in fact the method took longer time than sequential approach. This is because each thread must wait in a queue (in each iteration) to be able to update the ‘diff’ variable (as that is a critical section enclosed in mutex lock), i.e. the behavior effectively becomes sequential. And hence, the synchronization cost outweighs any benefit achieved by parallelizing here (using mutex lock).

The Pthread method (with mutex lock) has performed better than OpenMP approach using pragma omp critical, giving a speedup of 1.98 times compared to OpenMP.

Correctness



Section II.3.b RED-BLACK CELLS APPROACH

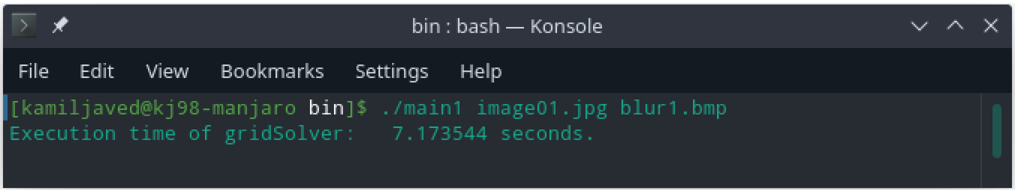
Parallel Implementation with Pthreads using per-thread 'diff' (no padding)

Methodology

To avoid the blockage occurring at identified critical section i.e. updating 'diff', each thread is provided with a variable that it can use to store its personal diff-sum. Hence, an array diffs[thread_count] is kept, and the actual diff value is computed at the end of entire-grid iteration by simply summing all the per-thread diffs in the diffs[] array. The following pseudocode sums up the process:

```
diff = 0

diffs[thread_count]

begin loop1 in thread_num=i

   diffs[i-1] = 0

   for each row, starting at i, step by thread_count
      for each red cell 'redc' in row
         tmp = redc
         redc = 0.2 * (redc + top_cell + bottom_cell + left_cell + right_cell)
         diffs[i-1] = diffs[i-1] + absolute(redc - tmp)
      endfor
   endfor

   barrier_wait()

   for each row, starting at i, step by thread_count
      for each black cell 'blackc' in row
         tmp = blackc
         blackc = 0.2 * (blackc + top_cell + bottom_cell + left_cell + right_cell)
         diffs[i-1] = diffs[i-1] + absolute(blackc - tmp)
      endfor
   endfor

  barrier_wait()

if i==1:
   diff = sum of all elements in diffs[]
   (test) if (diff/num_cells < threshold): end loop1 (for all threads)
   else diff = 0
   barrier_wait()
else:
   barrier_wait()
```

Results



$$\text{Execution Time} = 7.17 \text{ seconds}$$

Speedups

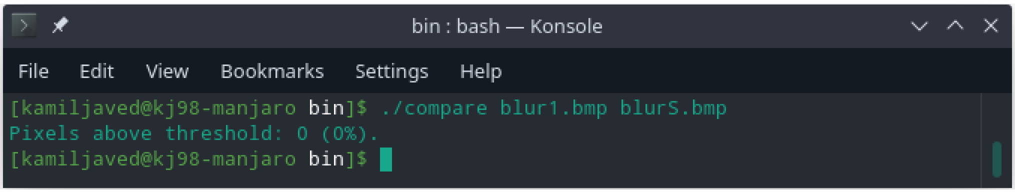
$$\text{Speedup (compared to sequential method)} = \frac{5.6}{7.17} = 0.78$$

$$\text{Speedup (compared to OpenMP per\_thread diff no\_padding method)} = \frac{8.99}{7.17} = 1.25$$

There is no speedup compared to sequential method, in fact the method still took longer time than sequential approach. The most probable cause of the delays is false-sharing here, as explained in Section II.1.b (speedups).

The Pthread method (with per-thread diff no-padding) has performed better than OpenMP approach using per-thread diff no-padding, giving a speedup of 1.25 times compared to OpenMP.

Correctness



Section II.3.c RED-BLACK CELLS APPROACH

Parallel Implementation with Pthreads using per-thread ‘diff’ (with padding)

Methodology

The false-sharing issue can be avoided by changing the relevant data structures so that the different addresses referenced in each thread are on different cache lines.

