

HIGH-PERFORMANCE SCIENTIFIC COMPUTING (HPSC) ME522

Instructor: Gaurav Bhutani

SMME, IIT Mandi

Feb-Jun 2023 semester

Venue: to be announced

Online: <https://meet.google.com/xyn-osvu-yys>

Lecture plan

- Course essentials
- Goals and learning outcomes
- Pre-requisites and software requirements
- Short demonstration

Course essentials

Notes

- Class: 4 hours per week (2 sessions).
- Venue to be announced.
- One hr lecture, followed by 1 hour lab / hands-on session. Use your own laptop.
- Teaching supporter: Ayush Sahu (SMME Metch
Help with labs, installing software, obtaining material, announcements)
- Announcements: Course mailing list / Google chat group
- Learn from peers
- Moodle – all course slides and link to material
- Codes on Github as we create
- Virtual machines on IIT Mandi Cloud

Evaluation

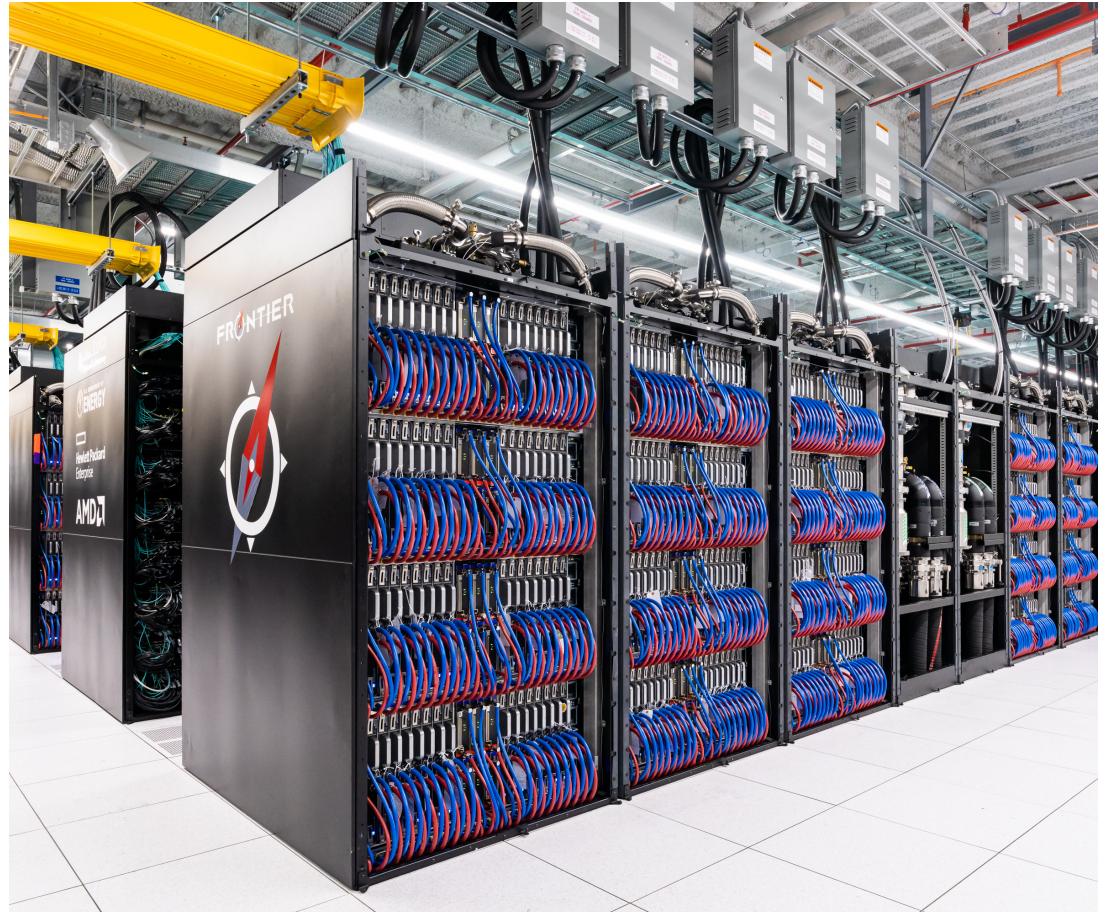
- Lab/class attendance and random lab viva (20)
- Midsem (30)
 - Written (10)
 - Lab exam (20)
- Endsem (50)
 - Written (20)
 - Lab exam (30)

