

Term Project

Parallel Fractal Image Generation

**A Study of Generating Sequential Data
With Parallel Algorithms**

Matthias Book

Table of Contents

| | |
|-------------------------------------|----|
| Table of Contents | 2 |
| Abstract | 2 |
| Introduction | 3 |
| The Mandelbrot Set | 3 |
| Fractal Image Characteristics | 5 |
| Parallel Algorithm Design | 7 |
| Partitioning | 7 |
| Agglomeration | 7 |
| Output Synchronization | 9 |
| Token-Passing Synchronization | 9 |
| Polling Synchronization | 12 |
| Performance Analysis | 14 |
| Conclusion | 16 |
| Bibliography | 17 |

Abstract

This paper discusses algorithms to implement an in-order output of sequential data generated by a parallel program, using the example of fractal image generation. After an introduction to the mathematics of fractals a parallel algorithm that spreads the workload evenly among processors and produces a sequential output is developed. In order to achieve the latter, several synchronization techniques (blocking and non-blocking token passing, non-blocking polling) are examined for their suitability, and polling synchronization is identified as the only certain way to realize an in-order output. A subsequent performance analysis shows that the parallel algorithm achieves an overproportional speedup over serial algorithms, since the non-blocking communication introduces a second layer of parallelization.

Introduction

Fractals are objects with dimensions that cannot be expressed as whole numbers – for example, a curve with the fractal dimension 1.3 is something between a line (euclidean dimension 1) and a plane (euclidean dimension 2). Fractals have three typical characteristics: They are infinitely complex, they show signs of self-similarity on all scales of magnification, and they contain areas of order and chaos.

Although the first fractal objects were discovered by mathematicians around 1870 and many more were conceived in the decades afterwards, they were considered intriguing but exceptional cases – mathematical irregularities that had little to do with the real world. It took about a hundred years before Benoit Mandelbrot found the common bond between all those seemingly unrelated cases and integrated them into the new field of fractal geometry in 1977. Mandelbrot also realized that fractals actually describe the real world much better than traditional mathematical models do – tree trunks are not perfect cylinders, the Earth is not a perfect sphere, the orbits of the planets are not perfect ellipsoids, etc. And as the weather forecast teaches us, nature also is not predictable on large timescales but shows chaotic behavior. "Reality is fractal", Mandelbrot discovered.

The Mandelbrot Set

Fractals are generated by calculating certain formulas over and over again, feeding the results of one step back into the next step. For example, the Mandelbrot set, the most well-known fractal object first described by Benoit Mandelbrot, is created by iterating a simple formula over complex numbers.

A complex number C consists of two components – a "real" part $\text{Re } C$, and an "imaginary" part $\text{Im } C$. The imaginary part is multiplied by the constant i , defined as the square root of -1 :

$$C = \text{Re } C + \text{Im } C \cdot i \quad \text{where } i := \sqrt{-1}$$

Although complex numbers look like an abstract mathematical contraption that has no connection with the real world, they are as realistic as the real number space that we are used to. In fact, complex numbers are routinely used by electrical engineers to design the devices we use every day, so they are indeed a valid means of describing the world. To make complex numbers a bit easier to grasp, we can visualize them as points with the coordinates (x, yi) on a plane, as shown in Figure 1:

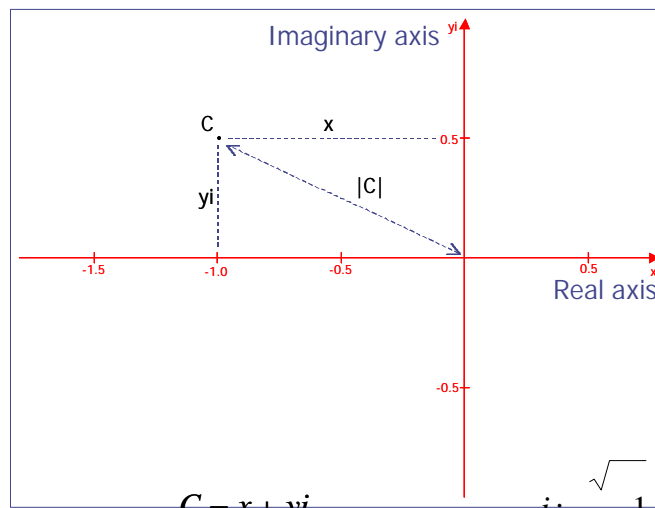


Figure 1: The complex plane

The magnitude of a complex number, $|C|$, is defined as that point's distance from the origin of the complex plane and calculated as

$$|C| = \sqrt{(\operatorname{Re} C)^2 + (\operatorname{Im} C)^2} = \sqrt{x^2 + y^2}$$

Now, to start creating a fractal image, we choose an arbitrary point C from the complex plane and plug it into the following simple iteration formula:

$$Z_0 = 0$$

$$Z_{n+1} = Z_n^2 + C$$

This is the formula Benoit Mandelbrot first described. It takes the current point Z_n (for the first iteration, this is Z_0 , the origin), squares it, adds the constant C we chose before, and so arrives at a new point Z_{n+1} . This point is again squared, C is added, and so on. We perform this iteration several times and see how the values of Z develop.