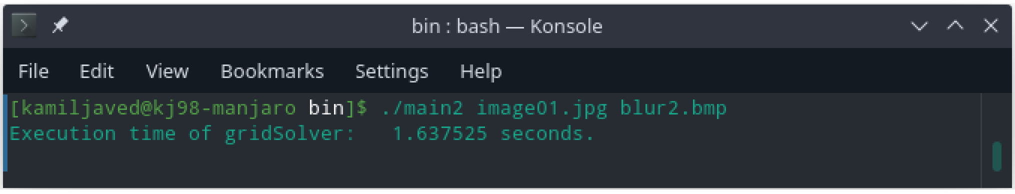
The cache line-size for the CPU used was determined to be 64 Bytes. Hence, a padding of 60 Bytes was used for each per-thread diff.

The following pseudocode sums up the process:

```
struct diffblk {
    float diff;
    char pad[60];
};

diff = 0

struct diffblk diffs[thread_count]

begin loop1 in thread_num=i

   diffs[i-1].diff = 0

   for each row, starting at i, step by thread_count
      for each red cell 'redc' in row
         tmp = redc
         redc = 0.2 * (redc + top_cell + bottom_cell + left_cell + right_cell)
         diffs[i-1].diff = diffs[i-1].diff + absolute(redc - tmp)
      endfor
   endfor

   barrier_wait()

   for each row, starting at i, step by thread_count
      for each black cell 'blackc' in row
         tmp = blackc
         blackc = 0.2 * (blackc + top_cell + bottom_cell + left_cell + right_cell)
         diffs[i-1].diff = diffs[i-1].diff + absolute(blackc - tmp)
      endfor
   endfor

  barrier_wait()
```

```
if i==1:
   diff = sum of all diffs[].diff
   (test) if (diff/num_cells < threshold): end loop1 (for all threads)
   else diff = 0
   barrier_wait()
else:
   barrier_wait()
```

Results



$$\text{Execution Time} = 1.64 \text{ seconds}$$

Speedups

$$\text{Speedup (compared to sequential method)} = \frac{5.6}{1.64} = 3.41$$

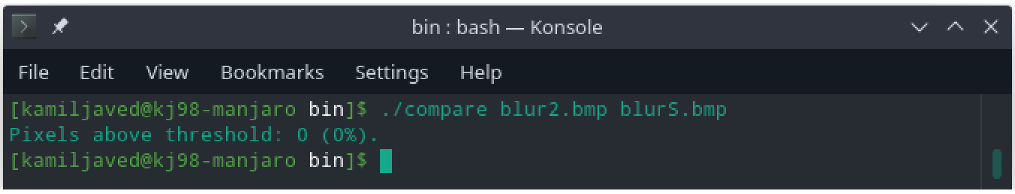
$$\text{Speedup (compared to per\_thread diff with no padding)} = \frac{7.17}{1.64} = 4.4$$

$$\text{Speedup (compared to OpenMP per\_thread diff with\_padding method)} = \frac{2.03}{1.64} = 1.24$$

Speedup of 3.41 times is achieved compared to the sequential method. Considerable speedup is also achieved as compared to (Pthread) parallel approach using per-thread diff with no padding (false-sharing).

Also, the Pthread method (with per-thread diff with-padding) has performed better than OpenMP approach using per-thread diff with-padding, giving a speedup of 1.24 times compared to OpenMP.

Correctness



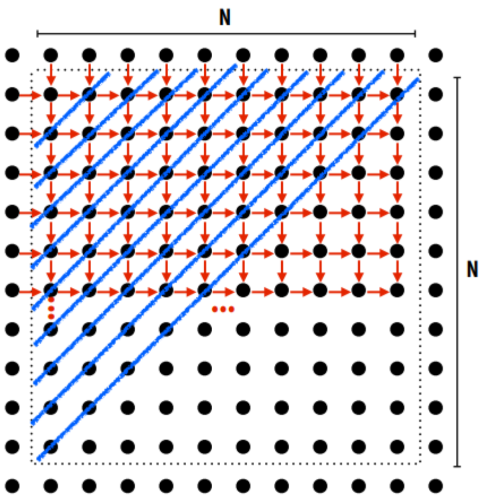
Section III.1

ANTI-DIAGONALS APPROACH

Sequential Implementation

Methodology

In this approach, the cells lying on an anti-diagonal are bunched into a group:

