Advanced Topics in Numerical Analysis: High Performance Computing

MATH-GA 2012.001 & CSCI-GA 2945.001

Georg Stadler
Courant Institute, NYU
stadler@cims.nyu.edu

Spring 2017, Thursday, 5:10–7:00PM, WWH #512

March 23, 2017

Outline

Organization issues

Sources of parallelism and locality

MPI Intro

Git revisited

Organization issues

- Final projects! Pitch your final project. I am available Friday (tomorrow) 1:30-2:30pm and Monday 1-2pm if you want to discuss your plans.
- I posted suggestions for final projects last weekend, and added more this morning (see Piazza).
- Final project presentations (max 10min each) in the week May 8–12.
- Short homework assignment posted by tomorrow.

Outline

Organization issues

Sources of parallelism and locality

MPI Intro

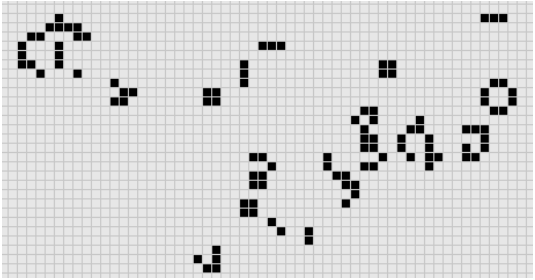
Git revisited

Parallelism and locality

- Moving data (through network or memory hierarchy) is slow
- Real world problems often have parallelism and locality, e.g.,
  - objects move independently from each other ("embarrassingly parallel")
  - objects mostly influence other objects nearby
  - dependence on distant objects can be simplified
  - Partial differential equations have locality properties
- Applications often exhibit parallelism at multiple levels

Example I: Conway's game of life

https://www.youtube.com/watch?v=C2vgICfQawE



- Played on a board of "cells"; simple rules decide on if a cell is alive or dead in the next generation
- Is an example of a cellular automaton
- Amounts to checking the 8 neighbor cells in every generation
- How to parallelize? Decompose domain into subdomains...

Example II: Particle systems

A particle system has a finite number of particles which move according to Newton's law ($F = ma$); particles can be stars subject to gravity, atoms in a molecule, swimming fish, . . .
Force on each particle:

$$F_{\text{overall}} = F_{\text{external}} + F_{\text{nearby}} + F_{\text{far}}$$

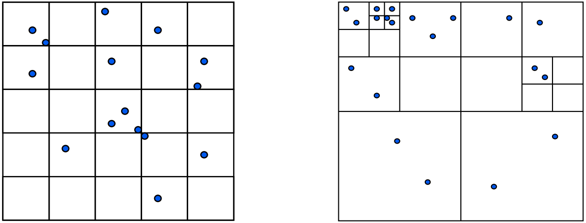
- external: background flow/ocean current/external electric field
- nearby attraction; collision force, Van der Waals forces
- far field: gravity, electrostatics

Example II: External and nearby forces

External force: independent, “embarrassingly parallel”: evenly distribute particles amongst processors.

Nearby force: requires neighbor communication; assume, for instance collisions; need to check in “ghost layer” for particles on neighboring processes

- interaction of particles near processor boundary
- load imbalance if particles cluster; must be adjusted



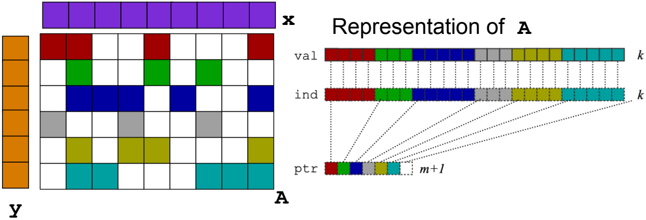
Example II: Far field forces

Far field forces involve all-to-all communication
Simple algorithm: $\mathcal{O}(n^2)$, where n is the number of particles.
More clever algorithms:

- Particle-mesh methods: interpolate particle force to nearest grid point; solve far field PDE (e.g., FFT); interpolate force back to particles
- Use tree construction; each node contains an approximation of descendants: Fast multipole method (FMM)

Example III: Sparse matrix-vector multiplication

Compressed sparse row (CSR) format:



