Parallel Divide and Conquer: Analysis and Implementation

Gabriele Marino*

Pisa, Sep 2, 2022

Abstract

The present work explores the improvements on execution times that can be achieved in Divide and Conquer (DAC) paradigm by mean of parallelization (specifically, multithreading). Starting from a general formulation of the paradigm, a theoretical study of the achievable speedup is performed. A parallel implementation of DAC is then tested on mergesort for benchmarking.

1 Introduction

1.1 Sequential DAC

Divide and Conquer (DAC) is a very well known paradigm in computer science, whose underlying idea is to divide a problem into subproblems, and to compute the result of the original problem by aggregating the subresults of such subproblems, which are supposed to be easier to solve. Even if the input data type of a DAC routine is very general, for the sake of simplicity, we restrict ourselves to considering the processing of arrays $\mathbf{a}$ of N elements. This does not imply such a loss of generality: as a matter of fact, most DAC algorithms can be stated in this form (Strassen algorithm, Quicksort, Mergesort, ...). DAC works recursively as follows:

1. **Divide Step.** Call a `divide` subroutine to split the input array $\mathbf{v}$ into d subarrays $\mathbf{a}_i, i = 1, 2, \ldots, d$ (usually, $d = 2$, and the subarrays have the same size);

2. **Solve or Recur Step.** For each subarray $\mathbf{a}_i$, check if it is a base case of the recursion by mean of a `basecase` subroutine. If it is, solve it with a subroutine `solve`, and retrieve its relative subresult R_i. Else, go on splitting it again;

3. **Conquer Step.** Collect each of the subresults R_i and process them to get the result R using a `conquer` subroutine.

The condition for a subarray to be considered a base case of the recursion will most often depend on its size. For the sake of simplicity, we will restrict ourselves

*Master's Degree Student in Artificial Intelligence, g.marino37@studenti.unipi.it

to considering as base case a subarray whose size is less or equal than a threshold B. This is in line with most DAC algorithms (a very usual value is $B = 1$). We note then that DAC defines a recursion tree with height $h^{seq} = \lceil \log_d \frac{N}{B} \rceil$ whose root is the given problem with input of size N and whose leaves are the basecase subproblems with input of size at most B.

1.2 Parallel DAC

The introduction of parallelism in a DAC scheme is straightforward. The most immediate way to do it is to create a new thread for each of the d subarrays produced by the subroutine `divide`, and let the thread recur on solving the subproblem relative to that specific subarray. A minimum of expediency is to realize that one of the subarrays can be managed by the same thread that invoked the `divide` subroutine, reducing the number of threads and thus the overhead.

To control the paralization degree, it may be useful to use a `stop` subroutine to stop the parallel execution and revert it to sequential when the size of the subarrays drops below a certain threshold S. If we assume that the `divide` subroutine splits the input array into equal sized subarrays, it is immediate to see that the number of threads created in such a process is equal to $n_w = d^{h^{par}}$, where $h^{par} = \lceil \log_d \frac{N}{S} \rceil$.

2 Theoretical Speedup Analysis

For the sake of simplicity, we will assume from now on that the `divide` subroutine splits the input array into $d = 2$ equal sized subarrays. We will also assume $N = 2^n$, $S = 2^s$, $B = 2^b$, for some integers n, s, b s.t. $s < n$, and $b \leq s$. Under these hypotheses, the recursion tree will be binary and balanced. We will limit our analysis to the case in which the `solve` subroutine returns as output an array of the same size of its input. This is the case of mergesort and of not so many others DAC algorithms, reason for which this may seem a very restrictive hypothesis. Still, the analysis presented is quite independent of this detail, and can be easily transposed to other cases.

We will refer to the times spent by the `divide`, `solve` and `conquer` subroutines to process an input of size N respectively as $T_d(N)$, $T_s(N)$ and $T_c(N)$, and we will neglect the times spent by the `basecase` and `stop` subroutine. $T_s(N)$ and $T_c(N)$ are equal in the sequential and parallel case, but $T_d(N)$ is not. In fact, when splitting in the parallel case, some overhead will be introduce by the creation of the new thread about to process one half of the input. For this reason we will use the superscripts *seq* and *par* to distinguish between the two cases, and we will expect $T_d^{seq}(N) < T_d^{par}(N)$.

Sequential Time. Let $T(1, N)$ be the time needed to process an input of size N by a DAC algorithm using just one thread. Then, by the expansion of the recursion tree, it is straightforward to see that:

$$T(1,N) = \sum_{i=0}^{h^{seq}-1} 2^i T_d^{seq}\left(\frac{N}{2^i}\right) + 2^{h^{seq}} T_s(B) + \sum_{i=0}^{h^{seq}-1} 2^i T_c\left(\frac{N}{2^i}\right),$$

where $h^{seq} = n - b$ is the height of the tree.

