Australian National University - Assignmemnt Cover Sheet

Report of Assignment Two: Advection Solver for Shared Address Space Programming Models

Hong-Bo Zhang

College of Engineering and Computer Science, Australian National University, Canberra, 2601, Australia
u6170245@anu.edu.au

UID	u6170245
Course	COMP8300 Parallel System
Assign No.	Assignment 2
Group No.	Individual
Lecturer	Peter Strazdins
Date	2017-May-26

Abstract: This is the assignment 2 of parallel system. I declared that all the work are done by myself.

*We declare that this work: upholds the principles of academic integrity, as defined in the ANU Policy: Code of Practice for Student Academic Integrity.

Contents

1. Task 1 Parallelization via 1D Decomposition and Simple Directives

In this section, we will use `parallel for` to parallelize two nesting loops, namely *Stage1* and *Stage2*. If one takes parallelizing the outer or inner loop, interchanging the loop order and scheduling the iterations in a block or cyclic style into consideration, there are $2 \times 2 \times 2 = 8$ possible configurations for each nesting loop. Therefore, there are $8 \times 8 = 64$ possible configurations of parallelization in total. However, most of them are not interesting. Consequently, in this report, we will restrict to the case that the parallelization configurations of *Stage1* and *Stage2* identical. In this case, only 8 configurations are considered. In certain subsection, we also add a brief comment on the case where configurations of *Stage1* and *Stage2* are different.

In scheduling part, we will only use `static`, rather than dynamics, since there is no obviously load inbalance. Furthermore, we notice that the `i` loop indice is the one contiguous in memory (column major). The convention in this report is that `i` is row indice and `j` is column indice.

1.1 Maximize performance

```
1  // 1. maximize performance
2  #pragma omp parallel for schedule(static) default(shared) private(i)
3      for (j=0; j < N+1; j++) // advection update stage 1
4        for (i=0; i < M+1; i++)
5          V(ut,i,j) = 0.25*(V(u,i,j) + V(u,i-1,j) + V(u,i,j-1) + V(u,i
               -1,j-1))
6            -0.5*dt*(sy*(V(u,i,j) + V(u,i,j-1) - V(u,i-1,j) - V(u,i-1,
                 j-1)) +
7                     sx*(V(u,i,j) + V(u,i-1,j) - V(u,i,j-1) - V(u,i-1,
                        j-1)));
8  #pragma omp parallel for schedule(static) default(shared) private(i)
9      for (j=0; j < N; j++) // advection update stage 2
10       for (i=0; i < M; i++)
11         V(u, i, j) +=
12           - dtdy * (V(ut,i+1,j+1) + V(ut,i+1,j) - V(ut,i,j) - V(ut,i
                ,j+1))
13           - dtdx * (V(ut,i+1,j+1) + V(ut,i,j+1) - V(ut,i,j) - V(ut,i
                +1,j));
```

Since the default value of chuck size is loop number divdes by number of threads. In this configuration, we assign (roughly) a block with $N/num\_threads$ columns to each thread. In this case, the performance will be improved according to following aspects.

- for each nesting loop, there is only one OpenMP parallel region entry/exit.
- a contiguous block of memory is operated in the for loop, since the inner loop indice is `i` (column major).
- the cache miss of coherent reads only lies at the boundary of adjacent blocks and we use a block fashion schedule (rather than cyclic shedule), the coherent read cache miss is also minimized.
- the cache miss of coherent writes only happens when the tail of one block and the head of the next block lies in the same cache line, so in this case, there is at most one coherent write cache miss between two nearby blocks.

Consequently, I choose this configuration as the best performing one.

