# MC48 Homework 8

Max Hailperin

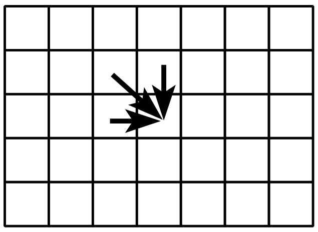
Due: December 8, 1995

Ideally, running a program on $p$ processors would be $p$ times faster than running it on one processor. There are a variety of implementation details that can stand in the way of this, such as communication time between the processors. However, it is also the case that many programs may not show linear speedup even if there is no overhead or communication cost. In this homework, we will look at one example.

Suppose you have two $n$-element sequences, called $s$ and $t$. We will use $s_1, \ldots, s_n$ to denote the elements of the $s$ sequence and $t_1, \ldots, t_n$ to denote the elements of the $t$ sequence. It doesn't really matter what they are sequences of—they could be the words in texts or the DNA units in genes. Our goal is to measure how similar the two sequences are, in the sense of how many insertions, deletions, or replacements it would take to turn one into the other.

We can compute this using a dynamic programming technique by using a two dimensional array of memory locations, which we can call $D$. The size of $D$ is $(n+1) \times (n+1)$, and we will use indices ranging from 0 to $n$ for both the rows and the columns. We will use the notation $D_{i,j}$ to refer to one of the elements of $D$, namely the one in row $i$ and column $j$.

Our algorithm will calculate each entry of $D$, and then will return $D_{n,n}$ as the answer. Except for the entries in row 0 and/or column 0, every entry of $D$ is calculated from three other entries, as shown below:



We will make sure that before we calculate some entry, we have already calculated all three of the entries it is based on.

When the algorithm is carried out on a single processor, it is possible to simply work left to right across the first row, then left to right across second row, etc. It is also possible to work down the first column, then down the second column, etc. Either way, the three entries at the tails of the arrows are calculated before the entry at the head of the arrows.

However, to do the calculation in parallel we will need to be more clever. We will calculate first all the entries $D_{i,j}$ such that $i + j = 0$, then all those such that $i + j = 1$, then all those where $i + j = 2$, then all those where $i + j = 3$, etc. Within each group, the calculations can be done simultaneously, because they all rely only on values calculated in earlier groups. If there are are $k$ entries in one of these groups, then we can calculate them all in $\lceil k/p \rceil$ time steps using $p$ processors, by dividing the computations as nearly evenly as possible among the $p$ processors. (We will call a "time step" the time it takes to calculate one entry of $D$; we will ignore any overhead time spent doing anything other than calculating the entries of $D$.)

1. Draw a four by four grid and label each of the sixteen entries with the $i + j$ sum. Circle each group of entries that shares the same sum.

2. The most processors that will do any good is when $p = n + 1$; clearly the least possible is $p = 1$. How many time steps will the computation take in each of these cases? What is the speedup of $n + 1$ processors *vs.* 1 processor? What is the limit of this as $n$ goes to infinity?

3. For the case $n = 1023 = 2^{10} - 1$, draw a graph of speedup as a function of $p$, plotting a point for $p$ being each power of 2 in the range from $2^0 = 1$ to $2^{10} = 1024$. How does your graph compare with the diagonal line of "perfect" speedup (i.e., a speedup of $p$ on $p$ processors)? How does your graph compare with the limiting speedup you calculated above?

In case you were wondering what is done in each time step to calculate a particular entry of $D$, although it is irrelevant to the problems above, here is the answer:

$$D_{i,j} = \begin{cases} j & \text{if } i = 0 \\ i & \text{if } j = 0 \\ D_{i-1,j-1} & \text{if } i \neq 0,\ j \neq 0, \text{ and } s_i = t_j \\ 1 + min(D_{i-1,j}, D_{i,j-1}, D_{i-1,j-1}) & \text{otherwise} \end{cases}$$