PARALLEL PROGRAMMING IN C LANGUAGE

Autor(i): Sergiu CRUCERESCU, Mihail KULEV
Conducător ştiinţific: dr. conf. Mihail KULEV
Universitatea Tehnică a Moldovei
Email: sergiu.crucerescu@gmail.com, mkmk@mail.md

Abstract: *Parallel computing is a form of computation in which many calculations are made at the same time. It is based on the principle that large problems can be divided in smaller ones. After the solutions of all small problems are found, they are combined to obtain the solution to the initial problem There are different forms of parallel computing: bit-level, instruction level, data, and task parallelism. Parallelism is used mainly in high-performance computing. Nowadays it became so important because of the physical constraints preventing frequency scaling. Parallel computer programs are more difficult to write than sequential ones, because there appear other types of software bugs, of which race conditions are the most common. Communication and synchronization between different tasks becomes very important in gaining performance increase.*

Cuvinte cheie: *parallel programming, race conditions, synchronization, threads, private / shared variables.*

1. Introduction

Subtasks in a parallel program are often called threads. Some parallel computer architectures use smaller versions of threads known as fibers, while others use bigger versions known as processes. Threads often need to update some data that is shared between them. This data is a part of the initial large problem that is needed for all subproblems. It might be constant, in case that threads don't modify it, or it might be variable, in case that all threads contribute in their way in modifying the data.

2. Basic parallel programming principles and issues

Parallel programs usually need to synchronize their threads at some steps, so that they were able to update the current states. This requires the use of a barrier. Barriers are typically implemented using a software lock.

Race conditions means controlling the access of shared variables by the threads. It must be done because threads should modify at the same time one variable. I race conditions are not respected, the computed result will be wrong. There might be critical sections in some algorithms that should be taken into account. For example, a variable is shared to all threads but it cannot be modified simultaneously by two or more of them. In this case we specify its section as critical. The execution of the program will be a little slower, but it will provide us the correct results.

When using concurrent programming we don't always get better performance. It could be actually even worse. Threads need to communicate with each other. This process needs resources. The more threads we use, the slower will be the program. At a moment, when we use too many threads, we will notice that the parallel computations take more time than the simple ones. This is known as parallel slowdown. So, there shouldn't be used more than we need threads. Their number is usually limited to the number of cores the processor has or to the number of threads at the physical layer.

Applications are often classified according to how often their subtasks need to synchronize or communicate with each other. An application uses fine-grained parallelism if its threads must communicate many times per second; it uses coarse-grained parallelism if they do not communicate many times per second, and it is embarrassingly parallel if they rarely or never have to communicate. Embarrassingly parallel applications are considered the easiest to make parallel.

3. About OpenMP

There are three major types of memory that are used for threads to communicate in an application: shared memory, distributed memory and shared distributed memory. In a shared memory program shared variables are are used. In distributed memory threads pass messages to each other. The shared distributed memory type combines the properties of the other two types of memory.

The C language doesn't have native support for multithreading. OpenMP extends its possibilities. It stands for Open Multi-Processing. It is a portable library used for working with threads. So, it enables the use multiple processor cores. OpenMP is available for the languages C/C++ and Fortran.
This is a “Hello world!” program for OpenMP:

```
int main()
{
 #pragma omp parallel
 printf("Hello, world.\n");
 return 0;
}
```

It should print Hello world on the screen as many times as many processors the computer has.

Here it is a simple parallelized loop:

```
int main()
{
   const int n = 10000;
   int i, a[n];

   #pragma omp parallel for
   for (i = 0; i < n; i++)
      a[i] = i;

   return 0;
}
```

Different threads will access array elements with different subscripts at the same time. By default the step between subscripts will be one.

OpenMP has some good advantages over other libraries. It is very simple. We don't have to change the entire section of the program that we want to make parallel. We use only directives that are easy to understand. The programmer has to take care only about the principles related to parallel programming. Everything else is done automatically.

There are some disadvantages, too. OpenMP doesn't have a good error handling. It runs efficiently only on shared-memory multiprocessor platforms. There are alos some minor disadvantages. In my program I noticed only the lack of error handling.

4. Using OpenMP in practice

My program computes the determinant of a square matrix using Gaussian elimination. I made a parallel implementation the algorithm and compared it with the initial one. I am also using the GMP library for computing large values of the determinant when the number of elements of the matrix is big.

```
double Gauss(float **matrix, int lines, mpf_t *result, float ***newArray, double *time)
{
        int i, j, k, l, sign;
        float **array = NULL, coefficient, temp;
        mpf_t determinant, number;
        mpf_init2(determinant, 1000);
        if (matrix)
        {
```

```
		mpf_set_d(determinant, 1);
		mpf_init_set_d(number, -1);
		sign = 1;

		array = declareArray(array, lines, lines);
		for (i = 0; i < lines; i++)
			for (j = 0; j < lines; j++)
				array[i][j] = matrix[i][j];


		*time = omp_get_wtime();


		for (k = 0; k < lines - 1; k++)
		{
			if (array[k][k] == 0)
			{
				l = lines - 1;
				while (array[l][k] == 0 && l > k)
					l--;
				if (l >= k + 1)
				{
					for (j = 0; j < lines; j++)
					{
						temp = array[k][j];
						array[k][j] = array[l][j];
						array[l][j] = temp;
					}
					sign = -1;
				}
			}

			#pragma omp parallel private(i, j, coefficient) shared(k, lines, array)
			{
			#pragma omp for
			for (i = k; i < lines - 1; i++)
			{
				coefficient = array[i + 1][k] / array[k][k];
				for (j = k; j < lines; j++)
				{
					array[i + 1][j] -= array[k][j] * coefficient;
				}
			}
			}
		}

		*time = omp_get_wtime() - *time;
		if (sign < 0)
			mpf_mul(determinant, determinant, number);
		for (i = 0; i < lines; i++)
		{
			mpf_set_d(number, array[i][i]);
			mpf_mul(determinant, determinant, number);
		}
	}
	else
		mpf_set_d(determinant, 0);
	*newArray = array;
	mpf_set(*result, determinant);
	mpf_clear(number);
	return mpf_get_d(determinant);
}
```

Table-1. Computation results.

Matrix size	Sequential, time (s)	Parallel, time, (s)
3x3	5.34346327e-008	8.3935447e-008
10x10	7.82310963e-008	1.08731911e-007
100x100	9.53441486e-008	0.0309998855
500x500	0.141000092	0.0780000382
1000x1000	0.935999912	0.655000097
2000x2000	7.33200007	5.74100003
3000x3000	24.227	18.564
5000x5000	111.556	85.738
10000x10000	886.597	706.759

5. Conclusions

Nowadays computing power is obtained from multicore processors. There are actual dual, quad and 8 core processors on the market. Why shouldn't we use the most of their power at computing something complex? Parallel programming will help us do this and it will save us some time. It isn't so hard to study concurrent computing. We should understand some important principles and then we'll be able to go further.

References

1. http://en.wikipedia.org/wiki/Parallel_computing
2. http://en.wikipedia.org/wiki/OpenMP
3. http://openmp.org
4. http://software.intel.com/en-us/articles/getting-started-with-openmp/