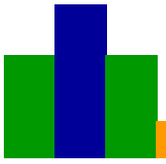


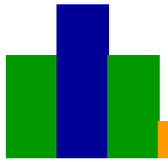
Lesson 3a: Message-passing (II)

Lesson 3b: Embarrassingly parallel computations

G Stefanescu — University of Bucharest

Parallel & Concurrent Programming
Fall, 2014

Lesson 3a: Message-passing (II)



Parallel program evaluation

Program evaluation:

- Both *theoretical* and *empirical* techniques may be used to determine the efficiency of (parallel) programs.
- The ultimate goal is to discriminate between various parallel processing techniques; a fine tune is also necessary to find the best *number of processes* and a good balance of the *computation time* and of the *communication time* of each process.
- An extra goal to find out if a parallel processing approach is actually better suited then a (usually simpler) sequential one.

Parallel execution time

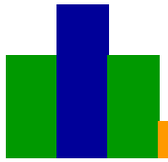
Theoretical evaluation of parallel programs is based on:

- Parallel execution time, t_p, consists of the *computation time* t_{comp} and the *communication time* t_{comm}, namely

$$t_p = t_{comp} + t_{comm}$$

- Computation time, t_{comp}, is estimated similarly as in the case of sequential algorithms. (It is supposed here that all processes use identical computers.)
- Communication time, t_{comm}, consists of the *startup time* $t_{startup}$ (also called "message latency") and the *time to send data* $n\,t_{data}$, where t_{data} is the time to transfer a data word, namely

$$t_{comm} = t_{startup} + n\,t_{data}$$



..Parallel execution time

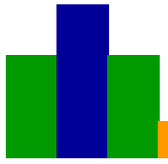
- Theoretical analysis is intended to give a *starting point* to how an algorithm might perform in practice.
- Parallel computation time is evaluated in terms of *arithmetical and logical operations* - their actual time depends on the computer system (also assume all computers are identical)
- Communication time should use the same time units as computation time; all (unit) data types are supposed to require the same time to be delivered

A typical example (IBM: SP-2; rough estimation):

—computation (1 arithmetical operation) - 1 unit

—time to transfer a data word - 55 units

—startup time - 8333 units



Latency hiding

- A general method to overcome the significant message communication time is to *overlap communication with subsequent computations*
- *Nonblocking send routines* are particularly useful to enable latency hiding
- A different technique is to *map multiple processes on a single processor*. (Due to the time-sharing facilities, the processor switches form one process to another when the first is stopped because of an incomplete message passing. *Threads*, a kind of "light processes" used in shared memory model, are particularly good for an efficient switching.)

Time complexity

Generalities:

- The starting point is to estimate the *number of computation steps*; here only *arithmetical* and *logical operations* are considered, ignoring other aspects such as computation tests, etc.
- The number of computation steps often depends on the *number of data items* handled by the algorithm.

We use various notations for the *order of magnitude* of a function. (They are listed below.)

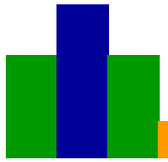
..Time complexity

- The O-notation: We says $f(x) = O(g(x))$ iff there exist positive constants c_2 and x_0 such that

$$0 \leq f(x) \leq c_2 g(x) \quad \text{for all } x \geq x_0$$

(Read as: for large values the growth of f is *"at most as"* the growth of g)

- Example: if $f(x) = 4x^2 + 2x + 12$, then $f(x) = O(x^2)$ (reason: for $x \geq 3$, $0 < 4x^2 + 2x + 12 \leq 6x^2$).
  Notice: x^{2003} is also good, i.e., $f(x) = O(x^{2003})$; hence, to be useful, always take the least growing function for g.



..Time complexity

- The Θ-notation: We says $f(x) = \Theta(g(x))$ iff there exist positive constants c_1, c_2, and x_0 such that

$$0 \leq c_1 g(x) \leq f(x) \leq c_2 g(x) \quad \text{for all } x \geq x_0$$

(Read as: for large values the growth of f is *"same as"* that of g. Clearly $f(x) = \Theta(g(x))$ implies $f(x) = O(g(x))$, but the converse is not true.)

