Embarrassingly Parallel Computations

Creating the Mandelbrot set

Péter Kacsuk

Laboratory of Parallel and Distributed Systems

MTA SZTAKI Research Institute

kacsuk@sztaki.hu

www.lpds.sztaki.hu

Definition of the Mandelbrot set

Definition:

A Mandelbrot set is a set of points in a complex plane computed by iterating a function. Usually the function is:

$$z_{k+1} = z_k^2 + c$$

where z_{k+1} is the (k+1)th iteration of the complex number:

$$z = a + bi \quad \text{and} \quad z_0 = 0$$

c is a complex number giving the position of the point in the complex plane.

First iterations of the Mandelbrot set

Notice that

$$z_k^2 = a^2 + 2abi + bi^2 = a^2 - b^2 + 2abi$$

therefore:

$$z_{k+1,real} = z_{k,real}^2 - z_{k,imag}^2 + c_{real}$$

$$z_{k+1,imag} = 2z_{k,eal}z_{k,imag} + c_{imag}$$

Computing the first iterations:

$$z_{1real} = c_{real}$$

$$z_{1imag} = c_{imag}$$

$$z_{2real} = c_{real}^2 - c_{imag}^2 + c_{real}$$

$$z_{2imag} = 2c_{real}c_{imag} + c_{imag}$$

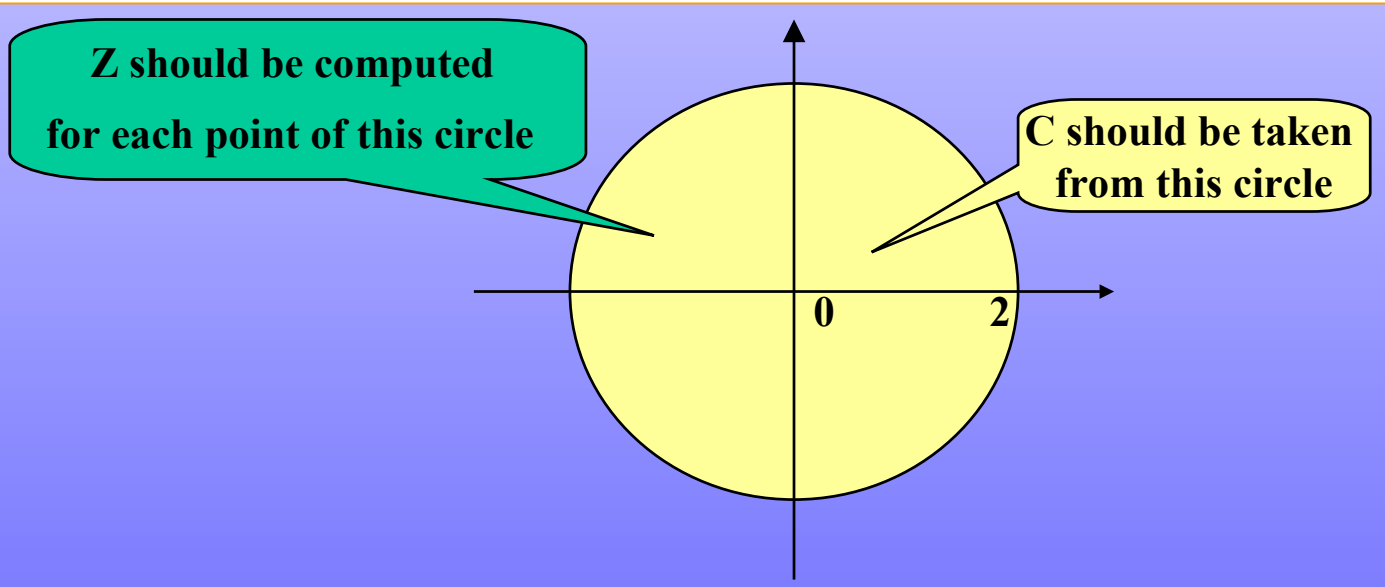
Termination condition of iterations

The iterations are stopped when the magnitude of *z* is greater than 2 (which indicates that *z* will eventually become infinite)

$$z_{length} = square(a^2 + b^2) < 2$$

or the number of iterations reaches some arbitrary limit.

Given the termination condition, all the Mandelbrot points must be within a circle with its center at the origin and of radius 2.

