GAIN ON MULTIPROCESSOR MACHINES

RICARDO J. JESUS

1. Generic problem

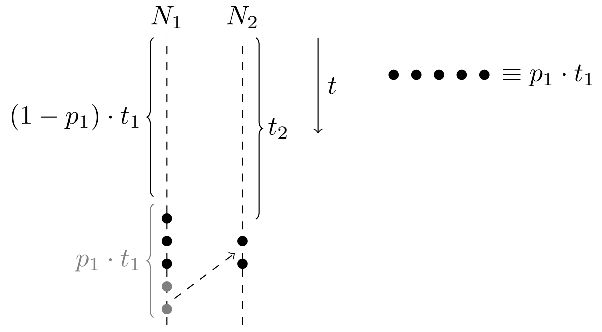
Given a set of P processes, each having a p_i parallelizable portion of their execution, what is the gain of running this set of processes on a N-core machine, assuming each p_i can be split into N' atomic components?

2. Simplified analysis

We will initially focus attention onto a simplified problem. Namely, it will be assumed that (i) there are only two processors and two processes ($N = P = 2$); (ii) only P_1 has a parallelizable portion p_1 of its code; and (iii) N' is assumed to be as large as desired (i.e. p_1 can be sliced into arbitrarily small chunks). One further assumption made is that of the structure of a process. In the discussion which follows processes are assumed to have one non-parallelizable portion of execution followed by a parallelizable one. This is clearly not the only possibility (it is not even the most common one). Yet, we will postpone dealing with this situation until a later time, where we will discuss it under the assumption of negligible context switch.

First, the baseline time for a single core machine should be established. In this simple scenario, and letting t_i represent the amount of time spent by the process P_i, it follows intuitively that $t_t = t_1 + t_2$.

The improved time (the time considering the two processors) needs to be determined. For this, let us consider the possibilities for the execution of the two processes, depicted below.



Before going into details, the general idea is that P_1 and P_2 will run concurrently for a given portion of their lifetime, in which neither can be speeded up by using both processors. The execution times for this phase are $(1 - p_1) \cdot t_1$ and t_2, respectively.

After running for $(1-p_1)\cdot t_1$, P_1 still needs to execute the equivalent to $p_1 \cdot t_1$. This portion can possibly be executed in parallel, but how much of it will actually use both processors depends on how soon P_2 finishes.

If the time t_1^p represents the portion of P_1 which will effectively use both cores, then with two processors it will be reduced to half (each core will handle half of it).

With this in mind we need to determine the time during which both P_1 and P_2 hold their respective processor, without any profit from removing it from them; afterwards t_1^p should be established, so that the actual gain from parallelizing P_1 can be computed.

Concurrent execution of P_1 and P_2 ($t^{\bar{p}}$). The time during which no profit can be harvested from parallelization is equal to

$$t^{\bar{p}} = \max((1-p_1)\cdot t_1,\ t_2)$$

This follows from the fact that while both P_1 and P_2 are running $(1-p_1)\cdot t_1$ and t_2, respectively, each process will hold their processor while running "sequentially". This effectively means that while P_2 is running, even if P_1 could run code in parallel, P_2 will still retain its processor (since otherwise P_2's termination would only be delayed).

This execution phase is illustrated in the previous picture by the black curly braces. Until both are closed, no parallel execution will happen.

Determination of t_1^p. Even though P_1 cannot run in parallel while P_2 runs, it can run parallel code sequentially (using a single processor), in a sense advancing work while waiting to use both processors.

As the figure illustrates, assuming that the parallel portion of P_1 can be sliced at will, a given amount of the resulting execution chunks can be consumed during the time P_1 has to wait until P_2 frees its processor. In the figure, this meant that 1 of the 5 chunks (there represented by a dot) was handled while P_2 was still running.

How much time is then available after P_2's termination, which can be used to run the remainder of P_1? This is clearly upper bounded by $p_1 \cdot t_1$, since P_1 can not run more than p_1 of its code in parallel. As for the lower bound, one has to consider the time window that P_2 leaves P_1 with in order to run whatever is left of its execution. This is $t_1 - t_2$ if $t_1 > t_2$ and 0 otherwise (the case for $t_1 \leq t_2$ is trivial and uninteresting, since clearly the total time of execution will be bounded by t_2).