Goals

HPC

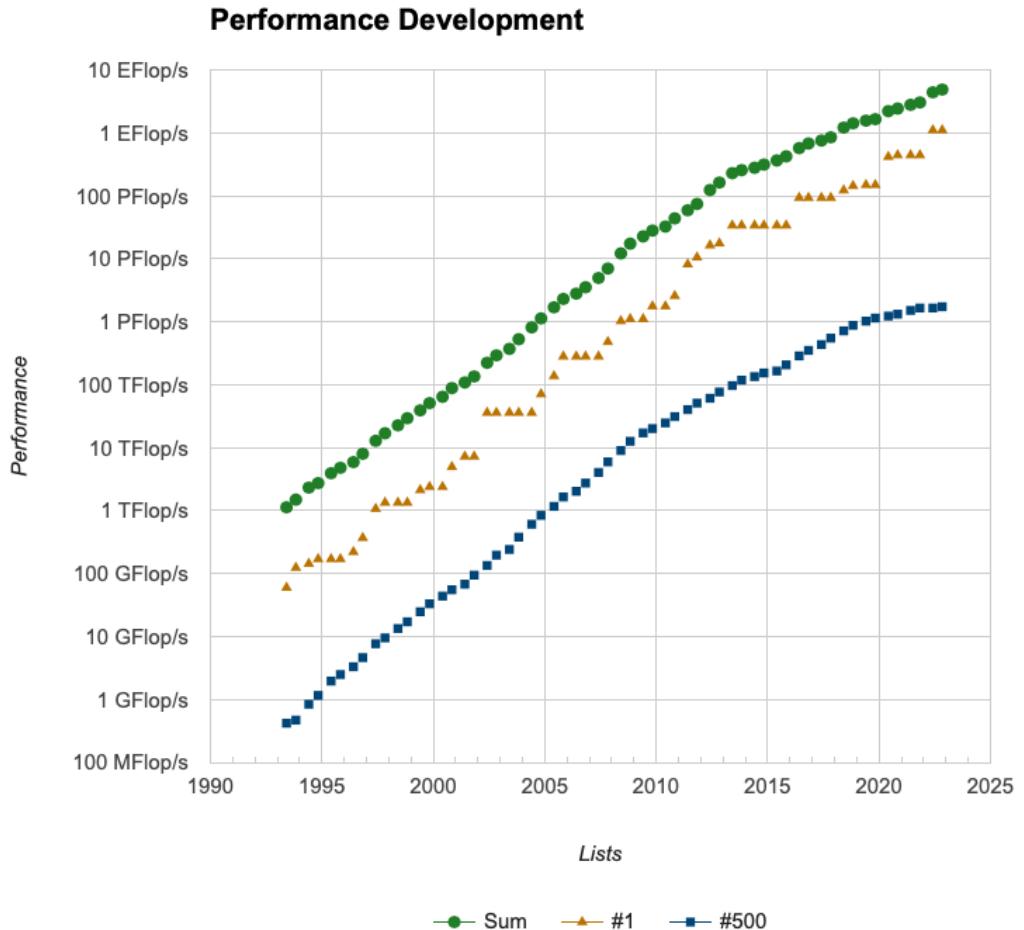
- **High Performance Computing (HPC)** often means heavy-duty computing on clusters or supercomputers with 100s of thousands of cores.
- “*World’s fastest computer*”
- #1. Frontier – HPE Cray (Oak Ridge National Lab, USA)
8.7M cores \approx 1100 Petaflops; 21MW power
- #3. Leonardo – Atos (CINECA, Italy)
1.4M cores \approx 174 Petaflops; 5.6MW
- Param Himalaya – Atos (IIT Mandi, India)
800 Teraflops; 150kW

See <http://top500.org> for current list.



Increasing speed

- Moore's Law: Processor speed doubles every 18 months.
⇒ factor of 1024 in 15 years.
- Going forward: Number of cores doubles every 18 months
- Top: Total computing power of top 500 computers
- Middle: #1 computer
- Bottom: #500 computer



<http://www.top500.org>

Our focus

- Our focus is more modest, but we will cover material that is:
 - Essential to know if you eventually want to work on supercomputers
 - Extremely useful for any scientific computing project, even on a laptop.
- Focus on scientific computing as opposed to other computationally demanding domains, for which somewhat different tools might be best.

Learning outcomes

Efficient processing

- Basic computer architecture, e.g. floating point arithmetic, cache hierarchies
- Using Unix (or Linux)
- Language issues, e.g. compiled vs. interpreted, object oriented, etc.
- Specific languages: Python (for scripting), Fortran 90/95 (for fast processing)
- Parallel computing with OpenMP, MPI
- Not included: GPU-based parallelisation

Good software practices

- Version control using Git, github
- Makefiles
- Debuggers, code testing
- Reproducability
- Use of high-performance computing (HPC) clusters

Strategy

- So much material, so little time...
- Concentrate on basics, simple motivating examples.
- Get enough hands-on experience to be comfortable experimenting further and learning much more on your own.
- Learn what's out there to help select what's best for your needs. New languages are introduced with time – similar ideas though.
- Teach many things “by example” as we go along.
- You'll be expected to read supplementary notes when they are provided.
- No specific book for the course. Internet search for help will be useful.

Pre-requisites & software requirements

Pre-requisites

- Some programming experience in some language, e.g., Matlab, C.
- You should be comfortable:
 - editing a file containing a program and executing it,
 - using basic structures like loops, if-then-else, input-output,
 - writing subroutines or functions in some language
- You are not expected to know Python or Fortran.
- Some basic knowledge of linear algebra, e.g.:
 - what vectors and matrices are and how to multiply them
 - How to go about solving a linear system of equations
- Comfort level for learning new software.

Software requirements

- You will need access to a computer with a number of things on it. All open-source software.
- Note: Unix is often required for scientific computing.
- Windows: Many tools we'll use can be used with Windows, but learning Unix is part of this class.
- Options:
 - Install everything you'll need on your own computer.
 - Install VirtualBox and use the Virtual Machine (VM) created for this class.
 - Use Amazon Web Services with an Amazon Machine Image (AMI) which will be created for this class. Other cloud computing services are also available.

Know the class

- A short intro by various class members
 - Name and program enrolled in
 - What are your academic interests and what research projects you are working on? For PG students please mention your research group at IIT Mandi.
 - Why interested in this course? What are your expectations from the course?

