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 two processors is modeled as:
    \[
    T_{\text{comm}}(n) = L + n/g
    \]
    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 homogeneous, static and symmetric.
  - All the processors are supposed to be equal,
  - Their number does not change during the computation,
  - A communication does not privilege 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 practice 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 \( n/p \) coefficients of ‘res’.
    - since each one of these computations takes time \( t(n) \), the total computing time is \( t(n/p) \) (by processor).
  - But then, each processor must send its coefficients to all the other.
    - By processor, this means \( p-1 \) messages of size \( n/p \), i.e. \( T_{\text{comm}}(n) = (p-1)g \times n^2/p^2 \).
    - The good point is that each message is “big” (good for latency).
    - Grain: approx. \( g^2 n^2 / (p-1)^2 p^2 \) \( \approx g n p \).

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 \( t(n/p) \) op. by processor
    - The problem is that a given proc needs \( (p-1) \times n/p \) columns that it does not own.
      - i.e., by proc: \( T_{\text{comm}}(m) = (p-1)n/L \times n/p \).
  - Grain: approx. \( g n^2 / n^2 = g n \)
    - 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 \( K x K \) elements.
  - \( g = K^2 \) i.e., \( K = n / \sqrt{p} \).
  - Each process must compute \( K^2 = n^2/p \) coefficients of res.
    - This still means \( t(n^2/p) \) op. by processor
    - Then, each processor needs to receive:
      - \( (n-K+1) K^2 \) coeff. from lines \( = p \times (L+K^2/g) \) \( \approx n^2 g / p \)
      - \( (n-K+1) K^2 \) coeff. from columns \( \approx n^2 g / p \)
        - i.e., by proc: \( T_{\text{comm}}(m) = 2 n^2 / g/p \).
  - Grain: approx. \( g n^2 / n^2 = g n / p \)
    - 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++)
      - \( M[i][k] = M[i][k] / M[k][k] \)
  - \( \text{if } (r == \text{rank}(k)) \)
    - 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 column,
  - 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):
  - \( r = \text{my_proc_rank}() \)
  - \( p = \text{number_of_procs}() \)
  - for (k = 0 ; k <= n-2; k++)
    - if (r == rank(k))
      - for (i = k+1 ; i <= n-1; i++)
        - \( M[i][k] = M[i][k] / M[k][k] \)
  - \( \text{if } (r == \text{rank}(k)) \)
    - for (i = k+1 ; i <= n-1; i++)
      - \( M[i][j] = M[i][j] - M[i][k] * M[k][j] \)

- Using a “local index” \( l \) (0...n/p) for the columns:
  - \( r = \text{my_proc_rank}() \)
  - \( p = \text{number_of_procs}() \)
  - for (k = 0 ; k <= n-2; k++)
    - if (r == rank(k))
      - 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 \( M[i][j] \) in the middle of the algorithm.
- This means sending n/p coefficients to all the p-1 other processors.
- Takes time (p-1)L x m/gp).
- Note: this is a worst case scenario – a broadcast can (should) be better implemented.
- It could take something like (L x n/g)(log)g).

- Probably, this broadcast needs to access contiguous elements in the local memory.
  - This means that the \( M[i][j] \) 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?

- \( \text{rank}(k) \) = rank of the processor that owns the column k.
- There are many options:
  - Block mapping: let \( B = n/p \), then \( \text{rank}(k) = k / B \)
    - \( / \) 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: \( \text{rank}(k) = k \% p \)
  - Simple to implement, minimizes locality
  - Good for load balancing.
  - Block cyclic: given a block size n/p \( \geq B \), \( \text{rank}(k) = (k / B) \% p \)
  - The best of two worlds.
Parallel complexity of LU

- Each processor performs (roughly):
  - \( n^3/2p \) divisions in the first phase (pivot computation)
  - Actually, they are products.
  - Broadcast: \((L + n^3/2p)\log(p)\)
  - \( n^3/3p \) products
- In the update phase.
- Total runtime:
  - \((n^3/2p + n^3/3p)T + (L + n^3/2p)\log(p)\)
  - Granularity: roughly \( gn/\log(p) \)
- This is not that bad (compare to the matrix products).
- But the parallel runtime is far from ideal.

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.

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 pollutants in a river,
  - The stock-exchange (bad example today...)
- Then you need to solve things like:

\[
\frac{\partial y}{\partial x} + \frac{\partial (mx)}{\partial x} - \frac{\partial (nx)}{\partial x} - \frac{\partial (ny)}{\partial y} + \frac{\partial (nz)}{\partial z} = \frac{\partial (V \Delta z)}{\partial x} - \frac{\partial (V \Delta z)}{\partial y} + f(x,y)
\]

(All the study in SCHEPKE et al., Performance improvement of the Parallel Lattice Boltzmann Method. SBAC'07)

Example of output – fluid in a channel

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

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 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 peripheral 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 neighbor 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...

Parallele Programmierung
Nicolas Malhant, Marcus Bitte

19

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.
- From the parallel point of view, the communication is directly conditioned by the size of the frontiers
  - Granularity is roughly NT / 2(N + Dg).
  - 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?

Parallele Programmierung
Nicolas Malhant, Marcus Bitte

20

Example: a “square” case

- The domain is a cube, containing N³ points.
- Divide it following one dimension in p slices
  - You have N³/p points by domain.
  - D is proportional to N³/p.
- Divide it following 2 dimensions in p “sticks”
  - You still have N³/p points by domain.
  - D is proportional to 4 N² / p.
- Divide it following 3 dimensions in p “small cubes”
  - You still have N³/p points by domain.
  - D is proportional to 6 N¹ / p³.
- The 3D solution is better!
  - But you need more technical manipulations of the communication!

Parallele Programmierung
Nicolas Malhant, Marcus Bitte

21

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.

Parallele Programmierung
Nicolas Malhant, Marcus Bitte

22

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).

Parallele Programmierung
Nicolas Malhant, Marcus Bitte

23