Project in Parallel Computing - Tom's puzzle

Christopher Vincenzo Febo, Irene Natale

1 Abstract

Tom's favorite activity is putting together *NxN*-piece puzzles (uniquely shaped cardboard pieces that are die-cut from a single complete scene). Using shape and colour as cues, Tom searches through the mixed up pieces trying to find adjacent ones, starting from the easy to find corner elements, from then onwards every time he picks up a piece and tries to fit it with the one he has just placed. often Tom misjudges, and the piece does not fit properly: he then has to put the piece back on the board and try again.
The aim of our project is to design a parallel algorithm to complete Tom's puzzle. The input data of our code will be a list of the individual pieces and their characterization. Each piece of the puzzle has 4 sides, so we will have the information about shape and colour for every side of a piece. The output of our algorithm will be the ordered puzzle, where every piece in the right position.
We have approached the problem from a combinatory point of view, considering shape and colour of adjacent sides as the key feature for two pieces to line up.
To figure out the best way to approach our final generalized problem we took three major steps before reaching a satisfactory algorithm. In the first part we will develop a sequential algorithm for a general *NxN* puzzle. Subsequently for a first attempt at parallelization we consider the problem with a *NxN*-piece puzzle divided into 4 subdomains solved by 4 processess. For this step we will study the efficiency and the speedup of the algorithm, comparing it with the sequential one. A more general version of the algorithm will then be described, still with an *NxN*-pieces puzzle and an even number *P* of processors. We will make the same comparisons also for this more general algorithm.

Christopher Vincenzo Febo
KTH e-mail: cvfebo@kth.se

Irene Natale
KTH e-mail: inatale@kth.se

2 Description of the Algorithm

2.1 Preliminary Assumptions

Before describing the sequential algorithm we want to list some important preliminary assumptions that we supposed before facing the problem.

1. The puzzle is supposed to be square, with *NxN* number of pieces.
2. Every piece of the puzzle has 4 sides. We have 2 different codes for each side: colour and shape. In general we have 3 different types of code for the shape: 0 for the flat side on the borders, 1 for ”female” shape and 2 for ”male” shape. The internal assignment of the shape code is done randomly. We also suppose that when two different pieces match, they have the same and unique colour code. The colour on the border sides is assigned to be always zero.
3. All the pieces of the puzzle are randomly positioned, but not randomly rotated. This means that when we assign every piece to its right position, the piece will be subject to shifts and not rotations.
4. We suppose that the number of processes and the number of puzzle elements are both divisible by 4 and that the following relationship holds

$$\frac{N^2}{4} \% \frac{P}{4} = 0$$

   where % represents the remainder operation. In this way we are sure that in every quarter of the puzzle the number of processors is the same, and that every processor has the same number of elements in its subdomain.

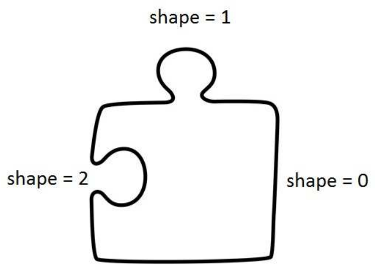


Fig. 1 Shape codes for puzzle pieces.

2.2 *Sequential Algorithm*

When the puzzle is randomly disordered on the table, the only pieces that can be identified without any further information are the 4 corners. This is due to the fact that they are the only pieces with two 0 shape codes. So, the first thing that our sequential algorithm does is to look for them.
When this is done we start to fill in all the frame of the puzzle. These kind of pieces are easy to find because we have to check if they have a 0-code for one sides' shape and the same shape and colour code of its neighbour. In this case we use the corner pieces as "roots" to start the research.
The third step is to find the right position of all the internal pieces, this is done by filling the frame in a linear way, checking if the north and the west side (both colour and shape) match with the neighbours. Therefore the internal part of the puzzle will be filled in like lines of a book.

