

Advanced OpenMP Programming

Pangfeng Liu
National Taiwan University

April 20, 2022

- We will observe the timing when threads access variables at different locations.
- The intuition is that local variables are much easier to access than the global ones, and we need to confirm that.

Compute Square Roots

- Write a program that computes the square root from 0 to 999999999 and assign the value to a variable v .
- We first observe the timing when we place the variable v at global area.

Example 1: (assign.c)

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <math.h>
4  #define N 999999999
5  int main()
6  {
7      double v;
8      #pragma omp parallel for
9          for (int i = 1; i <= N; i++)
10         v = sqrt(i);
11     return 0;
12 }
```

Demonstration

- Run the assign-uni program.
- Run the assign-omp program.

Discussion

- Compare the execution time of `assign-uni` and `assign-omp`.

Private Variable

- We now declare v as private and observe the timing.

Example 2: (assign-private.c)

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <math.h>
4  #define N 999999999
5  int main()
6  {
7      double v;
8      #pragma omp parallel for private(v)
9          for (int i = 1; i <= N; i++)
10         v = sqrt(i);
11     return 0;
12 }
```


Demonstration

- Run the assign-private-omp program.

Discussion

- Compare the execution time of `assign-private-omp` with previous `assign-uni` and `assign-omp`.
- What is the reason for this performance difference?

Heap Variable

- We now put v into the heap and observe the timing.

Example 3: (assign-heap.c)

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <math.h>
5  #define N 999999999
6  int main()
7  {
8      double *v = malloc(sizeof(double));
9      #pragma omp parallel for
10     for (int i = 1; i <= N; i++)
11         *v = sqrt(i);
12     return 0;
13 }
```

Demonstration

- Run the assign-heap-omp program.

Discussion

- Compare the execution time of `assign-heap-omp` with previous programs.
- What is the reason for this performance difference?

Prime Number Counting

- We want to count the number of prime numbers.
- We start with an array of numbers, and assuming that every number is a prime number.
- We start with the smallest prime number in the array, and mark *every* multiple of it as *composite* (non-prime number).
- We repeat this process until no new prime numbers are found.

Example 4: (prime.c)

```
2  #include <stdio.h>
3  #include <math.h>
4  #include <stdlib.h>
5  #include <assert.h>
6  #include "omp.h"
7
8  #define N 400000000
9
10 char notPrime[N];
11 int nPrime = 0;
12 char primes[N];
```



```
14 int main(int argc, char *argv[])
15 {
16     assert(argc == 2);
17     int n = atoi(argv[1]);
18     int bound = round(sqrt(n));
19
20     for (int i = 2; i <= bound; i++)
21         if (!notPrime[i]) {
22 #pragma omp parallel for
23             for (int j = 2 * i; j < n; j += i)
24                 notPrime[j] = 1;
25     }
```

```
27     int nPrime = 0;
28 #pragma omp parallel for reduction(+ : nPrime)
29     for (int i = 2; i < n; i++)
30         if (notPrime[i] == 0)
31             nPrime++;
32
33     printf("number of prime is %d\n", nPrime);
34     return 0;
35 }
```

Variables

- Array `notPrime` keeps track of the status of a number. If we know a number `i` is *not* prime, we set `notPrime[i]` to 1.
- The prime numbers will be kept in another array `primes`.
- The range to be tested (`n`) is given as a command line argument.

Try All Possibilities

- Try all numbers from 2 to \sqrt{n} .
- If i is a prime number, mark the all multiple of i as *not* prime.
- It is obvious that the first for loop *cannot* be parallelized because of dependency, so we parallelize the second for loop.

Reduction

- The number of prime number can be obtained by counting the number of zeros in array `notPrime`.
- We use a reduction on the variable `nPrime` to simplify the process.

Demonstration

- Run the prime-uni program.
- Run the prime-omp program.

Discussion

- Compute the speedup of the previous prime counting program.
- Is there any optimization that can improve the performance?

Efficiency

- The previous program parallelized the inner for loop only.
- The previous program will go through multi-threading whenever it finds a prime number.
- This incurs overheads of creating and destroying the threads.

Spawn and Join



- Single thread for the outer loop.
- Multi-thread for the inner loop.

parallel Once

- We would like to avoid the overheads in creating and destroying threads, so we put a `parallel` in front of the first for loop.
- All threads run the *entire* two-level nested loop.
- Then, we share the workload of the second for loop to improve performance since most work is in the second loop.

Example 5: (prime-inner.c)