- The Ω-notation: We says $f(x) = \Omega(g(x))$ iff there exist positive constants c_1 and x_0 such that

$$0 \leq c_1 g(x) \leq f(x) \quad \text{for all } x \geq x_0$$

(Read as: for large values the growth of f is *"at least as"* that of g.)

Clearly, Θ *is equivalent to* O & Ω.

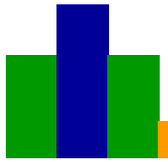
Time complexity of parallel algorithms

Time complexity analysis hides the lower terms, giving an estimation of the shape of time complexity function.

Example:

- Suppose we add n numbers on two computers using the following algorithm:

  1. Computer 1 sends $n/2$ numbers to computer 2.
  2. Both computers add $n/2$ numbers simultaneously.
  3. Computer 2 sends its partial sum back to computer 1.
  4. Computer 1 adds the partial sums to produce the final result.



..Time complexity of parallel algorithms

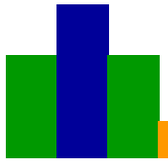
- Computation time (steps 2,4):

$$t_{comp} = n/2 + 1$$

- Communication time (steps 1,3):

$$\begin{aligned} t_{comm} &= (t_{startup} + n/2\, t_{data}) + (t_{startup} + t_{data}) \\ &= 2t_{startup} + (n/2+1)t_{data} \end{aligned}$$

- Both computational and communication complexities are $O(n)$, hence the overall time complexity is $O(n)$.



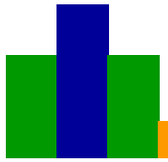
Cost-optimal algorithms

A *cost-optimal* (or *processor-time optimal*) algorithm is one such that

(parallel time complexity) $\times$ (number of processors)
= sequential time complexity

Example:

- Suppose the best know algorithm for problem P has time complexity $O(n \log n)$
- A parallel algorithm solving the same problem using n processes and having the time complexity $O(\log n)$ is cost-optimal, while a parallel algorithm which uses n^2 processes and has time complexity $O(1)$ is not cost-optimal.

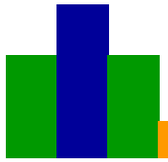


A case study: Broadcasting

Broadcast on a hypercube network: Consider a three-dimensional hypercube. To broadcast from node 000 to every other node an efficient algorithm is:

- 1st step: 000 → 001
- 2nd step: 000 → 010
  001 → 011
- 3rd step: 000 → 100
  001 → 101
  010 → 110
  011 → 111

The time complexity is $O(\log n)$ for an n-dimensional hypercube, which is optimal because the diameter is $\log n$.



..Broadcasting

Gather on a hypercube network: The *reverse algorithm* can be used to gather data from all nodes to a node, say 000 (in the 3-dim hypercube case):

- 1st step: $100 \rightarrow 000$
  $101 \rightarrow 001$
  $110 \rightarrow 010$
  $111 \rightarrow 011$
- 2nd step: $010 \rightarrow 000$
  $011 \rightarrow 001$
- 3rd step: $001 \rightarrow 000$

In such a case the messages become *longer* as the data are gathered, hence the time complexity is increased over $\log n$.

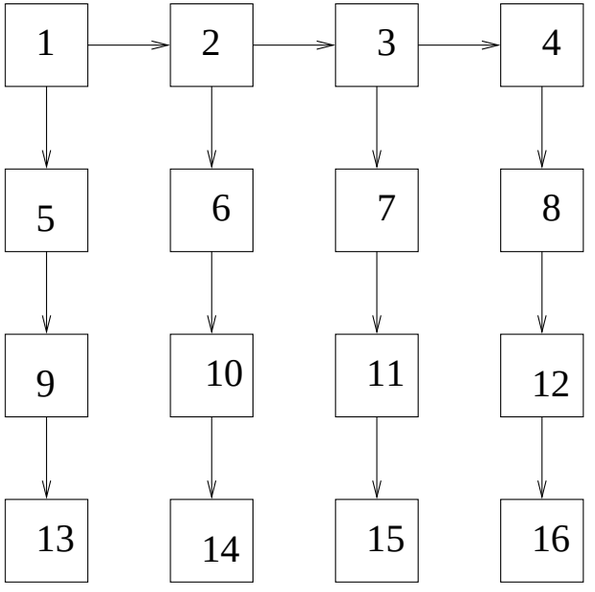
..Broadcasting

Broadcast on a mash network:

- Nodes of top line: firstly send to left (if any), then down.
- Other nodes: send down.