For example, if we choose $C = -1.0 + 0.5i$, we get the series of values of Z shown in Table 1:

| n | $\operatorname{Re} Z_n$ | $\operatorname{Im} Z_n$ | $ Z_n $ |
|-----|-----------------------------|------------------------------|------------------------------|
| 0 | 0.00000000000000 | 0.00000000000000 | 0.00000000000000 |
| 1 | -1.00000000000000 | 0.50000000000000 | 1.11803398874990 |
| 2 | -0.25000000000000 | -0.50000000000000 | 0.55901699437495 |
| 3 | -1.18750000000000 | 0.75000000000000 | 1.40451281589030 |
| 4 | -0.15234375000000 | -1.28125000000000 | 1.29027523446130 |
| 5 | -2.6183929443359 | 0.8903808593750 | 2.76563910980620 |
| 6 | 5.0632035362069 | -4.1627339199185 | 6.55472224713590 |
| 7 | 7.3076763610173 | -41.6535382072400 | 42.28970771924700 |
| 8 | -1682.6151113846000 | -608.2811530195500 | 1789.18964175920000 |
| 9 | 2461186.6519410000000 | 2047006.6200823000000 | 3201199.12507070000000 |
| 10 | 1867203633030.9000000000000 | 10076130739563.0000000000000 | 1024767583835.30000000000000 |
| ... | ... | ... | ... |

Table 1: Iteration approaching infinity with $C = -1.0 + 0.5i$

We observe that Z moves out towards infinity within very few iterations. Somehow, this is what we might expect since we are constantly squaring and adding numbers.

However, if we try a different choice of C , for example $C = 0.25 - 0.25i$, we get the results in Table 2:

| n | $\operatorname{Re} Z_n$ | $\operatorname{Im} Z_n$ | $ Z_n $ |
|-----|-------------------------|-------------------------|------------------|
| 0 | 0.00000000000000 | 0.00000000000000 | 0.00000000000000 |
| 1 | 0.25000000000000 | -0.25000000000000 | 0.35355339059327 |
| 2 | 0.25000000000000 | -0.37500000000000 | 0.45069390943300 |
| 3 | 0.17187500000000 | -0.43750000000000 | 0.47005027989035 |
| 4 | 0.088134765625000 | -0.40039062500000 | 0.40997608405816 |
| 5 | 0.097455084323883 | -0.32057666778564 | 0.33506252161220 |
| 6 | 0.156728093532030 | -0.31248365238264 | 0.34958508021450 |
| 7 | 0.176917662295790 | -0.34794993419571 | 0.39034473986338 |
| 8 | 0.160230702525410 | -0.37311697790776 | 0.40606669062459 |
| 9 | 0.136457598828770 | -0.36956959098863 | 0.39395730588684 |
| 10 | 0.132038993694610 | -0.35086115797288 | 0.37488377936362 |
| ... | ... | ... | ... |

Table 2: Beginning iteration with $C = 0.25 - 0.25i$

Surprisingly, Z stays within close range of the origin. If we iterate some more, we find that Z actually converges to a certain point (Table 3):

| n | $\text{Re } Z_n$ | $\text{Im } Z_n$ | $ Z_n $ |
|-----|------------------|-------------------|------------------|
| ... | ... | ... | ... |
| 114 | 0.14644660940674 | -0.35355339059328 | 0.38268343236510 |
| 115 | 0.14644660940673 | -0.35355339059328 | 0.38268343236510 |
| 116 | 0.14644660940672 | -0.35355339059328 | 0.38268343236509 |
| 117 | 0.14644660940672 | -0.35355339059327 | 0.38268343236509 |
| 118 | 0.14644660940673 | -0.35355339059327 | 0.38268343236509 |
| 119 | 0.14644660940673 | -0.35355339059327 | 0.38268343236509 |
| 120 | 0.14644660940673 | -0.35355339059328 | 0.38268343236509 |
| 121 | 0.14644660940673 | -0.35355339059328 | 0.38268343236509 |
| 122 | 0.14644660940672 | -0.35355339059327 | 0.38268343236509 |
| 123 | 0.14644660940673 | -0.35355339059327 | 0.38268343236509 |
| 124 | 0.14644660940673 | -0.35355339059327 | 0.38268343236509 |
| 125 | 0.14644660940673 | -0.35355339059327 | 0.38268343236509 |
| ... | ... | ... | ... |

Table 3: Converging iteration with $C = 0.25 - 0.25i$

Obviously, Z shows very different behaviour (converging vs. escaping to infinity) depending on our initial choice of C . To see how the two are related, we apply the iteration formula to every point C on the complex plane and color it according to the behavior if the iteration – if Z converges, we mark C black, and if Z escapes to infinity, we leave C white. The result is the graph shown in Figure 2:

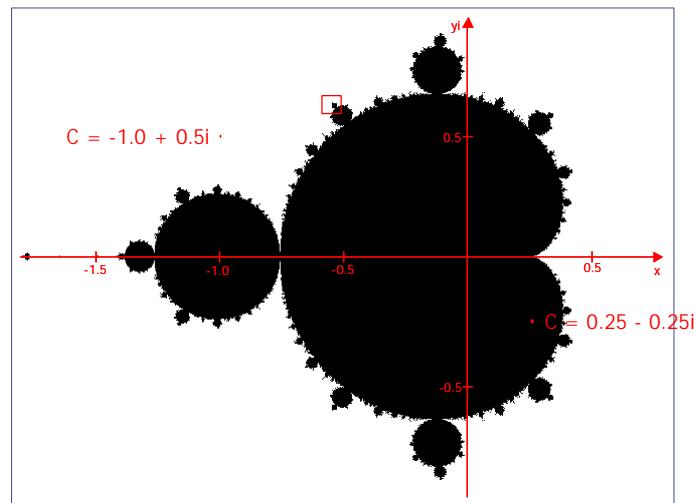


Figure 2: The Mandelbrot set

We have plotted the Mandelbrot set – the set of all points for which the iteration formula does not escape to infinity after a certain maximum number of iterations. It is a fractal with a solid interior, but an extremely convoluted edge that shows infinite detail and self-similarity when we look at it more closely.