```
13 int main(int argc, char *argv[])
14 {
15     assert(argc == 2);
16     int n = atoi(argv[1]);
17     int bound = round(sqrt(n));
18
19     #pragma omp parallel
20     for (int i = 2; i <= bound; i++)
21         if (!notPrime[i])
22     #pragma omp for
23         for (int j = 2 * i; j < n; j += i)
24             notPrime[j] = 1;
```

Efficient

- All threads run the first loop, and share the second loop.
- It is OK for all threads to run the first for loop simultaneously, since they will synchronize at every second loop.
- Index variables i and j are private.

Spawn and Join



- Multi-thread for both outer and inner loops.
- The work in the outer loop is duplicated, but it is more efficient than creating and joining threads.

Demonstration

- Run the prime-uni program.
- Run the prime-omp program.
- Run the prime-inner-omp program.

Discussion

- Compare the execution time of all three prime counting programs.
- Is there any further optimization that can improve the performance?

Synchronization

- A `for` pragma will synchronize all threads before leaving the `for` statement.
- If the `for` statement is still within the same `parallel` directive; then a barrier synchronization will do.
- If the `for` statement is at the end of the `parallel` directive, threads will be joined by the master thread and destroyed.

nowait

- Sometimes, we do not wish the threads to wait for each other.
- Those finished earlier can go on to the next statement to improve performance.
- We can only do this if the following statement does *not* depend on the previous statement.
- In this case, we can use a `nowait` clause.

nowait

```
1  nowait
```

Two loops

- We place two for directives with a parallel directive.
- The first for has an ascending workload, and the second loop has a descending workload.

Example 6: (2for.c)

```
7  int main(int argc, char *argv[])
8  {
9      assert(argc == 3);
10     omp_set_num_threads(atoi(argv[1]));
11     int n = atoi(argv[2]);
12     printf("# of proc = %d\n", omp_get_num_procs());
13     printf("# of loop iterations = %d\n", n);
```

```
15     double t = omp_get_wtime();
16     #pragma omp parallel
17     {
18     #pragma omp for
19         for (int i = 0; i < n; i++)
20             sleep(i);
21     #pragma omp for
22         for (int i = n - 1; i >= 0; i--)
23             sleep(i);
24     }
25     printf("time = %f\n", omp_get_wtime() - t);
```

Demonstration

- Run the `2for-omp` program with 4 threads and 8 iterations, and observe the timing.

Discussion

- Describe the the execution time of `2for-omp` and make sure that it is reasonable.

Wait

- In the previous program, if the work of the second loop does not depend on the first loop, then we can let the threads go to the second loop instantly.
- Since the first loop has an *ascending* workload and the second loop has a *descending* workload, if we let the threads go to the second loop instantly, then the workload imbalance will reduce.
- We only need to add a `nowait` clause into the directive of the first loop.

Example 7: (2for-nowait.c)

```
15     double t = omp_get_wtime();
16 #pragma omp parallel
17 {
18 #pragma omp for nowait
19     for (int i = 0; i < n; i++)
20         sleep(i);
21 #pragma omp for
22     for (int i = n - 1; i >= 0; i--)
23         sleep(i);
24 }
25 printf("time = %f\n", omp_get_wtime() - t);
```

Demonstration

- Run the `2for-nowait-omp` program with 4 threads and 8 iterations, and observe the timing.

Discussion

- Describe the the execution time of `2for-omp` and make sure that it is reasonable.

nowait

- We consider our previous prime number counting program.
- Previously, the second marking `for` will synchronize before going back to the outside `for`.
- We would like to remove this synchronization and let each thread start the prime finding as soon as possible.
- This will not cause a race condition since the threads will still synchronize at the beginning of the next inner loop.

Example 8: (prime-inner-nowait.c) nowait

```
19 #pragma omp parallel
20     for (int i = 2; i <= bound; i++)
21         if (!notPrime[i])
22 #pragma omp for nowait
23     for (int j = 2 * i; j < n; j += i)
24         notPrime[j] = 1;
```

Demonstration

- Run `prime-inner-omp` and `prime-inner-nowait-omp` and compare their execution time.

Discussion

- Why can the `nowait` clause improve performance?

Compute π

- We calculate π by integrating $f(x) = \frac{4}{1+x^2}$, where x is from 0 to 1.
- Divide the interval into N pieces, and assume the area in each interval is a trapezoid, then sum the area of these N trapezoids into a variable area.
- Note that in each interval you only need to compute $\frac{4}{1+x^2}$ once.

Parallel Version

- We use x to denote the x coordinate, and $area$ for the area in the integral.
- We parallelize the program by adding `parallel` for `pragma` and declare x as `private`.

Example 9: (pi.c) Compute π