Denoting the nodes in order from left-to-right and top-to-down, the broadcasting in a 4×4 mash is:

```
step-1:1->2
step-2:1->5;2->3
step-3:5->9;2->6;3->4
step-4:9->13;6->10;3->7;4->8
step-5:10->14;7->11;8->12
step-6:11->15;12->16
```

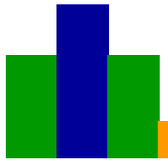


The algorithm is optimal as the number of steps is equal to the diameter of the mesh.

..Broadcasting

Broadcast on a workstation cluster:

- Broadcasting on a *single Ethernet connection* can be done using a single message that is read by all the destinations on the network *simultaneously*, hence time complexity is $O(1)$.

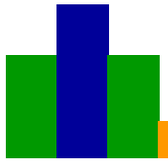


Evaluating programs empirically

Measuring execution time: To measure the execution time between point $L1$ and $L2$ in the code, one may use the following construction

```
L1:  time(&t1);                    /* start timer */
⋮
L2:  time(&t2);                    /* stop timer */
elapsedTime = difftime(t2,t1);   /* elapsedTime = t2-t1 */
pringf(``Elapsed time = %5.2f seconds'',elapsedTime);
```

Notice: MPI provides the routine `MPI_Wtime` for returning time (in seconds). Each processor may have its own clock, hence be careful not to mix such timers.



..Evaluating programs empirically

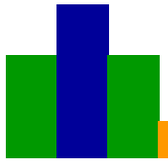
Communication time by Ping-Pong methods: To empirically estimate the communication time from a process P_1 to a process P_2 one may use the following method: *Immediately* after receiving the message P_2 *send the message back* to P_1.

P_1

```
L1:  time(&t1);
send(&x,P2);
recv(&x,P2)
L2:  time(&t2);
elapsedTime = 0.5 * difftime(t2,t1);
printf(``Elapsed time = %5.2f seconds'',elapsedTime);
```

P_2

```
recv(&x,P1)
send(&x,P1);
```

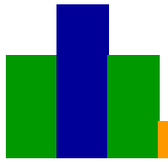


Debugging strategies

A useful *three-step approach* to debugging message-passing programs is:

1. If possible, run the program as a single process and debug as a normal *sequential program*.
2. Execute the program using *two to four multi-tasked processes* on a *single computer*. Now examine actions such as checking that messages are indeed being sent to correct places. It is very common to make mistakes with message tags and have messages sent to wrong places.
3. Execute the program using *two to four processes* but now across *several computers*. This step helps to find the impact of network delays on synchronization and timing constraints of your program.

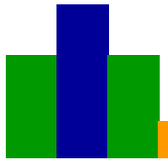
Lesson 3b: Embarrassingly parallel computations



b) Embarrassingly parallel computations

Embarrassingly parallel computations:

- A (truly) *embarrassingly parallel computation* is a computation that can be divided into a number of completely *independent parts*.
- More common is a *nearly embarrassingly parallel computation* where there is a *loose communication* between processes: a "master" process distributes initial data and collects the results; the set of "slave" processes is (truly) embarrassingly parallel.



Example 1: Image processing

A few (low level) image operations

- *Shifting*: Objects are shifted by Δx in x-dimension and by Δy in y-dimension

$$x' = x + \Delta x$$
$$y' = y + \Delta y$$

- *Scaling*: Objects are scaled by factor S_x in x-dimension and by S_y in y-dimension

$$x' = x\, S_x$$
$$y' = y\, S_y$$

- *Rotation*: Objects are rotated through an angle θ about the origin of the coordinate system:

$$x' = x\, cos\, \theta + y\, sin\, \theta$$
$$y' = -x\, sin\, \theta + y\, cos\, \theta$$

..Image processing

Algorithm

- Divide the area into regions for individual processes (square or row regions may be used).
- Each process independently performs the transformation for the pixels of its own region.

