PAR Laboratory Assignment

Course 2018/19 (Fall semester)

Lab 3: Divide and Conquer parallelism with OpenMP: Sorting

Pablo Vizcaino, Guillem Ramírez

User: Par2206

November 20, 2018

Contents

1 Introduction

In this laboratory assignment we are going to study and analyze the parallelization of Divide and Conquer algorithms using *OpenMP* , specifically Mergesort.

We will be working with a Multisort algorithm that combines a Divide and Conquer strategy (mergesort) with an iterative strategy (quicksort).

Multisort algorithm divides the vector in parts and when the divisions are small enough the quicksort algorithm is applied, named *basicsort* in the code. Finally the sorted parts get merged into bigger sorted parts until we get the sorted vector.

In this laboratory assignment we are going to analyze, parallelize and evaluate this Divide and Conquer aproach using *OpenMP* and Barcelona Supercomputing Center performance analysis tools[1]. We are going to explore different strategies and pragmas to program the optimum version.

[1]tools.bsc.es

2 Task decomposition analysis using Tareador

In this section we are going to analyze the task decomposition for the mergesort algorithm. We are going to use tareador to analyze dependencies and consequently set up the parallelization strategies.

First of all we nedded to add Tareador instrumentation in order to get dependency graphs. We added to the base code `tareador_start_task(task_name)` and `tareador_end_task(task_name)` clauses to the recursive calls of multisort:

```
tareador_start_task("multi1");
multisort(n/4L, &data[0], &tmp[0]);
tareador_end_task("multi1");
```

Also, we added them to mergesort recursive calls:

```
tareador_start_task("merge1");
merge(n, left, right, result, start, length/2);
tareador_end_task("merge1");
```

Finally we compiled[2] and runned[3] the modified version of the code obtaining the following graph.

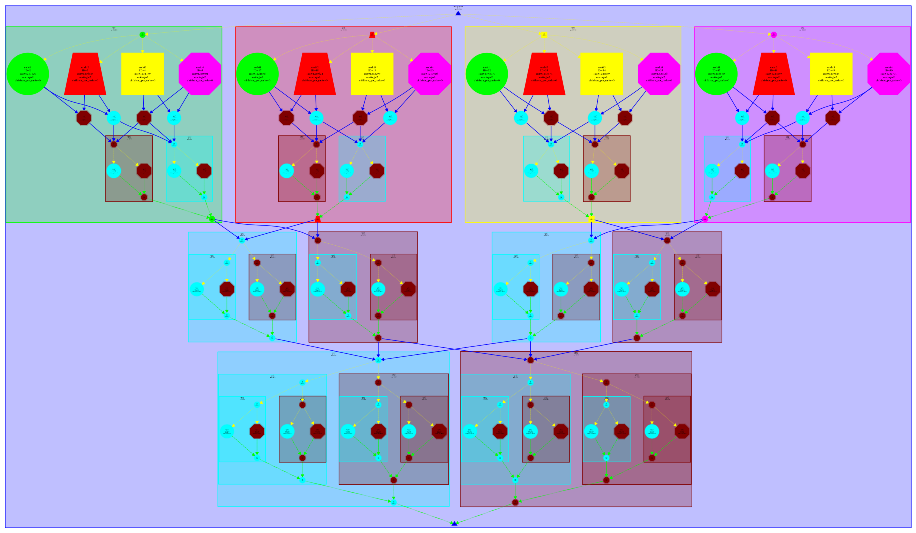


Figure 1: Tareador task decomposition graph

From the graph, we can observe that there are dependencies between tasks. Specifically, each merge task has to wait to the two multisort tasks that are sorting the two parts that it needs to merge. Lastly, another merge has to wait the last merges in order to merge both of them.

[2]make multisort-tareador

[3]./run-tareador.sh multisort-tareador

In order to evaluate its scallability we simulated its execution with 1,2,4,8,16,32 and 64 processors, gathered the execution time and calculated the respective speed-up from the version with 1 cpu.