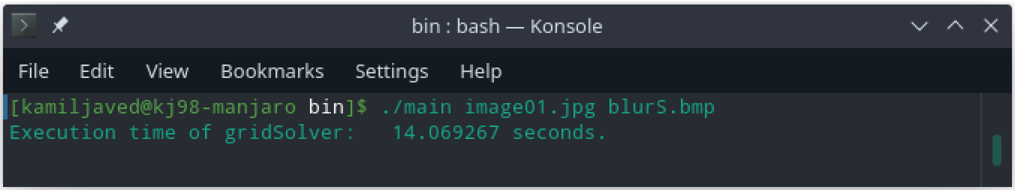


Update all cells in the first anti-diagonal, and move on to the next. Proceed until you reach the last anti-diagonal. Repeat until convergence.

The following pseudocode sums up the process (for sequential approach):

```
convert image to grayscale

begin loop1

  diff = 0

  for each anti-diagonal ‘ad’ in grid
    for each cell 'c' in ‘ad’
      tmp = c
      c = 0.2 * (c + top_cell + bottom_cell + left_cell + right_cell)
      diff = diff + absolute(c - tmp)
    endfor
  endfor

if (diff/num_cells < threshold): end loop1
```

Results



Execution Time = 14.1 seconds

Original Image

Resultant Image

Section III.2.a ANTI-DIAGONALS APPROACH

Parallel Implementation with OpenMP using "pragma omp critical"

Parallelization of Anti-Diagonals Approach

This approach can be parallelized by updating all the cells in a given anti-diagonal in parallel. After updating all cells in a given anti-diagonal in parallel, move on to the next anti-diagonal. Proceed until you reach the last anti-diagonal. Repeat until convergence.

Methodology

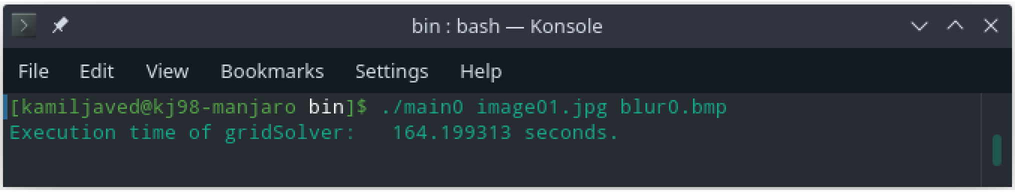
The OpenMP library is used to update cells on an anti-diagonal in parallel in threads, using interleaved assignment within an anti-diagonal (starting at thread_num for a thread, cells after step of thread_count assigned to that thread). Thread count was set to 4.

To avoid race condition on updating of the variable 'diff', the line(s) updating the 'diff' variable were wrapped inside a critical block.

The following pseudocode sums up the process:

```
begin loop1

  diff = 0

  for each anti-diagonal 'ad' in grid
    #pragma omp parallel for
    for each cell 'c' in 'ad'
      tmp = c
      c = 0.2 * (c + top_cell + bottom_cell + left_cell + right_cell)
      #pragma omp critical {
        diff = diff + absolute(c - tmp)
      }
    endfor
  endfor

if (diff/num_cells < threshold): end loop1
```

Results



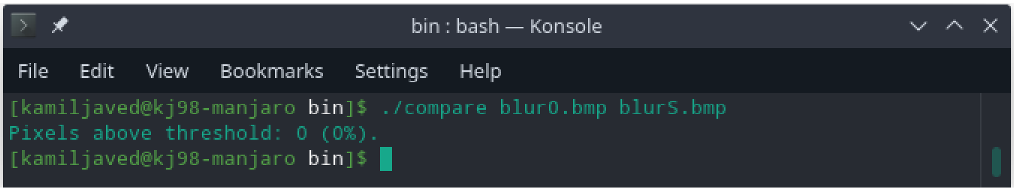
$$\text{Execution Time} = 164.2 \text{ seconds}$$

Speedups

$$\text{Speedup (compared to sequential method)} = \frac{14.1}{164.2} = 0.09$$

There is no speedup, in fact the method took longer time than sequential approach. This is because each thread must wait in a queue (in each iteration) to be able to update the 'diff' variable (as that is a critical section), i.e. the behavior effectively becomes sequential. And hence, the synchronization cost outweighs any benefit achieved by parallelizing here (using "pragma omp critical").

