Advanced OpenMP Programming

Pangfeng Liu
National Taiwan University

April 17, 2020

- 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)

```c
#include <omp.h>
#include <stdio.h>
#include <math.h>
#define N 999999999
int main()
{
  double v;
#pragma omp parallel for
  for (int i = 1; i <= N; i++)
    v = sqrt(i);
  return 0;
}
```

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)

```c
#include <omp.h>
#include <stdio.h>
#include <math.h>
#define N 999999999
int main()
{
  double v;
#pragma omp parallel for private(v)
  for (int i = 1; i <= N; i++)
    v = sqrt(i);
  return 0;
}
```

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)

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define N 999999999
int main()
{
  double *v = malloc(sizeof(double));
#pragma omp parallel for
  for (int i = 1; i <= N; i++)
    *v = sqrt(i);
  return 0;
}
```

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)

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <assert.h>
#include "omp.h"

#define N 400000000

char notPrime[N];
int nPrime = 0;
char primes[N];
```

```
int main(int argc, char *argv[])
{
  assert(argc == 2);
  int n = atoi(argv[1]);
  int bound = round(sqrt(n));

  for (int i = 2; i <= bound; i++)
    if (!notPrime[i]) {
#pragma omp parallel for
      for (int j = 2 * i; j < n; j += i)
        notPrime[j] = 1;
    }
```

```c
  int nPrime = 0;
#pragma omp parallel for reduction(+ : nPrime)
  for (int i = 2; i < n; i++)
    if (notPrime[i] == 0)
      nPrime++;

  printf("number of prime is %d\n", nPrime);
  return 0;
}
```

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.

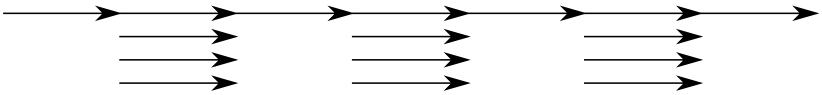
Discussion

- 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 every time a prime number is found.
- 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.
- The *entire* two level loop is run by all threads.
- Then we share the workload of the second for loop to improve performance, since most work is done in the second loop.

Example 5: (prime-inner.c)

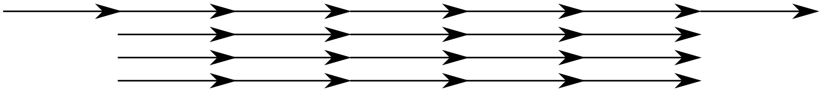
```c
int main(int argc, char *argv[])
{
  assert(argc == 2);
  int n = atoi(argv[1]);
  int bound = round(sqrt(n));

#pragma omp parallel
  for (int i = 2; i <= bound; i++)
    if (!notPrime[i])
#pragma omp for
      for (int j = 2 * i; j < n; j += i)
        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 the threads to wait for each other.
- Those finish 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 `nowait` clause.

nowait

```
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)

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

```c
  double t = omp_get_wtime();
#pragma omp parallel
  {
#pragma omp for
    for (int i = 0; i < n; i++)
      sleep(i);
#pragma omp for
    for (int i = n - 1; i >= 0; i--)
      sleep(i);
  }
  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 directly.
- Since the first for has an *ascending* workload, and the second loop has a *descending* workload, if we let the threads to go to the second loop then the workload imbalance will be reduced.
- We only need to add a `nowait clause` at the first `for` directive.

Example 7: (2for-nowait.c)

```
  double t = omp_get_wtime();
#pragma omp parallel
  {
#pragma omp for nowait
    for (int i = 0; i < n; i++)
      sleep(i);
#pragma omp for
    for (int i = n - 1; i >= 0; i--)
      sleep(i);
  }
  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 threads to 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

```
#pragma omp parallel
  for (int i = 2; i <= bound; i++)
    if (!notPrime[i])
#pragma omp for nowait
      for (int j = 2 * i; j < n; j += i)
        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

