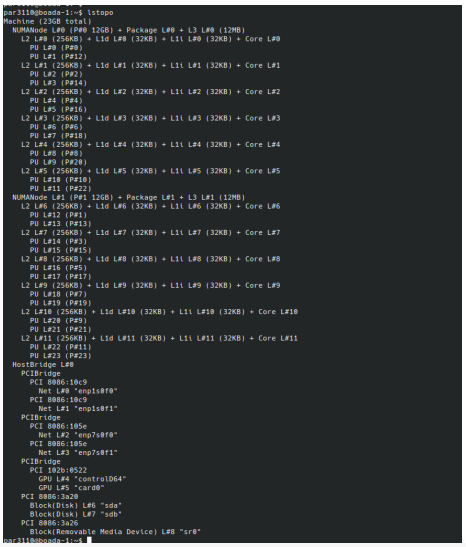
PARALLELISM
Laboratory 1:
EXPERIMENTAL
SET UP
AND TOOLS

Daniel Muñoz Arroyo, par3110
3/10/2021
3rd, Spring semester

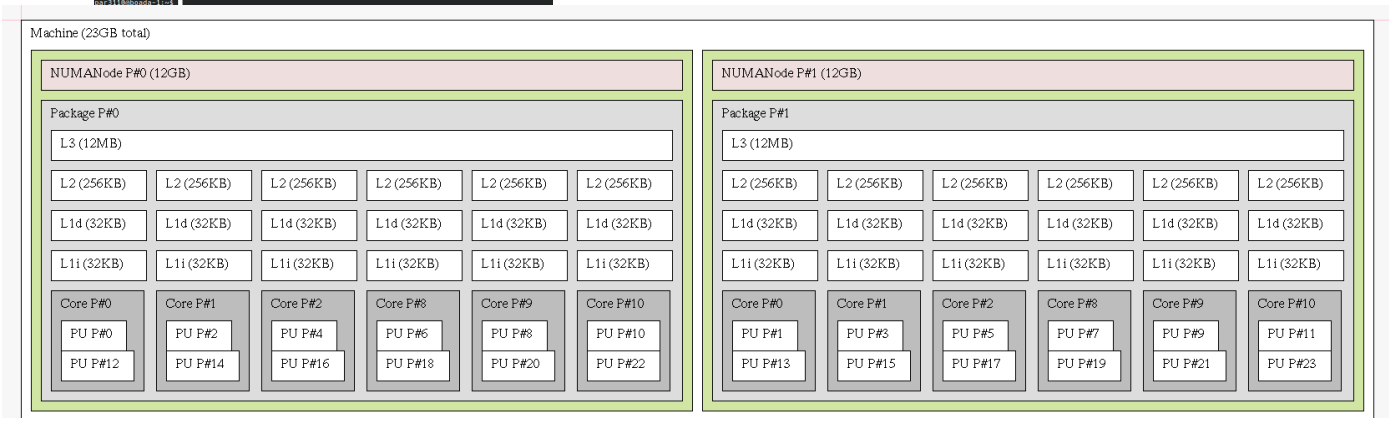
ÍNDEX

1.1 Boada structure

Boada is the computer that we will use to do the practises. To get information about it's structure, there are two commands (lstopo i lscpu).
One of them is lstopo and if we execute it, we can see the same as in the next picture:

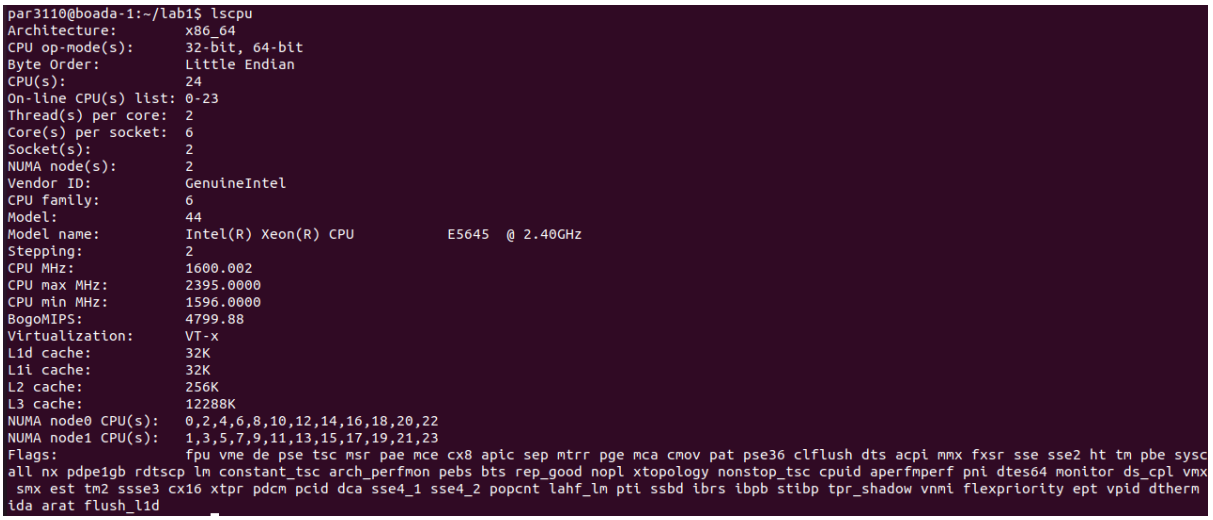


In this picture, we can see his internal structure, but I will explain better with the image that we obtain executing the command lstopo --of fig map.fig to create ot and "xfig map.fig" to see it.
After that, I made a screenshot of the result and we can see it in the picture above.