It then becomes clear that

$$t_1^p = \min(p_1 \cdot t_1,\ t_1 - t_2),\ t_1 > t_2$$

Furthermore, since this is the time during which P_1 can use both processors to run parallelizable code, this time is effectively halved.

Resulting speedup. The attained speedup will be given as a function of t_1 and t_2, with p_1 fixed, and follows from Amdahl's law. Combining the previous results, the following expression is obtained

$$\mathit{speedup}_{p_1}(t_1,\ t_2) = \begin{cases} t_t/t_2 & t_1 \leq t_2 \\ t_t/(t^{\bar{p}} + t_1^p/2) & t_1 > t_2 \end{cases}$$

$$= \begin{cases} 1 + t_1/t_2 & t_1 \le t_2 \\ \dfrac{t_1 + t_2}{\max((1-p_1)\cdot t_1,\ t_2) + \dfrac{\min(p_1 \cdot t_1,\ t_1 - t_2)}{2}} & t_1 > t_2 \end{cases}$$

The branch for $t_1 \le t_2$ stands for the speedup that can be attained when P_2 takes longer than P_1. Meanwhile the second branch is the ratio of sequential execution time over the sum of the timespan during which the processors are occupied executing P_1's non-parallel code and/or P_2 (whichever takes longer) plus the time of executing the remainder of P_1. This later value is effectively reduced to half since both processors are by now free to run P_1's instructions (in parallel).

Example 1. Let us consider a machine with two processors, running two processes (P_1 and P_2), with the first having 40% of its code parallelizable. This portion can be sliced into arbitrarily small atomic chunks (i.e. $0.4 \cdot P_1/N' \to 0$). Also, $t_1 = 120\,\text{ms}$ and $t_2 = 50\,\text{ms}$ when each process is run on a single core machine. What is the speedup of running these same processes on the dual core computer?

Resolution. We shall first get to the result intuitively, then compare it against the evaluation of the expression.

When both processes start, both processors are busy handling each of its designated process.

For $(1 - 0.4) \cdot t_1 = 72\,\text{ms}$ processor 1 is bound to run sequentially. Similarly, processor 2 will be busy for $t_2 = 50\,\text{ms}$.

As can be seen, since P_1's non-parallelizable code portion takes longer to execute than P_2, once P_1 gets to its parallel portion it can start running it immediately on both processors. Thus, the remaining time of P_1 will be reduced by half — $0.4 \cdot t_1/2 = 24\,\text{ms}$ — giving a grand total of $72 + 24 = 96$ (ms), for a speedup of $(120 + 50)/96 \approx 1.77$. Indeed, using the previously obtained expression one would obtain the expected $\mathit{speedup}_{0.4}(120,\ 50) \approx 1.77$.

Example 2. Consider now example 1 but assuming that 80% of P_1 is parallelizable.

Resolution. Under this new situation we have that $(1 - 0.8) \cdot t_1 = 24\,\text{ms}$ processor 1 will be running sequentially.

Recall that processor 2 is busy during the complete execution of P_2 (which lasts 50 ms). During this time, processor 1 will at first be running the non-parallel portion of P_1 (lasting 24 ms). Then, for the $50 - 24 = 26$ (ms) following processor 1 will be advancing part of P_1 which could be run in parallel, but due to lack of resources (processor 2 is still occupied), is instead run sequentially.

By the time P_2 finishes, freeing processor 2, P_1 has completed all of its non-parallelizable code as well as $26/(0.8 \cdot t_1) \approx 27\%$ of its parallelizable execution.

Its remainder ($0.8 \cdot t_1 - 26 = 70\,\text{ms}$) can now run using both processors, effectively taking half this time, leading to a grand total of execution time of $50 + 35 = 85\,\text{ms}$. This represents a speedup of $(120 + 50)/85 = 2$.

In fact, plugging in the expression obtained for speedup computation one obtains $\mathit{speedup}_{0.8}(120, 50) = 2$ as expected.

Configurations for P_1. We will now consider different configurations of P_1, dropping the constraint of it being of the form $\bar{p}p$ (i.e. non-parallelizable followed by parallelizable execution). We will do so by generalizing ideas previously presented in such a manner that it becomes possible to solve the speedup of running two processes P_1 and P_2 in a dual-core system where P_1 can be of any configuration.

