Parallel computing HW 1

Akshay Kumar Varanasi
(av32826)

May 6, 2018

Idea

I have used Fortran 90 to do this assignment.

Assignment 1

After parallelizing the code and running the code with different settings we get following result for counting kernel.(I did not do the serial case as Fortran users were told not to.)

Execution	Run-times for x	Run-times for y
Parallel with 1 thread	0.8309000	0.3430000
Parallel with 8 threads	0.1089000	4.9699999E-02

Table 1: Comparision of run-times of different no.of threads.

We can see that as the no.of thread increased the run-times for both x and y reduced by same factor and hence we can see the speed up in counting kernel i.e when threads become 8 times then times becomes 1/8th. Coming to number of elements below the threshold in serial and parallel, they are same as we used reduction option along with parallel do.

Assignment 2

2A

The speed up in smoothing due to parallelization with static scheduling with chunk size 100 is shown in the following figure 1.

The speedup seems to be good initially but after sometime the speedup is negligible and it starts to deviate from the ideal speedup. In this case, we can see that best speedup is at 8 threads but after that there is not much difference. So scaling is good till 8 but not after that. Deviation arises after some thread number(in our case it is 8) because as the threads increase, other costs such as the cost of forking,load imbalance, communication becomes significant.

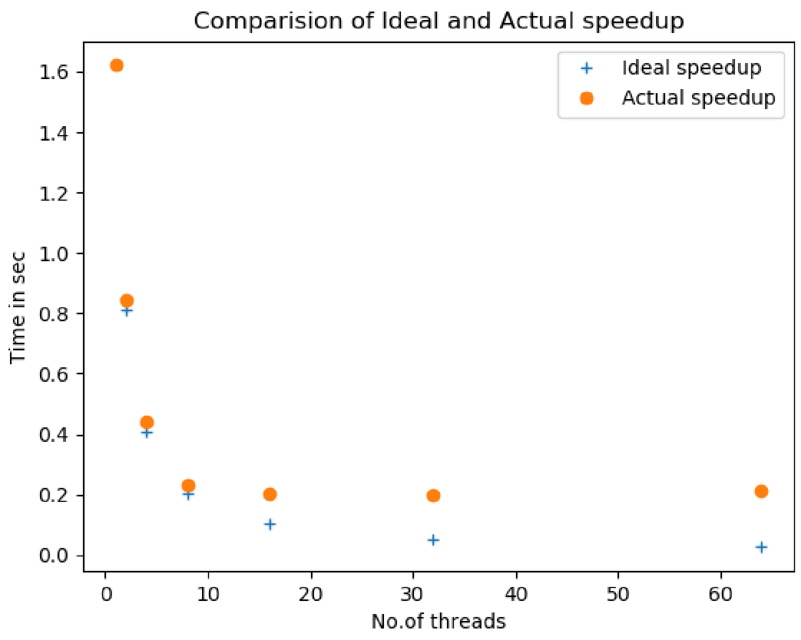

Figure 1: Speedup vs Thread count.

The output of the code with 8 threads

```
The time taken to allocate x is  1.1000000E-03
The time taken to allocate y is  0.0000000E+00
This is thread           3
This is thread           1
This is thread           7
This is thread           0
This is thread           6
This is thread           5
This is thread           4
This is thread           2
The time taken to initialize x is   3.322100
The time taken to smooth y is  0.2293000
The time taken to count x is  0.1094000
The time taken to count y is  4.9500000E-02
Summary
-------
Number of elements in a row/column ::         32770
Number of inner elements in a row/column ::          32768
Total number of elements ::     1073872900
Total number of inner elements ::     1073741824
Memory (GB) used per array ::  4194816.00000
Threshold ::           0.10
Smoothing constants (a, b, c) :: 0.05 0.10 0.40
Number of elements below threshold (X) ::      107384294
Fraction of elements below threshold ::    1.00009E-01
Number of elements below threshold (Y) ::          11727
Fraction of elements below threshold ::    1.09216E-05
```

2B

In this section, the results of static and dynamic scheduling will be discussed. As we can see from the table below that as the chunk size increased the time taken also increases in both types of scheduling. This is because with increase in chunk size time taken by each thread to do its job increases and thus waiting time for the thread to get next chunk will also increase. And we can also see that dynamic gives faster results than static because the threads take the chunk when they are free which need not be in round robin fashion like in the static case.