As we can see, boada have 1 NUMAnodes with 2 sockets and each socket have one line with cache level 3, one cache line of level 2, two cache lines of level 1 (one to instructions and the other for data) and 6 cores with to threads for every core but only one of the threads can be executed at a time.

The next picture is the result of the execution in the terminal of the command lscpu:



With the information and results that we took before, I can do the next table.

Number of sockets per node	2
Number of cores per socket	6
Number of threads per core	2
Maximum core frequency	2395 MHz
L1-I cache size (per-core)	32 KB
L1-D cache size (per-core)	32 KB
L2 cache size (per-core)	256 KB
Last-level cache size (per-socket)	12 MB
Main memory size (per socket)	12 GB
Main memory size (per node)	24 GB

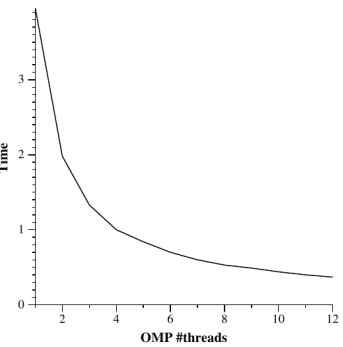
1.2 Strong vs Weak scalability

In this secction, we have to compile pi_omp with “make pi_omp” and execute it with 1000000000 iterations and changing between interactive and queued and modifying the number of threads (1,2,4 and 8). To do that in interactive mode, I executed with the command “./run-omp.sh pi_omp 1000000000 NUM_T”(NUM_T is the number of threads that I put (it changes between 1,2,4 and 8). This command shows the result in the terminal, but to execute in queued is in a file named Time-pi_omp-NUM_TE-boada-3 (NUM_TE is the number of threads executed.
I executed in queued mode using the command “sbatch ./submit-omp.sh pi_omp 1000000000 NUM_T” (NUM_T is the number of threads that I put (it changes between 1,2,4 and 8).

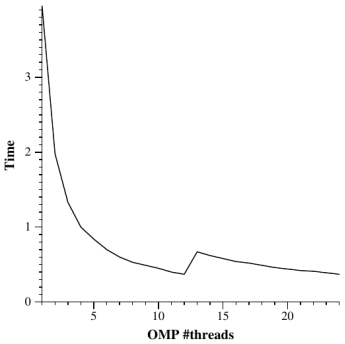
The results of the execution are in the next table:

#threads	Interactive				Queued			
	user	system	elapsed	% of CPU	user	system	elapsed	% of CPU
1	3,94	0	3,95	99	3,94	0	3,98	98
2	7,98	0	3,99	199	3,95	0	1,99	198
4	7,97	0,04	4,01	199	3,99	0	1,02	391
8	7,99	0,04	4,02	199	4,16	0,01	0,54	769

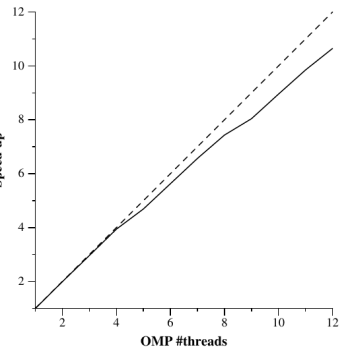
As we can see, there is a big difference in the percentage of CPU used executing with 4 and 8 threads and there is another big difference in the execution with 2,4 and 8 threads in the user time. It could be because if we execute interactively, the process starts when it arrives, but maybe is running because at least one CPU is not running any other thread or maybe not because there are all the CPU running other threads of other programs, so the elapsed time, in this case is more or less the same and in the percentage of CPU is 199 because it only were 2 CPUs executing the program or 1 in the case of 99%. The bit increment in the elapsed time, can be the overheads, because if we have more threads, we will have more overheads.
In queued mode, the CPU never will run process of different programs, so the program will use more CPUs at the same time and will stay less time to execute the program when we increment the threads because every thread will run at the same time (it is possible because there are 12 CPUs and at most 8 threads) and for that, in queued mode the percentage of CPU increments and the elapsed time decreases.



