A few parallel algorithms with communication

A simplified LogP model

- Let us assume now that:
  - A parallel program is run by p distinct and equal processors, each one with its own private memory;
  - The time to communicate n Bytes between 2 processors is modeled as:

    $T_{com}(n) = L + n/g$

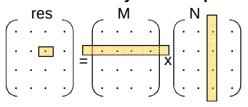
    1. L is a latency (in sec.),
    2. 1/g (in B/sec) is the throughput (g is the “gap” between the transmission of 2 Bytes).
- This model is homegeneous, static and symetric.
  - All the processors are supposed to be equal,
  - Their number does not change during the computation,
  - A communication does not privelegiate the sender or the receiver.

Granularity and Distribution

- Having a notion of “remote memory” vs. “local memory” enables to define two notions:
  1. The granularity of a parallel program is the ratio “number of instructions” / “volume of communication”;
     - » Or better, the ratio “CPU time” / “Communication time”
     - » A program is called “fine-grained” or “coarse-grained” depending of its granularity.
  2. The way the data have to be distributed between the processors, in order to minimize the communication.
     - » A processor can only compute with data in its local memory.
- Notice that in practive you may have overlap between computation and communication.

Revisiting the matrix product

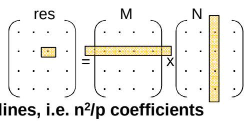
- 1st hypothesis: M and N are both copied in the memory of all the p processors.
  - It is reasonable to want to have ´res´ also copied.



  - Each processor can compute roughly n^2/p coefficients of ‘res’.
    - » since each one of these computations takes time $\theta(n)$, the total computing time is $\theta(n^3/p)$ (by processor).
  - But then, each processor must send its coefficients to all the other.
    - » By processor, this means p-1 messages of size n^2/p, i.e. $T_{com}(n) = (p-1)(L + n^2/pg)$
    - » The good point is that each message is “big” (good for latency).
  - Grain: approx. : $g.n^3 / (p-1)n^2 =\pm g.n /p$

With distributed matrices

- 2nd hypothesis: M and N are distributed, and ‘res’ should be distributed.
  - M is distributed by lines,
  - N is distributed by columns,
  - Res is distributed by lines.



- Each process must compute n/p lines, i.e. n^2/p coefficients of res.
  - This means again $\theta(n^3/p)$ op. by processor
  - The problem is that a given proc needs (p-1) x (n/p) columns that it does not own.
  - i.e., by proc, $T_{com}(n) = (1-1/p)n(L+n/g)$.
- Grain: approx. : $g.n^3 / n^2 =\pm g.n$
  - Same thing as previous, not that good for Latency, better for throughput.

With block-distributed matrices

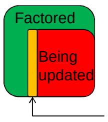
- 3rd hypothesis: M and N are distributed, and ‘res’ should be distributed.
  - M, N and res are distributed by blocks of size KxK elements.
  - $pK^2 = n^2$, i.e. $K = n / \sqrt{p}$
- Each process must compute $K^2 = n^2/p$ coefficients of res.
  - This still means $\theta(n^3/p)$ op. by processor
- Then, each processor needs to receive:
  - $(n/K-1)\ K^2$ coefs from lines = $\sqrt{p}$ x $(L+n^2/gp)$ = approx $n^2/ g\sqrt{p}$
  - $(n/K-1)\ K^2$ coefs from columns = approx $n^2/ g\sqrt{p}$
  - i.e., by proc, $T_{com}(n) = 2\ n^2/ g\sqrt{p}$.
- Grain: approx. : $g.n^3 / \sqrt{p}n^2 =\pm g.n / \sqrt{p}$
  - Good for latency, and much better than previous results.

System of linear equations

- Coming back to the LU original (non D&C) factorization...

```
for (k = 0 ; k <= n-2; k++) {
    for (i = k+1 ; i <= n-1 ; i++)
        M[i][k] = -M[i][k]  / M[k][k];
    for (j = k+1; j <= n-1; j++)
        for (i=k+1 ; i<=n-1; i++)
            M[i][j] = M[i][j] + M[i][k]*M[k][j];
}
```

- How do we distribute the computations?
  - Distribute M by columns,
  - Each processor only computes the coefficients of its own columns.
  - Let us note rank(k) the rank of the processor which owns column k.