1.2 Maximize the number of OpenMP parallel region entry/exits

```
// 2. maximize the number of OpenMP parallel region entry/exits
    if ( N > M ) {
      for (j=0; j < N+1; j++) // advection update stage 1
#pragma omp parallel for schedule(static) default(shared)
        for (i=0; i < M+1; i++)
        V(ut,i,j) = 0.25*(V(u,i,j) + V(u,i-1,j) + V(u,i,j-1) + V(u,i
           -1,j-1))
          -0.5*dt*(sy*(V(u,i,j) + V(u,i,j-1) - V(u,i-1,j) - V(u,i-1,
             j-1)) +
                   sx*(V(u,i,j) + V(u,i-1,j) - V(u,i,j-1) - V(u,i-1,
                      j-1)));
      for (j=0; j < N; j++) // advection update stage 2
#pragma omp parallel for schedule(static) default(shared)
        for (i=0; i < M; i++)
        V(u, i, j) +=
          - dtdy * (V(ut,i+1,j+1) + V(ut,i+1,j) - V(ut,i,j) - V(ut,i
             ,j+1))
          - dtdx * (V(ut,i+1,j+1) + V(ut,i,j+1) - V(ut,i,j) - V(ut,i
             +1,j));
    } else {
      for (i=0; i < M+1; i++) // advection update stage 1
#pragma omp parallel for schedule(static) default(shared)
        for (j=0; j < N+1; j++)
        V(ut,i,j) = 0.25*(V(u,i,j) + V(u,i-1,j) + V(u,i,j-1) + V(u,i
           -1,j-1))
          -0.5*dt*(sy*(V(u,i,j) + V(u,i,j-1) - V(u,i-1,j) - V(u,i-1,
             j-1)) +
                   sx*(V(u,i,j) + V(u,i-1,j) - V(u,i,j-1) - V(u,i-1,
                      j-1)));
      for (i=0; i < M; i++) // advection update stage 2
#pragma omp parallel for schedule(static) default(shared)
        for (j=0; j < N; j++)
        V(u, i, j) +=
          - dtdy * (V(ut,i+1,j+1) + V(ut,i+1,j) - V(ut,i,j) - V(ut,i
             ,j+1))
          - dtdx * (V(ut,i+1,j+1) + V(ut,i,j+1) - V(ut,i,j) - V(ut,i
             +1,j));
    }
```

The reason of choosing this configuration is obviously. Parallel the inner loop will increase the OpenMP parallel region entry/exits. For each outer loop iteration, there will be one parallel region entry, exit and synchronization. Therefore, there will be max $\mathcal{O}(M), \mathcal{O}(N)$ times entry/exit in total. I add a branch statement `if ( N > M )` at the begining to enforce the maximum between N and M.

1.3 Maximize shared cache misses involving coherent reads

```
// 3. maximize shared cache misses involving coherent reads
#pragma omp parallel for schedule(static,1) default(shared) private(
    i)
```

```
3     for (j=0; j < N+1; j++) // advection update stage 1
4       for (i=0; i < M+1; i++)
5         V(ut,i,j) = 0.25*(V(u,i,j) + V(u,i-1,j) + V(u,i,j-1) + V(u,i
              -1,j-1))
6           -0.5*dt*(sy*(V(u,i,j) + V(u,i,j-1) - V(u,i-1,j) - V(u,i-1,
                j-1)) +
7                    sx*(V(u,i,j) + V(u,i-1,j) - V(u,i,j-1) - V(u,i-1,
                         j-1)));
8 #pragma omp parallel for schedule(static,1) default(shared) private(
      i)
9     for (j=0; j < N; j++) // advection update stage 2
10      for (i=0; i < M; i++)
11        V(u, i, j) +=
12          - dtdy * (V(ut,i+1,j+1) + V(ut,i+1,j) - V(ut,i,j) - V(ut,i
                ,j+1))
13          - dtdx * (V(ut,i+1,j+1) + V(ut,i,j+1) - V(ut,i,j) - V(ut,i
                +1,j));
```

The coherent read cache misses occur at the boundary of two nearby blocks. To be more specific, in the *Stage1*, the read cache miss happens at `V(u,*,threadId*blockSize)`, where `u` is the top-left corner of halo. (Of course, read cache miss also happens at halo.) While in the *Stage2*, the read cache miss happens at `V(ut,*,(threadId+1)*blockSize)`. Every thread will meet $(N/num\_threads/blockSize)*(M/cache\_width)$ read cache misses, where $cache\_width$ is the width of cache line in the unit of `sizeof(double)`. Therefore, in order to maximize the read cache misses, I reduce the block size to 1 (which is essentially interleave the iterations). In this case, every thread will meet $(N/num\_threads)*(M/cache\_width)$ read cache misses.