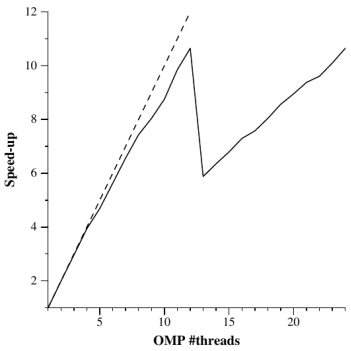
par3110
Min elapsed execution time
Thu Sep 16 13:19:16 CEST 2021



par3110
Min elapsed execution time
Thu Sep 16 13:32:15 CEST 2021



par3110
Speed-up wrt sequential time
Thu Sep 16 13:19:16 CEST 2021



par3110
Speed-up wrt sequential time
Thu Sep 16 13:32:15 CEST 2021

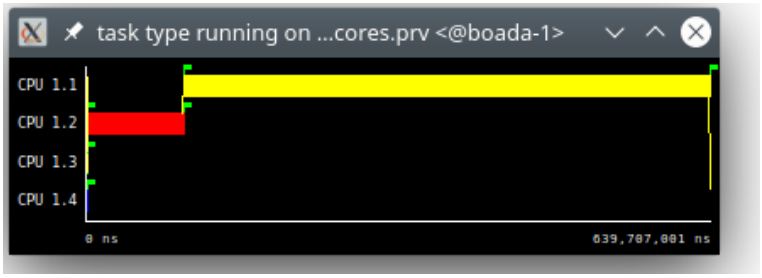
The left-left picture is the plots that I obtained. As we can see, every time that increments the threads, the time is decreasing because every process executes the parallel part of the program/(number of threads). It happens when there are at most 12 threads, because every CPU will execute one thread at the same time. But, in the left-right plot, if we execute more than 12 threads, we can see that the execution time will increase because now we have more threads than CPUs and some of them will wait until the CPUs are executing the other threads. The time can increase too, because if we have more threads, the overheads will increase too.

2 Exploring new task decompositions for 3DFFT

In this section I have to parallelize the 3dfft_tar.c file in a few versions.

2.1 Original version

The original version (v0), T1 = 639780 , as we can see in the TDG shown in the pdf of the practise and Tinf is the result of the execution of the program with infinite processors. In this case, with 4 processors is enough because it executes the program sequentially by the dependencies shown in the TDG. This is the picture of the execution:

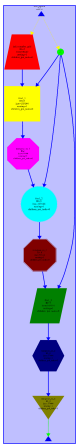


As we can see, Tinf = 639707 ns (is the number that is in the bottom-left of the picture).

Parallelism = T1/Tinf = 639780/639707 = 1 approximately, because it is the sequential version.

The calculations of the other versions of T1, Tinf and Parallelism I will do the same (T1 will be the same because is the same program but with a different number and size of the tasks, Tinf I will execute with paraver but incrementing the number of processors (128 because is the maximum that I can put, but is enough because not all the processors are doing something and Parallelism = T1/tinf)), so I won't show how I calculate it more, but you can see the results of the other versions in the next table:

Version	T1 (µs)	Tinf (µs)	Paral·lelisme
seq	639780	639707	1
v1	639780	639707	1
v2	639780	361439	1.77
v3	639780	154603	4.14
v4	639780	56080	11.41
v5	639780	35140	18.21

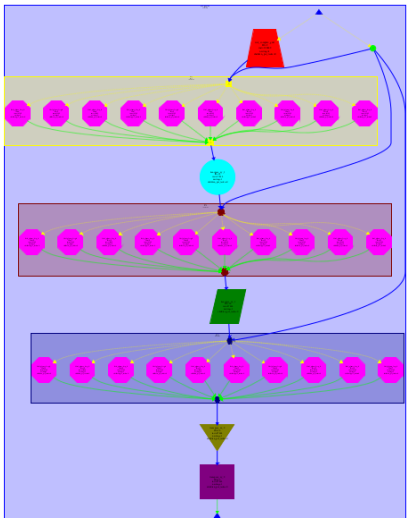


For the version 1 (v1), the difference is that the task ftts1_and_transpositions is decomposed in 7 tasks (one by every call to a function). But it is sequentially by the dependencies that we can see in the picture on the left. It is the TDG of v1.

2.2 version 2

For the version 2 (v2), we have to change the task creatiion in the ftts1_planes. Insteat of creating a single task in the call of the function, we will create a task in the external for. As we can see in the picture above.

```
void ffts1_planes(fftwf_plan p1d, fftwf_complex in_fftw[][N][N]) {
    int k,j;
    for (k=0; k<N; k++)  {
        tareador_start_task("ffts1_planes_loop_k");ç
        for (j=0; j<N; j++)
            fftwf_execute_dft( p1d, (fftwf_complex *)in_fftw[k][j][0],
                               (fftwf_complex *)in_fftw[k][j][0]);
        tareador_end_task("ffts1_planes_loop_k");
    }
}
```