Parallel Time. Let now $T(n_w, N)$ be the time needed to process an input of size N by a DAC algorithm using $n_w = 2^{h^{par}}$ threads, where $h^{par} = n - s$. By the expansion of the recursion tree, and considering that, once generated, two subtrees proceeds in parallel, it follows that:

$$T(n_w, N) = \sum_{i=0}^{h^{par}-1} T_d^{par}\left(\frac{N}{2^i}\right) + T(1, S) + \sum_{i=0}^{h^{par}-1} T_c\left(\frac{N}{2^i}\right),$$

where $T(1, S)$ is the time needed to process sequentially an input of size S. Note that the term n_w does not appear explicitly, but is implicit in h^{par} and $T(1, S)$.

Speedup. Let us assume now that the `divide` and `conquer` subroutines takes linear time, which is almost always the case: $T_d^{seq}(N) = a_d N + b_d^{seq}$, $T_d^{par}(N) = a_d N + b_d^{par}$, $T_c(N) = a_c N + b_c$, for some constants a_d, b_d^{seq}, b_d^{par}, a_c and b_c. We will also use the following abbreviations: $a = a_d + a_c$, $b^{seq} = b_c + b_d^{seq}$, $b^{par} = b_c + b_d^{par}$, $T_s = T_s(B)$, $T^{seq} = T(1, N)$ and $T_{n_w}^{par} = T(n_w, N)$. The sequential time can then be expressed as:

$$T^{seq} = (n - b)aN + (2^{n-b} - 1)b^{seq} + \frac{N}{B}T_s.$$

The parallel time, instead, can be rewritten as:

$$T_{n_w}^{par} = 2\left(1 - \frac{1}{2^{n-s}}\right)aN + (n - s)b^{par} + (s - b)aS + (2^{s-b} - 1)b^{seq} + \frac{S}{B}T_s.$$

By expliciting n_w, we get:

$$T_{n_w}^{par} = 2(1 - \frac{1}{n_w})aN + \log_2 n_w \cdot b^{par} + \frac{T^{seq}}{n_w} - \frac{\log_2 n_w}{n_w}aN - (1 - \frac{1}{n_w})b^{seq},$$

where the first two terms are due to the parallel computations, and the last three to the sequential ones.
The speedup is:

$$\mathcal{S}(n_w) = \frac{T^{seq}}{T_{n_w}^{par}},$$

and it is straightforward to show that:

$$\lim_{N \to \infty} S(n_w) = n_w.$$

As a matter of fact, the term of highest complexity is T^{seq} (which is the only one to be $\mathcal{O}(N \log N)$): the value of the limit immmediately follows by neglecting the lower complexity terms.

The maximum feasible number of threads in the proposed parallelization scheme is $n_w^* = \frac{N}{B}$, which is obtained by imposing $S = B$. For big values of N, as already stated, the dominating term of $T_{n_w}^{par}$ is $\frac{T^{seq}}{n_w}$. For n_w^* threads this term is minimized, and we achieve the minimum parallel execution time:

$$T_{min}^{par} = 2(N - B)a + (n - b)b^{par} + T_s,$$

and the maximum speedup:

$$\mathcal{S}^* = \frac{T^{seq}}{T_{min}^{par}} = \frac{(n - b)a^{seq}N + (2^{n-b} - 1)b^{seq} + \frac{N}{B}T_s}{2(N - B)a + (n - b)b^{par} + T_s}.$$

We note that T_{min}^{par} is exactly the time spent in the non-parallelizable part of the program, i.e. a path from the root to the leaf of the recursion tree. The result we got for $\mathcal{S}^*$, then, perfectly matches Amdahl's law.