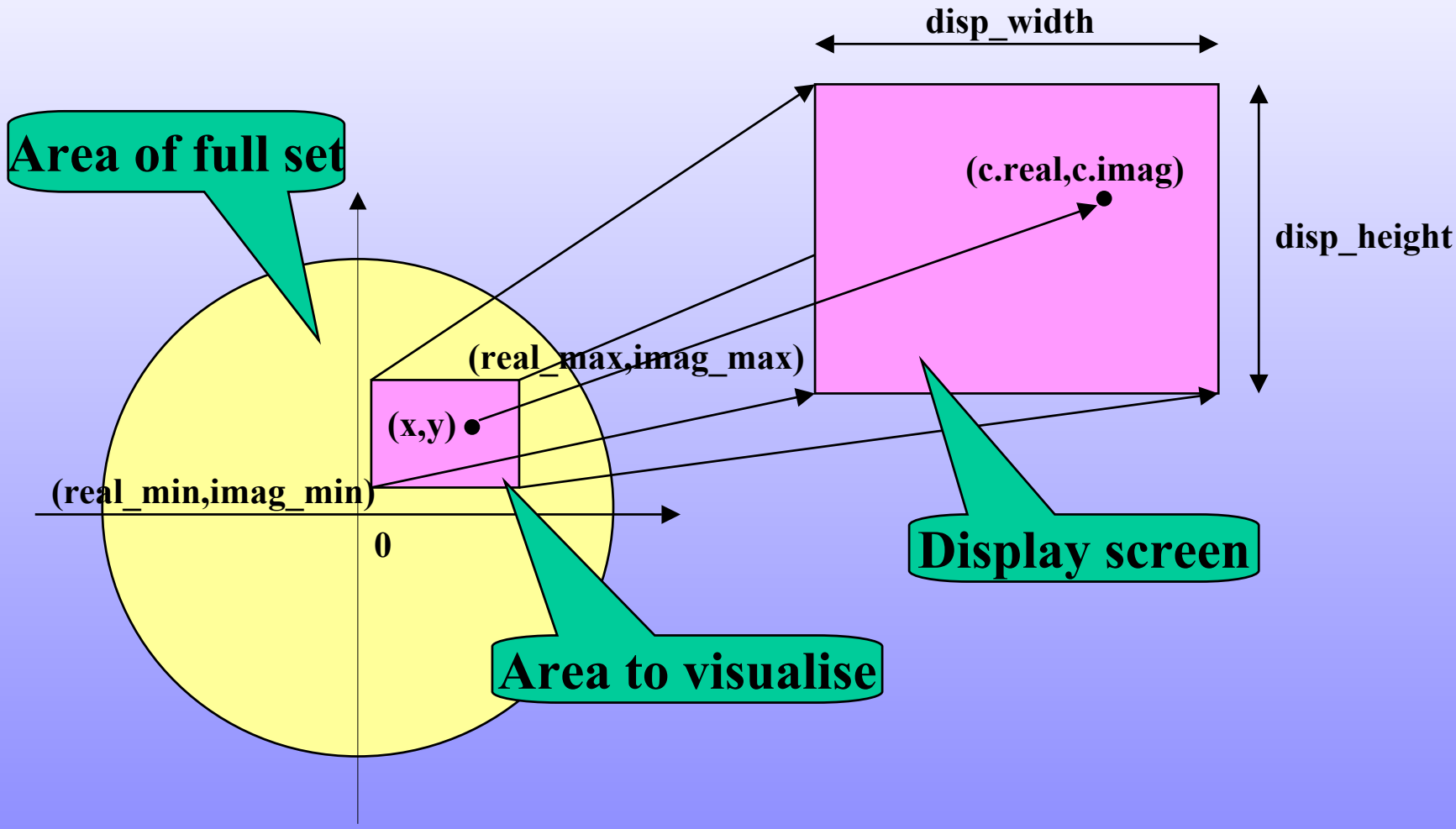


Computing the Mandelbrot set

A routine for computing the value of one point of the set and returning the number of iterations could be of the form:

```
int cal_pixel(complex c)
{ …                          /* declarations: z_k = 0 */
   count = 0;
   do { …                    /* compute z_{k+1} = z_k^2 + c
   } while ((lengthsq < 4.0) && (count < max));
   return count;
}
```

Visualisation of the Mandelbrot set



c.real = real_min + x * (real_max - real_min)/disp_width
c.imag = imag_min + y * (imag_max - imag_min)/disp_heigth

Visualisation of the Mandelbrot set

For each point of a rectangle of the circle we compute the iterations. We allocate different colours to the iteration numbers where the iteration terminated for a particular *c* point.