Short demonstration

Demo code

```
$ cd <folder name>
$ export HPSC=$PWD
$ cd $HPSC/lecture1

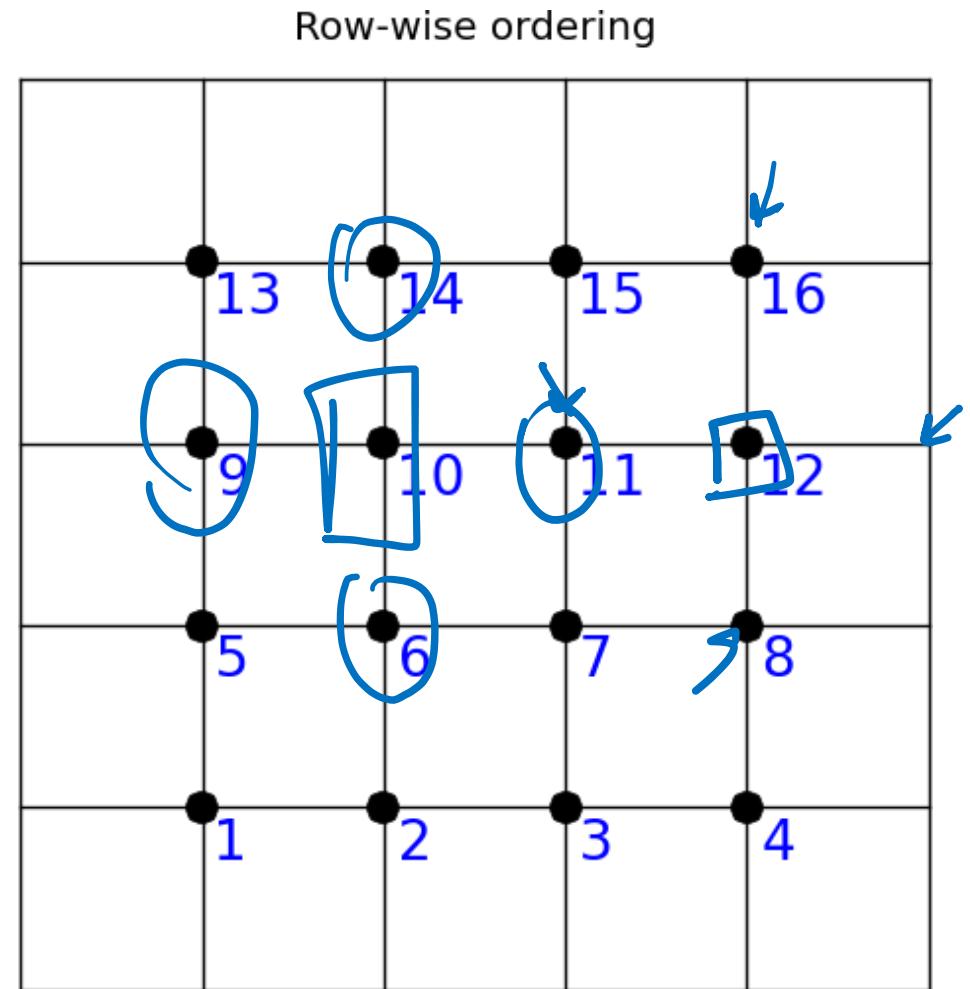
$ make plots

$ display *.png
```

Steady-state heat conduction

- Discretize on an $N \times N$ grid with N^2 unknowns
- Assume temperature is fixed (and known) at each point on boundary.
- At interior points, the steady state value is (approximately) the average of the 4 neighbouring values.

$$(A) \underbrace{\{x\}}_{N^2} = \underbrace{\{b\}}_{N^2}$$
$$\sqrt{N^2}$$



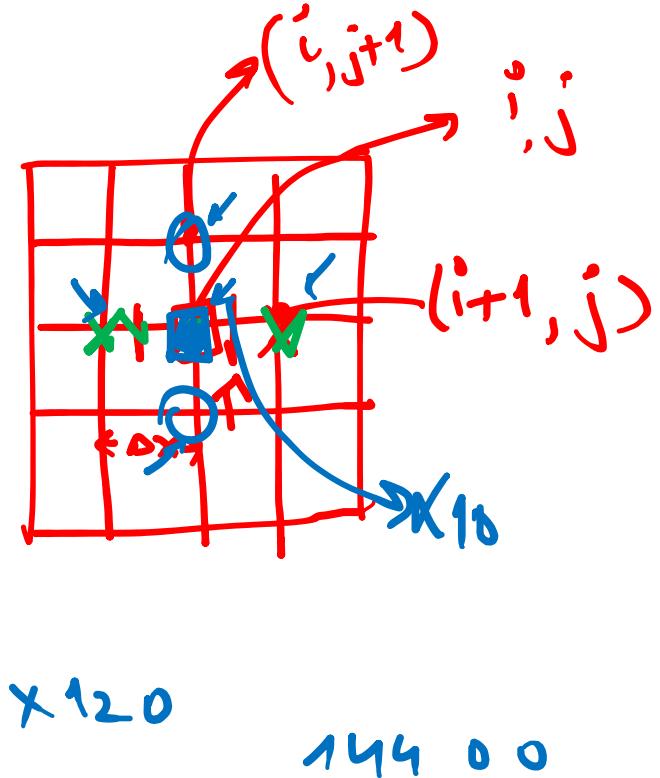
$$\nabla^2 u = 0$$

2-D

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

$$[A]_{N^2 \times N^2} \{x\} = \begin{matrix} L \\ \text{Boundary Condition} \end{matrix} \quad \{x\} = \begin{matrix} x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \end{matrix}$$