The parallel algorithm (1)

- Writing in a SPMD way (all the processors run this same code):

```
r = my_proc_rank()
p = number_of_procs()
for (k = 0 ; k <= n-2; k++) {
    if (r == rank(k)) then
        for (i = k+1 ; i <= n-1 ; i++)
            M[i][k] = -M[i][k]  / M[k][k];
    }
    /* each processor owns only part of the M[i][k] */
    for (j = k+1; j <= n-1; j++)
        for (i=k+1 ; i<=n-1; i++)
            M[i][j] = M[i][j] + M[i][k]*M[k][j];
}
```

The parallel algorithm (1)

- Writing in a SPMD way (all the processors run this same code):

```
r = my_proc_rank()
p = number_of_procs()
for (k = 0 ; k <= n-2; k++) {
    if (r == rank(k)) then
        for (i = k+1 ; i <= n-1 ; i++)
            M[i][k] = -M[i][k]  / M[k][k];
    }
    /* each processor owns only part of the M[i][k] */
    for (j = k+1; j <= n-1; j++)
        for (i=k+1 ; i<=n-1; i++)
            M[i][j] = M[i][j] + M[i][k]*M[k][j];
}
```

Small problem here: The matrix is distributed, so j and k should not run from 0 to n-1

The parallel algorithm (2)

- Using a “local index” l (=0...n/p) for the columns:

```
r = my_proc_rank()
p = number_of_procs()
l=0
for (k = 0 ; k <= n-2; k++) {
    if (r == rank(k)) then
        for (i = k+1 ; i <= n-1 ; i++)
            M[i][l] = -M[i][l]  / M[k][l];
        l= l+1
    }
    /* each processor owns only part of the M[i][k] */
    for (j = l ; j <= n/p-l ; j++)
        for (i=k+1 ; i<=n-1; i++)
            M[i][j] = M[i][j] + M[i][k]*M[k][j];
}
```

Two more “implementation” details

- You need to broadcast the elements M[i][l] in the middle of the algorithm.
  - This means sending n^2/p coefficients to all the p-1 other processors.
  - Takes time $(p-1)(L+ n^2/gp)$.
    - » Note: this is a worst case scenario – a broadcast can (should) be better implemented.
    - » It could take something like $(L+ n^2/gp)\log(p)$.
- Probably, this broadcast needs to access contiguous elements in the local memory.
  - This means that the M[i][l] coefs. probably should be stored in column-major order (Fortran order)
  - Else (in C), you have to use an intermediate buffer.
  - This is typical of MPI + C programming.

So what is the “rank()” function?

- Rank(k) = rank of the processor that owns the column k.
- There are many options:
  - Block mapping: let B = n/p, then rank(k) = k / B
    - » / is the euclidean division.
    - » With this formula, the last processor gets a little bit more elements than the other.
    - » Very simple to implement, and maximizes locality.
  - Cyclic (round-robin) mapping: rank(k) = k % p
    - » Simple to implement, minimizes locality
    - » Good for load balancing.
  - Block cyclic: given a block size $n/p \geq B \geq 1$, rank(k) = (k / B) % p
    - » The best of two worlds.

Parallel complexity of LU

- Each processor performs (roughly):
  - $n^2/2p$ divisions in the first phase (pivot computation)
    - » Actually, they are products.
  - Broadcast: $(L + n^2/gp)\log(p)$.
  - $n^3/3p$ products

  in the update phase.

```
for (k = 0 ; k <= n-2; k++) {
      if (r == rank(k)) then
         for (i = k+1 ; i <= n-1 ; i++)
             M[i][k] = -M[i][k]  / M[k][k];
      }
     /* brodcast */
      for (j = k+1; j <= n-1; j++)
          for (i=k+1 ; i<=n-1; i++)
               M[i][j] = M[i][j] + M[i][k]*M[k][j];
   }
```

- Total runtime:
  - $(n^2/2p + n^3/3p)T. + (L + n^2/gp)\log(p)$.
  - Granularity: roughly

    $gn/3\log(p)$
- This is not that bad (compare to the matrix products).
  - But the parallel runtime is far from ideal.

Solving a System of Differential Equations

- You want to simulate:
  - The heat diffusion in a metallic bar,
  - The behavior of a fluid flow when it meets an obstacle,
  - The diffusion of polluents in a river,
  - The stock-exchange (bad example toda...)
- Then you need to solve things like:

$$\frac{\partial\rho}{\partial t} + \frac{\partial(\rho v_x)}{\partial x} + \frac{\partial(\rho v_z)}{\partial z} = 0$$

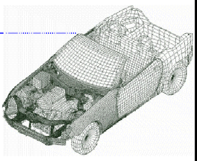
$$\rho\left(\frac{\partial v_x}{\partial t} + v_x\frac{\partial v_x}{\partial x} + v_z\frac{\partial v_x}{\partial z}\right) = -\frac{\partial p}{\partial x} - \left(\frac{\partial\tau_{xx}}{\partial x} + \frac{\partial\tau_{zx}}{\partial z}\right)$$

$$\rho\left(\frac{\partial v_z}{\partial t} + v_x\frac{\partial v_z}{\partial x} + v_z\frac{\partial v_z}{\partial z}\right) = -\frac{\partial p}{\partial z} - \left(\frac{\partial\tau_{xz}}{\partial x} + \frac{\partial\tau_{zz}}{\partial z}\right) + \rho g$$

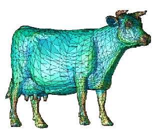
$$\frac{\partial}{\partial t}ui + \Sigma uj\frac{\partial ui}{\partial xj} = V\Delta ui - \frac{\partial p}{\partial xi} + fi(x,t)$$

(Wikipedia)

How do you do it?