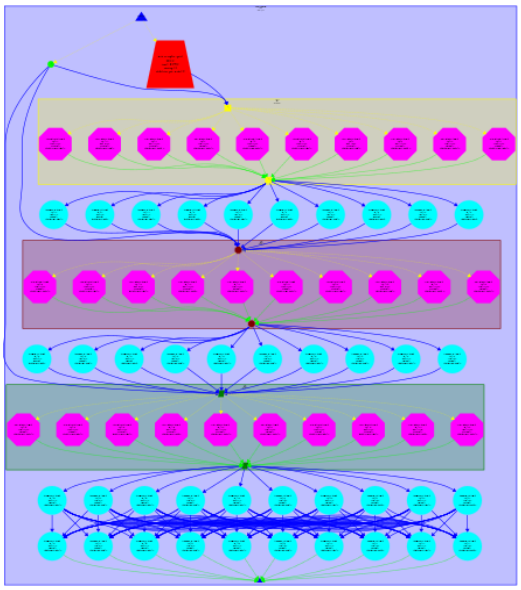


As we can see in the TDG (the picture on the left), some tasks will be in the same level (without dependencies between each one) and for that, the program won't be sequential, as we can see in the last table with the results.

2.3 Version 3

For version 3, we have to create tasks in every iteration of for k instead of in every call to function transpose_xy_plane and transpose_zx_plane, like in the picture below.

```
void transpose_xy_planes(fftwf_complex  tmp_fftw[][N][N], fftwf_complex in_fftw[][N][N]) {
    int k,j,i;

    for (k=0; k<N; k++) {
        tareador_start_task("transpose_zx_loop_k");
     for (j=0; j<N; j++) {
       for (i=0; i<N; i++)
       {
         tmp_fftw[k][i][j][0] = in_fftw[k][j][i][0];
         tmp_fftw[k][i][j][1] = in_fftw[k][j][i][1];
       }
     }
     tareador_end_task("transpose_xy_loop_k");
    }
}

void transpose_zx_planes(fftwf_complex in_fftw[][N][N], fftwf_complex tmp_fftw[][N][N]) {
    int k, j, i;

    for (k=0; k<N; k++) {
        tareador_start_task("transpose_zx_loop_k");
     for (j=0; j<N; j++) {
       for (i=0; i<N; i++)
       {
         in_fftw[i][j][k][0] = tmp_fftw[k][j][i][0];
         in_fftw[i][j][k][1] = tmp_fftw[k][j][i][1];
       }
     }
     tareador_end_task("transpose_zx_loop_k");
    }
}
```



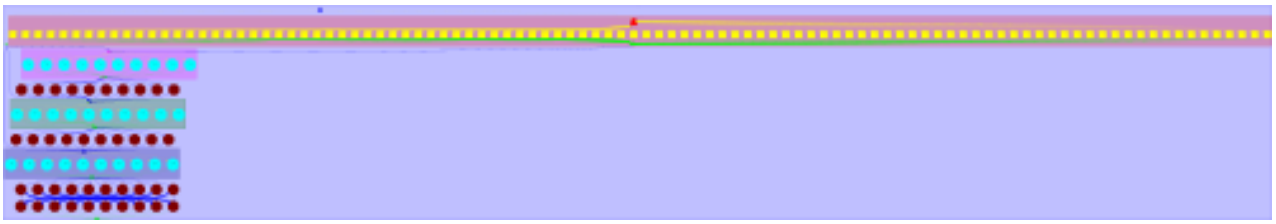
As we can see in the TDG (on the left), we divided the two tasks of both functions (one task for every iteration of the for k). Now we have more tasks in the same level without dependencies between them and the program will be more parallelizable than in the other versions.

2.4 Version 4

In version number 4, I have to create a Tareador task in the function init_complex_grid with the finer-grained task inside the body function, like the picture below.

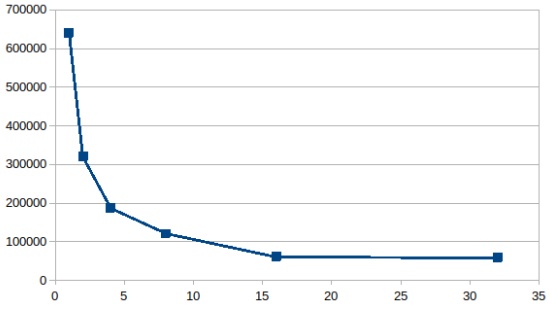
```
void init_complex_grid(fftwf_complex in_fftw[][N][N]) {
  int k,j,i;

  for (k = 0; k < N; k++) {
    for (j = 0; j < N; j++) {
    tareador_start_task("init_complex_grid_loop_j");
      for (i = 0; i < N; i++)
      {

        in_fftw[k][j][i][0] = (float) (sin(M_PI*((float)i)/64.0)+sin(M_PI*((float)i)/32.0)+sin(M_PI*((float)i/16.0)));
        in_fftw[k][j][i][1] = 0;
#if TEST
        out_fftw[k][j][i][0]= in_fftw[k][j][i][0];
        out_fftw[k][j][i][1]= in_fftw[k][j][i][1];
 #endif
      }
      tareador_end_task("init_complex_grid_loop_j");
    }
  }
}
```

Now, the TDG generated with tareador is the next:



We divided the function init_complex_grid into a lot of tasks, and for that reason, we have more parallelism than before.