```c
#include <omp.h>
#include <stdio.h>
#define N 100000000
int main()
{
  double x;
  double area = 0.0;
  double t = omp_get_wtime();
#pragma omp parallel for private(x)
  for (int i = 0; i < N; i++) {
    x = (i + 0.5) / N;
    area += 4.0/(1.0 + x * x);
  }
  double pi = area / N;
  t = omp_get_wtime() - t;
  printf("execution time is %f\n", t);
  printf("pi = %f\n", pi);
  return 0;
}
```

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

```
  double x;
  double area = 0.0;
  double t = omp_get_wtime();
#pragma omp parallel for private(x)
  for (int i = 0; i < N; i++)
#pragma omp critical
    {
      x = (i + 0.5) / N;
      area += 4.0 / (1.0 + x * x);
    }
  double pi = area / N;
  t = omp_get_wtime() - t;
  printf("execution time is %f\n", t);
  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

```c
  double x;
  double area = 0.0;
  double t = omp_get_wtime();
#pragma omp parallel for private(x)
  for (int i = 0; i < N; i++)
    {
      x = (i + 0.5) / N;
#pragma omp critical
      area += 4.0 / (1.0 + x * x);
    }
  double pi = area / N;
  t = omp_get_wtime() - t;
  printf("execution time is %f\n", t);
  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)

```
  double x;
  double area[MAXT] = {0.0};
  double t = omp_get_wtime();
#pragma omp parallel for private(x)
  for (int i = 0; i < N; i++) {
    x = (i + 0.5) / N;
    area[omp_get_thread_num()] +=
      4.0 / (1.0 + x * x);
  }
  t = omp_get_wtime() - t;
  double areaSum = 0.0;
  for (int i = 0; i < omp_get_num_procs(); i++)
    areaSum += area[i];
  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

```c
  double x;
  double area = 0.0;
  double t = omp_get_wtime();
#pragma omp parallel for private(x) \
  reduction(+ : area)
  for (int i = 0; i < N; i++) {
    x = (i + 0.5) / N;
    area += 4.0 / (1.0 + x * x);
  }
  double pi = area / N;
  t = omp_get_wtime() - t;
  printf("execution time is %f\n", t);
  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 cell depends on the status of others, if we modify a cell directly it will affect the computation on other cells. This causes “race” condition during the update.
- Instead we use *two* arrays A and B to store the status of cells. In 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 – a live cell has 1 and a dead cell has 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)

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define MAXN 4096
#define SIDE (MAXN + 2)

#define nLiveNeighbor(A, i, j) \
  A[i + 1][j] + A[i - 1][j] + A[i][j + 1] + \
  A[i][j - 1] + A[i + 1][j + 1] + A[i + 1][j - 1] + \
  A[i - 1][j + 1] + A[i - 1][j - 1]

char A[SIDE][SIDE];
char B[SIDE][SIDE];
```

Print

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

```c
void print(char A[SIDE][SIDE], int n)
{
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++)
      printf("%2d ", A[i][j]);
    printf("\n");
  }
}
```

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.

```
int main()
{
  int n, generation, cell;
#ifdef READINPUT
  scanf("%d%d", &n, &generation);
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++) {
      scanf("%d", &cell);
      A[i][j] = cell;
    }
#else
  n = 4096;
  generation = 20;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
      A[i][j] = rand() % 2;
#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 only if it is dead now and has three live neighbors, or it is live and it has two or three live neighbors now.
- We simply use a `parallel for` directive to distribute the workload.

```c
  int nln;
  for (int g = 0; g < generation; g++)
    if (g % 2 == 0)
#pragma omp parallel for          /*  from A to B */
      for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++) {
          nln = nLiveNeighbor(A, i, j);
          B[i][j] = ((A[i][j] == 0 && nln == 3) ||
            (A[i][j] == 1 && (nln == 2 || nln == 3)));
        }
    else
#pragma omp parallel for          /*  from B to A */
      for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++) {
          nln = nLiveNeighbor(B, i, j);
          A[i][j] = ((B[i][j] == 0 && nln == 3) ||
            (B[i][j] == 1 && (nln == 2 || nln == 3)));
        }
```

Final Status

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

```
#ifdef PRINT
  if (generation % 2 == 0)
    print(lifeA, n);
  else
    print(lifeB, n);
#endif
  return 0;
}
```

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?