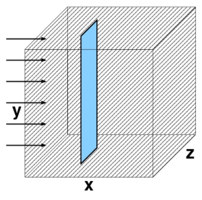
- These equations apply mathematical operators to 3D points.
  - E.g.: temperature(x,y,z).
- In order to solve them by approximation, the 3D spatial domain is discretized by a mesh.
  - The continuous operators turn to matricial operators.

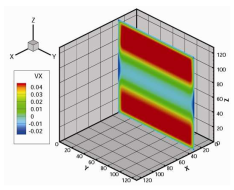


You then have to iteratively apply these operators to compute the values for each vertex
- (this means matrix-vector products)
- Hopefully this converges.

Example of output – fluid in a chanel

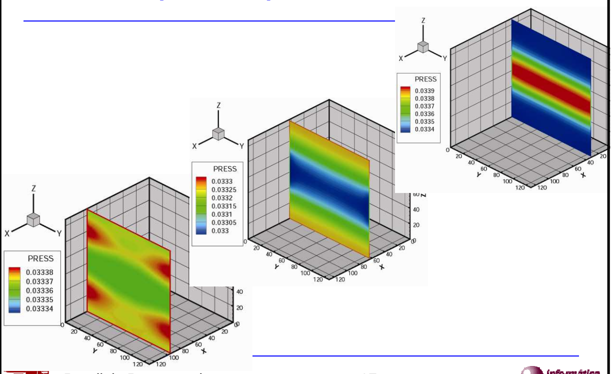
- A fluid flows in a square channel,
- There is an obstacle in the middle.
- How does it impact on the velocity/ pression of the fluid?





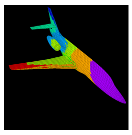
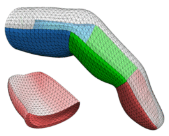
(All the study in SCHEPKE et al., *Performance Improvement of the Parallel Lattice Boltzmann Method*. SBAC´07)

Example of output – fluid in a chanel



Domain decomposition (1)

- So you end up with a mesh that you have to distribute.

- How do you distribute it?
  - If it is "regular" (structured), e.g. with rectangles or triangles, all with the same size, it is more or less like the matrices.
  - If it is unstructured (like the examples above), then it is much more difficult
    - » Graph partitioning techniques.

Domain decomposition (2)

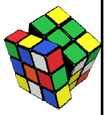
- Anyway, you end up with a distributed data-structure (usually a d-dimensional array), with:
  - N internal vertices,
  - D peripherical vertices.
- The parallel computation will consist in an iterative process. At each iteration:
  - Each processor applies its (discretized) operator on the N internal points;
  - Each processor sends to those which own the neighboor domains the values of the points that lie on the frontier.
    - » And receives from its neighbors their values.
  - Each processor updates its frontier with these new received values.
    - » Either overwrite them, either uses a mean...

Frontier

- The notion of frontier is crucial:
  - The more overlap between the frontiers, more continuous the solution will be.
    - » The convergence will be faster, less risks of diverging.
    - » There are numerical results that prove this.
  - The more overlap there is, more duplicated computation there is.
    - » Time lost.
- From the parallel point of view, the communication is directly conditionned by the size of the frontiers
  - Granularity is roughly NT. / 2(L+ D/g).
  - So you want small frontiers.
- What is best?
  - A more parallel algorithm, which performs more iterations to converge?
  - A less parallel algorithm, which performs less iterations?

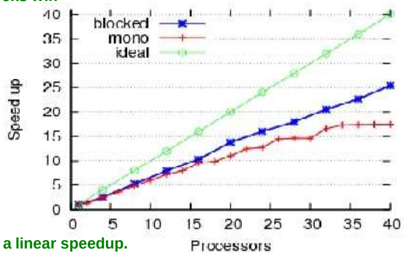
Example: a “square” case

- The domain is a cube, containing N^3 points.
- Divide it following one dimension in p slices
  - You have N^3/p points by domain.
  - D is proportional to N^2
- Divide it following 2 dimensions in p “sticks”
  - You still have N^3/p points by domain
  - D is proportional to $4 N^2 / \sqrt{p}$
- Divide it following 3 dimensions in p “small cubes”
  - You still have N^3/p points by domain
  - D is proportional to $6 N^2 / p^{2/3}$
- The 3D solution is better!
  - But you need more technical manipulations of the communication!



Performance analysis [Schepke´07]

- Back to the fluid dynamics.
  - The 3D blocks win



  - They have a linear speedup.
  - But you can see the distance from optimum.

Conclusion

- Taking communication into account leads to other preocupations:
  - Granularity,
  - Interleaving comm with computation.
- This is good, but is highly specific to each application and architecture
  - You have to measure L, g, etc.
  - You lose the “big picture”.
- You end up having to think about the implementation...
  - See the Broadcast in the LU factorization.
- Talking about implementation... This is the subject of next lecture!
  - Message Passing Interface (MPI).