$N^2 = 120 \times 120$



$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2}$$

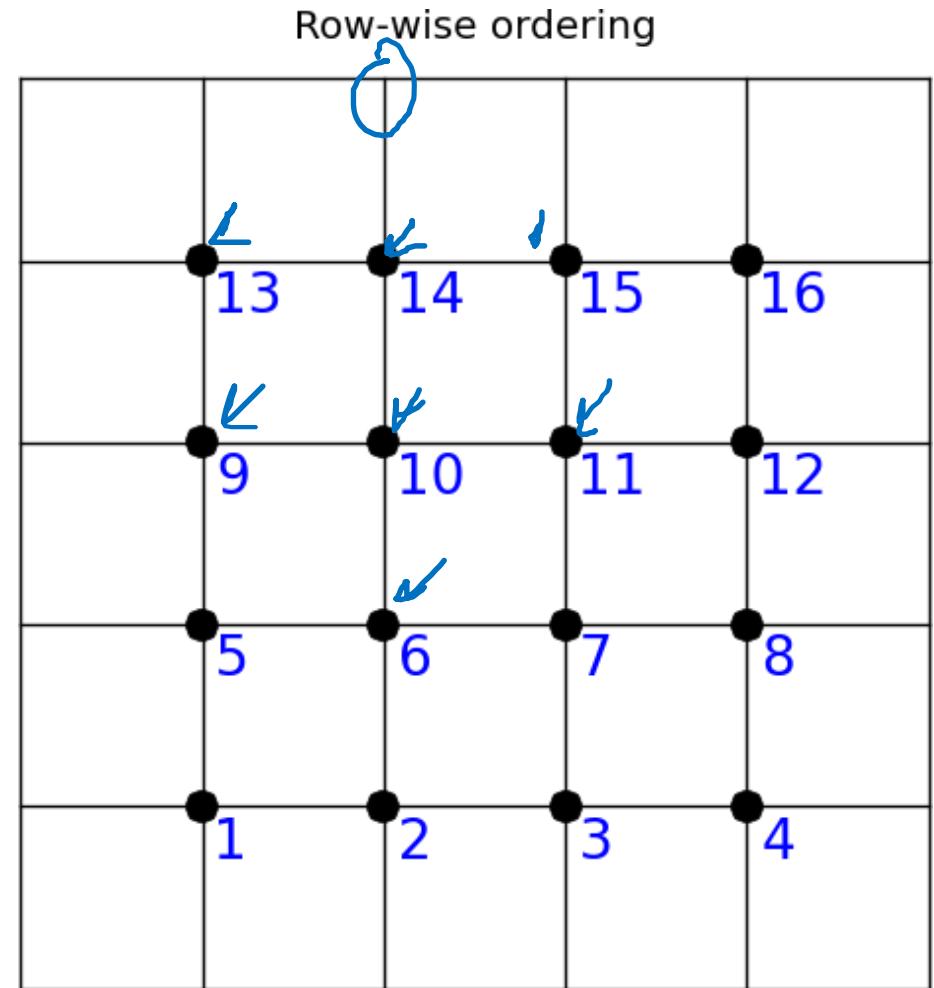
$$u_{i,j} = f(u_{i-1,j}, u_{i+1,j}, u_{i,j-1}, u_{i,j+1})$$

Steady-state heat conduction

$$u_{i,j} = \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1})$$

- Holds for $i, j = 1, 2, \dots, N$ with $u_{0,j}$ known on boundary. Gives a linear system $Au = b$, with N^2 equations N^2 unknowns. Matrix A is $N^2 \times N^2$, for $N = 120, N^2 = 14400$.
- Very sparse: each row of matrix A has at most 5 nonzeros. Gaussian elimination is not the best approach.
- Jacobi method (not the best method)

$$u_{(i,j)}^{[k+1]} = \frac{1}{4}(u_{(i-1,j)}^{[k]} + u_{(i+1,j)}^{[k]} + u_{(i,j-1)}^{[k]} + u_{(i,j+1)}^{[k]})$$