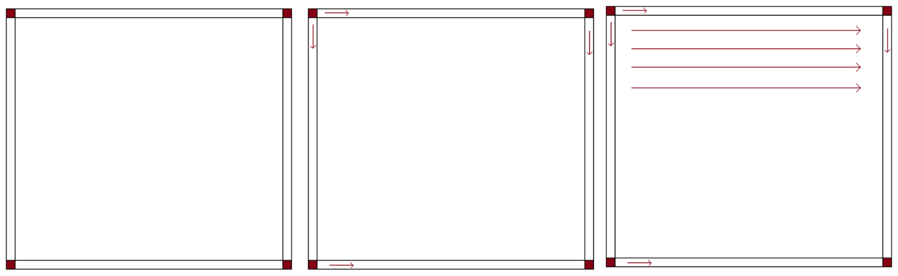


Fig. 2 Sequential Algorithm

2.3 *Parallel Algorithm*

In this section we describe two versions of a parallel algorithm, in both cases we start from the assumption of splitting the domain of the puzzle into 4 parts, and solving them independently.

2.3.1 4 processors

Firstly we consider a simple version in which each of the 4 subdomains is assigned to a single processor. Each process searches in the whole domain but only switches elements in its area of competence. The starting point as always a the corner element, from which we can then start a sequential search for elements by looking for matching sides (same colour and opposite shape). By doing this each of the 4 processes fills in its subdomain. The last step of this algorithm consists of an all-to-one communication. We use process 0 as a hub to which the other 3 processes send their ordered data. Process 0 then saves all the pieces from the other processes and prints the ordered output.

2.3.2 P processors

Taking into consideration assumption 4 from section 2.1 we have a near identical approach to the problem as in the simpler (4 process) version. This time round each of the 4 subdomains is split vertically into $P/4$ sub-subdomains. The search for the 4 corner elements is done by the outer process of each subdomain. Once the starting piece is in place these four ”corner” processes start filling in their first row, once they get to the end of this row a message is passed to the next process in the subdomain. The message received contains the information needed to find the corner element of the inner process and subsequently fill in the area of competence. This gets repeated $P/4$ times in a cascade effect until the whole subdomain is ordered. At this point we have an all-to-one communication thanks to which each process sends its ordered elements to process 0, ready for print-out. All this can be seen in figures 3 and 4.

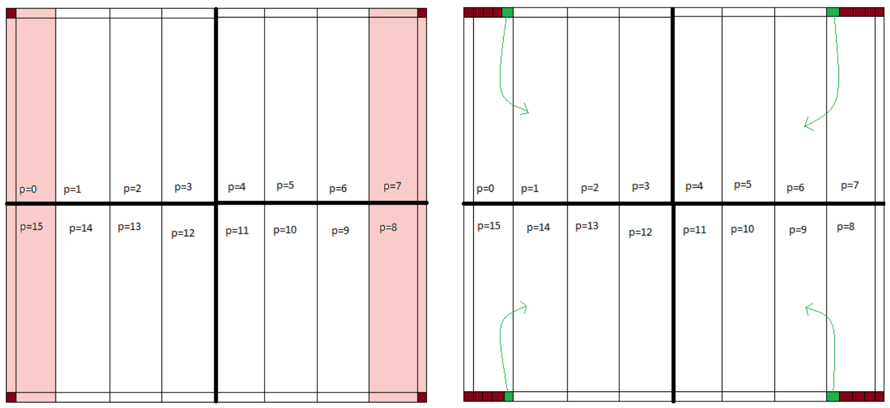


Fig. 3 Parallel Algorithm with P=16

3 Theoretical Performance Estimation

In this section we will try to predict the behaviour of the speedup for our parallel algorithm. We know the followings for both the parallel and the sequential algorithm