Suppose now that we want to find the parallelization degree p needed to achieve a speedup $\mathcal{S}_p$ such that:

$$\mathcal{S}_p \geq (1-f)\mathcal{S}^*,$$

for a given fraction f. We can also express this inequality as:

$$T_p^{par} \leq \frac{T_{min}^{par}}{1-f}.$$

By expanding the terms of the inequality, we get:

$$T_p^{par} \leq 2(1-\frac{1}{p})aN + \log_2 p \cdot b^{par} + \frac{T^{seq}}{p} \leq \frac{T_{min}^{par}}{1-f}.$$

If N is big enough to let us neglect the second addend of the middle term, then it follows that:

$$p \geq \frac{T^{seq} - 2aN}{T_{min}^{par} - (1-f)2aN}(1-f) \implies \mathcal{S}_p \geq \mathcal{S}^*.$$

Suppose now $T_s(B) = \mathcal{O}(B) = \mathcal{O}(N)$. Since $S^* = \mathcal{O}(\log N)$, $T^{seq} = \mathcal{O}(N \log N)$, $T_{min}^{par} = \mathcal{O}(N)$, this result shows that with a $\mathcal{O}(\log N)$ parallelization degree we can achieve $\mathcal{O}(\log N)$ speedup. In other words, by a $\mathcal{O}(\log N)$ parallelization degree we can reduce the execution time from $\mathcal{O}(N \log N)$ to $\mathcal{O}(N)$.

3 Experimental Set Up

Both a parallel and a sequential version of DAC pattern have been implemented in C++17, managing the parallelism using the native C++ threads library `std::thread`. The implemented DAC scheme was tested on sorting integer arrays of variable sizes with mergesort, using a variable number of threads. The base cases of the recursion were solved using the `std::sort` function from the standard C++ algorithms library.

The experiments have been performed using an Ubuntu machine with 32 Intel Xeon Gold 5120 CPUs at 2.20GHz.

3.1 Open Source Code

The whole code is available at this GitHub repository. You can compile the project using the command `make`. This will create some intermediate compilation results in the sub-directory `build/` and four executable files in the sub-directory `bin/`, which you can run using the command `bin/<executable> [<args>]`. The executable files produced are:

- `bin/Experiment1`: program to run experiment 1, as described in 3.4;
- `bin/Experiment2`: program to run experiment 2, as described in 3.4;

- `bin/measure_Tnw`: program to measure the overhead due to threads instantiation, as described in 3.3;
- `bin/run`: program to run a single simulation of mergesort for given values of N, B and n_w. These must be passed as command line arguments, along with a `random_seed` argument for the array random initialization, to guarantee the reproducibility of the result. To run this executable, then, use the command `bin/run <N> <B> <nw> <random_seed>`.

The sub-directory `results/` contains two Python scripts to plot the results of the experiments done. To install the required dependencies you need to run the command `pip install -r requirements.txt` (consider creating and activating a virtual environment for the project before doing this). The mentioned scripts are `exp1-plot.py` and `exp2-plot.py`. These requires respectively the files `exp1-results.txt` and `exp2-results.txt` to be present in the folder `results/outputs/`. These are the outputs produced by running respectively `bin/Experiment1` and `bin/Experiment2` (consider creating them by `bin/<exp_executable> > results/outputs/<exp_results_file>`). The plots produced by the two scripts are created into the folder `results/images/` as `.png` files.

3.2 Theoretical Speedup

To simulate a sensible computational workload, and to prevent the parallel tasks to become too granular, which would cause their execution times to be highly variant and difficult to estimate, we chose to add an active delay of $d = 1$ms when performing `divide`, `conquer` and `solve` subroutines. This delay is big enough to approximate effectively the whole execution times of these routines (at least for not too big values of N, to which we restricted our experiments). We will then compute the theoretical speedup by assuming $T_d^{seq} = T_d^{par} = T_c = T_s = d$.

3.3 Overheads

The parallelization scheme proposed introduces two main sources of overhead.

The first one is unavoidable, and is due to threads instantiation (creation and joining of a thread). In our experiments, this overhead was estimated to amount to tens of microseconds per thread, which is completely negligible with respect to the other quantities involved.

The second one is due to threads synchronization, and is intrinsically linked to the parallelization scheme architecture. Specifically, it is due to the time wasted by a thread after completing its recursion step while waiting for another thread completion before the conquer step. This time is not negligible at all, and is responsible for most of the discrepancy between the theoretical and the experimental speedup. Also, note that such an overhead accumulates easily, and thus increases as the number of threads increases.

3.4 Experiments

Experiment 1. In this first experiment we set $B = 1$, and measure the speedup using $n_w = 16$, for $N = 2^i$, with $i = 4, 1, ..., 13$, averaging over a set of 10 random runs. Figure 1 shows the results we got. We see that for low values of N the