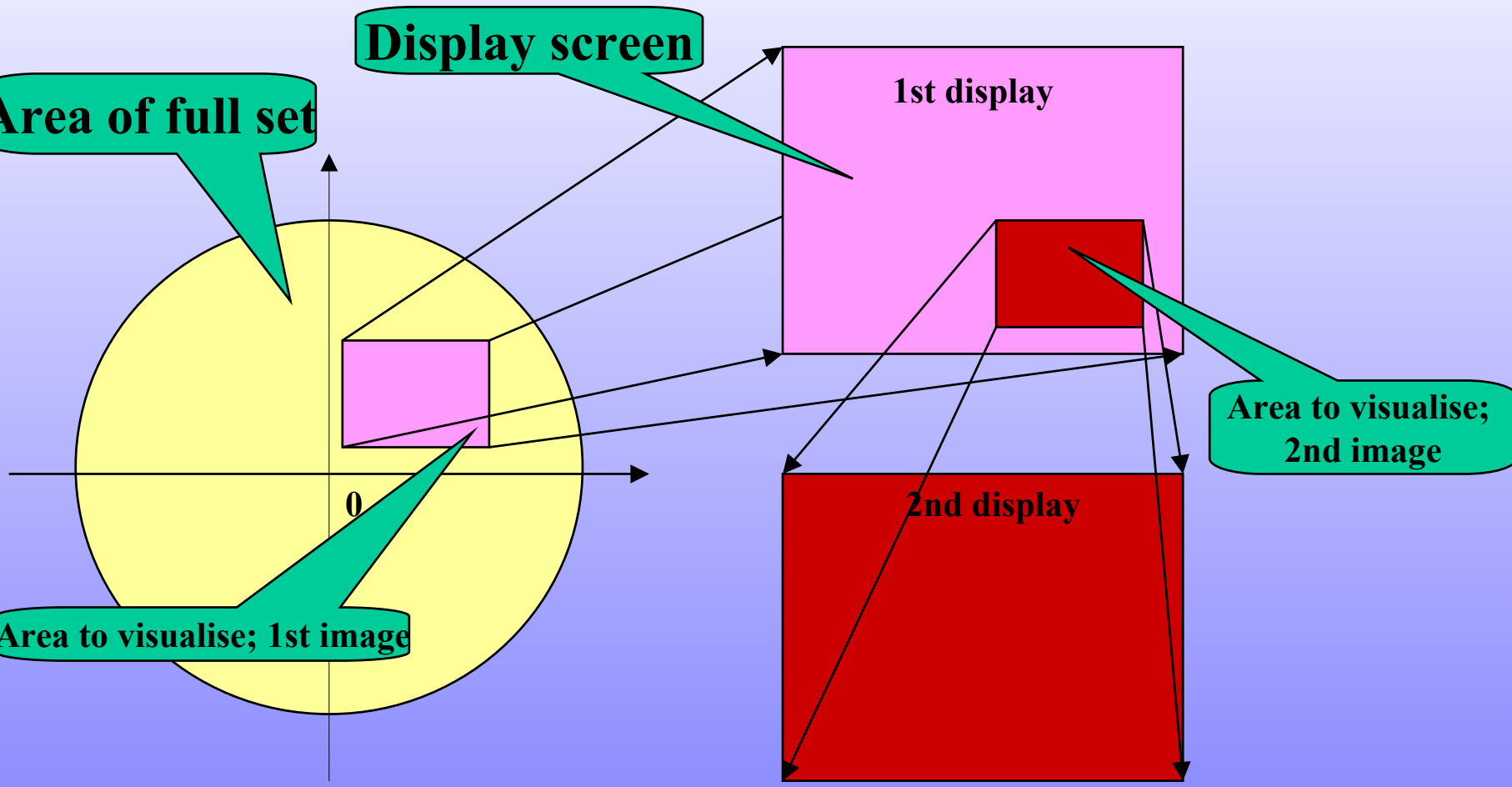
```
for (x = 0; x < disp_width; x++)          /* x, y are screen coord.s */
        for (y = 0; y < disp_heigth; y++)  {
                c.real = real_min + ((float)x * scale_real);
                c.imag = imag_min + ((float)y * scale_imag);
                color = cal_pixel( c );
                display(x, y, color); }
where
        scale_real = (real_max - real_min)/disp_width;
        scale_imag = (imag_max - imag_min)/disp_heigth;
```

A typical image of the Mandelbrot set

Expanding resolution during the visualisation of the Mandelbrot set

- The resolution is expanded at will to obtain fascinating images.
- After computing one image we can zoom to a smaller part of the original area and enlarge it to the size of the screen window.
- Computing and visualising again the Mandelbrot set we have the impression to travel in the world of fractals deeper and deeper.
- The zoom area is selected by the mouse.