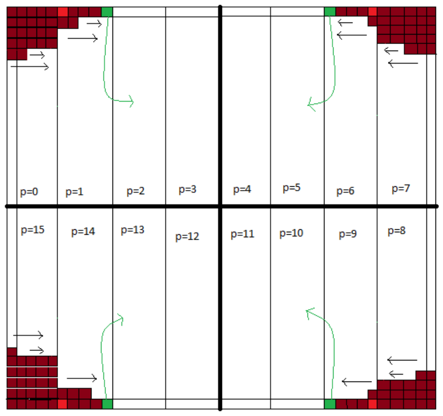


Fig. 4 Parallel Algorithm with P=16

- when we look for the corner elements, we have to check if two sides of the same piece have shape 0, so we need to compute 2 commands
- when we look for the corner elements we always check if the element is already ordered or not, and we also check two sides: one has to have shape 0 and the other has to have compatible shape and colour with its neighbour. In conclusion we need 4 comparisons
- when we look for a specific internal piece of the puzzle, we always check both shape and colour for 2 sides and we also check if the piece is already ordered or not. This conditions yield to 5 comparisons.

We created the following unit of measure:

- t_c = time to do one comparison between two integers
- t_s = time to swap all the values of two different pieces of the puzzle
- t_{sup} = time to start up an MPI communication
- t_d = time to send one "word" as a MPI message

3.1 Sequential Algorithm

In the sequential algorithm we first look for the 4 corners, then we complete the 4 borders of the puzzle, and then we order all the internal pieces. If T^* indicates the serial algorithm runtime, we can write

$$T^* = T_{corners} + T_{borders} + T_{internal pieces}$$

where

$$T_{corners} = (2t_c N^2 + t_s)4$$

$$T_{borders} = (4t_cN^2 + t_s)(N-2)4$$

$$T_{internalpieces} = (5t_cQ + t_s)(N-2)^2$$

When we look for the correct internal pieces, the research does not include the previous rows, so it seems fair to consider an average between the number of elements that has to be checked to find elements in the first row and the number of elements that need to be checked to find elements in the last one.

$$Q = \frac{(N-2)^2 + (N-2)}{2} = \frac{N^2 - 3N + 2}{2}$$

In conclusion

$$T^* = (2t_cN^2 + t_s)4 + (4t_cN^2 + t_s)(N-2)4 + (5t_cQ + t_s)(N-2)^2$$

3.2 4 processors, *NxN* puzzle

When we use the 4-processors algorithm we have each processor looking for a different corner, then each processor order all the internal pieces of its subdomain using the corner that he found as "root". At the end of these operations, all the processors send their ordered part of the puzzle to th first one, which prints the result after having copied in his own puzzle the right elements. So, if T_4 indicates the 4-processors algorithm runtime, we can write

$$T_4 = T_{1corner} + T_{subdomain} + T_{communications}$$

where

$$T_{1corner} = (2t_cN^2 + t_s)$$

$$T_{subdomain} = T_{1st-row} + T_{1st-column} + T_{internal} = T_{borders} + T_{internal} =$$

$$= (4t_cN^2 + t_s)(\frac{N}{2} - 1)2 + (5t_cN^2 + t_s)(\frac{N}{2} - 1)^2$$

$$T_{communications} = (t_{sup} + t_d + t_s)(\frac{N^2}{2^2})3$$

To be able to do a SpeedUp analysis in this case, we take the liberty to consider the communication runtime in the parallel algorithm negligible. Therefore we obtain

$$S_4 = \frac{T^*}{T_4} \approx$$

$$\approx \frac{(2t_cN^2 + t_s)4 + (4t_cN^2 + t_s)(N-2)4 + (5t_cQ + t_s)(N-2)^2}{(2t_cN^2 + t_s) + (4t_cN^2 + t_s)(\frac{N}{2} - 1)2 + (5t_cN^2 + t_s)(\frac{N}{2} - 1)^2} \approx$$

If we neglect the first and the second term of both numerator and denominator we obtain

$$\approx \frac{(5t_cQ+t_s)(N-2)^2}{(5t_cN^2+t_s)(\frac{N}{2}-1)^2} \approx \frac{(5t_c\frac{N^2}{2}+t_s)N^2}{(5t_cN^2+t_s)N^2}4 \approx \frac{(5t_c\frac{N^2}{2}+t_s)}{(5t_cN^2+t_s)}4 \approx$$

We notice that the 4 that multiplies the fraction comes from the number of processors that we are using in this case. The only difference is the number of times that we need to do the 5 comparisons, that is $\frac{N^2}{2}$ for the sequential algorithm and N^2 for the parallel one. In conclusion the fraction is always something that takes values less then 1. If we also neglect the swap runtime we obtain

$$\approx \frac{(5t_c\frac{N^2}{2})}{(5t_cN^2)}4 = 2$$

Now we can continue with the more general algorithm analysis, and see if the behaviour is similar.