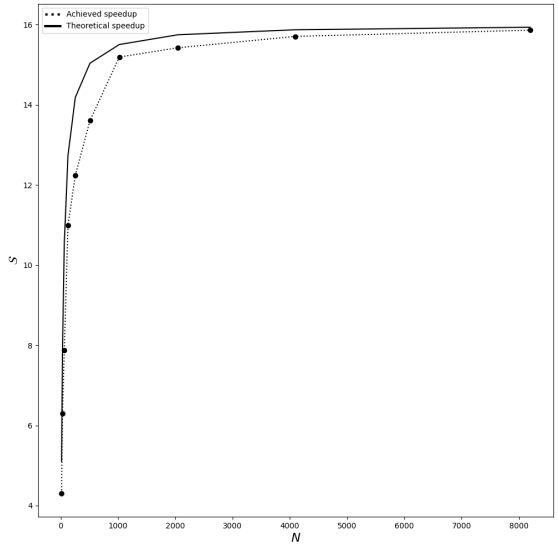


Figure 1: Experiment 1. Achieved and theoretical speedup ($B = 1$)

overhead is high and the achieved speedup is far from the ideal one. However, by increasing it by a few thousands, the achieved speed up sticks to the ideal one, both tending to n_w, as predicted by theory.

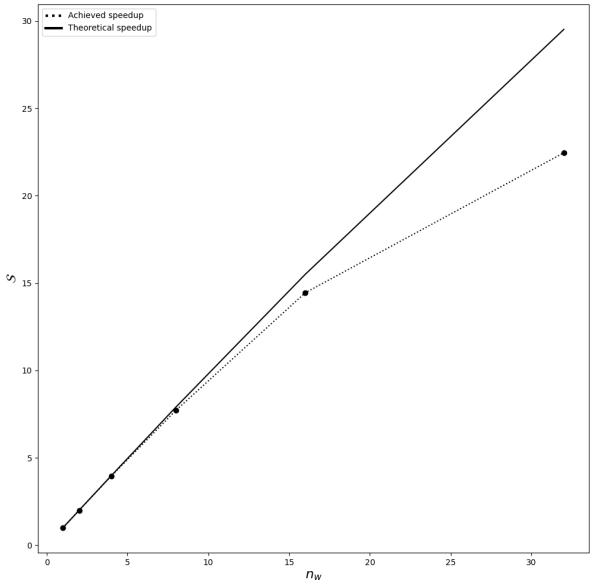
Experiment 2. In this second experiment we set $N = 1024$ and $B = 1$, and measure the mean execution time and the speedup for $n_w = 2^i$, with $i = 0, 1, ..., 5$, averaging over a set of 10 random runs. Figure 2 shows the results we got. We can see that the results achieved match closely the theoretical predictions up to $n_w = 16$ and detach from them for $n_w = 32$, when the overhead begins to weight and the ratio between the ideal and achieved speed up ends up dropping to $\approx 75\%$.

4 Conclusions

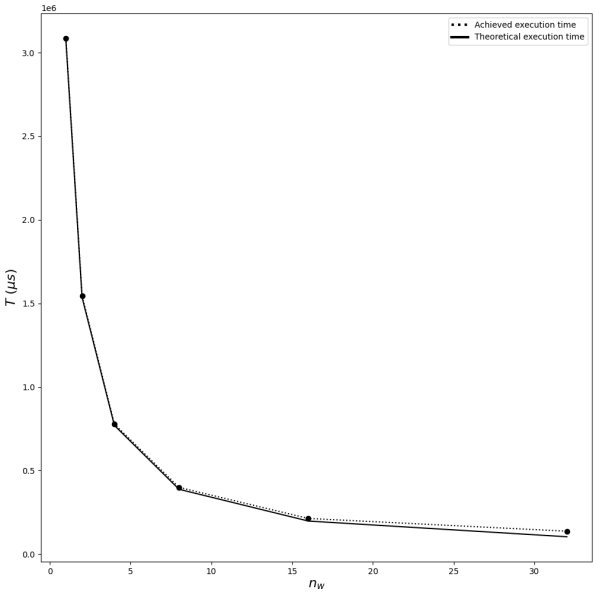
In this work a parallelization scheme for Divide and Conquer paradigm has been introduced and analysed. First, the theoretical speedup of the proposed scheme has been investigated in depth. Then, an open source C++ implementation using native threads has been presented and tested using mergesort.

In the first experiment done, the achieved speedup was analysed as a function of the size of the input, using a fixed number of threads, to show that the relative weight of the overhead decreases as the input size increases, eventually vanishing for large inputs, as predicted by theory.

In the second experiment, instead, the achieved speedup was analysed as a function of the number of threads used. The proposed implementation speedup proved to stick to the theoretical one for a low number of threads, progressively deviating for larger values, until stopping at $\approx 75\%$ of it.



(a) Achieved and theoretical speedup



(b) Mean execution time

Figure 2: Experiment 2. Speedup and execution time ($N = 2^{10}$, $B = 1$)