Speedup of linear solvers

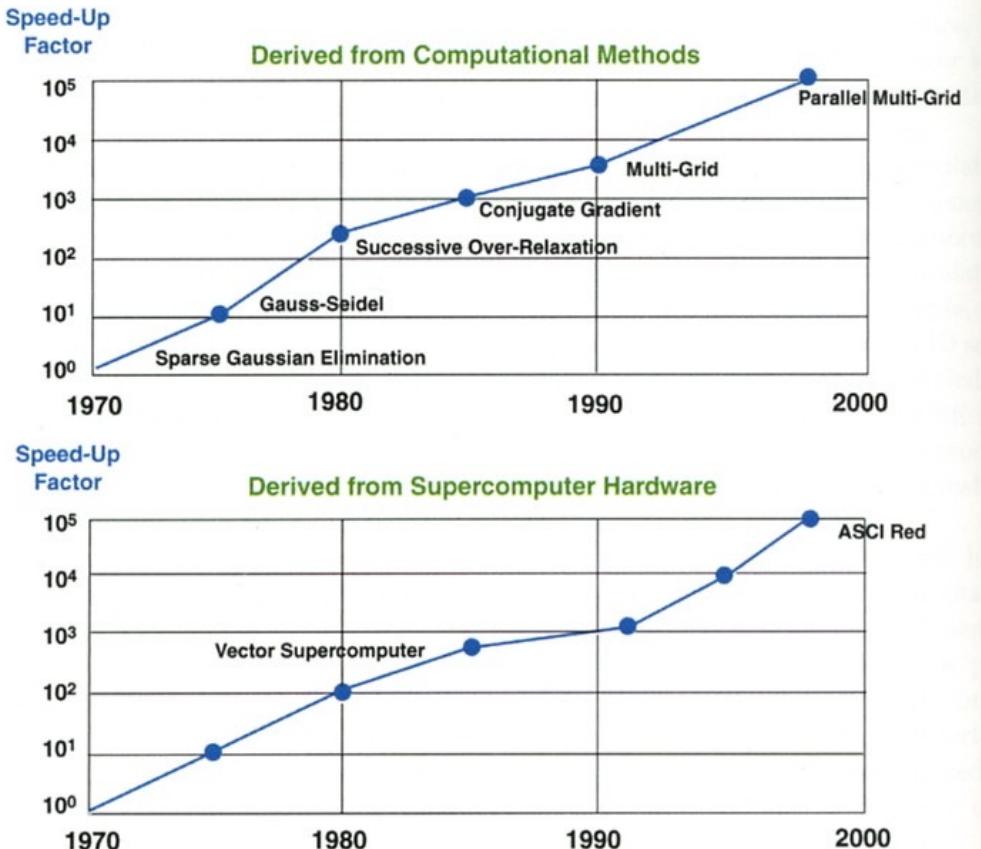


Fig. 2 Comparison of the contributions of mathematical algorithms and computer hardware.

Source: SIAM Review 43(2001), p. 168.

- Exponential increase in the speed of numerical algorithms.
- Exponential increase in the computer hardware performance with time.

Class virtual machine

- Available on Google Drive -
<https://drive.google.com/drive/folders/1mnMJoJNqiOpcuFzVZeW8EfoxqlqHWWVG?usp=sharing>
- Username: hpsc; password: me522
- This file is large! About 5 GB compressed. After unzipping, about 10 GB.

References

- Slides are adapted from HPSC, AM483, Uni of Washington by Randall J Leveque
[\(https://faculty.washington.edu/rjl/teaching.html\)](https://faculty.washington.edu/rjl/teaching.html)
Refer to his page for more useful material and notes.

HPSC 101 — Lecture 2

Outline:

- Binary storage, floating point numbers
- Version control — main ideas
- Distributed version control, e.g., **git**

Reading:

- Storing information in binary
- version control & git
- Github

Outline of course

Some topics to be covered:

- Unix
- Version control (git)
- Python
- Compiled vs. interpreted languages
- Fortran 90
- Makefiles
- Parallel computing
- OpenMP
- MPI (message passing interface)
- Graphics / visualization

Unix (and Linux, Mac OS X, etc.)

See the [Software Carpentry Unix training material](#)

Unix commands will be introduced as needed and mostly discussed in the context of other things.

Some important ones...

- cd, pwd, ls, mkdir
- mv, cp

Commands are typed into a terminal window shell,

We will use [bash](#). Prompt will be denoted \$, e.g.

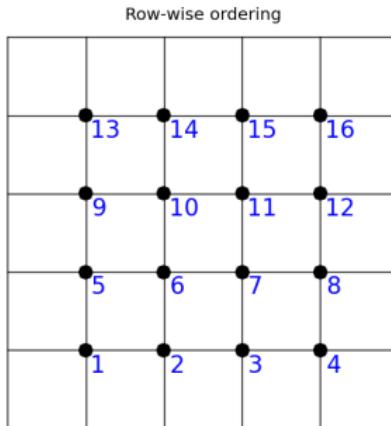
```
$ cd ..
```

Other references and sources

- Wikipedia often has good intros and summaries.
- Software Carpentry, particularly these videos (also see their YouTube playlist for lessons)
- Other courses at universities or supercomputer centers.

Steady state heat conduction

Discretize on an $N \times N$ grid with N^2 unknowns:



Assume temperature is fixed (and known) at each point on boundary.

At interior points, the steady state value is (approximately) the average of the 4 neighboring values.

Storing a big matrix

Recall: Approximating the heat equation on a 100×100 grid gives a linear system with 10,000 equations, $Au = b$ where the matrix A is $10,000 \times 10,000$.

Question: How much disk space is required to store a $10,000 \times 10,000$ matrix of real numbers?

Storing a big matrix

Recall: Approximating the heat equation on a 100×100 grid gives a linear system with 10,000 equations, $Au = b$ where the matrix A is $10,000 \times 10,000$.

Question: How much disk space is required to store a $10,000 \times 10,000$ matrix of real numbers?

It depends on how many bytes are used for each real number.

1 byte = 8 bits, bit = “binary digit”

Storing a big matrix

Recall: Approximating the heat equation on a 100×100 grid gives a linear system with 10,000 equations, $Au = b$ where the matrix A is $10,000 \times 10,000$.

Question: How much disk space is required to store a $10,000 \times 10,000$ matrix of real numbers?

It depends on how many bytes are used for each real number.

1 byte = 8 bits, bit = “binary digit”

Assuming 8 bytes (64 bits) per value:

A $10,000 \times 10,000$ matrix has 10^8 elements,

so this requires 8×10^8 bytes = **800 MB**.

Storing a big matrix

Recall: Approximating the heat equation on a 100×100 grid gives a linear system with 10,000 equations, $Au = b$ where the matrix A is $10,000 \times 10,000$.

Question: How much disk space is required to store a $10,000 \times 10,000$ matrix of real numbers?