1.4 Maximize shared cache misses involving coherent writes

```
1 // 4. maximize shared cache misses involving coherent writes
2     for (j=0; j < N+1; j++) // advection update stage 1
3 #pragma omp parallel for schedule(static,1) default(shared)
4       for (i=0; i < M+1; i++)
5         V(ut,i,j) = 0.25*(V(u,i,j) + V(u,i-1,j) + V(u,i,j-1) + V(u,i
              -1,j-1))
6           -0.5*dt*(sy*(V(u,i,j) + V(u,i,j-1) - V(u,i-1,j) - V(u,i-1,
                j-1)) +
7                    sx*(V(u,i,j) + V(u,i-1,j) - V(u,i,j-1) - V(u,i-1,
                         j-1)));
8
9     for (j=0; j < N; j++) // advection update stage 2
10 #pragma omp parallel for schedule(static,1) default(shared)
11      for (i=0; i < M; i++)
12        V(u, i, j) +=
13          - dtdy * (V(ut,i+1,j+1) + V(ut,i+1,j) - V(ut,i,j) - V(ut,i
                ,j+1))
14          - dtdx * (V(ut,i+1,j+1) + V(ut,i,j+1) - V(ut,i,j) - V(ut,i
                +1,j));
```

The coherent write cache miss happens when the data (`u` or `ut`) used by different threads lie in the same cache line. Let's look into this configuration with more details. In this configuration, we use cyclic scheduling with the block size is 1 and parallel the inner

loop (`i` the row indice), therefore the nearby `u` or `ut` are written by different threads. For example, there are 4 threads in total, `V(u,1,j)` is operated by thread0, `V(u,2,j)` is operated by thread1, `V(u,3,j)` is operated by thread2, `V(u,4,j)` is operated by thread3, and then cyclic. If the width of cache line is longer than 4 doubles, then every write by one thread (e.g. `V(u,1,j)` by thread0) will cause the corresponding cache line of all other processors change into invalidate state. When other threads write (e.g. `V(u,2,j)` by thread1), the coherent write cache miss occurs. Consequently, almost every write will cause coherent write cache miss. Hence this configuration maximize the cache misses involving coherent write.

1.5 Other configurations

Here we will briefly discuss other configurations which hasn't mention above. It is also possible that the first nesting loop and second nesting loop don't have the same parallel configurations. However, in this case, there will lots of nromal cache misses. While the coherent read/write cache misses will not become more than those shown in previous subsections.

1.6 Experiment results

We choose $M = N = 2000$ and a large $r = 100$ to maximize the difference between different configurations. We conduct experiments for all the four configurations with $p = 8$ and $p = 16$ to reflect the difference between inner socket and intra socket. For the case of $p = 8$, we invoke `numactl --cpunodebind=0 --membind=0` to enforce all 8 threads run on a single socket. The boundary update is not parallized for all the configurations. The boundary parallelization will be shown in next subsection. The "Max Perform" is abbreviation of "Maximize Performance". The "Max Paral Entry" is short for "Maximize Parallel Region Entry/Exists". The "Max Read/Write Miss" is short for "Maximize Shared Cache Misses Involving Coherent Read/Writes".

Table 1: The performance of four different OpemMP parallel configurations.