Fractal Image Characteristics

To demonstrate these characteristics, we zoom into the section marked by the small rectangle in Figure 2, i.e., we apply the iteration formula to every point in the area delimited by the corner coordinates

$$C_{\text{top-left}} = 0.562261026 + 0.629245084i$$

$$C_{\text{bottom-right}} = 0.534635782 + 0.608525360i$$

We end up with the image in Figure 3:



Figure 3: Mandelbrot set detail with an iteration maximum of 150

As predicted, the result is not only a magnification of the pixels from the previous image, but actually contains new, intricate detail that was not visible before. We also recognize familiar shapes, like the bulges and tendrils on the edge of the set, and even a smaller, rotated image of the mandelbrot set at the top of this section.

The image looks a bit "dirty" like a bad photocopy, though – the edges are not clearly defined, and there are black points scattered outside the solid interior. The reason is that this image was computed with an iteration maximum of 150 – i.e., if Z did not escape to infinity after 150 iterations, the point is considered be part of the Mandelbrot set. However, we can imagine that there are points which take a many more iterations, like 1,000 or even more, to escape to infinity. These points would have been mistakenly colored black in the above image.

So, we recalculate the image, and this time iterate up to 1,500,000 times over each point before we decide that Z indeed does not escape to infinity. The result is shown in Figure 4:

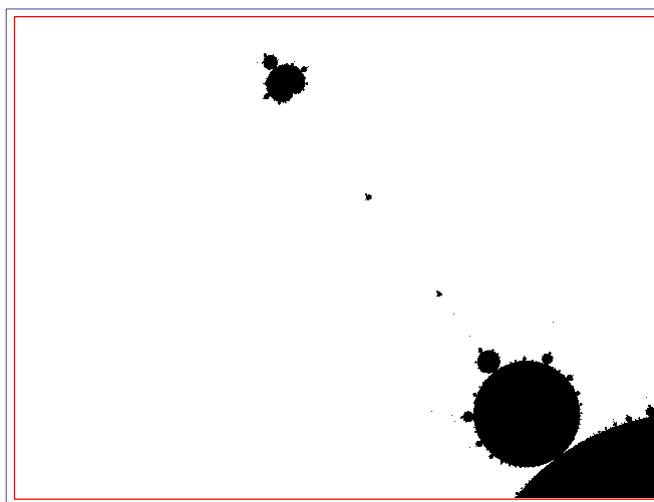


Figure 4: Mandelbrot set detail with an iteration maximum of 1,500,000

The image is much clearer now; the bulges and the small replica of the Mandelbrot set are clearly discernible. We note that **the higher the iteration count, the higher the accuracy of the fractal image.**

Parallel Algorithm Design

We have seen that in order to obtain an accurate image, we need to set the maximum iteration count sufficiently high, especially when zooming deeply into a tiny section of the set – which is necessary to see the most interesting areas of the Mandelbrot set. For example, it can be mathematically proven that the inside of the Mandelbrot set is solid, i.e. all points inside set are connected. The fact that we saw isolated black points and "islands" in the previous images is due to the limited resolution of our calculation. If we zoomed in extremely deep and used a very high resolution and iteration count, we could see some of the points connecting the seemingly "isolated" parts of the set.

Obviously, fractal image generation is computationally expensive. For example, to print a highly accurate black and white fractal image on a letter-size sheet of paper (8.5 x 11") at a resolution of 600 dots per inch, we have to apply the iteration formula to $8.5 * 11 * 600^2 = 33,660,000$ points, and perform up to 1,500,000 iterations on each of those points, leading to an upper bound of 50,490,000,000,000 (over 50 million million) computations.

While there are several ways to optimize the fractal image generation algorithm, like bailing out of the iteration loop early when it becomes obvious that Z is growing without limit, or if Z has converged, the computation can still take a long time. This is especially true when using the serial approach traditionally used by most fractal plotting programs, with images computed one pixel at a time.

We assume that parallel processing techniques can help us to reduce the computation time by spreading the workload over several processors. Our goal therefore is to develop a parallel fractal computation algorithm that poses no inherent restrictions size of the computation job.

Partitioning

As we saw in the first part, a fractal image is computed the following way:

```
FOR each line
  FOR each column in the line
    WHILE NOT maximum iteration count reached AND NOT Z escaping to infinity
      apply iteration formula
    END WHILE
    IF maximum iteration count reached
      color point black
    ELSE
      color point white
    END IF
  END FOR
END FOR
```

The first phase in the design of the parallel algorithm is to partition the problem into its elementary parts [IF95]. In this case, the most elementary part is performing one iteration of the formula on one point.

Agglomeration

In the second phase (agglomeration), we have to decide how to group all the elementary computations that make up the complete job into separate tasks that we can assign to the processors. The first decision is pretty obvious: Since the Mandelbrot iteration formula feeds the result of the previous step back into the next, all the computations for one point should be performed on the same processor to avoid the need for communication among processors.

Next, we have to decide how to agglomerate the points. Since no information from other points of the image goes into the iteration formula, there is no need for communication among the points. This gives us a lot of freedom when assigning points to processors since we don't have to find an agglomeration that minimizes communication. However, in order to accommodate very large output sizes, no processor should ever have to store the whole image.