The (pseudo)code for shifting transformation (using row partition) is given below: the image area in 480×640; each slave process is supposed to handle 10 lines.

..Image processing

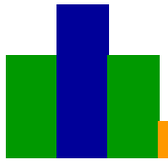
Master code

```
/* send starting row number to processes */
for (i=0,row=0; i<48; i++,row=row+10)
      send(row,Pi);

/* initialize tmp */
for (i=0; i<480; i++)
      for (j=0; j<640; j++)
            tmpMap[i][j] = 0;

/* accept new coords for each pixel */
for (i=0; i < (480*640); i++){
      recv(oldRow,oldCol,newRow,newCol,Pany);
      if !((newRow<0)||(newRow>=480)||(newCol<0)||(newCol>=640))
            tmpMap[newRow][newCol] = map[oldRow][oldCol];
}

/* update bitmap */
for (i=0; i<480; i++)
      for (j=0; j<640; j++)
            map[i][j] = tmpMap[i][j];
```

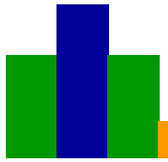


..Image processing

Slave code

```
/* receive starting row number */
recv(row,P_master);

/* for each pixel compute new coords and send to master */
for (oldRow=row; oldRow<(row+10);oldRow++)
      for (oldCol=0; oldCol<640;oldCol++){
            newRow = oldRow + deltaX
            newCol = oldCol + deltaY
            send(oldRow,oldCol,newRow,newCol,P_master);
}
```



..Image processing

Analysis (the time to compute new coordinates is fixed and small):

- Sequential: $t_s = n^2 = O(n^2)$
- Parallel ($p+1$ processes):
  —Communication: In general $t_{comm} = t_{startup} + m\,t_{data}$, hence for p processes
  $t_{comm} = p(t_{startup} + t_{data}) + n^2(t_{startup} + 4\,t_{data})$
  $= (n^2 + p)t_{startup} + (4\,n^2 + p)t_{data} = O(p + n^2)$
  —Computation: (2 additions; each process handle n^2/p pixels)
  $t_{comp} = 2\,\frac{n^2}{p} = O(n^2/p)$
  —Overall execution time: $t_p = t_{comm} + t_{comp}$

Conclusion: *Perform badly*, as the *communication part far exceeds the computation part* and the overall parallel time is significantly grater than the sequential time.