It depends on how many bytes are used for each real number.

1 byte = 8 bits, bit = “binary digit”

Assuming 8 bytes (64 bits) per value:

A $10,000 \times 10,000$ matrix has 10^8 elements,

so this requires 8×10^8 bytes = 800 MB.

And less than 50,000 values are nonzero, so 99.95% are 0.

Measuring size and speed

Kilo = thousand (10^3)

Mega = million (10^6)

Giga = billion (10^9)

Tera = trillion (10^{12})

Peta = 10^{15}

Exa = 10^{18}

Computer memory

Memory is subdivided into **bytes**, consisting of 8 bits each.

One byte can hold $2^8 = 256$ distinct numbers:

$$00000000 = 0$$

$$00000001 = 1$$

$$00000010 = 2$$

...

$$11111111 = 255$$

Might represent integers, characters, colors, etc.

Usually programs involve integers and real numbers that require more than 1 byte to store.

Often 4 bytes (32 bits) or 8 bytes (64 bits) used for each.

Integers

To store integers, need one bit for the sign (+ or -)
In one byte this would leave 7 bits for binary digits.

Two-complements representation used:

10000000	= -128
10000001	= -127
10000010	= -126
...	
11111110	= -2
11111111	= -1
00000000	= 0
00000001	= 1
00000010	= 2
...	
01111111	= 127

$$215 = \underbrace{2 \times 10^2}_{+ 1 \times 10^1} + \underbrace{5 \times 10^0}_{+ 5 \times 10^0}$$
$$215 = 01011111_2$$
$$0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

Advantage: Binary addition works directly.

Integers

Integers are typically stored in 4 bytes (32 bits). Values between roughly -2^{31} and 2^{31} can be stored.

In Python, larger integers can be stored and will automatically be stored using more bytes.

Note: special software for arithmetic, may be slower!

```
$ python
```

```
>>> 2**30  
1073741824
```

```
>>> 2**100  
1267650600228229401496703205376L
```

Note L on end!

Fixed point notation

Use, e.g. 64 bits for a real number but always assume N bits in integer part and M bits in fractional part.

Analog in decimal arithmetic, e.g.:

5 digits for integer part and

6 digits in fractional part

Could represent, e.g.:

00003.141592 (pi)

00000.000314 (pi / 10000)

31415.926535 (pi * 10000)

Fixed point notation

Use, e.g. 64 bits for a real number but always assume N bits in integer part and M bits in fractional part.

Analog in decimal arithmetic, e.g.:

5 digits for integer part and

6 digits in fractional part

Could represent, e.g.:

00003.141592 (pi)

00000.000314 (pi / 10000)

31415.926535 (pi * 10000)

Disadvantages:

- Precision depends on size of number
- Often many wasted bits (leading 0's)
- Limited range; often scientific problems involve very large or small numbers.

Floating point real numbers

Base 10 scientific notation:

$$0.2345e-18 = 0.2345 \times 10^{-18} = 0.000000000000000000002345$$

Mantissa: 0.2345, Exponent: -18

Floating point real numbers

Base 10 scientific notation:

$$0.2345e-18 = 0.2345 \times 10^{-18} = 0.000000000000000000002345$$

Mantissa: 0.2345, Exponent: -18

Binary floating point numbers:

Example: Mantissa: 0.101101, Exponent: -11011 means:

$$\begin{aligned}0.101101 &= 1(2^{-1}) + 0(2^{-2}) + 1(2^{-3}) + 1(2^{-4}) + 0(2^{-5}) + 1(2^{-6}) \\&= 0.703125 \text{ (base 10)}\end{aligned}$$

$$-11011 = -1(2^4) + 1(2^3) + 0(2^2) + 1(2^1) + 1(2^0) = -27 \text{ (base 10)}$$

So the number is

$$0.703125 \times 2^{-27} \approx 5.2386894822120667 \times 10^{-9}$$

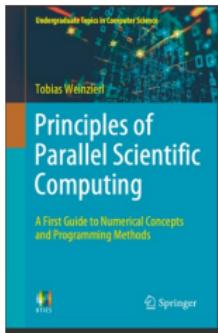
Floating point real numbers

Python `float` is 8 bytes with IEEE standard (754) representation.
53 bits for mantissa and 11 bits for exponent (64 bits = 8 bytes).

We can store 52 binary bits of precision.

$$2^{-52} \approx 2.2 \times 10^{-16} \Rightarrow \text{roughly 15 digits of precision.}$$

Read Chapter 4: Floating Point Numbers from
Principles of Parallel Scientific Computing by
Tobias Weinzierl



Floating point real numbers

Since $2^{-52} \approx 2.2 \times 10^{-16}$ this corresponds to roughly 15 digits of precision.

For example:

```
>>> from numpy import pi
```

```
>>> pi
```

```
3.1415926535897931
```

```
>>> 1000 * pi
```

```
3141.5926535897929
```

```
>>> pi/1000
```

```
0.0031415926535897933
```

Note: storage and arithmetic is done in base 2
Converted to base 10 only when printed!

Version control systems

Originally developed for large software projects with many developers.

Also useful for single user, e.g. to:

- Keep track of history and changes to files,
- Be able to revert to previous versions,
- Keep many different versions of code well organized,
- Easily archive exactly the version used for results in publications,
- Keep work in sync on multiple computers.

Server-client model:

Original style, still widely used (e.g. CVS, Subversion)

One **central repository** on server.