First, we will represent a process's execution as

$$\bar{p}_0(p\bar{p})_1(p\bar{p})_2 \dots (p\bar{p})_n$$

Under this representation, $\bar{p}_i$ represents non-parallelizable chunk i and similarly p_i parallelizable chunk i. Meanwhile, $(p\bar{p})_i$ is simply shorthand for $p_i\bar{p}_i$. We will also be considering that p_i and $\bar{p}_i$ represent the fraction of slot i which can be parallelized and non-parallelized, respectively. Where such does not rise ambiguity, these may instead represent the time of executing p_i or $\bar{p}_i$.

Under this representation a process is simply an ordered queue of parallelizable and non-parallelizable execution "tokens"/chunks.

Now, consider two processes, P_1 and P_2, where P_1's configuration is as presented above and P_2 is non-parallelizable. These have execution times t_1 and t_2, respectively. We are now interested in determining how long it takes to execute these two processes optimally in a dual core machine. We will denote this time by t'_t.

First, note that if $t_2 \geq t_1$ then t'_t (the duration of execution using two cores) will simply be equal to t_2, since optimally the two processes will run each in its own processor, and P_2 will finish after (or at the same time as) P_1 does. This has been seen previously.

Meanwhile, if $t_1 > t_2$, it is important to realise that the system only benefits from running P_1 concurrently (i.e. using both processors) at most $t_1 - t_2$ units of time, since otherwise the termination of P_2 is simply being postponed. This is the key idea behind understanding this generalization.

Consider now the sequence (a_i) defined as

$$\begin{aligned}
a_0 &= \max(0,\, t_1 - t_2)\\
a_1 &= a_0 - \frac{\min(a_0,\, p_1)}{2}\\
a_2 &= a_1 - \frac{\min(a_1,\, p_2)}{2}\\
&\vdots\\
a_n &= a_{n-1} - \frac{\min(a_{n-1},\, p_n)}{2}
\end{aligned}$$

It simply represents for $1 \leq i \leq n$ the amount of time that P_2 may be preempted (if it has not terminated already) in order to run P_1's parallelizable slot i (or part of it).

Using this sequence one can compute t'_t with (recall that $t_1 > t_2$)

$$\begin{aligned}
t'_t &= \bar{p}_0 + \sum_{i=1}^{n} \left[(p\bar{p})_i - \frac{\min(a_{i-1},\, p_i)}{2} \right]\\
&= \bar{p}_0 + \sum_{i=1}^{n} \left((p\bar{p})_i + a_n - a_{n-1} \right)
\end{aligned}$$

$$= t_1 + a_n - a_0$$

The resulting speedup is then given by

$$speedup = \frac{t_t}{t'_t}$$

As a closing remark, we shall unfold the expression for the original configuration that had been explored in the beginning of this paper, $\bar{p}p$. In this situation, we will have (recall that $t_1 > t_2$)

$$\begin{aligned} speedup_p(t_1,\, t_2) &= \frac{t_1 + t_2}{\bar{p}_1 \cdot t_1 + p_1 \cdot t_1 + a_1 - a_0} \\ &= \frac{t_1 + t_2}{t_1 + \cancel{a_0} - \dfrac{\min(a_0,\, p_1 \cdot t_1)}{2} - \cancel{a_0}} \\ &= \frac{t_1 + t_2}{t_1 - \dfrac{\min(\max(0, t_1 - t_2),\, p_1 \cdot t_1)}{2}} \\ &= \frac{t_1 + t_2}{t_1 - \dfrac{\min(p_1 \cdot t_1,\, t_1 - t_2)}{2}} \end{aligned}$$

This not only is an equivalent expression to that found previously but it also appears to be simpler.