In the table we can see that the speed-up is really close to the ideal case , because doubling the resources nearly doubles the speed-up. We can also observe that after 16 processors, adding more resources doesn't improve the performance. This is due to the size of the problem; we only divide the vector a finite amount of times, creating a finite amount of parallel tasks. Once we reach that limit, adding more resources doesn't increase the performance.

#cpu	t(s)	s
1	20.33	1
2	10.17	1.99
4	5.086	3.99
8	2.55	7.97
16	1.29	15.76
32	1.29	15.76
64	1.29	15.76

3 Parallelization strategies

In this section we are going to compare two recursive parallelization strategies: leaf and tree.

3.1 Leaf strategy

First we implemented the leaf strategy. To do so, we don't create tasks until the code reaches the base case of the recursion. In the merge function we just have to add a task creation in the base case.

```
void merge(long n, T left[n], T right[n], T result[n*2],
           long start, long length) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                #pragma omp task
                basicmerge(n, left, right, result, start, length);
        } else {
                // Recursive decomposition
                merge(n, left, right, result, start, length/2);
                merge(n, left, right, result, start + length/2,
                length/2);
        }
}
```

In the multisort routine we added a task creation pragma in the base case and also implemented a task synchronization mechanism using `#pragma omp taskwait` in order to respect data dependencies.

```
void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                multisort(n/4L, &data[0], &tmp[0]);
                multisort(n/4L, &data[n/4L], &tmp[n/4L]);
                multisort(n/4L, &data[n/2L], &tmp[n/2L]);
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
                #pragma omp taskwait
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0,
                      n/2L);
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L],
                      0, n/2L);
                #pragma omp taskwait
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        } else {
                // Base case
                #pragma omp task
                basicsort(n, data);

        }
}
```

We can observe this strategy's impact in the paraver timeline. It can be seen that only the first thread is creating tasks and synchronizing them. The other threads are only executing tasks, which are the base cases.

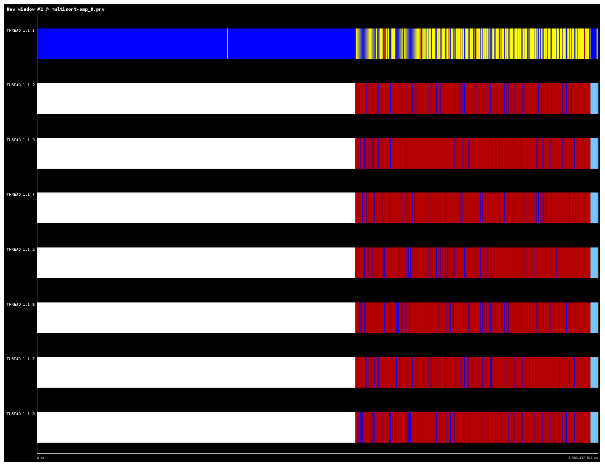


Figure 2: Paraver timeline for leaf strategy

The number of task creations (bursts) that we obtained is 11264 and the execution time of this parallelizable part is 1.525s. This will be usefull to compare it with the next strategy.

In order to analyze the strategy's performance, we obtained the strong speed-up plots.

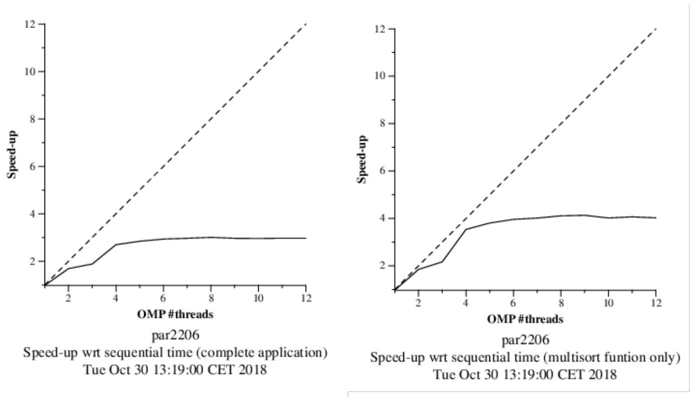


Figure 3: Strong speed-up plots for leaf strategy

We can observe that after 4 threads the speed is nearly constant, making it far from ideal.