3.2.1 P processors, *NxN* puzzle

The more general algorithm for an arbitrary number P of processors has only one difference from the previous one: each processor, to be able to start ordering his own subdomain, has to wait until the processor next to it finishes to order its line on the border and sends him the last neighbour element. For example, if we use 16 processors, the most internal processes are $p = 3$, $p = 4$, $p = 11$ and $p = 12$ (we remains the reader that the subdomains of the puzzle are assigned to the various processors in a clockwise order). In fact for this case we can write the runtime of the algorithm as

$$T_P = T_{1corner} + T_{ordering-most-internal-processor} + T_{communications}$$

where

$$T_{1corner} = (2t_cN^2+t_s)$$

$$T_{ordering-most-internal-processor} = T_{borders-previous-processors} + T_{singular-communic} + T_{ord-subdomain}$$

$$T_{borders-previous-processors} = (4t_cN^2+t_s)(\frac{2N}{P})(\frac{P}{4}-1)$$

$$T_{singular-communications} = (t_{sup}+t_c)(\frac{P}{4}-1)$$

$$T_{ordering-subdomain} = T_{1strow} + T_{internal-elements} = (4t_c+t_s)(\frac{2N}{P}) + (5t_c+t_s)(\frac{2N}{P})(\frac{N}{2}-1)$$

$$T_{communications} = (t_{sup}+t_c+t_s)(\frac{N^2}{P})(P-1)$$

Now it is possible to look at the speed up also in this case. To do so we neglect all the communications runtime. We obtain

$$S_P = \frac{T^*}{T_P} \approx \frac{(2t_cN^2+t_s)4+(4t_cN^2+t_s)(N-2)4+(5t_cQ+t_s)(N-2)^2}{(2t_cN^2+t_s)+(4t_cN^2+t_s)(\frac{2N}{P})(\frac{P}{4})+(5t_cN^2+t_s)(\frac{N}{2}-1)(\frac{2N}{P})}$$

As we did in the previous case, we neglet the first and second term in both the numerator and the denomitor, obtaining

$$\approx \frac{(5t_cQ+t_s)(N-2)^2}{(5t_cN^2+t_s)(\frac{N}{2}-1)(\frac{2N}{P})} = \frac{(5t_cQ+t_s)(N^2+4-4N)}{(5t_cN^2+t_s)(\frac{N^2-2N}{P})} \approx \frac{(5t_cQ+t_s)N^2}{(5t_cN^2+t_s)(\frac{N^2}{P})} =$$

$$= \frac{(5t_cQ+t_s)}{(5t_cN^2+t_s)}P \approx \frac{(5t_c\frac{N^2}{2}+t_s)}{(5t_cN^2+t_s)}P$$

that is the same exact result as $P = 4$.
When we increase the number of processes, the speedup increases as a straight line, that has inclination $\frac{(5t_c\frac{N^2}{2}+t_s)}{(5t_cN^2+t_s)}$. If we neglect t_s, then the speedup becomes

$$\approx \frac{5t_c\frac{N^2}{2}}{5t_cN^2}P = \frac{P}{2}$$

In conclusion, after several approximations, we obtain as result that when P increases, the speedup increases as a straight line that has inclination close to $1/2$. We will successively see in section "*Experimental Speedup*" if our result holds in reality.
The last thing that we want to underline is that theoretically, the best parallel speedup of a parallel algorithm is $S_P = P$. In our case this is impossible to obtain since the only way to have this result is when

$$\frac{N^2}{2} = N^2 \leftrightarrow N = 0$$

4 Implementation Details

Here we analyze some parts of the coding which have been essential and not trivial in the writing of our parallel program.