As we can see in the next plot (X axis Processors and Y axis time in microseconds), we can reduce the execution time to 0,058180 s with 32 processors. But we won't have less significant time because if we put more processors, we will decrease the execution time of the init_compled_grid function. For that reason, we have a limitation that is that we don't have more tasks to parallelise more. For that reason, we can see in the plot that we will have an horizontal asymptote near the 0,056 s

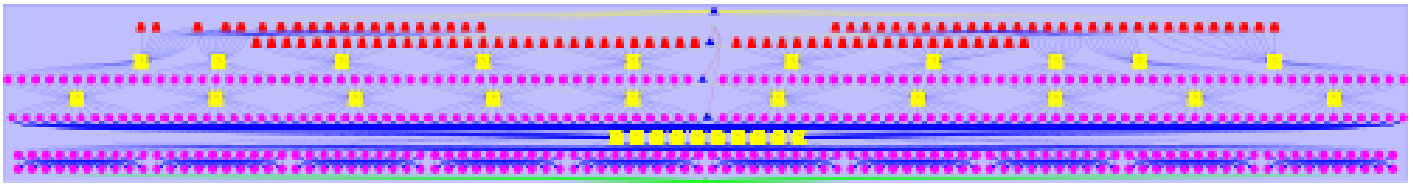


2.5 Version 5

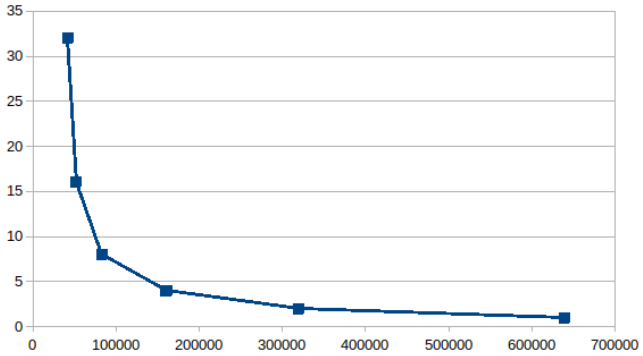
```
void transpose_xy_planes(fftwf_complex  tmp_fftw[][N][N], fftwf_complex in_fftw[][N][N]) {
    int k,j,i;

    for (k=0; k<N; k++) {
     for (j=0; j<N; j++) {
         tareador_start_task("transpose_zx_loop_j");
       for (i=0; i<N; i++)
       {
         tmp_fftw[k][i][j][0] = in_fftw[k][j][i][0];
         tmp_fftw[k][i][j][1] = in_fftw[k][j][i][1];
       }
     tareador_end_task("transpose_xy_loop_j");
     }
    }
}

void transpose_zx_planes(fftwf_complex in_fftw[][N][N], fftwf_complex tmp_fftw[][N][N]) {
    int k, j, i;

    for (k=0; k<N; k++) {
     for (j=0; j<N; j++) {
        tareador_start_task("transpose_zx_loop_j");
       for (i=0; i<N; i++)
       {
         in_fftw[i][j][k][0] = tmp_fftw[k][j][i][0];
         in_fftw[i][j][k][1] = tmp_fftw[k][j][i][1];
       }
     tareador_end_task("transpose_zx_loop_j");
     }
    }
}
```

Finally, the version 5 consists in make a finer-grained size for all the functions that call boada. For that, I will change the creation tasks of transpose_xy_planes and transpose_zx_planes to the more interior for like the picture shown below:

Whith that modification, we obtain the next TDG:



As we can see, it has a lot of nodes and there are a lot of nodes in the same levels. For that reason it will have more parallelism.



On the left plot (X axis Processors and Y axis time in microseconds), we can reduce the execution time to 0,051710 s with 32 processors.

We can have less time execution if we put more processors to achieve Tinf (0,35140) because if we put more processors, we will decrease the execution time of the function, and specially, the functions that is modified in this version (transpose_xy_planes and transpose_zx_planes). With that modification we can parallelise more the modified function and we will have less execution time than the version 4.

But the bad thing is that we can only have those times in tareador because it doesn't take the time to create the task. In real life, having the most finer-grained size is a bad idea because it will take a lot of time to create functions and more time for overheads. For that reason, it usually takes more time than if we execute in parallel with a bit less finer-grained size.

3 Understanding the parallel execution

I will show my results in the next table table(the explanations are above):

Version	Φ	Ideal S8	T1 (s)	T8	Real S8
Initial version in 3dfft_omp.c	0,729	2,760	2,233	1,628	1,913
New version improved Φ	0,968	6,536	2,336	0,599	3,900
Final version with reduced parallelization overheads	0,990	7,466	2,208	0,374	5,905

3.1 Initial version

To calculate T1, we have to execute the command “sbatch submit-omp.sh 3dfft_omp 1” and it generates a file .txt. If we look at it, it shows the time needed to execute the program (it’s divided into 3 times (3D FFT Plan Generator, Init Complex Grid FFT3D and Execution FFT3D), so we have to sum the three). In this case, the values are 0.000468 s, 0.585286 s and 1.647386 s respectively. The total is 2.23314 s. In the case of T8, it is the same as T1 but changing 1 by 8 threads in the command(“sbatch submit-omp.sh 3dfft_omp 8”) and the sum in file .txt is 1,167222 s (0,000419 + 0,589337 + 0,577466).