The most simple method that comes to mind would be to assign each processor a continuous block of the image. For example, with four processors, the first processor would compute the first quarter of the image, the second would compute the second quarter, and so on. This concept of blockwise agglomeration is visualized in Figure 5, where the colors and the numbers in front of each line denote the processor working on that line.

As a side note, Figure 5 also shows the solution of a minor implementation problem: Since graphical output is not available on the nodes of a parallel processing cluster, the nodes return an ASCII string for each line, where a " " character denotes a point outside the fractal set, and a "#" character denotes a point inside the set.

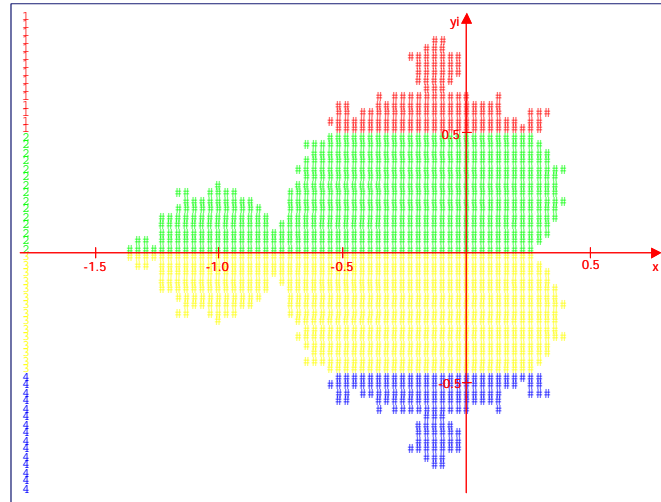


Figure 5: Blockwise agglomeration of points

Blockwise agglomeration is easy to implement, however, Figure 5 hints at a potential problem: As we can see, the largest part of the Mandelbrot set's inside is computed by processors 2 and 3, while most of the points assigned to processors 1 and 4 lie outside of the set. Since points inside the set always reach the maximum iteration count, processors 2 and 3 have a much greater workload than 1 and 4. This means that processors 1 and 4 will likely finish their computations and sit idle long before 2 and 3 are done, which is a waste of processing power if 1 and 4 can not be used for other jobs until the whole image is generated.

Consequently, we have to find a way to balance the load more evenly. A simple method to achieve this is assigning each processor not whole blocks, but alternating lines of the image, i.e. the first processor computes the first, fifth, ninth line etc, the second processor computes the second, sixth, tenth line etc., and so on (Figure 6):

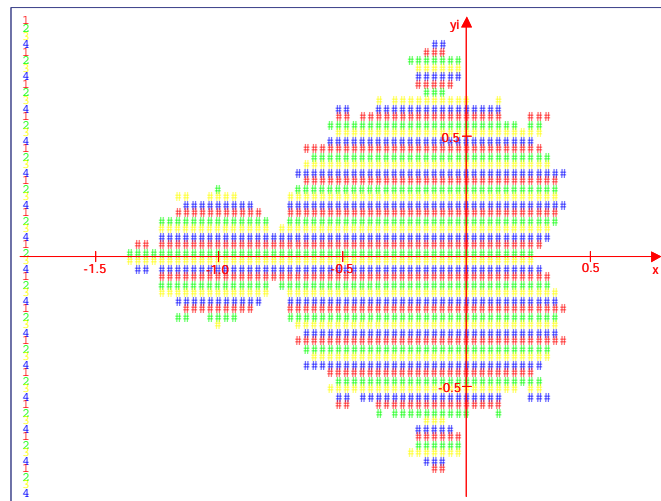


Figure 6: Linewise agglomeration of points

While this linewise agglomeration method does not balance the load precisely equally, it is a good approximation for most fractal images (unless somebody comes up with a formula that generates an image where only every fourth row is inside the set).

Output Synchronization

Although no communication is necessary to compute all the points, the nodes ultimately have to communicate in order to output the complete image. When designing the output algorithm, we have to keep in mind our requirement that no node should ever have to store the whole image – rather, every node should be responsible for its own part of the output. At this time, we face the problem that no node has a continuous block, but only non-consecutive lines of the image. If each node was to output its lines as soon as they are completed, the resulting overall output would be completely out-of-order since the nodes will take different amounts of time to compute different lines. Thus, **we need a synchronization mechanism to output the distributed lines sequentially and in order.**

Token-Passing Synchronization

As a first approach, we can make use of the fact that the lines are distributed among the nodes in a cyclic fashion (node 1 has lines 1, 5, 9, ...; node 2 has lines 2, 6, 10, ...; etc.). After all nodes have completed their share of the lines, they use a simple token-passing algorithm to output their lines one after another (Figure 7):

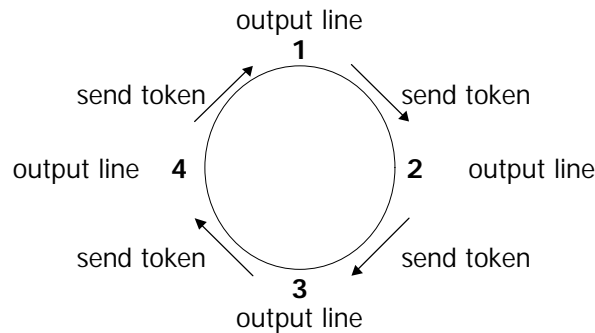


Figure 7: Token-passing synchronization

In this model, every node must wait to receive a token from its previous node before it can output a line. Then, it passes the token on to the next node, which has the next line of the image, and so on. Of course, one "master" node (the one that has the very first line) must initiate the cycle as described in the following pseudocode:

Master node:

```
compute all my lines
output my line
send token to next node
continue with WHILE loop as every other node
```

Every other node:

```
compute all my lines
WHILE NOT all my lines are output
    wait for reception of token from previous node
    output my next line
    send token to next node
END WHILE
```

An obvious drawback of this approach is that **only one node is busy outputting a line at any given point in time while all other nodes have to be idle, waiting until it is their turn.** Processing power is further wasted by the fact that all nodes have to finish computation before the output cycle begins.