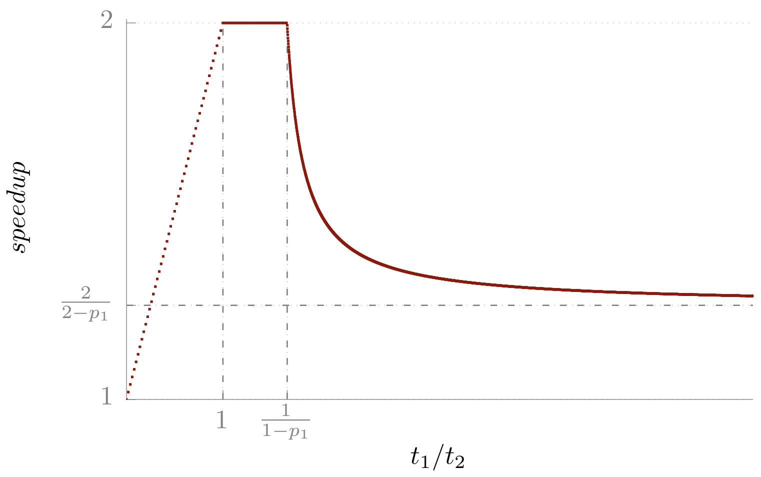
Final remarks. To conclude this discussion, a plot of how the speedup evolves over time is presented[1].

The function plotted can be defined as (for each branch the underlined argument of max and min functions is the one in effect)

$$\begin{aligned} speedup_{p_1}(t_1,\; t_2) &= \begin{cases} t_t/t_2 & t_1 \le t_2 \\ t_t/(t^{\bar{p}} + t_1^p/2) & t_1 > t_2 \end{cases} \\ &= \begin{cases} 1 + t_1/t_2 & \dfrac{t_1}{t_2} \in [0, 1[\\ \dfrac{t_1 + t_2}{\max((1 - p_1) \cdot t_1,\; \underline{t_2}) + \dfrac{\min(p_1 \cdot t_1,\; \underline{t_1 - t_2})}{2}} & \dfrac{t_1}{t_2} \in \left[1, \dfrac{1}{1 - p_1}\right[\\ \dfrac{t_1 + t_2}{\max(\underline{(1 - p_1) \cdot t_1},\; t_2) + \dfrac{\min(\underline{p_1 \cdot t_1},\; t_1 - t_2)}{2}} & \dfrac{t_1}{t_2} \in \left[\dfrac{1}{1 - p_1}, +\infty\right[\end{cases} \end{aligned}$$

[1]The plot's time axis is nonlinear. To better represent its horizontal asymptote, for $t_1/t_2 > 1/(1 - p_1)$ the plot was scaled by a factor of $10 : 1$.

$$= \begin{cases} 1 + \dfrac{t_1}{t_2} & \dfrac{t_1}{t_2} \in [0, 1[\\ 2 & \dfrac{t_1}{t_2} \in \left[1, \dfrac{1}{1-p_1}\right[\\ \dfrac{2}{2-p_1} \cdot \left(1 + \dfrac{t_2}{t_1}\right) & \dfrac{t_1}{t_2} \in \left[\dfrac{1}{1-p_1}, +\infty\right[\end{cases}$$



First, there is a stage during which the speedup grows linearly as t_1/t_2 increases until $t_1 = t_2$, which makes sense since during this time the total execution time will be equal to t_2 (P_1 will fully run while P_2 is running).

Then, a plateau of maximum speedup is reached, lasting until $(1 - p_1) \cdot t_1 = t_2$. One interpretation for this is to notice that by now $t_1 \geq t_2$, so some portion of P_1 will eventually run in parallel. Furthermore, the non-parallel portion of P_1 will be completed at most by the time P_2 terminates. What this implies is that both processors will work like two jars being filled. One is filled with $(1 - p_1) \cdot t_1$, the other with t_2. Then, some portion of $p_1 \cdot t_1$ is used to top up the first till it meets the second. Whatever is left of P_1 is then equally distributed among each processor. The result is that both processors are equally filled and thus, where one processor would take some time executing both processes, the two processors will do so in half the time.

Finally, the speedup obtained starts to observe a downfall once $t_1 > t_2/(1 - p_1)$. The difference is that now, since the non-parallel portion of P_1 takes longer to execute than t_2, processor 2 will have to wait until P_1 reaches its parallel portion in order to do useful work, which introduces a time delay during which the second processor is not contributing to the final result (and this is wasted time). Borrowing from the previous analogy, what this means is that the two processors will not be equally filled, and thus optimal speedup cannot be reached. Using the previously presented speedup expression it becomes clear that $speedup_{p_1} \longrightarrow 2/(2 - p_1)$.

3. Generalizing N'

Building up on the previous section's results, we will now drop the assumption on the infinite granularity of parallel execution, i.e. N' (the number of small chunks into which parallel execution can be sliced) can no longer be taken as large as desired.