Correctness



Section III.2.b ANTI-DIAGONALS APPROACH

Parallel Implementation with OpenMP using per-thread 'diff' (no padding)

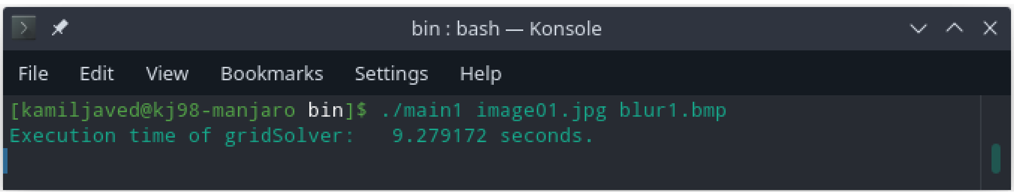
Methodology

To avoid the blockage occurring at identified critical section i.e. updating 'diff', each thread is provided with a variable that it can use to store its personal diff-sum. Hence, an array diffs[thread_count] is kept, and the actual diff value is computed at the end of entire-grid iteration by simply summing all the per-thread diffs in the diffs[] array.

The following pseudocode sums up the process:

```
begin loop1

  diffs[thread_count]

  initialize all elements of diffs[] to 0

  for each anti-diagonal 'ad' in grid
    #pragma omp parallel for
    for each cell 'c' in 'ad'
      tmp = c
      c = 0.2 * (c + top_cell + bottom_cell + left_cell + right_cell)
      diffs[thread_num] = diffs[thread_num] + absolute(c - tmp)
    endfor
  endfor

  diff = sum of all elements in diffs[]

if (diff/num_cells < threshold): end loop1
```

Results



Execution Time = 9.3 seconds

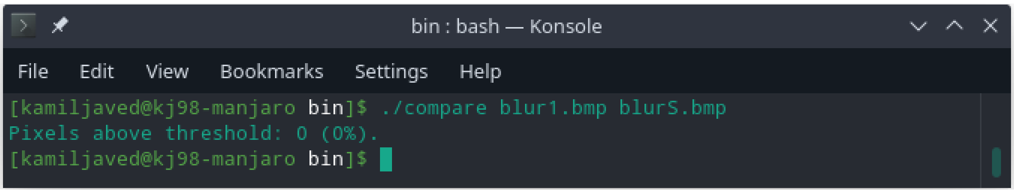
Speedups

$$\text{Speedup (compared to sequential method)} = \frac{14.1}{9.3} = 1.5$$

Speedup of 1.5 times is achieved compared to the sequential method, although further improvements can be made, as this implementation still has some avoidable delays.

The most probable cause of the delays is false-sharing here. False sharing is a term which applies when threads unwittingly impact the performance of each other while modifying independent variables sharing the same cache line. More about false-sharing has been discussed in Section II.1.b (speedups).

Correctness



Section III.2.c ANTI-DIAGONALS APPROACH

Parallel Implementation with OpenMP using per-thread 'diff' (with padding)

Methodology

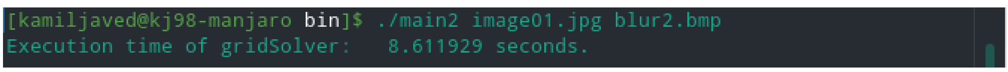
The false-sharing issue can be avoided by changing the relevant data structures so that the different addresses referenced in each thread are on different cache lines.

The cache line-size for the CPU used was determined to be 64 Bytes. Hence, a padding of 60 Bytes was used for each per-thread diff.