Schedule	Chunk=100	Chunk=1000	Chunk=10000
Static	0.2365000	0.2614000	0.5360000
Dynamic	0.2295000	0.2609000	0.5304000

Table 2: Comparision of run-times of different scheduling.

2C

Numactl option lets us control NUMA() policy for processes or shared memory where NUMA stands for Non Uniform Memory Access.In SMP,systems with multiple CPUs access time to different parts of memory may vary, and this helps us to control it.

The **"–cpunodebind=nodes, -N nodes"** option makes command execute on the CPUs of given nodes only. And **"–preferred=node"** option allocates memory on given nodes preferably, but if memory cannot be allocated it falls back to other nodes. This option takes only a single node number. Relative notation may be used. Whereas **"–interleave=nodes, -i nodes"** sets a memory interleave policy. Memory will be allocated using round robin on nodes. When memory cannot be allocated on the current interleave target then it falls back to other nodes. Multiple nodes may be specified on –interleave and –cpunodebind.

So you can see when only ./a.out is run, memory allocation and threads are shared between 0 and 1 nodes. Whereas in second case, it is mainly on node 0 and thus it is faster as it is easier for threads to access the memory as both are on same node. While in third case, though the memory is allocated on node 0 but threads are running on both the nodes so threads on node 1 need to access memory on node 0, which is slower. In the fourth case it is even slower compared to third case where some threads are on 0 so it was little faster so they can access faster but now all the threads are on 1 and memory is on 0 so it is even more difficult to access the memory.

Numactl settings	Run-times	Memory allocation	Threads
./a.out (no numactl)	0.2667000	0,1	0,1
numactl –cpunodebind=0 –preferred=0 ./a.out	0.2306000	0	0
numactl –cpunodebind=0 –interleave=0,1 ./a.out	0.2873000	0	0,1
numactl –cpunodebind=0 –preferred=1 ./a.out	0.3872000	0	1

Table 3: Comparision of run-times of different numactl settings.

When **numactl –hardware** command is used it shows inventory of available nodes on the system. The output is shown below

```
available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88
90 92 94
node 0 size: 96965 MB
node 0 free: 92384 MB
node 1 cpus: 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41
43 45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89
91 93 95
node 1 size: 98304 MB
node 1 free: 94627 MB
node distances:
node 0 1
0: 10 21
1: 21 10
```

Assignment 3

The **"-xHost"** option instructs the compiler to generate instructions for the highest instruction set available on the compilation host processor. It basically does high level optimization based on the architecture of the computer being used. If this is not used the compiler will give instructions such that it is compatible with different systems rather than optimizing for each system's architecture and specification.

Part 2

A

In the first case, where we don't parallelize initialization, following are the coding changes

```
  !$omp parallel do schedule(runtime)
      do i=2, N,   2;  a(i) = (a(i) + a(i-1)) / 2.0; end do
      !$omp parallel do schedule(runtime)
      do i=1, N-1, 2;  a(i) = (a(i) + a(i+1)) / 2.0; end do

      error=0.0d0; niter = niter+1

      !$omp parallel do reduction(+:error) schedule(runtime)
      do i=1,N-1; error=error + abs(a(i)-a(i+1)); end do
```

Various scheduling methods do make a difference, from the figure 2 we can clearly see this. In this case the result is not as expected because dynamic is slower than static. This may be because of load imbalance. Not only that but