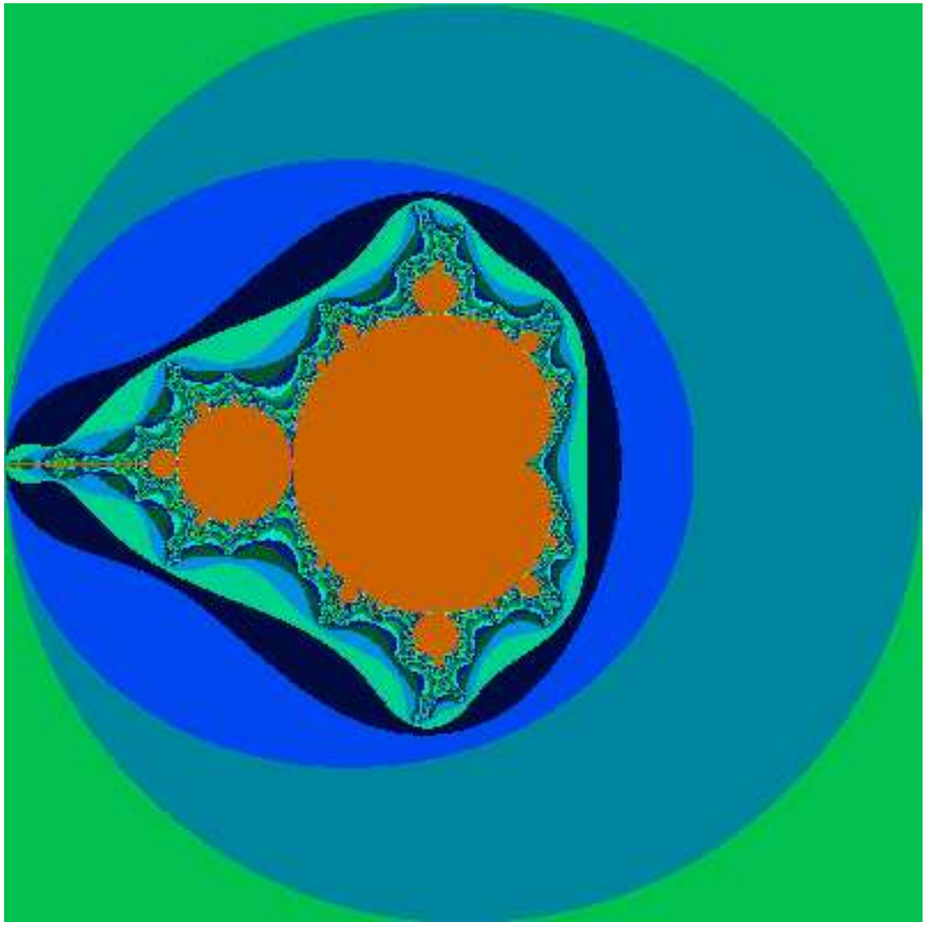
Example 2: Mandelbrot set

Mandelbrot set: Use the transformation

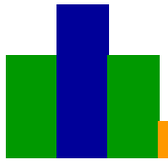
$$z \mapsto z^2 + c$$

on complex numbers. For given c iterate this starting with 0 till

—the module of the number is grater than 2 or
—the number of iterations reaches some arbitrary limit.



Notice: This is an example of an intensive computation for each pixel. The colors describe the numbers of steps necessary to get a module grater than 2.

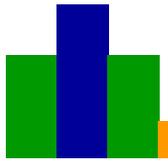


..Mandelbrot set

A sequential program

```
structure complex{
      float real;
      float imag;
}

int calcPixel(complex c){
      int count, max;
      complex z;
      float tmp, lengthSq;
      max = 256;
      z.real = 0; z.imag = 0;
      count = 0;
      do{
            tmp = z.real * z.real - r.imag * z.imag + c.real;
            z.imag = 2 * z.real * z.imag - c.imag;
            z.real = tmp;
            lengthSq = z.real * z.real + z.imag * z.imag;
            count++;
      } while (lengthSq < 4.0) && (count < max));
      return count;
}
```



..Mandelbrot set

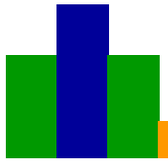
Parallel version (Static task assignment):

Master code

```
for (i=0,row=0; i<48; i++,row=row+10)
     send(row,Pi);
for (i=0; i<(480*640); i++){
     recv(c,color,Pany);
     display(c,color);
}
```

Slave code (scaling factors are used to fit the display, i.e., `scaleReal = (realMax-realMin)/dispWidth` and similarly for `scaleImg`)

```
recv(row,Pmaster);
for (x=0; x<dispWidth, x++)
     for (y=row; y<(row+10), y++){
          c.real = minReal + ((float)x * scaleReal);
          c.imag = minImag + ((float)y * scaleImag);
          color = calcPixel(c);
          send(c,color,Pmaster);
     }
```



..Mandelbrot set

Rough analysis (a more precise analysis is complicate as the number of iterations per pixel is difficult to estimate):

- Sequential: $t_s \leq max \times n = O(n)$
- Parallel ($p+1$ processes):

  —Comm-1: send row number to each slave

  $t_{comm1} = p(t_{startup} + t_{data})$

  —Computation: slaves perform their computation in parallel

  $t_{comp} \leq \frac{max \times n}{p}$

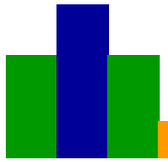
  —Comm-2: results are passed to master using individual sends

  $t_{comm2} = \frac{n}{p}(t_{startup} + t_{data})$

  —Overall execution time:

  $t_p \leq \frac{max \times n}{p} + (\frac{n}{p} + p)(t_{startup} + t_{data})$

Conclusion: When *max is large* the first factor is dominating and speedup may get closed to $p-1$.