3.2 Tree strategy

The second strategy that we implemented was the tree strategy. To do so, we create the tasks in each recursion call. In the merge function we added a `#pragma omp task` before each recursion call and a `#pragma omp taskwait` at the end for synchronization.

```
void merge(long n, T left[n], T right[n], T result[n*2],
           long start, long length) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                basicmerge(n, left, right, result, start, length);
        } else {
                // Recursive decomposition
                #pragma omp task
                 merge(n, left, right, result, start, length/2);
                #pragma omp task
                merge(n, left, right, result, start + length/2,
                      length/2);
                #pragma omp taskwait
        }
}
```

In the multisort function we applied the same logic. We added a `#pragma omp task` before each recursion call and then added the needed task synchronization pragmas guided by the dependencies we found at section 2.

```
void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                #pragma omp task
                multisort(n/4L, &data[0], &tmp[0]);
                #pragma omp task
                multisort(n/4L, &data[n/4L], &tmp[n/4L]);
                #pragma omp tas
                multisort(n/4L, &data[n/2L], &tmp[n/2L]);
                #pragma omp task
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
                #pragma omp taskwait

                #pragma omp task
                merge(n/4L, &data[0], &data[n/4L], &tmp[0],0,n/2L);
                #pragma omp task
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L],
                      0, n/2L);
                #pragma omp taskwait

                #pragma omp task
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
                #pragma omp taskwait
        } else {
                // Base case
                basicsort(n, data);
        }
}
```

When looking at the paraver timeline of this strategy, we can see that unlike the leaf strategy all threads are creating and synchronizing tasks.

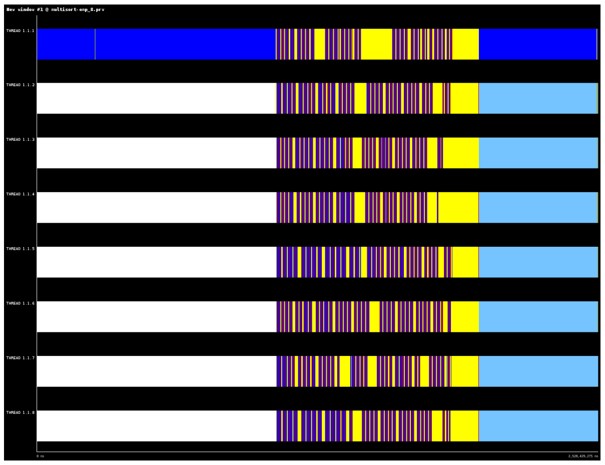
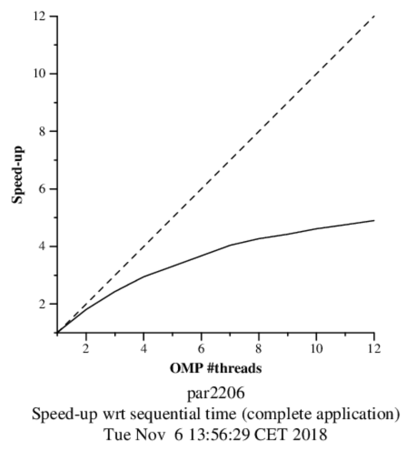


Figure 4: Paraver timeline for tree strategy

Another metric when comparing it to the leaf strategy is the number of task creations of the new strategy. In this case, we obtained 20821 burst, 78% more than the leaf version. This makes sense considering that the tree strategy creates tasks as it travels the recursive tree.

The most distinctive metric that helped us choose the tree strategy as the better one (in this given problem) is the execution time of the parallelizable part. The time we obtained is 0.88s, a speed-up of 1.73 when comparing it to the leaf strategy.