Developers' workflow (simplified!):

- Check out a **working copy**,
- Make changes, test and debug,
- Check in (**commit**) changes to repository (with comments).
This creates new **version number**.
- Run an **update** on working copy to bring in others' changes.

The system keeps track of **diffs** from one version to the next
(and info on who made the changes, when, etc.)

A **changeset** is a collection of **diffs** from one commit.

Server-client model:

Only the server has the full history.

The working copy has:

- Latest version from repository (from last [checkout](#), [commit](#), or [update](#))
- Your local changes that are not yet committed.

Server-client model:

Only the server has the full history.

The working copy has:

- Latest version from repository (from last **checkout**, **commit**, or **update**)
- Your local changes that are not yet committed.

Note:

- You can retrieve older versions from the server.
- Can only *commit* or *update* when connected to server.
- When you *commit*, it will be seen by anyone else who does an *update* from the repository.

Often there are **trunk** and **branches** subdirectories.

Distributed version control

Git uses a distributed model:

When you clone a repository you get all the history too,

All stored in .git subdirectory of top directory.

Usually don't want to mess with this!

Ex: (backslash is continuation character in shell)

```
$ git clone \
  https://bitbucket.org/rjleveque/uwhpsc \
  mydirname
```

will make a complete copy of the class repository (from uwhpsc) and call it mydirname. If mydirname is omitted, it will be called uwhpsc.

This directory has a subdirectory .git with complete history.

Distributed version control

Git uses a distributed model:

- `git commit` commits to your clone's .git directory.
- `git push` sends your recent changesets to another clone by default: the one you cloned from (e.g. bitbucket), but you can push to any other clone (with write permission).
- `git fetch` pulls changesets from another clone by default: the one you cloned from (e.g. bitbucket)
- `git merge` applies changesets to your working copy

Next lecture: simpler example of using git in a single directory.

Distributed version control

Advantages of distributed model:

- You can commit changes, revert to earlier versions, examine history, etc. without being connected to server.
- Also without affecting anyone else's version if you're working collaboratively. *Can commit often while debugging.*

Distributed version control

Advantages of distributed model:

- You can commit changes, revert to earlier versions, examine history, etc. without being connected to server.
- Also without affecting anyone else's version if you're working collaboratively. *Can commit often while debugging.*
- No problem if server dies, every clone has full history.

Distributed version control

Advantages of distributed model:

- You can commit changes, revert to earlier versions, examine history, etc. without being connected to server.
- Also without affecting anyone else's version if you're working collaboratively. *Can commit often while debugging.*
- No problem if server dies, every clone has full history.

For collaboration will still need to push or fetch changes eventually and `git merge` may become more complicated.

Bitbucket

You can uwhpsc examine class repository at:

<https://bitbucket.org/rjleveque/uwhpsc>

Experiment with the “Source”, “Commits” tabs...

See also Software Carpentry for more git references and tutorials.

Github

<https://github.com>

Another repository for hosting git repositories.

Many open sources projects use it, including Linux kernel.

(Git was developed by Linus Torvalds for this purpose!)

Github

<https://github.com>

Another repository for hosting git repositories.

Many open source projects use it, including Linux kernel.

(Git was developed by Linus Torvalds for this purpose!)

Many open source scientific computing projects use github, e.g.

- Ipython, Jupyter notebook
- NumPy, Scipy, matplotlib

Other tools for git

The [gitkraken](#) tool is useful for working with git in GUI.

This is not installed on the VM, but you can install it from their website by downloading the .deb or

```
sudo snap install gitkraken
```

Demo...

Hands-on exercise

1. Explore the Software Carpentry Github repo page for Unix Shell training
2. Modify a file in the SC Unix repo using gitkraken, then commit the change, then checkout the previous version again to access the version of the file you cloned.
3. Apply for HPC account over the IIT Mandi HPC website

<https://sites.google.com/iitmandi.ac.in/hpc-iit-mandi>

Hands-on exercise - bash

1. List the largest file in the folder shell-novice/shell-lesson-data/exercise-data/proteins, print its name, then copy it a new folder called largest.
2. What does: ls pe?tane.* give? And cat pe?tane.* ?
3. Count the number of files and directories in a given folder.
Generate a new file with the text: “the number of files in the folder is xx”.
4. Write a bash script/loop to create a file that contains the second last line of all pdb files available in a folder
5. in a folder containing 10 files, write a bash script to prepend the name of each file with the year it was created.
- 6.What does this command do: cut -d, -f 2 animals.csv | sort | uniq -c? animals.csv is in shell-novice/shell-lesson-data/exercise-data/animal-counts

Answers

```
echo cp $(ls -S | head -1) ./largest
```

```
ls | wc -w
```

```
echo The number of the files is $(ls | wc -w) >  
blah.txt
```

```
for filename in *.pdb  
do  
tail -2 $filename | head -1 >> tt.txt  
Done
```

```
for filename in *.pdb; do echo mv $filename  
$(ls -lT $filename | cut -w -f 9)-$filename;  
done
```

Hands-on exercise - bash

1. You have a set of 20 text files and would like to concatenate them into one super text file.
2. Write a script that prints the name of the file with the longest name in a given directory.

Answer:

```
for filename in *.pdb  
do  
    echo ${filename}  
    cat ${filename} >> total.pdb
```

Done

```
for line in $(ls); do echo ${#line} ${line} >>  
namelen.txt; done
```

HPSC 101 — Lecture 3

This lecture:

- computing square roots
- Python demo
- git demo

Computing square roots

Hardware arithmetic units can add, subtract, multiply, divide.