If we execute the command “sbatch submit-extrae.sh 3dfft_omp 1” it generates a 3dfft_omp-1-boada-2.prv file to see it with paraver. To do that, we have to execute “wxparaver 3dfft_omp-1-boada-2.prv”. If we load the configuration implicit_task_profile, we can see the next picture:

2D Implicit tasks profile @ 3dfft_omp-1-boada-2.prv <@boada-1>

Default

	47 (3dfft_omp.c, 3dfft_omp)	61 (3dfft_omp.c, 3dfft_omp)	75 (3dfft_omp.c, 3dfft_omp)	91 (3dfft_omp.c, 3dfft_omp)	146 (3dfft_omp.c, 3dfft_omp)
THREAD 1.1.1	283,284.24 us	715,384.86 us	629,538.31 us	16.21 us	2.50 us
Total	283,284.24 us	715,384.86 us	629,538.31 us	16.21 us	2.50 us
Average	283,284.24 us	715,384.86 us	629,538.31 us	16.21 us	2.50 us
Maximum	283,284.24 us	715,384.86 us	629,538.31 us	16.21 us	2.50 us
Minimum	283,284.24 us	715,384.86 us	629,538.31 us	16.21 us	2.50 us
StDev	0 us	0 us	0 us	0 us	0 us
Avg/Max	1	1	1	1	1

47 (3dfft_omp.c, 3dfft_omp)

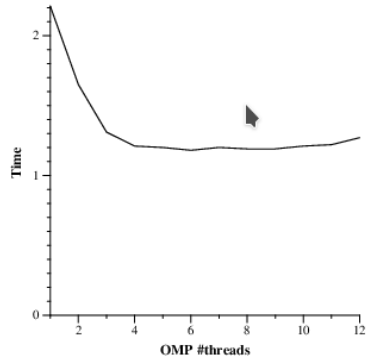
With that, we can calculate the execution time in parallel with the sum of all the parallel regions that is shown in the picture. Tpar = 283284,24+715384,86+629538,31+16,21 = 1628223,62 γs = 1,628 s

With T1 and Tpar, we can know Tseq because T1 = Tseq + Tpar => Tseq = T1-Tpar = 2,23314 - 1,628 = 0,60514 s.

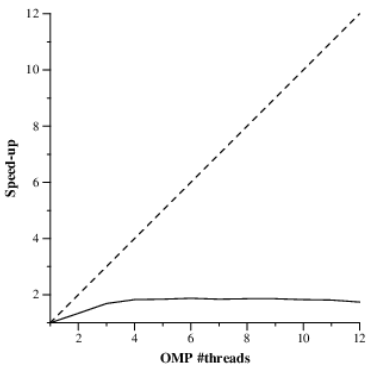
Now we can calculate Φ because Φ = Tpar/(Tpar+Tseq) = 1,628/(1,628 + 0,60514) = 0,729

The ideal S8 = 1/((1-Φ) + (Φ/8)) = 1/((1-0,729)+(0,729/8)) = 2,76
The real S8 = T1/T8 = 2,23314 / 1.167222 = 1,9132 s.

The real is not the same as the ideal because it has overheads and needs time to create tasks in real life, but ideally it doesn't exist.



par3110
Min elapsed execution time
Wed Sep 29 13:08:00 CEST 2021



par3110
Speed-up wrt sequential time
Wed Sep 29 13:08:00 CEST 2021

With the execution of "sbatch submit-strong-omp.sh 3dfft_omp" we can obtain the 3dfft_omp-1-12-3-strong-boada-2.ps and open it with "gs 3dfft_omp-1-12-3-strong-boada-2.ps". I made a screenshot of the plots with 1 to 12 threads of the execution of 3dfft_omp that it's shown in the picture on the left. As we can see, it takes less time with 2 to 6 threads but with more threads it takes a bit more time than with 6. It could be by overheads.

3.2 Second version

The calculations will be the same as before, but modifying the 3dfft_omp.c.
The first modify to do is uncomment the #pragma omp parallel, #pragma omp single and the second taskloop like the picture below.

```
void init_complex_grid(fftwf_complex in_fftw[][N][N]) {

  #pragma omp parallel
  #pragma omp single
//    #pragma omp taskloop
  for (int k = 0; k < N; k++)
    #pragma omp taskloop firstprivate(k)
    for (int j = 0; j < N; j++)
      for (int i = 0; i < N; i++) {
        in_fftw[k][j][i][0] = (float) (sin(M_PI*((float)i)/64.0)+sin(M_PI*((float)i)/32.0)+sin(M_PI*((float)i/16.0)));
        in_fftw[k][j][i][1] = 0;
#if TEST
        out_fftw[k][j][i][0]= in_fftw[k][j][i][0];
        out_fftw[k][j][i][1]= in_fftw[k][j][i][1];
#endif
      }
}
```