```
1  #include <omp.h>
2  #include <stdio.h>
3  #define N 1000000000
4  int main()
5  {
6      double x;
7      double area = 0.0;
8      double t = omp_get_wtime();
9      #pragma omp parallel for private(x)
10     for (int i = 0; i < N; i++) {
11         x = (i + 0.5) / N;
12         area += 4.0/(1.0 + x * x);
13     }
14     double pi = area / N;
15     t = omp_get_wtime() - t;
16     printf("execution time is %f\n", t);
17     printf("pi = %f\n", pi);
18     return 0;
19 }
```

Demonstration

- Run `pi-omp`.

Discussion

- Is the answer from the parallel version correct?
- Find out why the sequential version will not compile and fix it.
Then compare the the execution time of these two programs.

Atomic

- We did not declare `area` private because it has the final global answer.
- As a result the operation on `area` must be *atomic* to avoid race condition.
- We simply use `critical` directive on the loop body to ensure atomic condition.

Example 10: (pi-critical.c) Compute π

```
9  double x;  
10 double area = 0.0;  
11 double t = omp_get_wtime();  
12 #pragma omp parallel for private(x)  
13   for (int i = 0; i < N; i++)  
14 #pragma omp critical  
15   {  
16       x = (i + 0.5) / N;  
17       area += 4.0 / (1.0 + x * x);  
18   }  
19 double pi = area / N;  
20 t = omp_get_wtime() - t;  
21 printf("execution time is %f\n", t);  
22 printf("pi = %f\n", pi);
```

Demonstration

- Run the `pi-critical-omp` program.

Discussion

- Is the answer from the parallel version correct?
- Compare the the execution time of with the two previous programs.

Atomic

- It appears that we do not need to make the entire loop body critical because `x` is already private.
- We now only use critical directive on the area summation.

Critical Section

- It is essential to reduce the size of the critical section,
- We want a thread to get through a critical section as soon as possible, so that other threads can get into the critical section as well.

Example 11: (pi-critical-small.c) Compute π

```
9  double x;  
10 double area = 0.0;  
11 double t = omp_get_wtime();  
12 #pragma omp parallel for private(x)  
13   for (int i = 0; i < N; i++)  
14     {  
15         x = (i + 0.5) / N;  
16 #pragma omp critical  
17         area += 4.0 / (1.0 + x * x);  
18     }  
19 double pi = area / N;  
20 t = omp_get_wtime() - t;  
21 printf("execution time is %f\n", t);  
22 printf("pi = %f\n", pi);
```

Demonstration

- Run the `pi-critical-small-omp` program.

Discussion

- Is the answer from the parallel version correct?
- Compare the the execution time of with the three previous programs.

Private Variables

- The performance improvement is very limited by a smaller critical section because it is not very different from the previous implementation.
 - Two statements v.s. one statement.
- If there are many statements in the loop body the benefit will more obvious.

Critical Section

- The number of critical sections is enormous.
- We would like to remove these time consuming critical sections by letting all threads to work on its integrals by summing into its own area.
- The idea is to use a global array to store the individual area, then the master threads can add sum them up.

Array Implementation

- We need an array for threads to store its area.
- During each iteration a thread needs to call `omp_get_thread_num()` to know where to store its area, which is a significant overheads.

Example 12: (pi-array.c)

```
8  double x;  
9  double area[MAXT] = {0.0};  
10 double t = omp_get_wtime();  
11 #pragma omp parallel for private(x)  
12 for (int i = 0; i < N; i++) {  
13     x = (i + 0.5) / N;  
14     area[omp_get_thread_num()] +=  
15         4.0 / (1.0 + x * x);  
16 }  
17 t = omp_get_wtime() - t;  
18 double areaSum = 0.0;  
19 for (int i = 0; i < omp_get_num_procs(); i++)  
20     areaSum += area[i];  
21 double pi = areaSum / N;
```

Demonstration

- Run the `pi-array-omp` program.

Discussion

- Is the answer from the parallel version correct?
- Compare the the execution time of with the previous programs.
- Is there any way to reduce the overheads in calling `omp_get_thread_num()`?

Reduction

- We now use a reduction to compute the integral.
- The implementation is much cleaner and (hopefully) with better performance because it has been optimized by the OpenMP library.

Example 13: (pi-reduction.c) Compute π