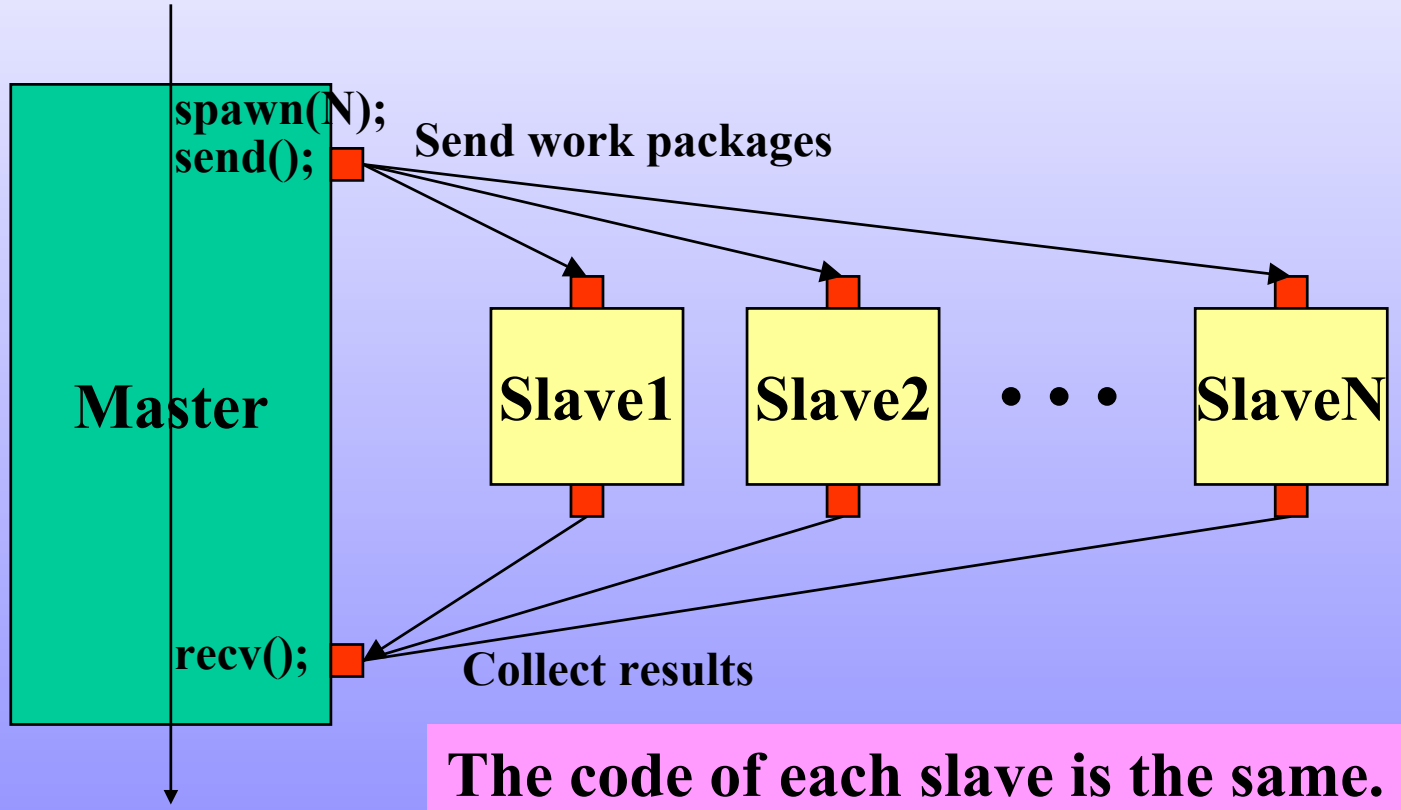
Expanding resolution



Parallelising the Mandelbrot set computation

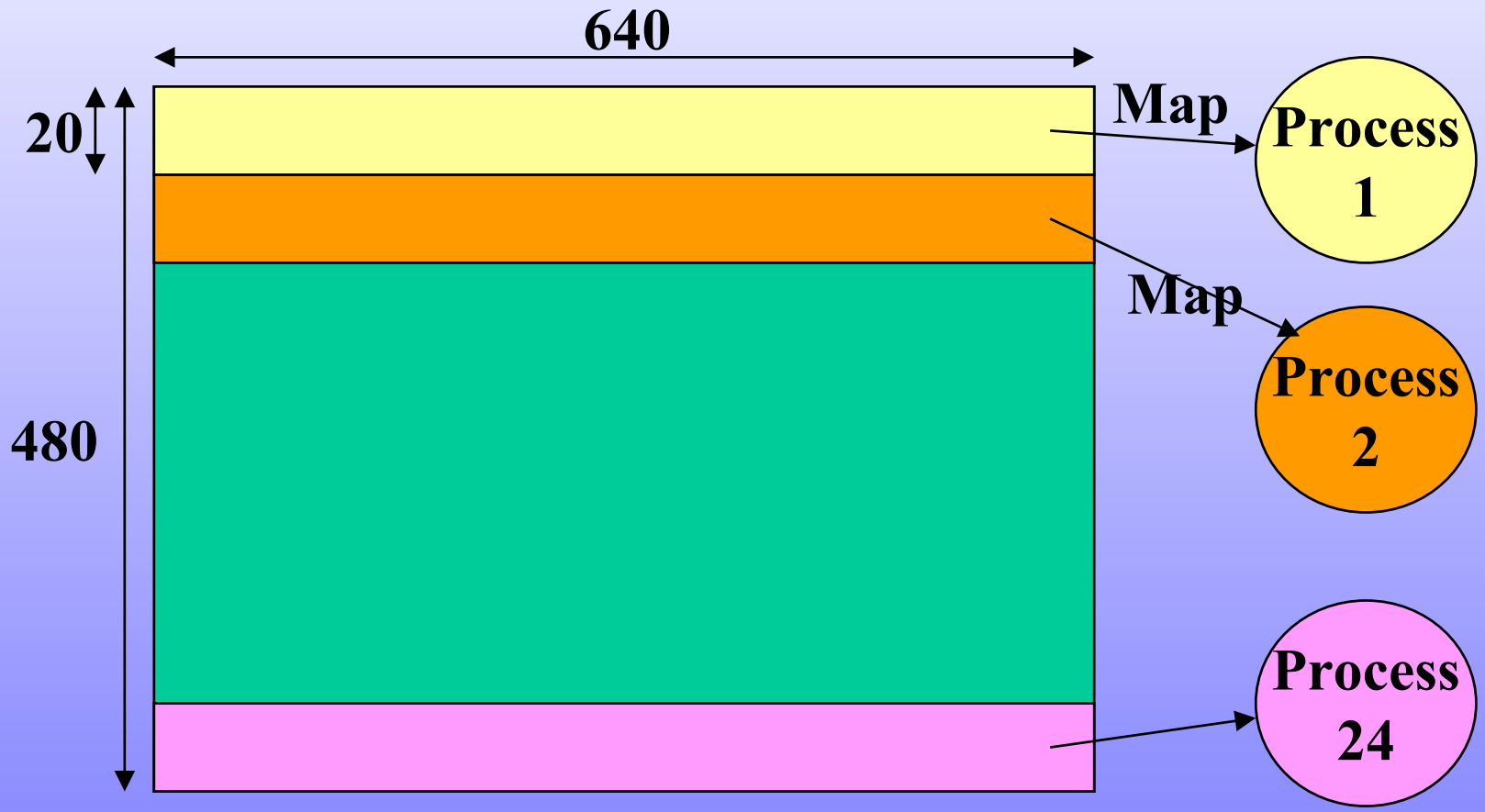
- Basic principle: **process farm (master-slave)** approach
- Two orthogonal methods of parallelising the code:
  1. **Data partitioning**
     - by square regions
     - by row (or column) regions
  2. **Task assignment**
     - Static
     - Dynamic