After executing a make, we can know the value of T1, like before, we have to execute the command “sbatch submit-omp.sh 3dfft_omp 1” and it generates a file .txt. If we look at it, it shows the time needed to execute the program. In this case, T1 = 0,000452 + 0,585318 + 1,750059 = 2,335829 s. In the case of T8, it is the same command as T1 but changing 1 by 8 threads(“sbatch submit-omp.sh 3dfft_omp 8”) T8 = 0,000440 + 0,090570 + 0,507881 = 0,598891 s

If we execute the command “sbatch submit-extrae.sh 3dfft_omp 1” it generates a 3dfft_omp-1-boada-2.prv file to see it with paraver with the command “wxparaver 3dfft_omp-1-boada-2.prv”. If we load the configuration implicit_task_profile, we can see the nexts pictures:

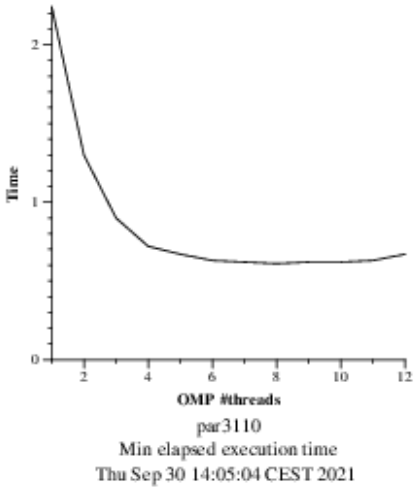
	29 (3dfft_omp.c, 3dfft_omp)	47 (3dfft_omp.c, 3dfft_omp)	61 (3dfft_omp.c, 3dfft_omp)	75 (3dfft_omp.c, 3dfft_omp)
THREAD 1.1.1	602,781.55 us	283,807.87 us	744,291.89 us	631,121.29 us
Total	602,781.55 us	283,807.87 us	744,291.89 us	631,121.29 us
Average	602,781.55 us	283,807.87 us	744,291.89 us	631,121.29 us
Maximum	602,781.55 us	283,807.87 us	744,291.89 us	631,121.29 us
Minimum	602,781.55 us	283,807.87 us	744,291.89 us	631,121.29 us
StDev	0 us	0 us	0 us	0 us
Avg/Max	1	-	1	1

91 (3dfft_omp.c, 3dfft_omp)	146 (3dfft_omp.c, 3dfft_omp)
17.14 us	2.62 us
17.14 us	2.62 us
17.14 us	2.62 us
17.14 us	2.62 us
17.14 us	2.62 us
0 us	0 us
1	1

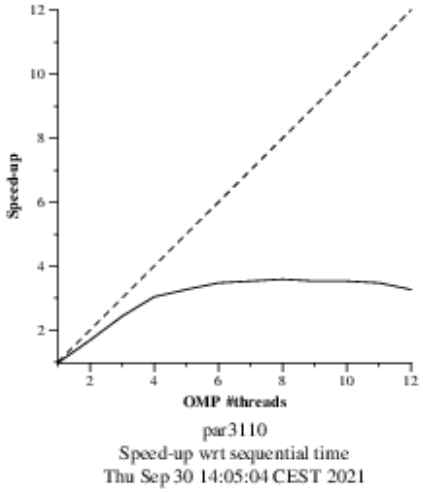
We can see in the two previous pictures, every parallelization time of each piece of codes that are in the program with #pragma omp parallel. With that, Tpar = 602781,55+283807,87 + 744291,89 + 631121,29 + 17,14 +2,62 = 2262022,36 γs = 2,262 s
T1 = Tseq + Tpar => Tseq = T1-Tpar = 2,335829 - 2,262 = 0,073829 s.
Φ = Tpar/(Tpar+Tseq) = 2,262/(2,262 + 0,073829) = 0,968

The ideal S8 = 1/((1-Φ) + (Φ/8)) = 1/((1-0,968)+(0,968/8)) = 6,536
The real S8 = T1/T8 = 2,335829 / 0,598891 = 3,90

The real is not the same as the ideal because it has overheads and needs time to create tasks in real life, but ideally it doesn't exist.



With the execution of "sbatch submit-strong-omp.sh 3dfft_omp" we can obtain the 3dfft_omp-1-12-3-strong-boada-2.ps and open it with "gs 3dfft_omp-1-12-3-strong-boada-2.ps". I made a screenshot of the plots with 1 to 12 threads of the execution of 3dfft_omp that it's shown in the picture on the left and bottom. As we can see, it takes less time with 2 to 6 threads, but with more threads it takes a bit more time than with 6. It could be by overheads.



