Performance Analysis - OpenMP

Parallel Programming

Arturo del Cerro Nil Bellmunt

November 2020

1 Introduction

In this work, we expose the experiments we carried out in order to improve the `lap.c` program with OpenMP tools. This experiments consist on parallelizing '`for`' loops, changing the number of threads, changing the schedule mode and other OpenMP functionalities. Also, in order to get a more trustworthy values in our experiments, we have executed all of them in MobaXterm using nodes exclusively by us, working out with the slurm workload manager using `sbatch`. In this work are presented, analyzed and discussed the improvement or deterioration in the performance of the code.

2 Results and conclussions

To begin with, unless it is specified, all the executions are made with a 1024x1024 matrix and 100 iterations. Also, we are parallelizing our programs with 4 threads. This will be analyzed and discussed later.
Just to compare with, here we show the performance of the original (baseline) code with the metrics above.

	Cycles $\cdot 10^9$	Instructions $\cdot 10^9$	Cache misses $\cdot 10^6$	CPUs	Time (s)
Baseline	31	17	265	0,99	9,45

Table 1

2.1 Parallelizing '`for`'

As in C accessing a matrix by columns is rather more efficient than accessing by rows, these changes also must be showed up in parallelizing a for in the columns or in the rows. In this case, we add the `#pragma omp for` clause into the `i` or `j` '`for`' in the `laplace_step` and `laplace_copy` functions. We do not do it in the `laplace_init` because it is called only one time, neither the `laplace_error` function because the well functioning of the program (commented in the Code section).

	Cycles $\cdot 10^9$	Instructions $\cdot 10^9$	Cache misses $\cdot 10^6$	CPUs	Time (s)
i for	67	34	242	3,96	5,24
j for	64	34	224	3,93	5,18

Table 2

In both cases we see a reduction of the execution time compared to the original one. This was expected, otherwise we could call in question the well functioning of OpenMP. Taking a look for the number of cycles and instructions we see that they increase nearly the double, but it seems to be normal. The CPUs metric will be commented later on.

2.2 Reduction

We try other parallelizing techniques, this time the `#pragma omp reduction` clause applied only in the `laplace_error` function with the operators `max` and `+`. Finally we try this improvements with the `#pragma omp for j` clause.

	Cycles $\cdot 10^9$	Instructions $\cdot 10^9$	Cache misses $\cdot 10^6$	CPUs	Time (s)
reduction (max)	124	69	467	3,98	9,65
reduction (+)	123	69	455	3,95	9,62
reduction (max,+)	122	69	441	3,97	9,48
j for, reduction (max,+)	65	34	221	3,98	5,14

Table 3

The results by separate are hugely disappointing. We see that separately, compared to the original code, both the `max` and `+` reduction make worse performance by taking almost the same time but with 4 more times the number of cycles and instructions and nearly the double of cache-misses. Mixing max and + in reduction also gets a worse performance but a bit better than separately in terms of time and cache-misses. One could think in discarding this reduction clause, but at the end we show that this changes, with the previous one, improve in a better way the code: the execution time is the best achieved until now, getting a 50% less execution time. Going a bit deeper into the results we can conclude that, in this particular case, the j for controls or limit the number of cycles, instructions and cache-misses.

2.3 Number of threads

When parallelizing we can choose how many threads we can have and depending on this number the performance can change extremely. We study the behaviour of different threads combined with the previous improvements, this is parallelization in the j for and reduction on max and +.

threads	Cycles $\cdot 10^9$	Instructions $\cdot 10^9$	Cache Misses $\cdot 10^6$	CPUs	time (s)
1	31	17	271	0,99	9,35
2	43	23	257	1,99	6,84
3	52	28	244	2,98	5,45
4	67	35	221	3,97	5,14
5	91	45	261	4,67	6,08
6	104	49	263	5,5	5,86
7	162	53	326	6,67	7,54
8	163	54	317	6,62	7,61
9	202	69	378	5,08	12,5
10	220	75	447	5,37	12,8

Table 4

Some of the metrics, like cycles, instructions, cache-misses and time seem to have linear-like behaviour respect the number of threads. This was expected because at some point, distributing the amount of work between too threads turn an easy task into a problem. But CPUs seem to have a minimum (1, as a non-parallelized program) and a maximum ≈ 8. This might be due to the number of cores of the system. Surely in a more powerful system this number would be greater.