The following pseudocode sums up the process:

```
struct diffblk {
    float diff;
    char pad[60];
};

begin loop1

   struct diffblk diffs[thread_count]

   initialize all of diffs[].diff to 0

   for each anti-diagonal 'ad' in grid
      #pragma omp parallel for
      for each cell 'c' in 'ad'
         tmp = c
         c = 0.2 * (c + top_cell + bottom_cell + left_cell + right_cell)
         diffs[thread_num].diff = diffs[thread_num].diff + absolute(c - tmp)
      endfor
   endfor

   diff = sum of all diffs[].diff

if (diff/num_cells < threshold): end loop1
```

Results



Execution Time = 8.6 seconds

Speedups

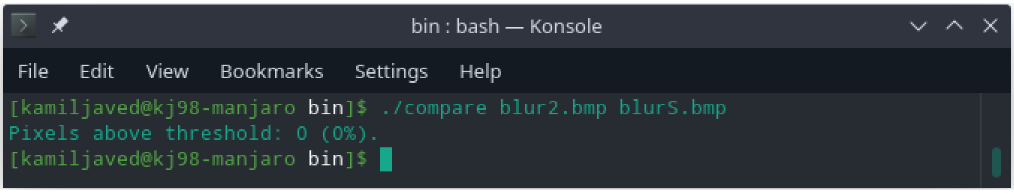
$$\text{Speedup (compared to sequential method)} = \frac{14.1}{8.6} = 1.64$$

$$\text{Speedup (compared to per\_thread diff with no padding)} = \frac{9.3}{8.6} = 1.1$$

Speedup of 1.64 times is achieved compared to the sequential method.

Compared to parallel approach using per-thread diff with no padding (false-sharing), speedup of 1.1 times is achieved.

Correctness



Section III.3.a ANTI-DIAGONALS APPROACH

Parallel Implementation with Pthreads using mutex lock

Methodology

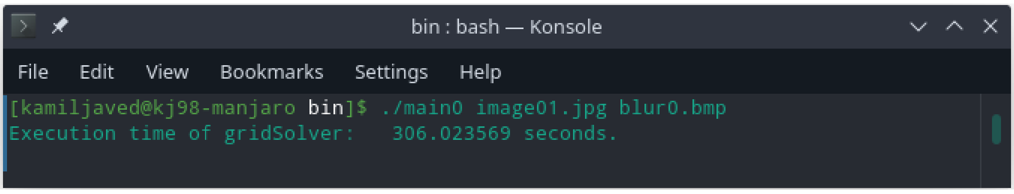
The Pthreads library is used to update cells on an anti-diagonal in parallel in threads, using interleaved assignment within an anti-diagonal (starting at thread_num for a thread, cells after step of thread_count assigned to that thread). Thread count was set to 4.

To avoid race condition on updating of the variable ‘diff’, the line(s) updating the ‘diff’ variable were wrapped inside a critical block.

The following pseudocode sums up the process (for a thread):

```
diff = 0

begin loop1 in thread_num=i

    for each anti-diagonal ‘ad’ in grid
        for each cell 'c' in ‘ad’, starting at i, step by thread_count
            tmp = c
            c = 0.2 * (c + top_cell + bottom_cell + left_cell + right_cell)
            mutex_lock {
                diff = diff + absolute(c - tmp)
            }
        endfor
        barrier_wait()
    endfor

if i==1:
    (test) if (diff/num_cells < threshold): end loop1 (for all threads)
    else diff = 0
    barrier_wait()
else:
    barrier_wait()
```

Results



Execution Time = 306 seconds

Speedups

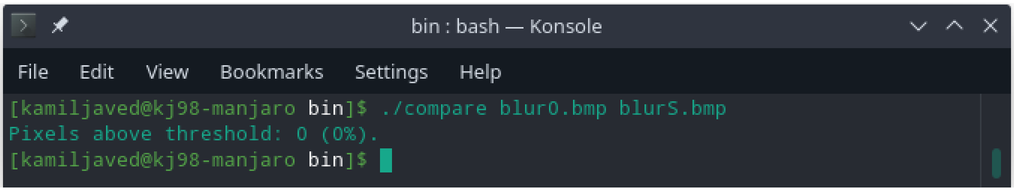
$$\text{Speedup (compared to sequential method)} = \frac{14.1}{306} = 0.05$$

$$\text{Speedup (compared to OpenMP pragma omp critical method)} = \frac{164.2}{306} = 0.54$$

There is no speedup compared to sequential method, in fact the method took longer time than sequential approach. This is because each thread must wait in a queue (in each iteration) to be able to update the 'diff' variable (as that is a critical section enclosed in mutex lock), i.e. the behavior effectively becomes sequential. And hence, the synchronization cost outweighs any benefit achieved by parallelizing here (using mutex lock).