	Max Perform	Max Paral Entry	Max Read Miss	Max Write Miss
$p = 8$	4.978×10^{-1}s	1.876s	4.983×10^{-1}s	4.203s
$p = 16$	3.717×10^{-1}s	2.452s	5.968×10^{-1}s	15.90s

As shown in Tab 1, we can see that the max performance configuration is indeed fastest among the all 4 configurations. Furthermore, the maximize parallel region entry/exists is the worst among all. The difference among maximize performance, max read miss and max write miss is relative small for $p = 8$, at the order of 10^{-4}s. Since in this case, all the buffer are storaged in the same socket, the difference among these three are relative small. However, in the case of $p = 16$, i.e., all threads are distributed on two sockets, so a read/write cache miss will be updated from the other socket, which is much more expensive. Hence, the difference among max performance, max read caches misses and max write caches miss are relative large.

Furthermore, due to the more threads, the more coherent cache misses, the cost of $p = 16$ is larger than half of cost of $p = 8$.

1.7 Boundary update loops

Both boundary update loops are parallel with scheduling the iterations in a block fashion.

The data in left and right halos are contiguous in memory storage. Therefore parallel in block will reduce cache misses, hence improve performance.

The data in top and bottom halos are not contiguous in memory. However, we also parallel them in block with block size identical to best performance configuration. Therefore, in this way, it will reduce the coherent cache miss when update field `u` and `ut` at later stage.

Actually, if `M` and `N` are small enough, it is better to use the first thread to update left halo and to use the last thread to update the right halo.

Furthermore, parallel the boundary update loop will reduce the computation cost as well.

However, the drawback of parallel the boundary update loop is that it will double the parallel region entry/exists, which is expensive. So there is a competition between this two terms.

We do experiments with $M = N = 2000, r = 100$, and the results are shown in Tab 2.

Table 2: Whether to parallel boundary update

	Not parallel boundary update	parallel boundary update
$p = 8$	4.978×10^{-1}s	4.934×10^{-1}s
$p = 16$	3.717×10^{-1}s	3.687×10^{-1}s

From the above results,we can see that the performance get better with the order of 10^{-2}s.

2. Task 2 Performance Modelling of Shared Memory Programs

2.1 Modelling

The overall cost of the program can be divided into sequential part cost and parallel part cost.

$$t = t_{seq} + t_{par}$$

However, the computational load of sequential part is much smaller than the parallel part, so we neglect this term safely. The parallel term is constituted by four parts, including parallel region entry/exist cost t_s, the cost of double words of cache miss for coherent read $t_{w,R}$, the cost of double words of cache miss for coherent write $t_{w,W}$ and the cost of computation t_f.

Consequently, the overall cost can be modelled as

$$\begin{aligned} t &= n_E t_s + n_R t_{w,R} + n_W t_{w,W} + n_f t_f + \text{cache misses from halo part} \\ &= 4 \times r \cdot t_s + \frac{M \cdot Q}{8} \cdot (2 \times r - 1) \cdot t_{w,R} + Q \cdot (2 \times r - 1) \cdot t_{w,W} + r \cdot \frac{M \cdot N}{Q} \cdot t_f \end{aligned}$$

where n_E, n_R, n_W, n_f is the number of occurences of corresponding terms. We neglect the cache misses from halo part, since this term is much smaller than those of non-halo part. Since we both parallel the boundary update loops and advection field update loops, n_E is $4 \cdot r$. Since in the maximize performance configuration, we divide the whole field into blocks in column direction, the read cache miss occurs on the boundary of two adjacent blocks, and each cache line has 8 doubles, so $n_R = \frac{M \cdot Q}{8} \cdot (2 \times r - 1)$, the $(2 \times r - 1)$ reflects the fact that in the first r loop iteration, there is no coherent read cache miss of u field. The coherent write cache miss comes from two adjacent block shares one cache line, so there will be one write cache miss between two blocks, the term $(2 \times r - 1)$ is similar to that in $t_{w,R}$ term, so $n_W = Q \cdot (2 \times r - 1)$. Finally, the t_f contains 23 FLOPS and serveral interger operations.