The process farm parallelisation approach



The code of each slave is the same.

Static row region data partitioning



Analysis of the Mandelbrot set computation

- **Problem of exact analysis**: we do not know how many iterations are needed for each pixel

- **The number of iterations** for each (**n**) pixel is some function of **n** but cannot exceed **max**.

  Therefore, the sequential time is

  $$t_s <= max * n$$

  or a sequential **time complexity** of **O(n)**.

Analysis

- **Three phases of the parallel program**:
- **Phase 1: Communication**
  - First the row number is sent to each slave, one data item to each of **s** slaves:

$$t_{comm1} = s * (t_{startup} + t_{data})$$

- **Phase 2: Computation**
  - The slaves perform their Mandelbrot computation in parallel:

$$t_{comp} <= (max * n) / s$$

- **Phase 3: Communication**
  - The results are passed back to the master by each slave:

$$t_{comm2} = s * (t_{startup} + t_{data})$$

Analysis

- **Total parallel time:**

$$t_p <= (max * n) / s + 2s * (t_{startup} + t_{data})$$

- **Speed-up:**

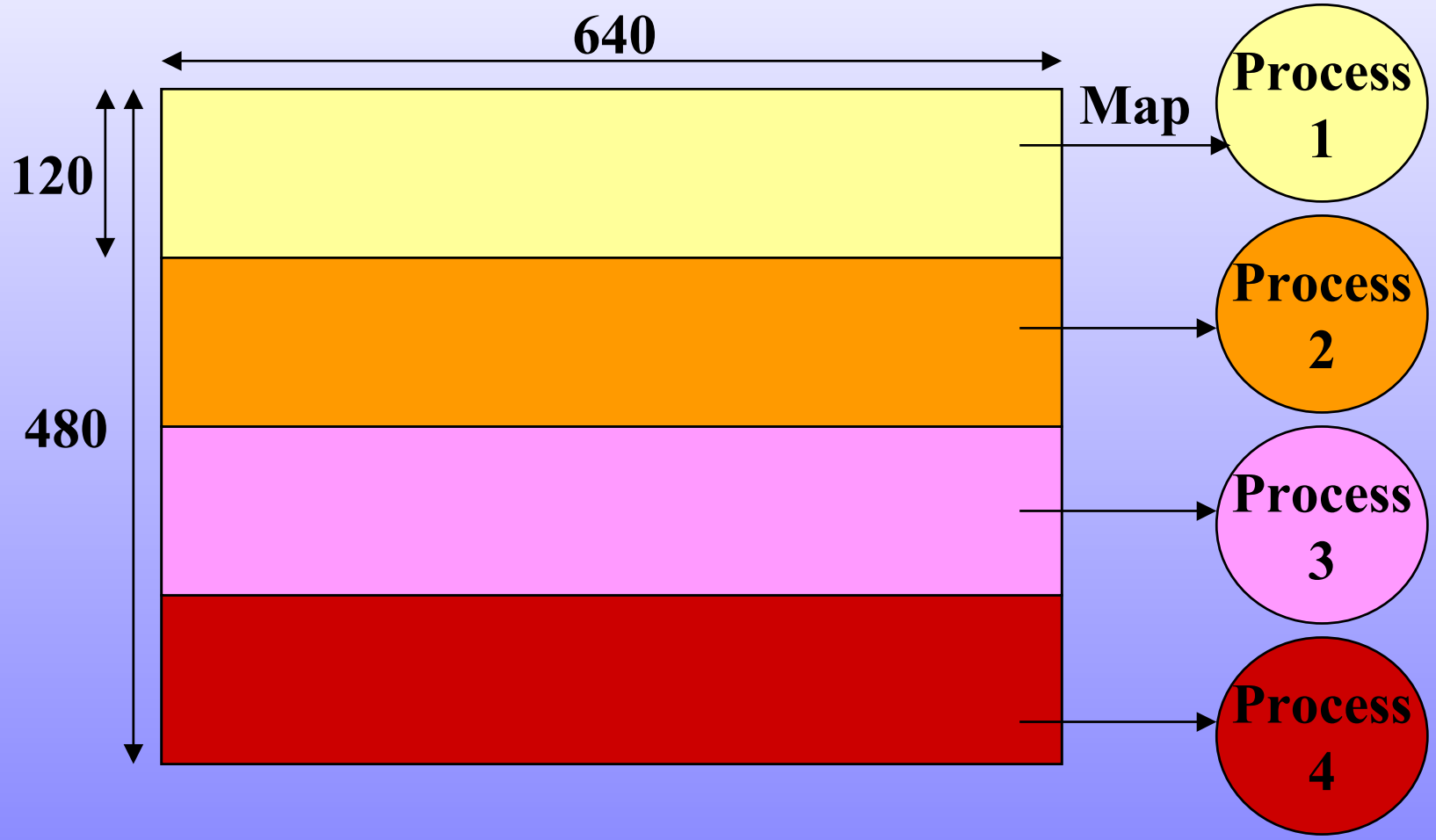
$$SP = t_s / t_p$$

$$1/SP = \frac{(max * n) / s + 2s * (t_{startup} + t_{data})}{max * n} =$$

$$= 1/s + 2s * (t_{startup} + t_{data}) / max * n \rightarrow 1/s$$

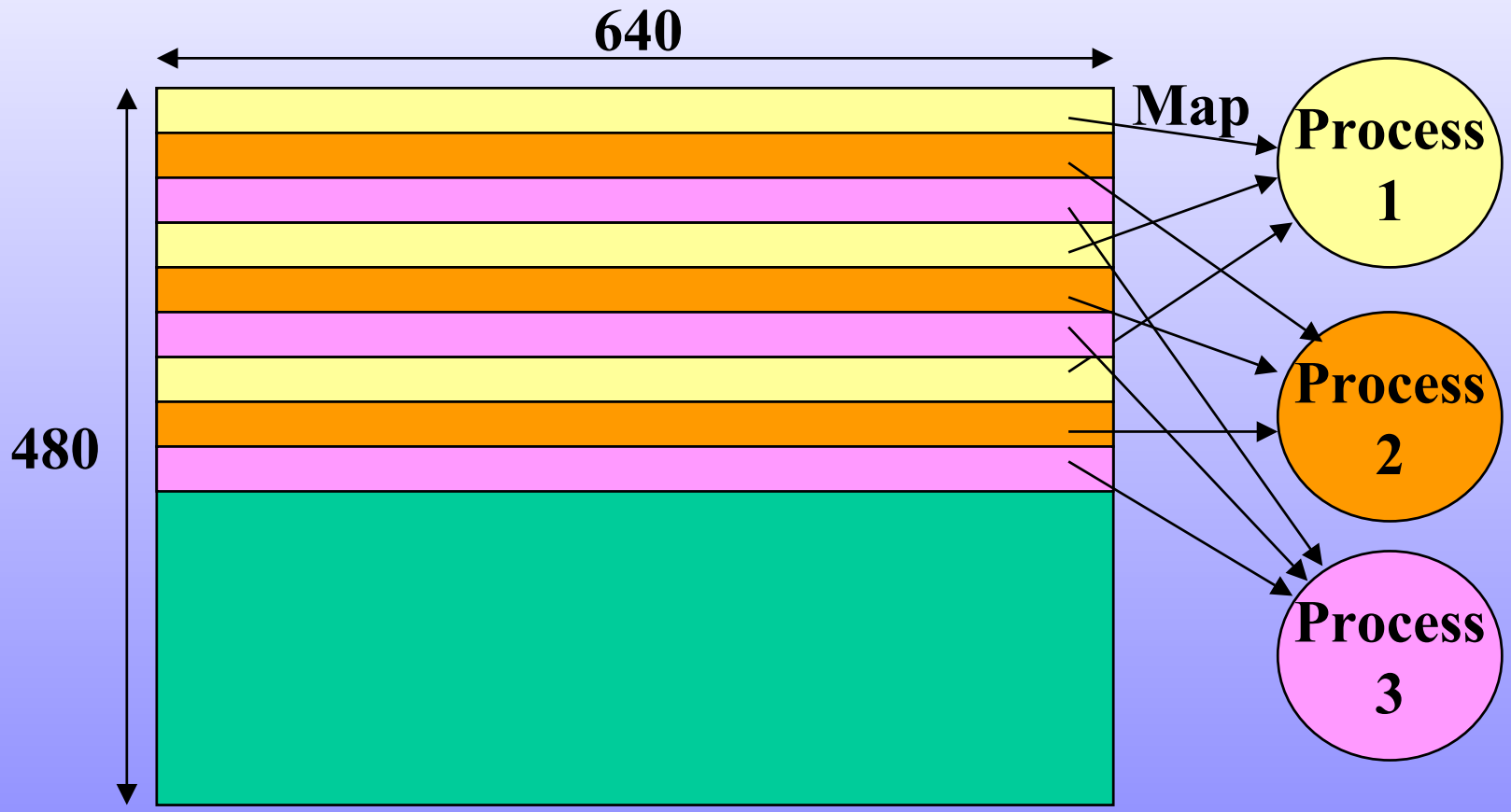
if max or n is large.