We can conclude, and now justify why in the previous programs we use that (4) specific number of threads, that, for this particular Laplace problem the best number of threads seems to be 4 as it minimizes time and cache-misses, although it does not minimize the other metrics.

2.4 Schedule

The `schedule` clause helps us distribute the computation in chunks for each thread. The different clauses `schedule` has distribute the work in chunks of equal or not sizes. We try the different clauses with only the `#pragma omp for` modification in the j for and the results are showed in Table 5.

schedule (chunk size)	Cycles $\cdot 10^9$	Instructions $\cdot 10^9$	Cache Misses $\cdot 10^6$	CPUs	time (s)
static (256)	67	35	231	3,98	5,18
static (128)	66	34	232	3,98	5,15
static (64)	70	34	244	3,98	5,3
static (512)	85	46	260	3,98	6,65
dynamic	176	41	248	3,98	13,6
dynamic (32)	72	34	250	3,98	5,6
dynamic (64)	68	34	250	3,98	5,32
guided	68	34	240	3,95	5,37
guided (8)	71	35	255	3,97	5,58
guided (16)	69	34	241	3,98	5,31

Table 5

As we can see, the best performance of the code is achieved when we choose the static schedule. Both default (256 chunk-size) and 128 chunk-size gives us a very similar result and, for the sake of simplicity, we stuck with the default. With this schedule type we achieve same results as the previous better ones: same number of cycles, instructions and cache-misses. For other values of static gives a worse performance but not as worse as the dynamic schedule, that when its value is default, then its chunk-size is 1 and that is the reason it performs so bad. For other values of chunk-size, dynamic schedule performs well and same does the guided schedule. We can conclude that, there are not huge differences between different schedules because the issues that the schedule parallelization works are not extremely complicated.
Finally, combining this improvements with the previous ones we achieve the best values for the time until now in terms of time. The parallelized parts of the code are the `j for, reduction max +` and `static schedule`.

	Cycles $\cdot 10^9$	Instructions $\cdot 10^9$	Cache Misses $\cdot 10^6$	CPUs	time (s)
Final improved code	64	34	214	3,99	4,96

Table 6

In comparison with the initial code the cache-misses are reduced and also is the time. Due to the OpenMP parallelization the number of cycles and instructions is the double.

2.5 Matrix size

We want to check how the parallelization depends of the size of the matrix. In order to see this, we compute the metric values on a 1024x4096 and 4096x1024 matrix with all the previous improvements (j for, max + reduction and static schedule). In Table 7 we can see the results for this computations.

	Size	Cycles $\cdot 10^9$	Instructions $\cdot 10^9$	Cache Misses $\cdot 10^6$	CPUs	time (s)
Baseline	4096 x 1024	171	69	2564	0,99	50,7
Improved code	4096 x 1024	504	143	2474	3,99	39
Baseline	1024 x 4096	171	69	2487	0,99	50,7
Improved code	1024 x 4096	439	135	2577	3,99	33

Table 7

As we expected the numbers for the baseline are the same and as C access better by columns rather than rows it is reasonable that it performs better with more columns that rows.