..Mandelbrot set

A few improvements may be used to increase the efficiency of the parallel program:

- The startup time is generally long, hence a better approach for slaves may be to wait till all the pixels in a row are computed and then to send the full row to the master

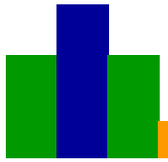
- The time to compute may differ from slave to slave, hence a better approach is to use a *dynamic task assignment*:
  —Allocate the rows one by one to slaves;
  —When a slave is ready and return the results a new task is send to him for processing.

..Mandelbrot set

Parallel version (Dynamic task assignment):

Master code

```
count = 0;
row = 0;
for (k=0; k < procNo; k++){
     send(row,Pk,dataTag);
     count++;
     row++;
}
do{
     recv(slave,r,color,Pany,resultTag);
     count--;
     if(row<dispHeight){
          send(row,Pslave,dataTag);
          count++;
          row++;
     } else
          send(row,Pslave,terminatorTag);
     rowRecv++;
     display(r,color);
} while(count>0);
```



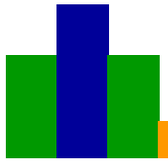
..Mandelbrot set

Slave code

```
recv(y,Pmaster,sourceTag);
while(sourceTag == dataTag){
      c.imag = imagMin + ((float) y * scaleImag);
      for (x=0; x < dispWidth; x++){
            c.real = realMin + ((float) x * scaleReal);
            color[x] = calcPixel(c);
      }
      send(i,y,color,Pmaster,resultTag);
      recv(y,Pmaster,sourceTag);
}
```

(Variable `count` *is used to count the number of busy slaves.)*

Notice: For this type of algorithms an empirical estimation of the overall execution time is for sure a better suited method.



Example 3: Monte-Carlo methods

Monte-Carlo methods are *approximative* methods based on random selections in computation.

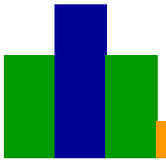
Examples:

- *Computing π:* Take a circle of unit radius included into a 2×2 square. Then one gets

$$\frac{\text{Area of circle}}{\text{Area of square}} = \frac{\pi \times 1^2}{2 \times 2} = \frac{\pi}{4}$$

  Then, choose randomly points within the square and score how may points happen to lie within the circle.

Notice: May be used for arbitrary integrals, but it's not too efficient.



..Monte-Carlo methods

- *Computing an Integral:* An integral

$$\int_a^b f(x)dx$$

is approximatively computed taking (uniformly distributed) randomly generated values between a and b, say, $x_1, \ldots, x_n$ and computing

$$\frac{b-a}{n}\sum_{i=1}^{n} f(x_i)$$

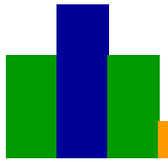
..Monte-Carlo methods

Pseudo-random number generation:

- Monte-Carlo methods are based on a procedure for generating pseudo-random numbers.
- A common method is to use the function

$$x_{i+1} = (ax_i + c) \bmod m$$

  where a, c, m are well-chosen constants
- A good selection is with $m = 2^{31} - 1$ (prime number), $a = 16807$ and $c = 0$. (For any non-zero a, a sequence including all $2^{31} - 2$ different numbers is created before a repetition occurs.)
- A disadvantage is that these generators are relatively slow.



..Monte-Carlo methods

Parallel random number generation:

- One can see that

$$x_{i+1} = (ax_i + c) \bmod m$$

gives

$$x_{i+k} = (Ax_i + C) \bmod m$$

where

$A = a^k \bmod m$ and
$C = c(a^{k-1} + \ldots + a^1 + a^0) \bmod m$

- Using such a recurrence relation one may do the generation in parallel:
—the first k numbers are generated sequentially
—then, one generates in parallel appropriate subsequences: $1, k+1, 2k+1, \ldots$ and $2, k+2, 2k+2, \ldots$ and so on.