One key difference from the previous analysis is that now one cannot think about execution (especially parallel) as something continuous, but should instead focus on time slots, or chunks, a concept already introduced in the picture of a possible execution flow by using dots to represent parallel execution.

We will consider t' (meaning the speedup time) for the different possible configurations of t_1 and t_2. The resulting speedup will then be given by

$$speedup_{p_1}^{N'}(t_1,\ t_2) = \frac{t_1 + t_2}{t'}$$

with t' defined according to the following conditions.

$0 \leq t_1/t_2 < 1$. As in the previous section, with $t_1 \leq t_2$ the total running time will be governed by the largest of the two (t_2 in this case). Thus, $t' = t_2$.

$1 \leq t_1/t_2 < 1/(1 - p_1)$. This is the case in which P_1 starts executing its parallel portion while P_2 is still running.

The idea now will be to determine how many chunks processor 1 can handle while P_2 is running, and then how long it takes to finish processing the remaining chunks using both processors. Denoting $p_1 \cdot t_1/N'$, the time of executing a chunk, by t_c, the amount of chunks answering the first question is given by

$$N_c^{\bar{p}} = \left\lfloor \frac{t_2 - (1 - p_1) \cdot t_1}{t_c} \right\rfloor = \left\lfloor \frac{N' \cdot (t_2 - (1 - p_1) \cdot t_1)}{p_1 \cdot t_1} \right\rfloor$$

Notice that when multiplied by t_c this is analogous to the $t^{\bar{p}}$ of the previous section, specially if the chunks' granularity is enough to perfectly fit the time between $(1 - p_1) \cdot t_1$ and t_2. Either way, the time during which processor 1 will be handling the execution of the non-parallel component of P_1 in addition to $N_c^{\bar{p}}$ chunks is

$$t_c^{\bar{p}} = (1 - p_1) \cdot t_1 + N_c^{\bar{p}} \cdot t_c$$

This value, under the restraints previously defined, will always be less than or equal to t_2.

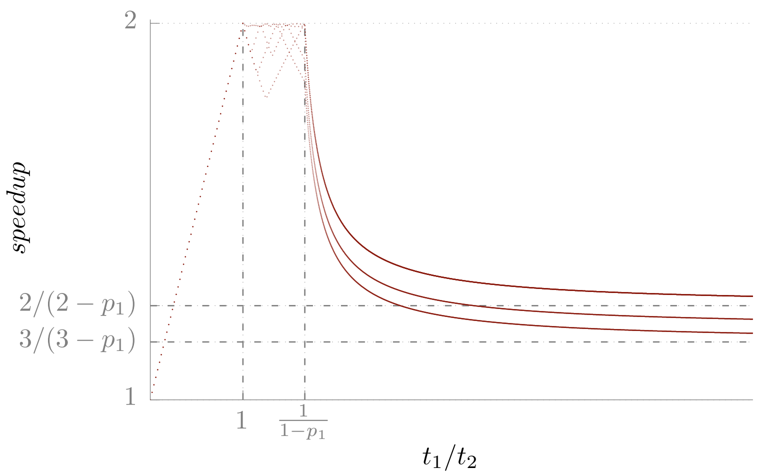
As for the second part of execution, during which processor 2 will also be available, one only needs to distribute whatever chunks are left among the two processors. Since $t_c^{\bar{p}} \leq t_2$, in case the division is not perfect, processor 1 gets the remaining chunk (hence the ceiling/floor dichotomy in the expression). The final expression for t' is then

$$t' = \max\left(t_c^{\bar{p}} + \left\lceil \frac{N' - N_c^{\bar{p}}}{2} \right\rceil \cdot t_c,\ t_2 + \left\lfloor \frac{N' - N_c^{\bar{p}}}{2} \right\rfloor \cdot t_c\right)$$

$t_1/t_2 \geq 1/(1 - p_1)$. In this branch, processor 2 has to wait until P_1 completes its non-parallel portion of execution, thus the parallel chunks will start being distributed immediately. As a result half the chunks will be run on each processor (eventually one processor will run one more in case the number of chunks is not divisible by two). The speedup time is then

$$t' = (1 - p_1) \cdot t_1 + \left\lceil \frac{N'}{2} \right\rceil \cdot t_c$$

Final remarks II. Similarly to what was done in the analogous section above when arbitrary resolution in the parallel execution phase was considered, a plot describing the evolution of the speedup with $N' \in \{2, 3, 5, 50\}$ is provided.