3.3 Third version

For the last version, we have to move the taskloop one line up of all the functions (comment the taskloop with firstprivate and uncomment the taskloop without firstprivate), like I did in init_complex_grid function shown above:

```
void init_complex_grid(fftwf_complex in_fftw[][N][N]) {

  #pragma omp parallel
  #pragma omp single
  #pragma omp taskloop
  for (int k = 0; k < N; k++)
//     #pragma omp taskloop firstprivate(k)
    for (int j = 0; j < N; j++)
      for (int i = 0; i < N; i++) {
        in_fftw[k][j][i][0] = (float) (sin(M_PI*((float)i)/64.0)+sin(M_PI*((float)i)/32.0)+sin(M_PI*((float)i/16.0)));
        in_fftw[k][j][i][1] = 0;
#if TEST
        out_fftw[k][j][i][0]= in_fftw[k][j][i][0];
        out_fftw[k][j][i][1]= in_fftw[k][j][i][1];
#endif
      }
}
```

Finally I have to calculate everything like before.
After executing a make, we can know the value of T1, we have to execute the command “sbatch submit-omp.sh 3dfft_omp 1” and it generates a file .txt. If we look at it, it shows the time needed to execute the program. In this case, T1 = 0,009351 + 0,592857 + 1,605333 = 2,207541 s. In the case of T8, it is the same command as T1 but changing 1 by 8 threads (“sbatch submit-omp.sh 3dfft_omp 8”): T8 =0,000478 + 0,083731 + 0,289664 = 0,373873 s

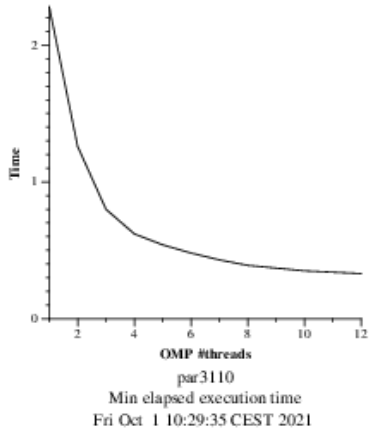
If we execute the command “sbatch submit-extrae.sh 3dfft_omp 1” it generates a 3dfft_omp-1-boada-2.prv file to see it with paraver with the command “wxparaver 3dfft_omp-1-boada-2.prv”. If we load the configuration implicit_task_profile, we can see the next picture:

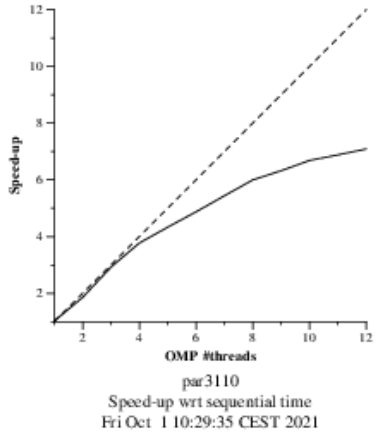
	91 (3dfft_omp.c, 3dfft_omp)	146 (3dfft_omp.c, 3dfft_omp)	t_omp)	61 (3dfft_omp.c, 3dfft_omp)	75 (3dfft_omp.c, 3dfft_omp)
T	15.88 us	2.57 us	86.36 us	807,803.54 us	587,857.35 us
	15.88 us	2.57 us	86.36 us	807,803.54 us	587,857.35 us
	15.88 us	2.57 us	86.36 us	807,803.54 us	587,857.35 us
	15.88 us	2.57 us	86.36 us	807,803.54 us	587,857.35 us
	15.88 us	2.57 us	86.36 us	807,803.54 us	587,857.35 us
	0 us	0 us	0 us	0 us	0 us
	1	1	1	1	1

We can see in the two previous pictures, every parallelization time of each piece of codes that are in the program with #pragma omp parallel. With that,
Tpar = 591003,44 + 255386,36 + 807803,54 + 587857,35 + 15,88 + 2,57 = 2242069,14 γs = 2,242 s.

We can see that T1 < Tpar, but i could be because I calculate T1 without paraver and Tpar with it and generate the trace of paraver takes some time, so I will assume Tpar = 0,99 * T1 because The code is approximately 0,99 % parallelizable.

Tpar = 0,99 * 2,207541 = 2,18546559 s
T1 = Tseq + Tpar => Tseq = T1 - Tpar = 2,207541 - 2,18546559 = 0,02207541 s.
Φ = Tpar/(Tpar+Tseq) = 2,18546559/(2,18546559+0,022541) = 0,989791244
The ideal S8 = 1/((1-Φ) + (Φ/8)) = 1/((1-0,989791244) + (0,989791244/8)) = 7,466438648
The real S8 = T1/T8 = 2,207541/0,373873 = 5,904521054





With the execution of “sbatch submit-strong-omp.sh 3dfft_omp” we can obtain the 3dfft_omp-1-12-3-strong-boada-3.ps and open it with “gs 3dfft_omp-1-12-3-strong-boada-3.ps”. I made a screenshot of the plots with 1 to 12 threads of the execution of 3dfft_omp that it’s shown in the picture on the left and bottom. As we can see, it takes less time when we increment the threads.

This is the best version of the program because
It’s 99% parallelizable and if we increment the threads (1 to 12), the overheads are less than the time that we reduce parallelizing.