2.2 Measurement

Firstly, we measure the parallel region entry/exits time t_s. We run two experiments with $M = N = 20$ and $r = 100$. In the first experiment, the OpenMP will not be used, In this case, the time cost is 2.5×10^{-4}s. While in the second experiment, we will use OpenMP with $p = 1$. In this case, the time cost is 5.4×10^{-4}s. Consequently, the $t_s = (5.4 - 2.5) \times 10^{-4}/r = 2.9 \times 10^{-6}$s.

Next we will measure the $t_{w,R}$ and $t_{w,W}$. Since the $t_{w,W}$ term will not depend on M, but $t_{w,R}$ does, so we perform two set of experiments with $p = 8$ and $p = 16$. For each set, we keep $N = 2000, r = 10000$, but varying $M = 1000, 2000, 3000$.

The experiment results are shown in Tab 3

Table 3: Measure cost of coherent cache miss

	$M = 1000$	$M = 2000$	$M = 3000$
$p = 8$	22.33s	49.03s	73.16s
$p = 16$	8.764s	24.37s	39.12s

From above results, we can get for inner socket ($p = 8$), $t_{w,R} = 1.2 \times 10^{-6}$s, and for intra socket ($p = 16$), $t_{w,R} = 3.2 \times 10^{-7}$s. In fact, intra socket time should be larger than inner socket time. $t_{w,W}$ is negative in both cases. These un-reasonable results are due to my inappriciate choose of M and N.

I have no time to choose a set of better M, N, r.

2.3 Experiments with $p = 8$ and $p = 16$

2.4 Prediction

3. Task3 Parallization via 2D Decomposition and an Extended Parallel Region

We parallel via 2D decomposition within one parallel region. It requires several synchronize to make sure it work correctly. The directive should be put outside the r loop. And it is faster than put it inside the r loop by 10^{-3}s with $M = N = 2000, r = 100$ from my experiments.

We choose $M = N = 2000, r = 100$ and $p = 16$, but varying P. The experiment results are shown in Tab 4.

Table 4: 2D distribution with varying P

p	1	2	4	8	16
time	3.677×10^{-1}s	3.696×10^{-1}s	3.726×10^{-1}s	3.750×10^{-1}s	8.455×10^{-1}s

From the results, comparing to 1D 3.687×10^{-3}s, when $p = 1$ increase the performance by 10^{-3}s. while the other value of p doesn't improve the performance at all. This is because, for $p = 1$, the 2D case will have less parallel region than 1D case, since there is only one parallel region. So indeed, having only one parallel region will improve the performance. However, for other value of p, although it will reduce the number of parallel region entry/exits, it also increases the coherent cache miss. The number of synchronize is the same from 1D and 2D cases. As a result, for $p > 1$, the performance reduced.

In MPI case, comparing to 1D, 2D case will increase the number of t_s. The number of t_w may increase or decrease, depending on the value of M, N, P and Q. But in OpenMP, the number of t_s in 2D is unchanged or decrease comparing to 1D, depending on how to program. And generally Openmp will increase t_w terms.

4. Task 4 Optional

We use the `#pragma omp simd` to improve the performance further. I only apply this extra optimization to 1D case. However, it is easy to extends to 2D case.

I do two experiments with $M = N = 200$ and $r = 100$. With simd, the time cost is 4.295×10^{-3}s. While without simd, the time cost is 5.222×10^{-3}s.

Obviously, it has improved a lot.

5. Task 5 Baseline GPU Implementation

I create new kernal functions to parallel 2D.

5.1 Comparison of $1 \times B$ vs $B \times 1$ blocks

I have run two experiments with $M = N = 600, r = 10, Bx = By = 16$, and vary $(Gx, Gy) = (1, 32), (32, 1)$. The time cost of $(1, 32)$ is 6.359×10^{-2}s, while the time cost of $(32, 1)$ is 5.867×10^{-2}s.

5.2 Optimize combination

Since a warp include 32 threads, the $Bx \cdot By$ should be better a multiplier of 32.