4.1 Puzzle creation and randomization

Input data is very important, so creating a randomized version of our puzzle model is essential and needs to be produced by another program altogether. This consists of creating a 3D-array, *NxNx*8, where the 8 parameters are shape and colour of each side of an element. The positions of the elements, before being printed to a .txt file are then scrambled using a randomizing function which gives a number from 0 to *NXN* as so:

```
for i = 0 : N
  for j = 0 : N
     p = random_position();
     switch element (i,j) with element (p/N, p mod N)
```

4.2 Datatype

To be able to allocate the puzzle elements in dynamic memory we implement a struct containing the 8 parameters which are then copied into each element of a 2D-array of structs from the .txt file. We add a ninth parameter to indicate if the element is ordered yet.

```
struct {
int n_s; north shape
int n_c; north colour
int s_s; south shape
int s_c; south colour
int w_s; west shape
int w_c; west colour
int e_s; east shape
int e_c; east colour
int ord; ordered?
} element;
```

So as to use this structure in MPI message passing it is necessary to create a new `MPI_Datatype` which we call `mpi_element`.

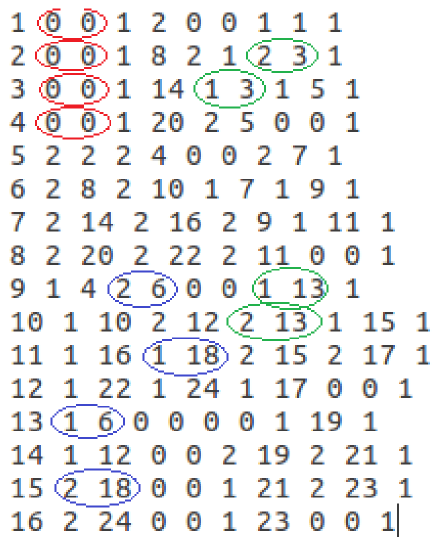
4.3 Loops

A hidden but not trivial aspect of the code is represented by the `for` loops used to fill in each subdomain with the right elements. The difficulty occurs because of the fact that matrix numbering goes from left to right, and top to bottom, whilst are

processes are numbered in a clockwise order, so apart from the top left subdomain, the numbering differs in many ways. For ease of explanation we take the case of the bottom right subdomain. In a general case elements are cycled as:
`k = 0 : N/2` over rows
`h = p*(N/(P/2)) : (p+1)*(N/(P/2))` over columns, where `p` is the number of the process and `(N/(P/2))` is the width of each sub-subdomain
In this case however we have to go, from the bottom right corner, left and upwards, so our new indices `n` and `m` become:
`n = N-k-1`
`m = (2*N-h-1).`

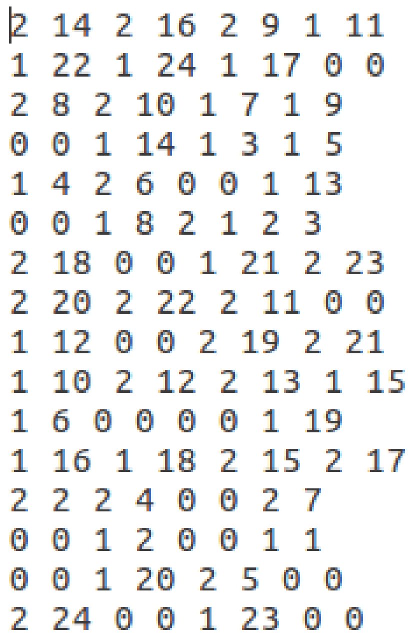
5 Results of a typical problem run

A typical problem run gives as output the elements numbered from 1 to *NxN*, each followed by its 9 parameters, below we show the simple output for a 4*x*4 problem. In the figure of the ordered output we highlight some of the matching parameters on sides of different pieces. To bare in mind is the order of the parameters: north, south, west, east, ord; the fact that shape 1 matches with shape 2, colours are equal, and borders are 0 everywhere



```
1 0 0 1 2 0 0 1 1 1
2 0 0 1 8 2 1 2 3 1
3 0 0 1 14 1 3 1 5 1
4 0 0 1 20 2 5 0 0 1
5 2 2 2 4 0 0 2 7 1
6 2 8 2 10 1 7 1 9 1
7 2 14 2 16 2 9 1 11 1
8 2 20 2 22 2 11 0 0 1
9 1 4 2 6 0 0 1 13 1
10 1 10 2 12 2 13 1 15 1
11 1 16 1 18 2 15 2 17 1
12 1 22 1 24 1 17 0 0 1
13 1 6 0 0 0 0 1 19 1
14 1 12 0 0 2 19 2 21 1
15 2 18 0 0 1 21 2 23 1
16 2 24 0 0 1 23 0 0 1
```