For comparing both strategies we also obtained the strong scalability plots of the tree version.



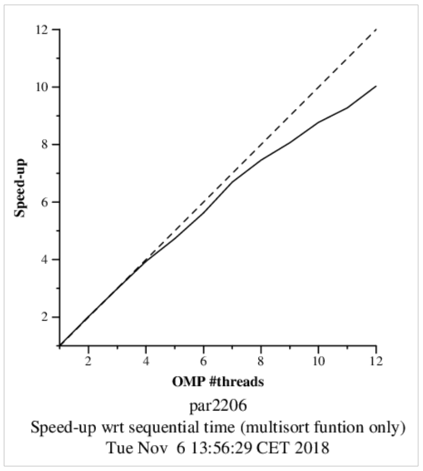


Figure 5: Strong scalability plot for tree strategy

From the strong scalability plots we can observe that this strategy is much better than the leaf one, as it gets much closer to the ideal speed-up.

4 Improving tree strategy

As wee seen in the section 3.2, the tree strategy went much better on solving the problem, as its execution time nearly halved. In this section we are going to implement some improvements with the goal to get a smaller execution time, reducing overheads.

4.1 Cut off

A problem that the original tree version has is that there is no cutoff mechanism implemented. So in big problem sizes we can find out that we have overheads; the program would create exponentially too many tasks as it goes deeper in recursion.

To control the creation of tasks, we need to implement a cut off control mechanism. This is done adding `#task final(d>=CUTOFF) mergeable` and modifying our functions to have a depth counter. What the `final` clause does is stopping the parallelization of the program once the condition evaluated is true. When that happens, the threads that reach the pragma will switch to the execution of the new task instead of running it simultaneously, effectively stopping the parallelization. One important note in our implementation is that the merge function doesnt get a new value of depth when it's called from the multisort function; even tho it does its own recursion.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start,
           long length, int d) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base casea
                basicmerge(n, left, right, result, start, length);
        } else {
                // Recursive decomposition
                #pragma omp task final(d>=CUTOFF) mergeable
                 merge(n, left, right, result, start, length/2,d+1);
                #pragma omp task final(d>=CUTOFF) mergeable
                merge(n, left, right, result, start + length/2,
                      length/2,d+1);
                #pragma omp taskwait
        }
}

void multisort(long n, T data[n], T tmp[n], int d) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                #pragma omp task final(d>=CUTOFF) mergeable
                multisort(n/4L, &data[0], &tmp[0],d+1);
                #pragma omp task final(d>=CUTOFF) mergeable
                multisort(n/4L, &data[n/4L], &tmp[n/4L],d+1);
                #pragma omp task final(d>=CUTOFF) mergeable
                multisort(n/4L, &data[n/2L], &tmp[n/2L],d+1);
                #pragma omp task final(d>=CUTOFF) mergeable
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L],d+1);
                #pragma omp taskwait
                #pragma omp task final(d>=CUTOFF) mergeable
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L
                      ,d+1);

                #pragma omp task final(d>=CUTOFF) mergeable
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L]
                      , 0, n/2L, d+1);
                #pragma omp taskwait

                #pragma omp task final(d>=CUTOFF) mergeable
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n,
                      d+1);
                #pragma omp taskwait
        } else {
                // Base case
                basicsort(n, data);