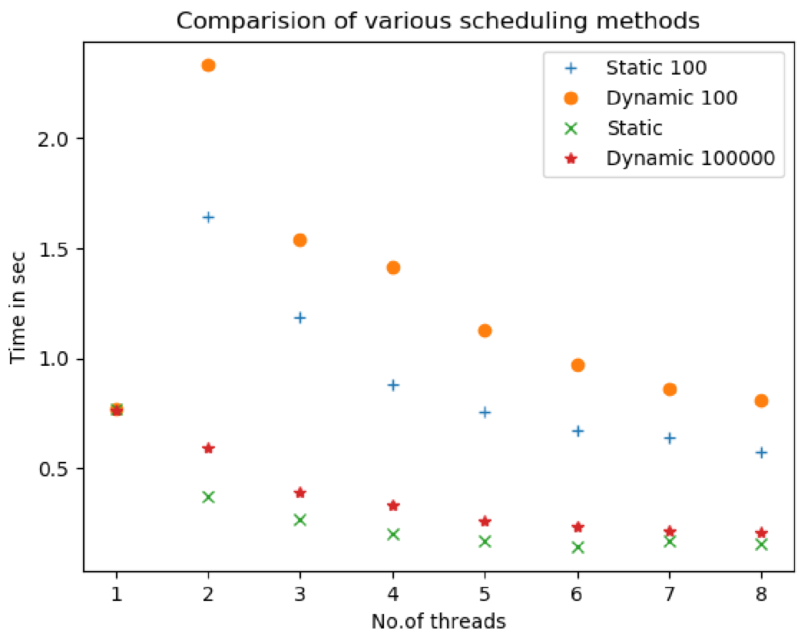

Figure 2: Comparision of various scheduling.

as the chunk size increased it gets faster, which is also not expected. It may be because the problem size is too big that the cost of taking chunks by the threads(communication) is significant.

Scaling is not good compared to ideal speedup but it does speedup till 6 but then saturates.

For parallelizing the initialization, we need to do the following changes

```
!$omp parallel do
    do i=1,N-1,2; a(i)   = 0.0; a(i+1) = 1.0d0; end do
```

This speeds up in general but the trend in different scheduling methods remains same.We can see from figure 3 that scaling is better when we parallelized the initialization because serial part of the code is reduced and thus it is closer to ideal case where it assumes whole code is parallelized. Even in this case, the speedup saturates after 6 threads.

When I tried to see where the threads are running in iv and v, I did not see any consistency to where they are running. I saw that different threads take their chunk with no pattern in how they take it, it was random.

B

The scheduling methods follow the same trend but they are almost similar to part A even though there is only one parallel directive instead of three and thus

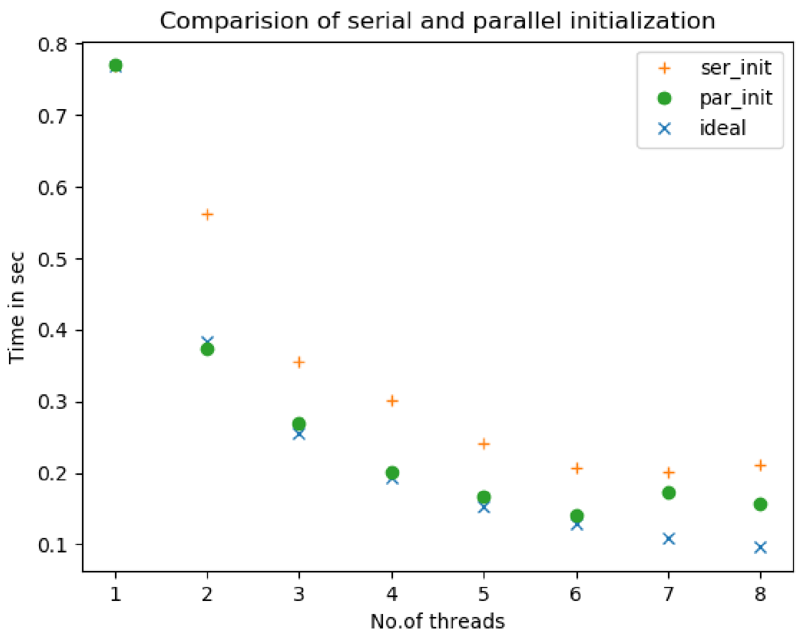

Figure 3: Serial vs Parallel initialization.

saving the cost of forking thrice because forking is immediate i.e there is no significant serial part in between which would have slowed down the code.