Fig. 5 ordered elements (output), border details in red, east-west in green, north-south in blue



```
2 14 2 16 2 9 1 11
1 22 1 24 1 17 0 0
2 8 2 10 1 7 1 9
0 0 1 14 1 3 1 5
1 4 2 6 0 0 1 13
0 0 1 8 2 1 2 3
2 18 0 0 1 21 2 23
2 20 2 22 2 11 0 0
1 12 0 0 2 19 2 21
1 10 2 12 2 13 1 15
1 6 0 0 0 0 1 19
1 16 1 18 2 15 2 17
2 2 2 4 0 0 2 7
0 0 1 2 0 0 1 1
0 0 1 20 2 5 0 0
2 24 0 0 1 23 0 0
```

Fig. 6 disordered elements (input)

6 Experimental Speedup Investigation with meaningful data

We measure the runtime using sequential and parallel algorithm for solving a $200x200$ piece puzzle, so with $4x10^4$ elements involved. To analyze the runtime in a more complete and meaningful fashion we compute only the time required to actually solve the puzzle, leaving the print-out time out of the measurement. The sequential algorithm takes on average 9.61 seconds to reach a final ordered result. The parallel runtime related to the algorithm is equal to the time it takes for the innermost processes to complete their own domain, since we have a cascade effect going inwards as the algorithm is designed. In Table 1 we can see the experimental data measured with $P = 4, 8, 16, 20$ processors.

Table 1 Experimental runtime

Sequential	**9.61s**
P=4	**4.34s**
P=8	**2.37s**
P=16	**1.32s**
P=20	**1.07s**

If we go back to our predicted speedup for the parallel algorithm, we predicted that it should be around

$$S_P = \frac{T^*}{T_P} = \frac{P}{2}$$

From this equation, and considering $T^* = T_1 = 9.61s$, we had a theoretical prediction of the speedup computed as

$$T_P = \frac{2T^*}{P}$$

and the values can be seen in Table 2. The results in the two tables are very similar, giving credibility to our theoretical analysis and to the efficiency of the algorithm.

Table 2 Theoretical speedup from sequential runtime

Sequential	**9.61s**
P=4	**4.805s**
P=8	**2.4025s**
P=16	**1.20125s**
P=20	**0.961s**

In the figure below (Figure 7) we plot the experimental runtime as a function of the number of processors used.

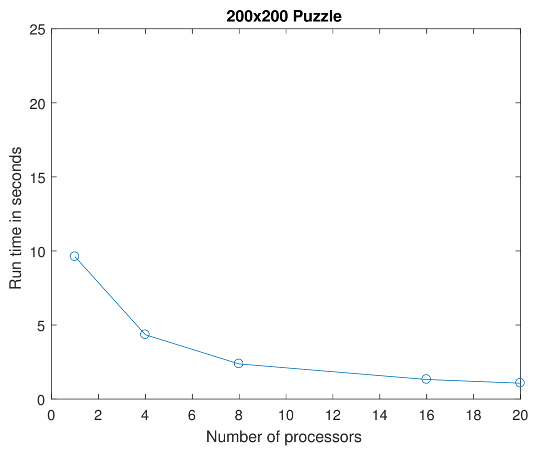


Fig. 7 Run time of the sequential (P=1) and parallel algorithm.

7 References

[1] B. Wilkinson and M. Allen. Parallel Programming Techniques Applications Using Networked Workstations Parallel Computers, 2nd ed. Toronto, Canada: Pearson, 2004.