Matrix-vector multiply kernel: **$y_{(i)} \leftarrow y_{(i)} + A_{(i,j)} \cdot x_{(j)}$**

Matrix multiplication kernel: $\boldsymbol{y} = \boldsymbol{y} + A\boldsymbol{x}$:

```
for each row $i$
    for $k = \text{ptr}[i]$ to $\text{ptr}[i+1] - 1$ do
        $\boldsymbol{y}[i] = A_{\text{val}}[k]\boldsymbol{x}[\text{ind}[k]]$
```

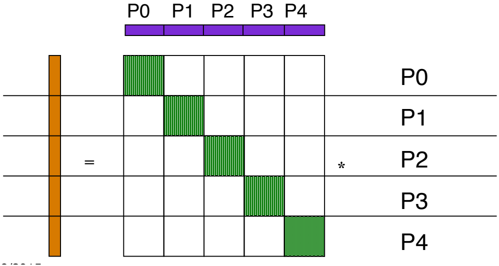
Example III: Sparse matrix-vector multiplication

How parallelize? Which processes compute/store which part of A, $\boldsymbol{x}$, $\boldsymbol{y}$?

Partition into index sets, and distribute to different processes. Requires communication if $\boldsymbol{x}$ is distributed as well.

Example III: Sparse matrix-vector multiplication

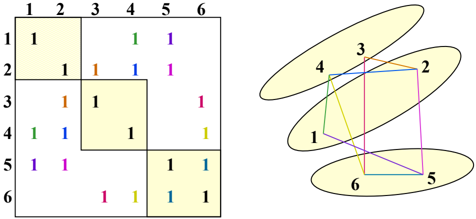
How parallelize? Which processes compute/store which part of A, $\boldsymbol{x}$, $\boldsymbol{y}$?



Communication can be reduced with proper ordering of rows/columns of A.

Example III: Sparse matrix-vector multiplication

How parallelize? Which processes compute/store which part of A, $\boldsymbol{x}$, $\boldsymbol{y}$?



$$\begin{array}{c|cccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 1 & & & 1 & 1 & \\
2 & & 1 & 1 & 1 & 1 & \\
3 & & 1 & 1 & & & 1 \\
4 & 1 & 1 & & 1 & & 1 \\
5 & 1 & 1 & & & 1 & 1 \\
6 & & & 1 & 1 & 1 & 1
\end{array}$$

Reordering and Graph Partitioning: Edges in graph correspond to nonzeros in matrix. Graph partitioning ↔ minimizing communication in parallel matrix-vector multiplication.

Example IV: Partial differential equations

Types of PDEs influence parallelism

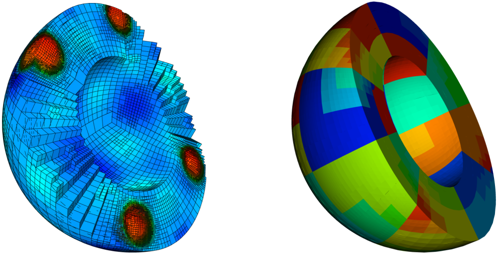
- Elliptic PDE (gravitation,elasticity,. . .): Steady-state, global dependence in space
- Hyperbolic PDE: (acoustic/electromagnetic waves,. . .): Time-dependent, local dependence in space
- Parabolic PDE (heat flow, diffusion,. . .): Time-dependent, global space dependence

Many PDEs (e.g., Navier-Stokes equation) combine properties of these basic types.

Example IV: Partial differential equations: elliptic

$$\begin{aligned} -\Delta u &= f \text{ on } \Omega \\ &+ \text{bdry cond.} \end{aligned}$$

After discretization, this is becomes a system with a positive definite, symmetric matrix. Efficient solvers include geometric or algebraic multigrid or FFT (requires proper boundary conditions and mesh). Parallel Gauss elimination allow limited parallelism.



Field governing mesh refinement (left). Mesh partitioning, each color illustrates mesh portion owned by a different processor (right).