Coarse-grain, static, row region data partitioning with few processors



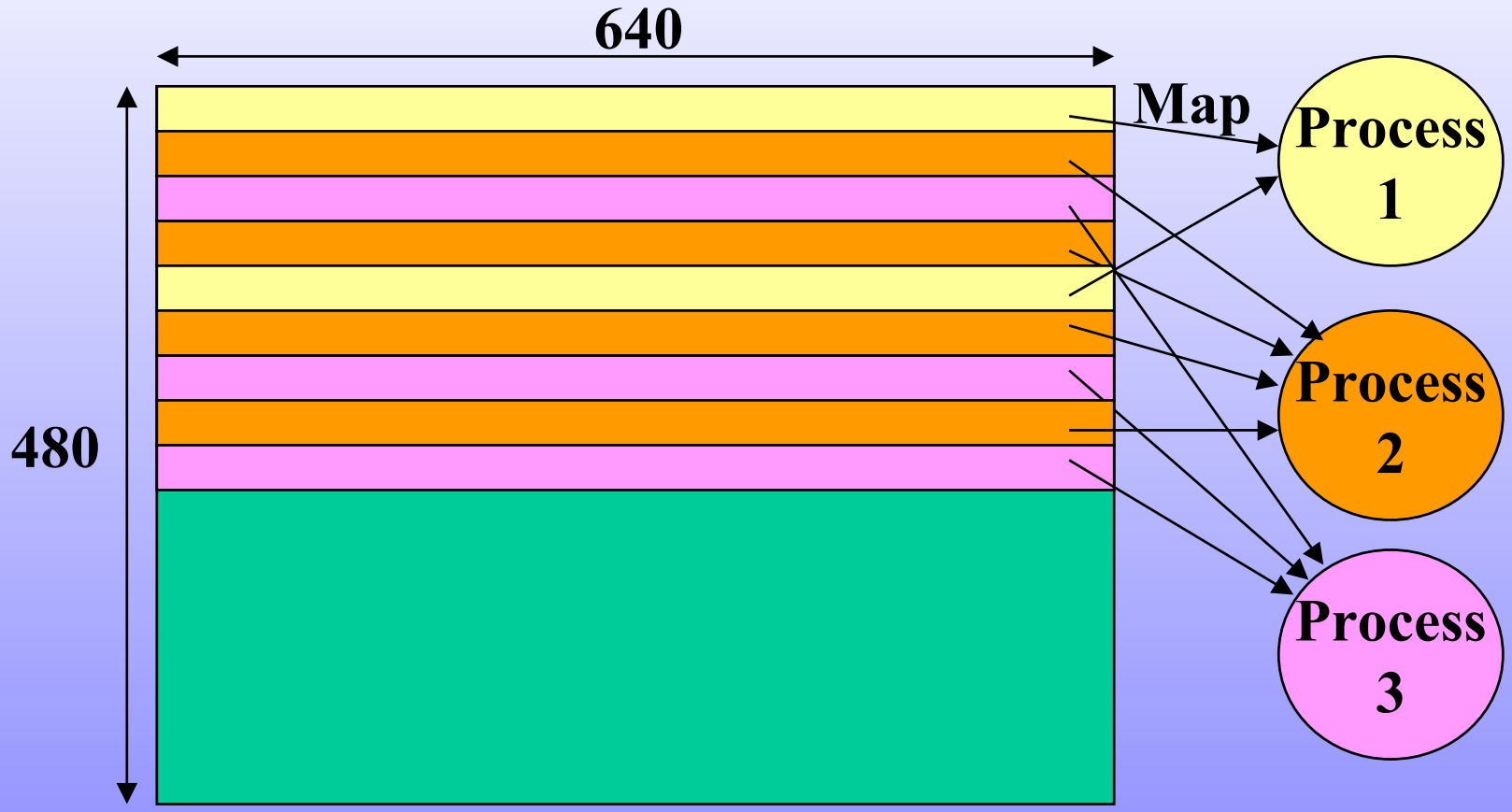
Drawback: unbalanced computation among processors