The Pthread method (with mutex lock) has performed worse than OpenMP approach using pragma omp critical, indicated by speedup of less than 1 (compared to OpenMP).

Correctness



Section III.3.b ANTI-DIAGONALS APPROACH

Parallel Implementation with Pthreads using per-thread 'diff' (no padding)

Methodology

To avoid the blockage occurring at identified critical section i.e. updating 'diff', each thread is provided with a variable that it can use to store its personal diff-sum. Hence, an array diffs[thread_count] is kept, and the actual diff value is computed at the end of entire-grid iteration by simply summing all the per-thread diffs in the diffs[] array. The following pseudocode sums up the process:

```
diff = 0

diffs[thread_count]

begin loop1 in thread_num=i

  diffs[i-1] = 0

  for each anti-diagonal 'ad' in grid
    for each cell 'c' in 'ad', starting at i, step by thread_count
      tmp = c
      c = 0.2 * (c + top_cell + bottom_cell + left_cell + right_cell)
      diffs[i-1] = diffs[i-1] + absolute(c - tmp)
    endfor
    barrier_wait()
  endfor

if i==1:
  diff = sum of all elements in diffs[]
  (test) if (diff/num_cells < threshold): end loop1 (for all threads)
  else diff = 0
  barrier_wait()
else:
  barrier_wait()
```

Results

```
[kamiljaved@kj98-manjaro bin]$ ./main1 image01.jpg blur1.bmp
Execution time of gridSolver:   26.516777 seconds.
```

Execution Time = 26.5 seconds

Speedups

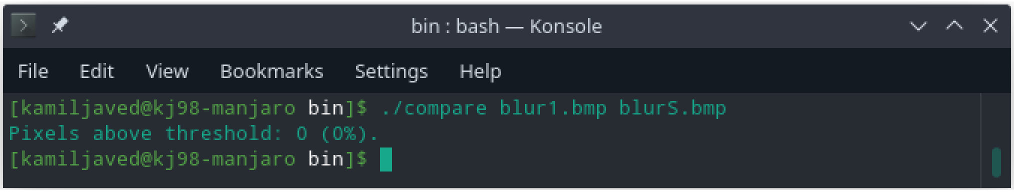
$$\text{Speedup (compared to sequential method)} = \frac{14.1}{26.5} = 0.53$$

$$\text{Speedup (compared to OpenMP per\_thread diff no\_padding method)} = \frac{9.3}{26.5} = 0.35$$

There is no speedup compared to sequential method, in fact the method still took longer time than sequential approach. The most probable cause of the delays is false-sharing here, as explained in Section II.1.b (speedups).

The Pthread method (with per-thread diff no-padding) has performed worse than OpenMP approach using per-thread diff no-padding, indicated by speedup of less than 1 (compared to OpenMP).

Correctness



Section III.3.c ANTI-DIAGONALS APPROACH

Parallel Implementation with Pthreads using per-thread 'diff' (with padding)

Methodology

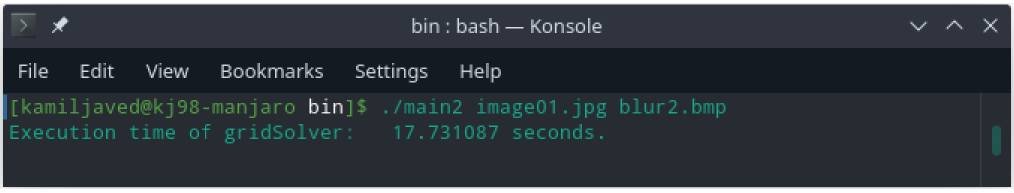
The false-sharing issue can be avoided by changing the relevant data structures so that the different addresses referenced in each thread are on different cache lines.

The cache line-size for the CPU used was determined to be 64 Bytes. Hence, a padding of 60 Bytes was used for each per-thread diff.