Example IV: Partial differential equations: hyperbolic

$$\begin{aligned} &u_{tt} - \Delta u = f \text{ on } \Omega \\ &+ \text{bdry cond.} \\ &+ \text{initial cond.} \end{aligned}$$

Often, method of choice is explicit time stepping, which requires a matrix-vector multiplication in each time step:

$$\boldsymbol{u}^{k+1} = \boldsymbol{u}^k + \delta t A \boldsymbol{u}^k$$

Explicit time stepping is commonly used. CFL stability does not restrict the size of the time step δt significantly.
Parallelization based on decomposition of mesh (leads to a similar decomposition as for sparse matrices).

Example IV: Partial differential equations: parabolic

$$u_t - \Delta u = f \text{ on } \Omega$$
$$+ \text{ bdry cond.}$$
$$+ \text{ initial cond.}$$

Stability is a problem for explicit time stepping (requires very small time step!). Thus, one usually uses implicit time stepping:

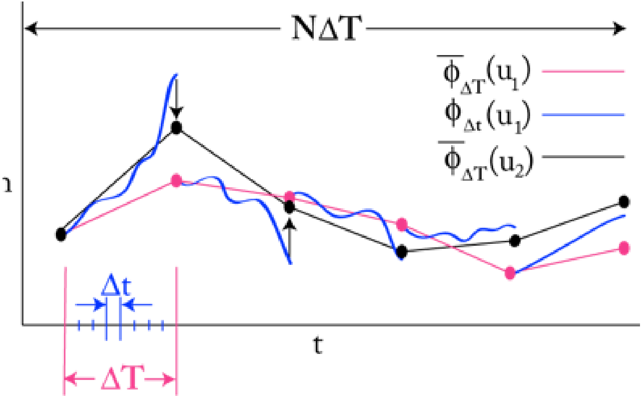
$$\boldsymbol{u}^{k+1} = \boldsymbol{u}^k + \delta t A \boldsymbol{u}^{k+1},$$

which requires to solve systems in every time step. These are similar as in the case of an elliptic PDE (solvers: multigrid, FMM,. . .) Parallelization based on decomposition of mesh.

Example IV: Partial differential equations: parallel-in-time

Parallelization-in-time is an active field of research. Basic idea:

- Use a fast and inaccurate serial time integration method $\bar{\Phi}$ as starting guess
- Iterative local-in-time parallel correction with more accurate time integration Φ



Outline

Organization issues

Sources of parallelism and locality

MPI Intro

Git revisited

Introduction to MPI

Use B. Gropp's PPT slides

https://github.com/NYU-HPC15/lecture7

Non-blocking MPI Send/Recv

- Non-blocking communication allows interlacing communication and computation.

  `MPI_ISend(..., MPI_Request *request)`

  `MPI_IRecv(..., MPI_Request *request))`

- Must check status to ensure that communication has finished.

  `MPI_Wait(MPI_Request *request, MPI_Status *status)`

Comparison with mailing a letter:

- Blocking Send: drop off letter at the mail box (copied to MPI buffer)
- Nonblocking Send: letter on kitchen table is ready to be taken to the mail box (MPI starts taking care of message)
- Blocking Recv: Letter has arrived (it's in the desired memory location)
- Nonblocking Recv: I'm expecting a letter (keep checking till it arrives using `MPI_Wait()`)

Outline

Organization issues

Sources of parallelism and locality

MPI Intro

Git revisited

Version control

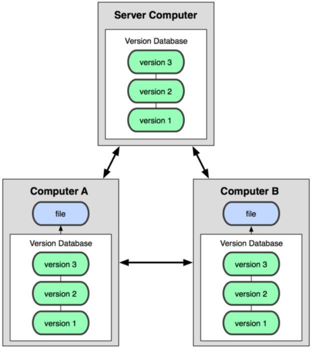
A Version Control System (VCS) is an integrated fool-proof framework for

- Backup and Restore
- Short and long-term undo
- Tracking changes
- Synchronization
- Collaborating
- Sandboxing

... with minimal overhead.

Git–a distributed Version Control Systems

Distributed VCSs keep a complete copy of database in every working directory.



Git Basics - Working with remotes

In Git **all remotes are equal**.

A *remote* in Git is nothing more than a link to another git directory.

In particular: There is nothing special about github, bitbucket and co—they only give you some storage space and the graphical interface.