However, this drawback only exists if we use blocking communication, i.e. if a node can not execute any other commands while it is waiting to receive a message. If we use non-blocking communication, the node can initiate the reception of a message at some point in time, then continue working on other things and just occasionally check if a message has arrived. Thus, if we use non-blocking communication for synchronization, the nodes can start outputting lines earlier and continue computation while it is not their turn to output a line, as described in the following pseudocode:

Master node:

```
compute my first line
output my first line
send token to next node (non-blocking)
continue with non-blocking reception as every other node
```

Every other node:

```
initiate non-blocking reception of token from previous node
WHILE NOT all my lines are output
  IF NOT all my lines are computed
    compute my next line
  END IF
  IF token received
    output my next completed line
    send token to next node (non-blocking)
    initiate non-blocking reception of token from previous node
  END IF
END WHILE
```

Here, each node checks after the completion of every line if it has received a token. If that is the case, it outputs the line, passes the token on, gets ready for the reception of the next token, and then continues to compute the next line. If a line is completed but no token has been received in the meantime, the node just goes ahead and computes the next line, and so on. This means that a node can finish the computation of all lines before it has finished the output of all lines. In that case, it will output the remaining lines as soon as the respective tokens arrive, just sitting idle between tokens.

One might argue that it would be better to check for an incoming token more frequently than only after each completed line (which might actually be quite a long time for high iteration maximums). However, checking after every point (or even after every iteration), produces considerable overhead that slows the calculation down. In addition, there is not really any performance gained by outputting a completed line and passing the token on as soon as possible, since we are not wasting many resources when we let lines accumulate: If the load is balanced relatively evenly, the final phase where some nodes are idle between communications should not be too long since all nodes need about the same time to compute all their lines.

The algorithm seems to work fine in theory: Each node computes its lines as fast as possible, and outputs them as soon as they can be under the in-order requirement. However, when we implement the algorithm and run it on a parallel processing cluster, we still observe that the lines in the output file are out-of-order:

```
PE0 started up on summit5
Maximum iterations: 150
Output size:      64 x 48 points
Top-left corner:  (-1.78, 0.96i)
Bottom-right corner: (0.78, -0.96i)

Number of PE's:   4
Rows per PE:     12

PE1 started up on summit4
PE2 started up on summit3

                                     #
PE3 started up on summit8           #####
                                     #####
                                     # ##### #
#####
```



Figure 8: Output of token-passing synchronization algorithm

To find the cause for this effect, we add debugging information to the output. In addition to the ASCII image, each node (process entity, PE) also prints out a line when sending and receiving a token. The value of each token consists of the PE number times 100, plus the number of the line was just output. For example, token 205 is sent by PE2 after outputting the fifth line in its share of lines (since PE's are numbered starting with 0, token 1 is sent by PE0 after outputting its first line). With this debugging information, we get the following output file:

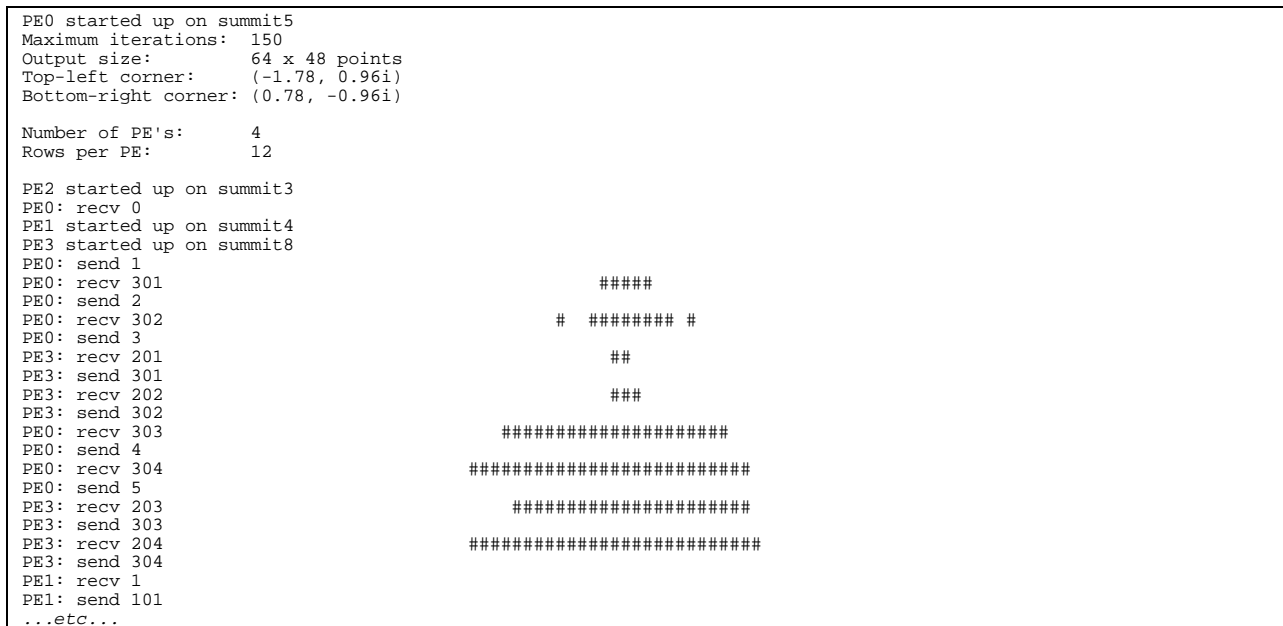


Figure 9: Output of token-passing synchronization algorithm with debugging information (truncated)