According to above analysis, we conduct following experiments. Firstly, we keep $M = N = 600$, then varying $(Gx, Gy, Bx, By) = (3, 3, 4, 8), (4, 8, 3, 3)$. The first set satisfies the warp condition, but the second doesn't satisfy. The time cost of first set is 4.112×10^{-2}s, while the time cost of second set is 8.339×10^{-2}s.

5.3 Speedup

With the parameter $M = N = 600, r = 10$, I run three experiments to calculate the speedups. The time cost on host is 5.756×10^{-2}s, the time cost of single GPU thread is 5.165s. While the time cost of $(Gx, Gy, Bx, By) = (3, 3, 4, 8)$ is 1.823×10^{-3}s. So the speedup to host is 1.4 and the speedup to a single thread is 62.

5.4 Overhead

In order to get the overhead of invoking kernal. We choose $M = N = r = Bx = By = Gx = Gy = 1$. The experiment result is 6.962×10^{-4}s. In a computation, it will invoke the kernal for 4 times, so the every time of each kernal is 1.7×10^{-4}s. Since it is much larger than the computation time, we can safely take this value as the overhead of invoking kernal.

6. Task 6 Optimized GPU Implementation

I use shared memory to do optimization. Since access to shared memory is much faster than the access to the global memory. It will improve the performance by reduce the memory access time. In every r iteration, using shared memory will have overhead of copying from global memory to local memory and several synchronization. Consequently, the two effects will compete.

I implement in following steps. 0. update halo of u field.
1. copy boundary from global memory u to shared memory cu.
2. copy center grid from global memory u to shared memory cu.
3. synchronize
4. updateAdvect1 of cu, cut
5. synchronize
6. updateAdvect2 of cu, cut
7. synchronize
8. write cu back to u

I do experiment with $M = N = 600, r = 10, Gx = Gy = 2, Bx = By = 16$. The optimized time cost is 1.987×10^{-3}. While the non-optimized time cost is 6.221×10^{-2}. So it improves a lot.

7. Task 7 Comparison of the Programming Models

OpenMP is the easiest to program. You just write the serial program firstly, and add some directives. It is relative easy to debug.

CUDA is also easy to program. But if one want to write a program with high efficieny, it will become difficult. Since there are many kinds of memories, including the global mem, shared mem, constant mem and texture mem, and it requires lots of skills and experiences to fully make use of all kinds of memories. It is also relative easy to debug. The speedup is

MPI is the most difficult to program. It is difficult to design the message passing strategy, to implement the parallel code. And the most important is that it is very difficult to debug.

The speedup cannot be compared directly. In fact, for $M = N = 600, r = 10$ and 4 threads/processors, the speedup of Cuda is 3%, which is much slower than CPU. Since the strategy of GPU is different from CPU. There are lots of SP on a GPU. The strategy of GPU speedup is to use lots of SP to do SIMD calculations. However, for OpenMP and MPI, there will not such many threads/processors as GPU's SP. But every thread/processor will run much faster than SP in GPU. The speedup between MPI and OpenMP is case by case. It will determined by the program. Generally, the prefactor of t_s term in OpenMP will be smaller than that in MPI.

8. Task 8 Optional

I have port my codes to Raijin GPUs.

To do that, you should `module load cuda` firstly. And then you need modify your Makefile to `nvcc $(CCFLAGS)-lcuda -lcudart -c $*.cu`. Then you should modify your batch to `#PBS -q gpu \n #PBS -l ngpus=2 \n #PBS -l ncpus=6` to use Kepler80.

I perform the experiment on Raijin with parameters $M = N = 600, r = 10$ with $Gx = Gy = 2, Bx = By = 16$ without -o option, The time cost is 2.617×10^{-2}s.

I also run the same experiment on Jetson1 with the same parameters as above without -o option. The time cost on Jetson1 is 6.288×10^{-2}s, which is much slower than Raijin.

Acknowledgments

Thanks Peter for nice lectures, Sara for her lab tutorial and Raijin for providing the computing resource.

References