Git Basics - Working with remotes

The easiest commands to get started working with a remote are

- *clone*: Cloning a remote will make a complete local copy.
- *pull*: Getting changes from a remote.
- *push*: Sending changes to a remote.

Remote repositories

Initialize repository

```
$ git init (--bare)
```

Add remote repository

```
$ git remote add origin https://github.com/...
```

What should (not) be added to a repository?

Git tracks diff-files to keep its memory requirements small. Main rule: mostly add *source files that compile*.

- .c, .cpp, .f files

What should (not) be added to a repository?

Git tracks diff-files to keep its memory requirements small. Main rule: mostly add *source files that compile*.

- .c, .cpp, .f files YES!
- .tex files

What should (not) be added to a repository?

Git tracks diff-files to keep its memory requirements small. Main rule: mostly add *source files that compile*.

- .c, .cpp, .f files YES!
- .tex files YES!
- .aux, .out, .dvi. . . files

What should (not) be added to a repository?

Git tracks diff-files to keep its memory requirements small. Main rule: mostly add *source files that compile*.

- .c, .cpp, .f files YES!
- .tex files YES!
- .aux, .out, .dvi. . . files NO!
- compiled files, object files

What should (not) be added to a repository?

Git tracks diff-files to keep its memory requirements small. Main rule: mostly add *source files that compile*.

- .c, .cpp, .f files YES!
- .tex files YES!
- .aux, .out, .dvi. . . files NO!
- compiled files, object files NO! (large, no diffs possible, conflicts)
- .pdf files

What should (not) be added to a repository?

Git tracks diff-files to keep its memory requirements small. Main rule: mostly add *source files that compile*.

- .c, .cpp, .f files YES!
- .tex files YES!
- .aux, .out, .dvi. . . files NO!
- compiled files, object files NO! (large, no diffs possible, conflicts)
- .pdf files YES/NO!
- large data files

What should (not) be added to a repository?

Git tracks diff-files to keep its memory requirements small. Main rule: mostly add *source files that compile*.

- .c, .cpp, .f files YES!
- .tex files YES!
- .aux, .out, .dvi. . . files NO!
- compiled files, object files NO! (large, no diffs possible, conflicts)
- .pdf files YES/NO!
- large data files NO. . . sometimes maybe
- photos, movies etc.

What should (not) be added to a repository?

Git tracks diff-files to keep its memory requirements small. Main rule: mostly add *source files that compile*.

- .c, .cpp, .f files YES!
- .tex files YES!
- .aux, .out, .dvi... files NO!
- compiled files, object files NO! (large, no diffs possible, conflicts)
- .pdf files YES/NO!
- large data files NO... sometimes maybe
- photos, movies etc. NO! (unless unavoidable)

My rule of thumb: Files in the repository are permanent, only the best should make it in there (it's not your trash can!) They should compile (code/Latex), be (more or less) cleaned up, unless it's avoidable only source/text files.

Some of my git wisedom

Should I have a few large repositories or many small ones?

- I recommend many small ones (like I use for this class).
- Easier to manage, commit messages easier to monitor.
- Small memory footprint and faster!
- It's easy to link two repositories (e.g., code libraries) using git submodules (look it up)!

Some of my git wisedom

Should I have a few large repositories or many small ones?

- I recommend many small ones (like I use for this class).
- Easier to manage, commit messages easier to monitor.
- Small memory footprint and faster!
- It's easy to link two repositories (e.g., code libraries) using git submodules (look it up)!

How often should you commit?

- As often as you like (in case of doubt, more often)
- Makes it easier to monitor changes, track down bugs
- If you collaborate, better to avoid conflicts
- For me: feels like a (small) achievement, supports clean/systematic working style (always look at diff before committing)

Some of my git wisedom

Should I have a few large repositories or many small ones?

- I recommend many small ones (like I use for this class).
- Easier to manage, commit messages easier to monitor.
- Small memory footprint and faster!
- It's easy to link two repositories (e.g., code libraries) using git submodules (look it up)!

How often should you commit?

- As often as you like (in case of doubt, more often)
- Makes it easier to monitor changes, track down bugs
- If you collaborate, better to avoid conflicts
- For me: feels like a (small) achievement, supports clean/systematic working style (always look at diff before committing)

. . . any others??