Fine-grain, static, row region data partitioning with few processors



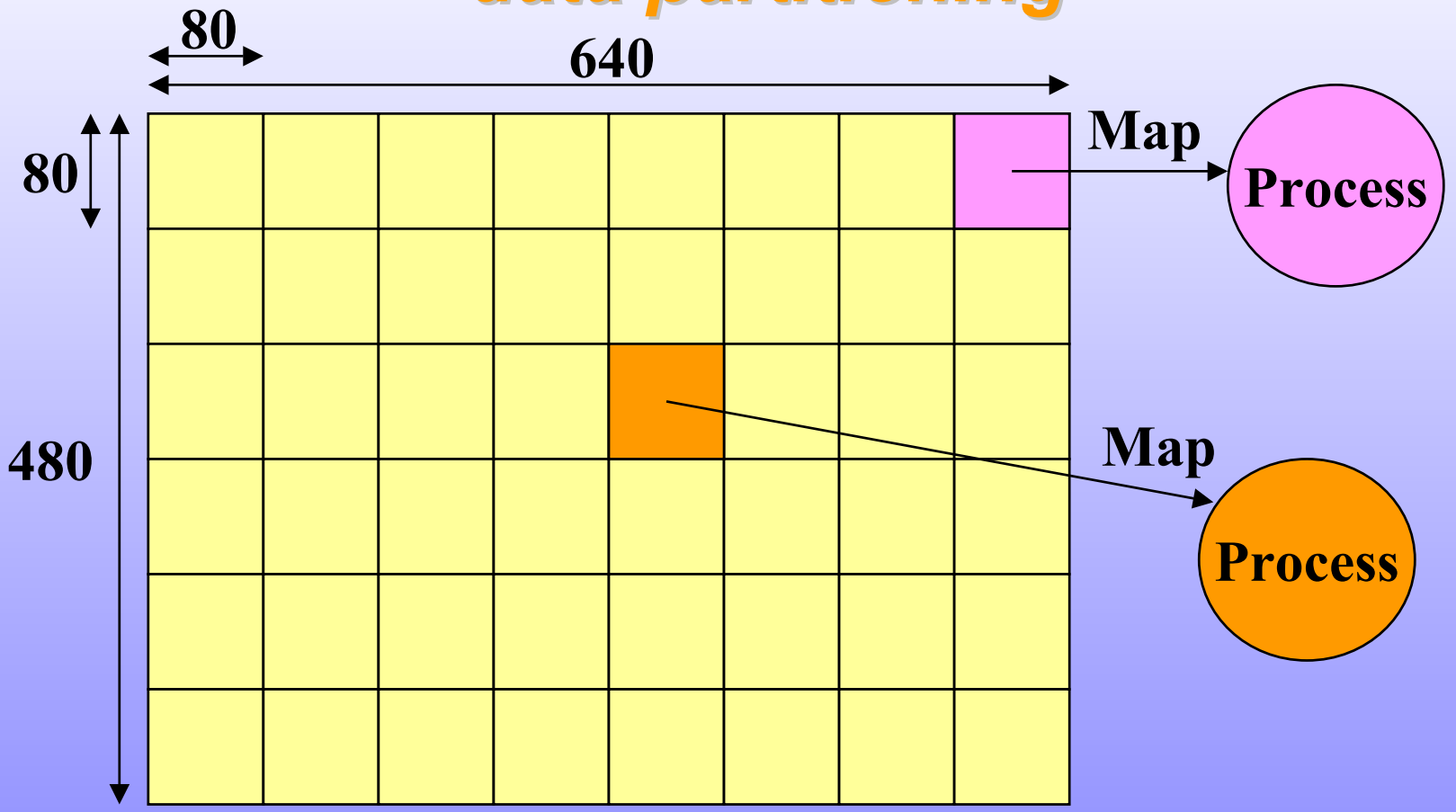
Advantage: more balanced computation among processors
See implementation in **GRAPNEL**

Dynamic row region data partitioning with few processors



Advantage: even more balanced computation among processors
Exercise: rewrite **GRAPNEL** program to dynamic partition, compare with the static one by **PROVE**.

Square region data partitioning



Exercise: rewrite the row partitioned **GRAPNEL** program to square partitioned one. Write both static and dynamic version. Compare all versions by **PROVE**.

A typical image of the Mandelbrot set

Thank You ...