The following pseudocode sums up the process:

```
struct diffblk {
    float diff;
    char pad[60];
};

diff = 0

struct diffblk diffs[thread_count]

begin loop1 in thread_num=i

   diffs[i-1].diff = 0

   for each anti-diagonal 'ad' in grid
      for each cell 'c' in 'ad', starting at i, step by thread_count
         tmp = c
         c = 0.2 * (c + top_cell + bottom_cell + left_cell + right_cell)
         diffs[i-1].diff = diffs[i-1].diff + absolute(c - tmp)
      endfor
      barrier_wait()
   endfor

if i==1:
   diff = sum of all diffs[].diff
   (test) if (diff/num_cells < threshold): end loop1 (for all threads)
   else diff = 0
   barrier_wait()
else:
   barrier_wait()
```

Results



$$\text{Execution Time} = 17.7 \text{ seconds}$$

Speedups

$$\text{Speedup (compared to sequential method)} = \frac{14.1}{17.7} = 0.8$$

$$\text{Speedup (compared to per\_thread diff with no padding)} = \frac{26.5}{17.7} = 1.5$$

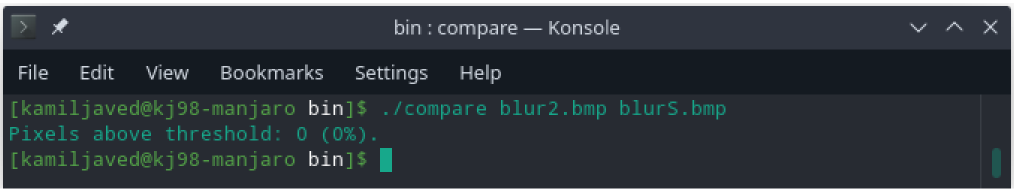
$$\text{Speedup (compared to OpenMP per\_thread diff with\_padding method)} = \frac{8.6}{17.7} = 0.5$$

There is no speedup compared to sequential method, in fact the method still took longer time than sequential approach. This could be due to frequent synchronization needed after computation at each anti-diagonal. Also, based on scheduling and due to varying CPU workloads, different executions may produce different results.

Nevertheless, compared to (Pthread) parallel approach using per-thread diff with no padding (false-sharing), notable speedup of 1.5 times is achieved.

The Pthread method (with per-thread diff with-padding) has performed worse than OpenMP approach using per-thread diff with-padding, indicated by speedup of less than 1 (compared to OpenMP).

Correctness



Section IV

Summary of Results

The execution times for various test performed have been gathered in the tables below.

RED-BLACK CELLS APPROCH					
METHOD			**EXEC. TIME (seconds)**	**SPEEDUP**	
seq./par.	***library***	***using***		***compared to***	***value***
Sequential			5.6		
Parallel	OpenMP	pragma omp critical	260.4	Sequential	0.02
Parallel	OpenMP	per-thread diff (no padding)	8.99	Sequential	0.62
Parallel	OpenMP	per-thread diff (with padding)	2.03	Sequential	2.76
				OMP – per-thread diff no padding	4.43
Parallel	Pthreads	mutex lock	131.3	Sequential	0.04
				OpenMP – pragma omp critical	1.98
Parallel	Pthreads	per-thread diff (no padding)	7.17	Sequential	0.78
				OpenMP – per-thread diff no padding	1.25
Parallel	Pthreads	per-thread diff (with padding)	1.64	Sequential	3.41
				Pthreads – per-thread diff no padding	4.4
				OpenMP – per-thread diff with padding	1.24

ANTI-DIAGONALS APPROCH					
METHOD			**EXEC. TIME (seconds)**	**SPEEDUP**	
seq./par.	***library***	***using***		***compared to***	***value***
Sequential			14.1		
Parallel	OpenMP	pragma omp critical	164.2	Sequential	0.09
Parallel	OpenMP	per-thread diff (no padding)	9.3	Sequential	1.5
Parallel	OpenMP	per-thread diff (with padding)	8.6	Sequential	1.64
				OMP – per-thread diff no padding	1.1
Parallel	Pthreads	mutex lock	306	Sequential	0.05
				OpenMP – pragma omp critical	0.54
Parallel	Pthreads	per-thread diff (no padding)	26.5	Sequential	0.53
				OpenMP – per-thread diff no padding	0.35
Parallel	Pthreads	per-thread diff (with padding)	17.7	Sequential	0.8
				Pthreads – per-thread diff no padding	1.5
				OpenMP – per-thread diff with padding	0.5