        }
}
```

Once we have this mechanism implemented, we need to evaluate the optimum value of the *CUTOFF* variable. To do so, we used the script that was given to us that tries different values and calculates the execution time with that *cutoff*. We made a small modification, making the script try some more and bigger values of the cutoff to really see the optimal one.

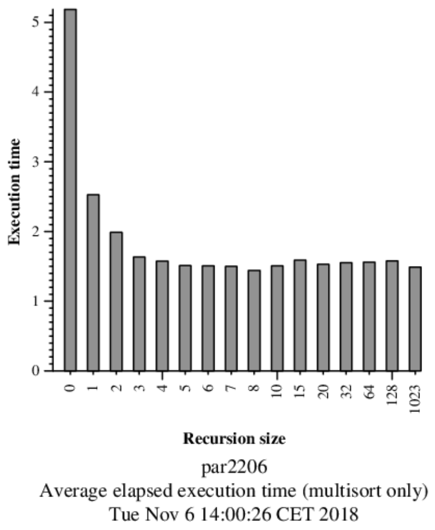


Figure 6: Execution time with different values of cutoff

As it can be seen in the figure, the optimal *cutoff* value is 8.

The next step was to evaluate this new version with this value. In order to do that, the first thing we did was to obtain the execution time. This version spent 0.879s on the parallelizable part, not really improving the previous time by any significant amount. This is due to the small number of recursive calls the program actually does. If the problem size was larger, the overheads would be much bigger and this mechanic would be much more effective.

In order to see some differences with the previous version, first we will look at the task profile table inside paraver. The numbers are the bursts.

	No cutoff		Cutoff	
#thread	instantiated	executed	instantiated	executed
THREAD 1.1.1	5924	3022	3244	3259
THREAD 1.1.2	4851	2496	2568	2573
THREAD 1.1.3	5467	2496	2428	2436
THREAD 1.1.4	5615	2868	3096	3101
THREAD 1.1.5	4071	2105	2449	2441
THREAD 1.1.6	4770	2454	2391	2391
THREAD 1.1.7	5118	2622	2499	2477
THREAD 1.1.8	4799	2454	2146	2143

The new version spends a bigger percentatge of time executing than instantiating tasks when compared to the one without cutoff.

As usual, another graph that is useful when evaluating the performance of a parallel program is the strong scalability plot.

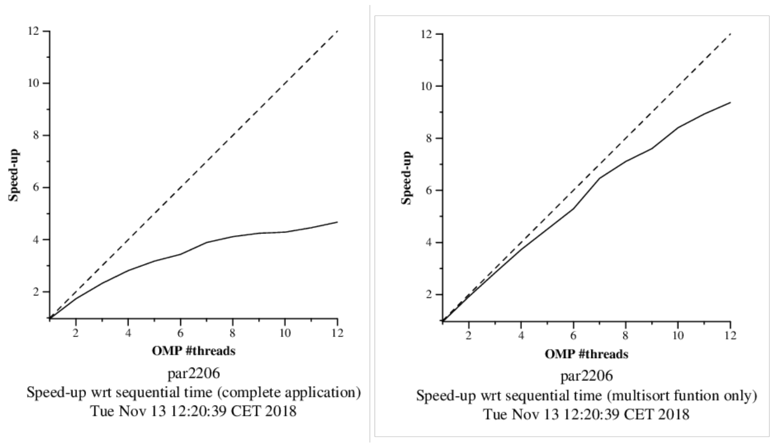


Figure 7: Execution time with different values of cutoff

This figure is very similar to its counterpart in the tree version without cutoff, because as we mentioned before the problem size isn't larger enough to create a big difference.

4.2 Depend

Another problem that can happen when parallelizing programs with dependences is that synchronization can be too strict. The pragmas that we used before wait until all other tasks of its level are finished before passing the barrier and executing new tasks, even if some of this new tasks don't depend on all the tasks that need to finish before passing the *taskwait*.

To remove some of the `#pragma omp taskwait` we can implement the task synchronization mechanism with `depend` OpenMP clauses. This clauses lets the programmer choose what work needs to be done before creating a specific task and what part of the work does it produces for other tasks to consume.

In our problem, we identified specified which quarter of the vector does each multisort recursive call produce and which half does the merge calls use. Inside the merge function there's no dependence between each pair of calls, so we did not add the depend clause.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start
            ,long length,int d) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base casea
                basicmerge(n, left, right, result, start, length);
        } else {
                // Recursive decomposition
                #pragma omp task final(d>=CUTOFF) mergeable
                 merge(n, left, right, result, start, length/2,d+1);
                #pragma omp task final(d>=CUTOFF) mergeable
                merge(n, left, right, result, start + length/2
                      length/2,d+1);
                #pragma omp taskwait
        }
}

void multisort(long n, T data[n], T tmp[n], int d) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                #pragma omp task depend(out:data[0])
                           final(d>=CUTOFF) mergeable
                multisort(n/4L, &data[0], &tmp[0],d+1);
                #pragma omp task depend(out:data[n/4L])
                              final(d>=CUTOFF) mergeable
                multisort(n/4L, &data[n/4L], &tmp[n/4L],d+1);
                #pragma omp task depend(out:data[n/2L])
                              final(d>=CUTOFF) mergeable
                multisort(n/4L, &data[n/2L], &tmp[n/2L],d+1);
                #pragma omp task depend(out:data[3L*n/4L])
                                 final(d>=CUTOFF) mergeable
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L],d+1);

                #pragma omp task depend(in:data[0], data[n/4L])
                    depend(out: tmp[0]) final(d>=CUTOFF) mergeable
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L
                      , d+1);
                #pragma omp task depend(in:data[n/2L],
                   data[3L*n/4L]) depend(out: tmp[n/2L])
                   final(d>=CUTOFF) mergeable
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L]
                      , 0, n/2L, d+1);
                #pragma omp task depend(in:tmp[0], tmp[n/2L])
                                  final(d>=CUTOFF) mergeable
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n
                      ,d+1);
                #pragma omp taskwait
        } else {
                // Base case
                basicsort(n, data);

        }
}
```