3 Code

3.1 Improved C code with OpenMP

```c
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

float stencil ( float v1, float v2, float v3, float v4 )
{
  return (v1 + v2 + v3 + v4) / 4;
}

void laplace_step ( float *in, float *out, int n, int m )
{
  int i, j;
    for ( i=1; i < m-1; i++ )
    #pragma omp for schedule(static)
    for ( j=1; j < n-1; j++ ){
      out[j*m+i]= stencil(in[j*m+i+1], in[j*m+i-1], in[(j-1)*m+i], in[(j+1)*m+i]);}
}

float laplace_error ( float *old, float *new, int n, int m )
{
  int i, j;
  float error=0.0f;
  for ( i=1; i < m-1; i++ )
    #pragma omp reduction (max: error,+:j)
    for ( j=1; j < n-1; j++ )
      error = fmaxf( error, sqrtf( fabsf( old[j*m+i] - new[j*m+i] )));
  return error;
}

void laplace_copy ( float *in, float *out, int n, int m )
{
  int i, j;
  for ( i=1; i < m-1; i++ )
    #pragma omp for schedule(static)
    for ( j=1; j < n-1; j++ )
      out[j*m+i]= in[j*m+i];
}


void laplace_init ( float *in, int n, int m )
{
  int i, j;
  const float pi  = 2.0f * asinf(1.0f);
  memset(in, 0, n*m*sizeof(float));
  for (i=0; i<m; i++)  in[    i     ] = 0.f;
  for (i=0; i<m; i++)  in[(n-1)*m+i] = 0.f;
  for (j=0; j<n; j++)  in[   j*m    ] = sinf(pi*j / (n-1));
  for (j=0; j<n; j++)  in[ j*m+m-1 ] = sinf(pi*j / (n-1))*expf(-pi);
}

int main(int argc, char** argv)
{
  int n = 1024, m = 1024;
  int iter_max = 100;
  float *A, *Anew;

  const float tol = 1.0e-4f;
  float error= 1.0f;

  // get runtime arguments: n, m and iter_max
  if (argc>1) {  n        = atoi(argv[1]); }
  if (argc>2) {  m        = atoi(argv[2]); }
  if (argc>3) {  iter_max = atoi(argv[3]); }

  A    = (float*) malloc( n*m*sizeof(float) );
  Anew = (float*) malloc( n*m*sizeof(float) );

  // set boundary conditions
  laplace_init (A, n, m);
  A[(n/128)*m+m/128] = 1.0f; // set singular point

  printf("Jacobi relaxation Calculation: %d x %d mesh,"
        " maximum of %d iterations\n",
        n, m, iter_max );

  int iter = 0;

  // Here we create the parallelized region
  #pragma omp parallel firstprivate(iter) num_threads (4)
  {
  while ( error > tol && iter < iter_max )
  {
    iter++;
    laplace_step (A, Anew, n, m);
    error= laplace_error (A, Anew, n, m);
    laplace_copy (Anew, A, n, m);
    // In order to only print one time the number of iterations
    #pragma omp master
    if (iter % (iter_max/10) == 0) printf("%5d, %0.6f\n", iter, error);
  }
  }
  printf("Total Iterations: %5d, ERROR: %0.6f, ", iter, error);
  printf("A[%d][%d]= %0.6f\n", n/128, m/128, A[(n/128)*m+m/128]);

  free(A);
  free(Anew);
}
```

It is a must to comment the issues that have not been commented until here in order to discuss and cover all the OpenMP issues. First, we comment that as we want our program only to print one time the number of iterations we use `#pragma omp master` to executate this line by the master thread one time. Also, we include the `firstprivate` clause applied to the variable `iter` because each thread must use this variable as private but initialized before the parallelization.

3.2 Script

Here we show the script used to compile and execute our tests. Note that we are using sbatch in order to get more stable resutls by using some cores in a exclusive mode.

```
#!/bin/bash -l
#SBATCH --job-name=laplace.test
#SBATCH --output=results.txt

#SBATCH --partition=aolin.q
#SBATCH --exclusive

gcc -lm  -fopenmp laplace.c -o laplace
perf stat -e cycles,instructions,cache-misses,task-clock laplace  1024 1024  100
```

4 Final conclusions

It is time the discuss the global obtained results. Although in the last work (about Ofast and O3) we obtained a improvement nearly the 6500% in time, we think that, this time, our improvement ($\approx$ 200 %) is still a good work. We expected a result like this and worse results would be a disappointing issue to us. In this terms, we think that our solution is quite efficient because the final error and matrix values are the same as the baseline values. This ensure us the correctness of our code while we use more and more parallelizing techniques. But we wonder, could our code be improved? Surely. Probably there are parallelizing clauses that we did not use or places in the code that could still be parallelized. Despite of this we think that our work covers the main functionalities that OpenMP provide us.