Stepping through the beginning of the output file shown in Figure 9, we see that the very first line output by PE0 is prefixed "recv 0", indicating correctly that no token was received before. Then, it sends token 1 ("PE0 just output line 1") to PE1, which is exactly what it is supposed to do. But next, PE0 immediately receives token 301 ("PE3 just output line 1"), which was not even sent yet, according to the output file. Obediently, PE0 outputs its next line, although it seems clearly out-of-order, then sends token 2. However, instead of the expected "PE1: recv 2", we then see that PE0 receives the next token (302) from PE3 already, outputs the respective line and sends token 3. Only now do we get a word from PE3, announcing it received token 201 and output its first line – which is not sent until much later, according to the output file. And then, finally, we see that PE3 sent token 301 as it was supposed to do. The output file continues in a similar unpredictable fashion until all lines are output.

Examining the output file more closely, we see that all nodes actually do exactly what they are supposed to do: They output the next line only when they receive a token from their predecessor, and then send a token to their successor. Every node sends and receives the correct tokens, and outputs the correct lines. The only problem is that the tokens seem to arrive in the wrong order, being received before they have been sent.

The reason for this effect is subtle: As we can see, every node obediently outputs a line only after receiving the respective token from the previous node. So, the nodes indeed perform their output operations sequentially and in-order. However, **the nodes have no control over how their individually output lines are collated in one overall file.** Through buffering and delays introduced by the parallel processing environment, the overall output file is still being pieced together in chunks that may be larger than one line per node, thereby garbling the output order of lines.

Polling Synchronization

It seems that we have come to a dead end: Even when we synchronize the output operations of the nodes, we can not make sure that their output is actually collated in the right order. We can only be sure that the order in which a node outputs lines is the same as the order in which those lines appear in the complete output file – although randomly interspersed with other nodes' output.

Thus, the only way of assuring an in-order output of all the lines is to have one node output all of them - although this seems to violate our requirement that no node should have to store the whole image. However, the node responsible for the output does not have to store all the lines – it would be sufficient if it just got the other nodes' lines that have to be output next, output them in order, and then discard them to make room for the next set of lines (called "chunk" in the following). For example, the first chunk consists of the first lines of all nodes, the second chunk comprises the second lines of all nodes, and so on. Figure 8 illustrates this by showing the chunk number (1 to 15) and the node number (0 to 3) in front of each line:

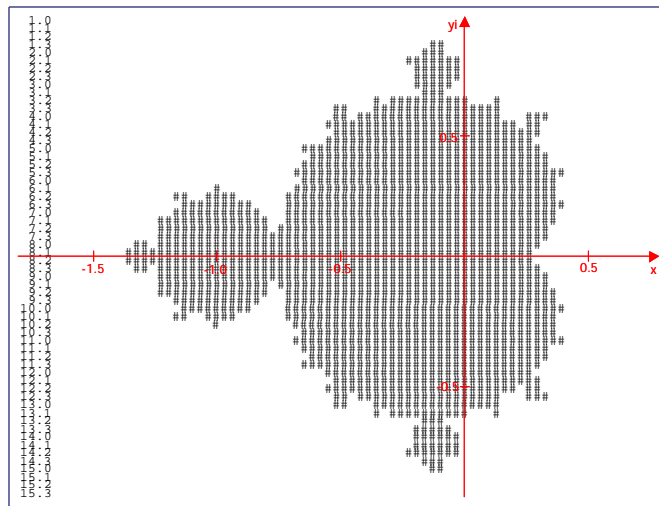


Figure 10: Output chunks of four nodes

This polling algorithm can be realized with the following algorithm, where requests and responses are implemented using non-blocking communication:

Master node:

```
WHILE NOT all chunks are output
  send request for lines of next chunk to other nodes (non-blocking)
  initiate non-blocking reception of other nodes' lines
  REPEAT
    IF NOT all my lines are computed
      compute my next line
    END IF
  UNTIL lines of chunk received
  output all lines of chunk in order (including my line)
  discard chunk
END WHILE
```

Every other node:

```
WHILE NOT all my lines are sent
  initiate non-blocking reception of master's request
  REPEAT
    IF NOT all my lines are computed
      compute my next line
    END IF
  UNTIL request received
  send requested line to master (non-blocking)
END WHILE
```

For each chunk of the image, the master requests the respective lines from the other nodes, then computes its own line for that chunk. If it received the other lines by the time it finished computation of its own line, it outputs all the lines of the chunk in order. Otherwise, it just continues to compute more of its own lines until the other nodes have responded. As soon as the chunk has been output, it is removed from memory, and the cycle repeats for the next chunk.