```
7  double x;  
8  double area = 0.0;  
9  double t = omp_get_wtime();  
10 #pragma omp parallel for private(x) \  
11     reduction(+ : area)  
12     for (int i = 0; i < N; i++) {  
13         x = (i + 0.5) / N;  
14         area += 4.0 / (1.0 + x * x);  
15     }  
16     double pi = area / N;  
17     t = omp_get_wtime() - t;  
18     printf("execution time is %f\n", t);  
19     printf("pi = %f\n", pi);
```

Demonstration

- Run the `pi-reduction-omp` program.

Discussion

- Is the answer from the parallel version correct?
- Compare the the execution time of with the previous programs.

Game of Life

- A two-dimensional board with cells. A cell could either *live* or *dead*.
- The status of a cell evolves according to its status and the status of its eight neighbors.
- A dead cell with exactly *three* live neighbors becomes a live cell.
- A live cell with two or three live neighbors stays live, otherwise it becomes dead.

Double Buffer

- It is intuitive that we keep the status of cells in a two-dimensional array.
- Since the status of a cell depends on the status of others, if we modify a cell directly, it will affect the computation of other cells and cause race conditions during the update.
- Instead, we use *two* arrays A and B to store the status of cells. Initially, the status is in A .

Iterations

- We repeat the following steps.
 - We set the cell status of B according to A in the first, third, etc. iterations.
 - We set the cell status of A according to B in the second, fourth, etc. iterations.

Double Buffers

- We use two buffers A and B to avoid data inconsistency in updating the status of cells.
- We use a macro to compute the number of live neighboring cells – alive cells have 1, and dead cells have 0.
- We pad the broad boundary so that we can use a single macro to compute the number of live neighboring cells

Example 14: (life.c)

```
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <omp.h>
5
6  #define MAXN 4096
7  #define SIDE (MAXN + 2)
8
9  #define nLiveNeighbor(A, i, j) \
10     A[i + 1][j] + A[i - 1][j] + A[i][j + 1] + \
11     A[i][j - 1] + A[i + 1][j + 1] + A[i + 1][j - 1] + \
12     A[i - 1][j + 1] + A[i - 1][j - 1]
13
14 char A[SIDE][SIDE];
15 char B[SIDE][SIDE];
```

Print

- We use a simple printing routine to print the status of cells.

```
17 void print(char A[SIDE][SIDE], int n)
18 {
19     for (int i = 1; i <= n; i++) {
20         for (int j = 1; j <= n; j++)
21             printf("%2d ", A[i][j]);
22         printf("\n");
23     }
24 }
```

Input

- if the flag READINPUT is set the main program will read the size of the board, the number of generations, and the cell status from stdin.
- Otherwise it will generate a random input of size 4096 by 4096, and repeat for ten generations.

```
26 int main()
27 {
28     int n, generation, cell;
29     #ifdef READINPUT
30         scanf("%d%d", &n, &generation);
31         for (int i = 1; i <= n; i++)
32             for (int j = 1; j <= n; j++) {
33                 scanf("%d", &cell);
34                 A[i][j] = cell;
35             }
36     #else
37         n = 4096;
38         generation = 20;
39         for (int i = 1; i <= n; i++)
40             for (int j = 1; j <= n; j++)
41                 A[i][j] = rand() % 2;
42     #endif
```


Dead or Alive

- Depending on the generation, the program will set the cell status of B with A or in the other direction.
- A cell will be alive if it is dead now and has three live neighbors, or it is alive now and has two or three live neighbors.
- We use a `parallel for` directive to distribute the workload.

```
44     int nln;  
45     for (int g = 0; g < generation; g++)  
46         if (g % 2 == 0)  
47             #pragma omp parallel for          /* from A to B */  
48                 for (int i = 1; i <= n; i++)  
49                     for (int j = 1; j <= n; j++) {  
50                         nln = nLiveNeighbor(A, i, j);  
51                         B[i][j] = ((A[i][j] == 0 && nln == 3) ||  
52                             (A[i][j] == 1 && (nln == 2 || nln == 3)));  
53                     }  
54             else  
55                 #pragma omp parallel for          /* from B to A */  
56                     for (int i = 1; i <= n; i++)  
57                         for (int j = 1; j <= n; j++) {  
58                             nln = nLiveNeighbor(B, i, j);  
59                             A[i][j] = ((B[i][j] == 0 && nln == 3) ||  
60                                 (B[i][j] == 1 && (nln == 2 || nln == 3)));  
61                         }
```

Final Status

- if the flag PRINT is set the main program will output the final cell status.

```
63 #ifdef PRINT
64     if (generation % 2 == 0)
65         print(lifeA, n);
66     else
67         print(lifeB, n);
68 #endif
69     return 0;
70 }
```

Demonstration

- Run the `life-uni` program.
- Run the `life-omp` program.

Discussion

- Compute the speedup of the parallel program.
- What kind of scheduling policy was used?
- Is there any optimization that can improve the performance?