First, one should notice that the plot obtained is in accordance with the previous which assumed N' arbitrarily large. As N' grows the current plot approaches more and more the first.

Moreover, during $t_1 \leq t_2$ no difference exists (again, expected since in this case P_1 does not contribute to the overall time).

The most interesting values appear to be contained within $t_2 < t_1 \leq t_2/(1-p_1)$. In this region the obtained speedup seems to be approaching the expected maximum value of 2 while describing an interesting pattern.

When $t_1 \geq t_2/(1-p_1)$ the speedup approaches to the previously computed limit of $speedup_{p_1} \longrightarrow 2/(2-p_1)$. Despite this fact, with low values of N' (up to 20 for example) there is a visible band of white space separating what appears to be functions with N' divisible by 2 from the remaining. This is probably a cause of the non-perfect division that arises, with the white space diminishing as N' grows.

As a closing remark, it is interesting to see that for every value of N' the respective speedup function is continuous everywhere, despite being defined by distinct branches. Although this does not prove the obtained expression correct, it serves as suggestive evidence.

4. Generic solution

This section will present a possible algorithm for the calculation of the speedup achievable when executing P processes on N processors, each process organized into arbitrary sequences of non-parallelizable execution followed by chunks of arbitrary size which can be parallelizable. It is not clear whether this algorithm will provide the maximum possible speedup.

The following discussion will assume $N \geq P$, but it could be extended with minor modifications.

First, we will model each process as a sequence of arbitrary parallelizable (p_i) and non-parallelizable ($\bar{p}_j$) chunks, e.g. $P_1 \equiv p_1 p_2 \bar{p}_3$. Note that in this notation parallel chunks following each other (i.e. at the beginning, between non-parallel chunks, or at the end of the process) are assumed to be independent execution units, so that the order in which they are executed is irrelevant.

Meanwhile, processors are modelled as queues which can be filled with processes' chunks of execution.

The algorithm is presented in textual form below. It assumes that in the worst case the total time of execution when using all the processors is going to be equal to the time of execution of the process that takes longer, in case it has to run fully sequentially. The idea is then to try to minimize this time, by if possible removing one parallelizable chunk of execution from the slowest process and adding it to the queue of a free processor.

Once no processor is empty or no chunks can be distributed, time is moved forward by an amount equal to the smallest (nonzero) chunk in some processor's queue. This time is subtracted from every processor that does not have an empty queue (possibly removing a chunk from a queue) and added to the total running time.

Algorithm 1: Algorithm for finding the expected speedup of running a set of processes $P_{1,\dots,m}$ in N processors

```
function speedup ($P_1,\ P_2,\ \dots,\ P_m,\ N$)
    $t' \leftarrow 0$
    $t_t \leftarrow \sum_i t(P_i)$
    while some process $P_i$ has chunks remaining do
        while some processor $N_j$ has an empty queue do
            Select $P_k$ taking longest time and having either no chunks queued
              or only parallel chunks queued and the next chunk in its head
              being also a parallel one
            if no $P_k$ could be selected then
                break
            end
            Add the first chunk of $P_k$ (its head) to $N_j$'s queue
            Pop the head of $P_k$
        end
        $min_c \leftarrow$ smallest nonzero chunk in some processors' queue
        Subtract/remove $min_c$ from all non-empty processors' queues
        $t' \leftarrow t' + t(min_c)$
    end
return $t_t/t'$
```

The resulting speedup is then the ratio of full sequential execution (in a single core machine) and the total sum computed.

Though it is not clear whether this process will always output the maximum speedup achievable, being a generalization from the ideas that have been presented in previous sections it will provide the best results for the scenarios that were there considered. Moreover, it does not impose any restrictions to a process' structure, neither does it assume one process to be parallelizable and the other fully sequential, which happened with the discussion carried up to this section.

References

1. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 4th edition.

(Ricardo J. Jesus) Department of Electronics, Telecommunications, and Informatics, University of Aveiro, Aveiro, Portugal

Email address: `ricardojesus@ua.pt`