The other nodes just compute their lines of the image all the time, and whenever they receive a request from the master, they send it the respective line. Since the master requests the lines in the order they are computed, and the other nodes always compute at least one line before checking for the reception of a request, it is ensured that they always have the requested line available. As in the previous algorithms, nodes may finish computation of all their lines before the whole image is output. In this case, they just sit idle while they are not handling requests.

One might argue that the polling mechanism is unnecessary communication overhead, and all the nodes could just send their lines to the master as soon as they are finished. However, in this case, the master would need a buffer to store all the lines that were received, but not yet due for output. Under some circumstances, the master could then end up storing almost all the lines of the image, which is something we were trying to avoid all the time. Using the request/response algorithm, on the other hand, the master just needs to buffer one line from each node at a time.

If we implement and run this algorithm on a parallel processing cluster, we finally get the desired result, as shown in the following output file:

```
PE0 started up on summit5
Maximum iterations: 150
Output size: 64 x 48 points
Top-left corner: (-1.78, 0.96i)
Bottom-right corner: (0.78, -0.96i)

PE1 started up on summit4
Number of PE's: 4
Rows per PE: 12

PE2 started up on summit3
PE3 started up on summit8
```



Figure 11: Output of polling synchronization algorithm

We now have a working algorithm for the in-order output of sequential data generated by a parallel program. The algorithm was implemented using the Message Passing Interface library [MPI97]. The serial and parallel programs were executed and timed on the University of Montana's supercomputer *summit.cs.umt.edu* [DM01].

Performance Analysis

Our next step is analyze to analyze the performance of the algorithm we just developed. We want to see how much time we actually save by computing the parts of the image in parallel, and we need to check if the synchronization mechanism introduces any overhead.

In order to see how the execution time changes with the problem size, we run the program with a maximum iteration count of $N = 5,000$, $N = 50,000$ and $N = 500,000$.

To get good measurements without too much influence of fluctuating system performance (e.g. background processes, other users' activity), we calculate the image of the whole Mandelbrot set with 160×120 pixels and run the program three times for each choice of N , averaging the results. After completing all the test runs, we end up with the following (averaged) timings:

| average $T_{P,N}$ | $P = 1$ (serial) | $P = 2$ | $P = 4$ | $P = 6$ |
|-------------------------------------|------------------|---------------|---------------|---------------|
| $N = 5,000$ | 1,43199333 s | 0,48863167 s | 0,25034200 s | 0,17215833 s |
| $N = 50,000$ | 14,18023330 s | 4,67098667 s | 2,36133667 s | 1,58265667 s |
| $N = 500,000$ | 141,93433300 s | 46,43553330 s | 23,48320000 s | 15,71240000 s |

Table 4: Average execution walltime

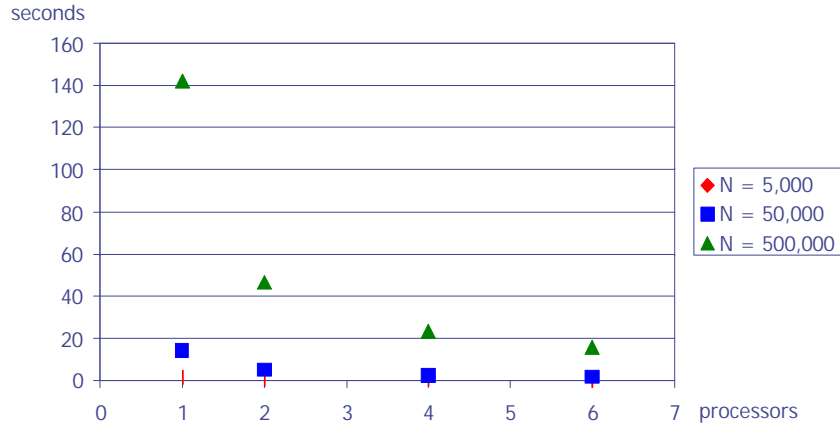


Figure 12: Execution walltime chart

Table 4 shows that the execution time grows almost proportionally with the maximum number of iterations. This makes sense since the number of points computed remains the same, but the time to compute the points inside the set grows with the number of iterations. Since the time required for most points outside the set stays the same no matter how high the iteration maximum, the relation can not be exactly proportional.

We are mainly interested in the performance gain that can be achieved when using a parallel algorithm on a certain number of processors as opposed to a serial algorithm on one processor. From Figure 12, we can deduce that with an increasing number of processors, the execution time decreases overproportionally – for example, we would expect that by doubling the number of processors, we can cut the execution time in half, but the actual execution time is even less than that. To examine this effect, we compute the speedups, defined as

$$S_{P,N} = \frac{T_{1,N}}{T_{P,N}}$$

where $T_{P,N}$ is the overall execution time when running on P processors with an iteration maximum of N .

| $S_{P,N}$ | $P = 2$ | $P = 4$ | $P = 6$ |
|---------------|------------|------------|------------|
| $N = 5,000$ | 2,93061915 | 5,72014816 | 8,31788581 |
| $N = 50,000$ | 3,03581113 | 6,00517219 | 8,95976592 |
| $N = 500,000$ | 3,05658884 | 6,04407973 | 9,03326882 |