To evaluate the improved version with the new definition of dependences, first we obtained the execution time of the parallelizable part and we got 0.87s, again a really small improvement. This is due to the fact that all recursive calls lasts nearly the same amount of time, so they most likely end together. The new tasks that are waiting for specific tasks and dependences to end (with the *depend* clauses) would benefit from this new implementation if the task that they are waiting ended before the other ones. If they end at the same time, the result is the same as waiting all of them (as we did with the *taskwait* clause).

Nonetheless, we also obtained the strong scalability plot and confirmed that the difference with the previous tree versions is minimal.

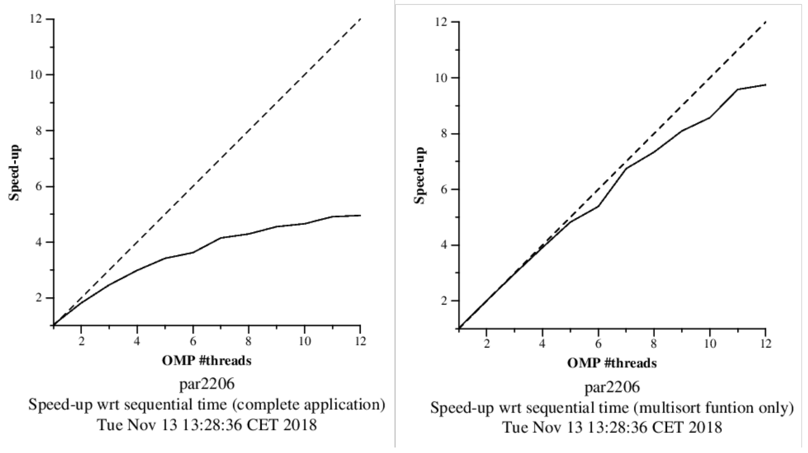


Figure 8: Speedup plots for depend version

5 Conclusions

In this laboratory assignment we consolidated the knowledge of the difference between leaf and tree parallelization strategies, and observed that in this kind of problems and with enough resources, the tree one is usually the best option.

Another important thing that we learned to implement, even if it didn't have a big impact on our problem, is the cut-off mechanism. As we commented, it would have a much more important role if the problem size was bigger and we couldnt afford the overhead of creating too many tasks.

Lastly, we also learned to implement the *depend* clause and most importantly to identify the exact dependences of each task in order to minimize the serialization of the program that implies using synchronization. The improvement wasn't big in our case, but we understood that it can greatly improve the execution time of another programs.