These are the coding changes for this part where there is single parallel directive in while loop.

```
    !$omp parallel do
    do i=1,N-1,2; a(i)   = 0.0; a(i+1) = 1.0d0; end do

    t0 = gtod_timer();

    do while (error .ge. 1.0d0)

       !$omp parallel
       !$omp do schedule(runtime)
       do i=2, N,   2;  a(i) = (a(i) + a(i-1)) / 2.0; end do
       !$omp do schedule(runtime)
       do i=1, N-1, 2;  a(i) = (a(i) + a(i+1)) / 2.0; end do

       !$omp single
       error=0.0d0; niter = niter+1
       !$omp end single

       !$omp do reduction(+:error) schedule(runtime)
       do i=1,N-1; error=error + abs(a(i)-a(i+1)); end do
       !$omp end parallel
    end do
```

C

The two steps of optimization of this code are

1.Using single parallel directive outside the while loop

```
!$omp parallel
   !$omp do
   do i=1,N-1,2; a(i)   = 0.0; a(i+1) = 1.0d0; end do

   !$omp single
   t0 = gtod_timer();
   !$omp end single

   !!$ omp do
   do while (error .ge. 1.0d0)

      !$omp do
      do i=2, N,   2;  a(i) = (a(i) + a(i-1)) / 2.0; end do
      !$omp do
      do i=1, N-1, 2;  a(i) = (a(i) + a(i+1)) / 2.0; end do

      !$omp single
      error=0.0d0; niter = niter+1
      !$omp end single

      !$omp do
      do i=1,N-1; error=error + abs(a(i)-a(i+1)); end do


   end do
!$omp end parallel
```

The output of the code is shown below

```
prb_c
schedule: STATIC,100
      0.7936
      1.6198
      1.1614
      0.8794
      0.7882
      0.6747
      0.6176
      0.5255
      0.4989
      0.4487
      0.4109
      0.3988
      0.4013
      0.3721
      0.3694
      0.3272
schedule: dynamic,100
      0.7637
      2.3205
      1.5559
      1.3946
      1.1479
      0.9764
      0.8594
      0.7793
      0.7488
      0.7120
      0.6663
      0.6366
      0.6133
      0.6011
      0.5842
      0.5624
```

```
prb_c
schedule: STATIC
      0.7654
      0.3765
      0.2699
      0.2060
      0.1673
      0.1411
      0.1681
      0.1311
      0.1239
      0.1353
      0.1446
      0.1035
      0.1191
      0.1058
      0.1051
      0.1000
schedule: dynamic,100000
      0.7633
      0.5884
      0.3967
      0.3148
      0.2573
      0.2425
      0.2114
      0.2037
      0.1879
      0.1804
      0.1718
      0.1638
      0.1625
      0.1548
      0.1536
      0.1458
```

2.Combining red-black loops

```
  !$omp do schedule(runtime)
  do i=2,N,2 ;
   a(i) = (a(i)+a(i-1))/2.0; a(i-1)=(a(i-1)+a(i))/2.0;
  end do
```

The output of the code is shown below

```
 prb_c
schedule: STATIC,100
      0.5501
      1.1720
      0.8750
      0.6929
      0.5923
      0.4668
      0.5189
      0.4339
      0.3949
      0.3686
      0.3165
      0.2998
      0.2843
      0.2831
      0.2750
      0.2329
schedule: dynamic,100
      0.5495
      1.6833
      1.2744
      1.0296
      0.8932
      0.7494
      0.6808
      0.6244
      0.5783
      0.5442
      0.5062
      0.4914
      0.4649
      0.4475
      0.4273
      0.4275
 prb_c
schedule: STATIC
      0.5492
      0.2696
      0.1924
      0.1449
      0.1196
      0.1391
      0.0923
      0.0796
      0.0770
      0.0926
      0.0838
      0.0839
      0.0757
      0.0704
      0.0689
      0.0611
schedule: dynamic,100000
      0.5513
      0.4329
      0.2909
      0.2497
      0.2008
      0.1715
      0.1454
      0.1398
      0.1303
      0.1171
      0.1200
      0.1073
      0.1062
      0.1039
      0.0994
      0.0952
```