Table 5: Speedups

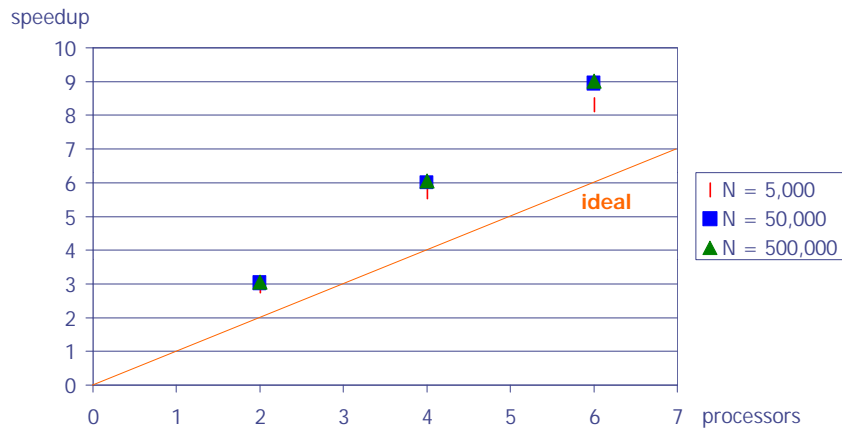


Figure 13: Speedup chart

Surprisingly, the achieved speedups are even higher than the "ideal" linear speedup, although the parallel program does even more work than the serial program, since it has to communicate in order to synchronize the output. We can rule out an erroneous measurement since all test runs were performed three times, and the timings were within remarkably close range. One might imagine some external influence (like a background process or another user's job) slowing the serial program down, but this would only have yielded a deviation in one measurement, but not in all nine test runs of the serial program. We can therefore assume that the timings must be correct.

The overproportional speedup can be explained when we look more closely at the differences between the serial and the parallel algorithm: In the serial algorithm, we compute a point, output it, compute the next point, output it, and so on - the whole algorithm is completely serial (Figure 14):

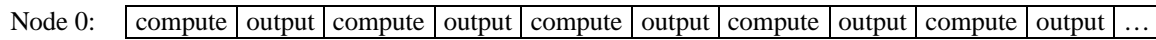


Figure 14: Operations in serial execution

In the parallel algorithm, we obviously have several nodes computing points in parallel. But in addition to this, **we have a second layer of parallelism introduced by the non-blocking communication because the output of a node (sending a line to the master) is performed in the background while the computation continues:**

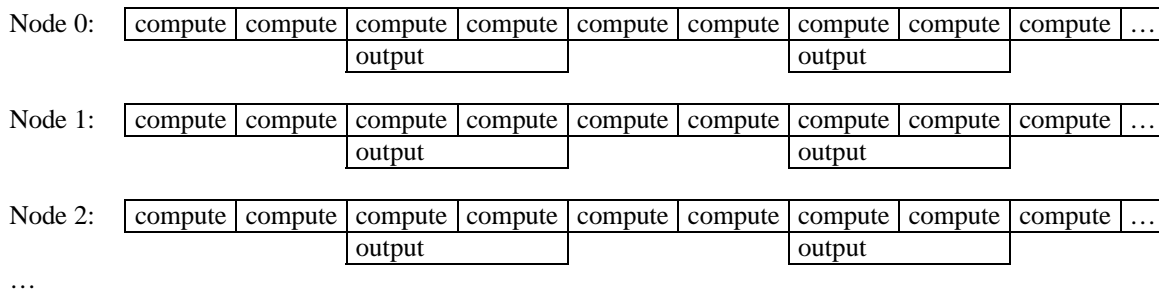


Figure 15: Operations in two-level parallel execution

It might be argued that the parallel execution of computation and output is an illusion created by the operating system's multi-tasking, since the output process running in the background takes CPU time slices away from the computation process. However, this is not really the case: The transmission of data is an I/O operation that does not involve the CPU anymore after being initiated.

The parallel execution of computation and output in every node is therefore real and a subtle, but deciding factor in the overproportional speedup that we observe.

Conclusion

While the generation of fractal images like the Mandelbrot set seems to be of little more than academic value, there are other fractal objects that are of more relevance to the real world. Fractal models are used today to predict all kinds of systems that show chaotic behavior, such as the weather, population growth, or brain waves. We used the Mandelbrot set here because it is simple to generate, yet nicely demonstrates all characteristics of a fractal object. The program could be easily adapted to generate other fractal objects by changing the iteration formula.

More important, though, are the insights we gained into the methods for generating sequential data (like the lines of a fractal image) with a parallel algorithm: We found that a synchronization mechanism is needed in order to output the sequential data in-order, but we also realized that the parallel processing environment introduces a random factor even if the nodes are synchronized. Thus, we concluded that the only way to achieve in-order output is to have one node responsible for performing all the output. While this sounds like bad parallel design, we realized that the memory requirements of the master node can be kept low by chunking the output, and we observed that by using non-blocking communication, we can actually output data more efficiently than with a serial algorithm.

Bibliography

- [BM77] Mandelbrot, B.: *Fractals: Form, Chance and Dimension*. W.H. Freeman & Co., 1977
- [DM01] Morton, D.: An Overview of *summit.cs.umt.edu*, April 9, 2001. Available online at http://www.cs.umt.edu/u/morton/cs580/LectureNotes/summit09-04-2001-1slide_per_page.pdf (cited: May 2001)
- [IF95] Foster, I.: *Designing and Building Parallel Programs*. Addison-Wesley, 1995. Available online at <http://www.mcs.anl.gov/dbpp/> (cited: May 2001)
- [MPI97] MPI Forum: *The Message Passing Interface (MPI) Standard*. Available online at <http://www.mcs.anl.gov/mpi/> (cited: May 2001)
- [TW00] Wegner, T. et al.: *Fractint 20.0 Documentation*, February 2000. Included in Fractint software packages available online at <http://www.fractint.org/ftp/release.20.0/> (cited: May 2001)