Other mathematical functions usually take some software.

Computing square roots

Hardware arithmetic units can add, subtract, multiply, divide.

Other mathematical functions usually take some software.

Example: Compute $\sqrt{2} \approx 1.4142135623730951$

In most languages, `sqrt(2)` computes this.

```
>>> from numpy import sqrt  
>>> sqrt(2.)
```

One possible algorithm to approximate $s = \sqrt{x}$

```
s = 1.      # or some better initial guess
for k in range(kmax):
    s = 0.5 * (s + x/s)
```

where k_{max} is some maximum number of iterations.

Note: In Python, `range(N)` is $[0, 1, 2, \dots, N - 1]$.

One possible algorithm to approximate $s = \sqrt{x}$

```
s = 1.      # or some better initial guess  
for k in range(kmax):  
    s = 0.5 * (s + x/s)
```

where k_{max} is some maximum number of iterations.

Note: In Python, `range(N)` is $[0, 1, 2, \dots, N - 1]$.

Why this works...

If $s < \sqrt{x}$ then $x/s > \sqrt{x}$

If $s > \sqrt{x}$ then $x/s < \sqrt{x}$

One possible algorithm to approximate $s = \sqrt{x}$

```
s = 1.      # or some better initial guess  
for k in range(kmax):  
    s = 0.5 * (s + x/s)
```

where k_{max} is some maximum number of iterations.

Note: In Python, `range(N)` is $[0, 1, 2, \dots, N - 1]$.

[Why this works...](#)

If $s < \sqrt{x}$ then $x/s > \sqrt{x}$

If $s > \sqrt{x}$ then $x/s < \sqrt{x}$

In fact this is [Newton's method](#) to find root of $s^2 - x = 0$.

Newton's method

Problem: Find a solution of $f(s) = 0$ (zero or root of f)

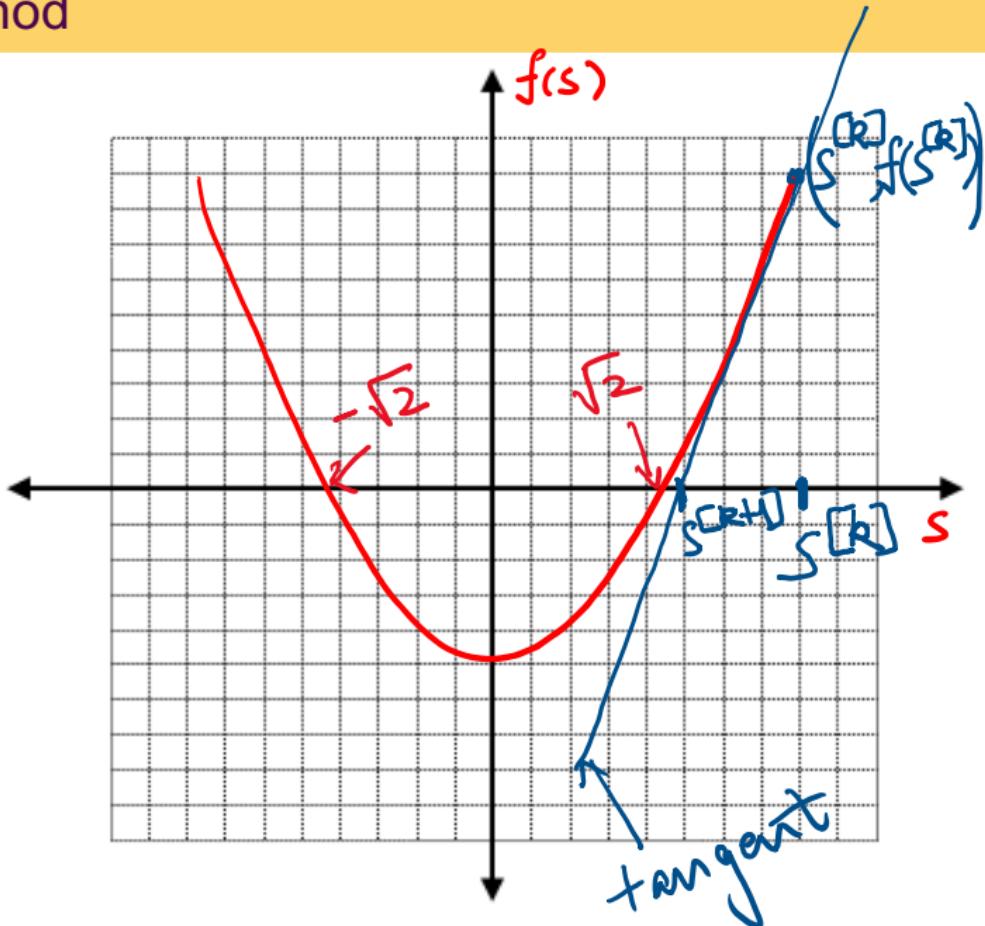
Idea: Given approximation $s^{[k]}$,

approximate $f(s)$ by a linear function,
the tangent line at $(s^{[k]}, f(s^{[k]}))$.

Find unique zero of this function and use as $s^{[k+1]}$.

Newton's method

$$f(s) = s^2 - x$$



Newton's method

Problem: Find a solution of $f(s) = 0$ (zero or root of f)

Idea: Given approximation $s^{[k]}$,

approximate $f(s)$ by a linear function,
the tangent line at $(s^{[k]}, f(s^{[k]}))$.

Find unique zero of this function and use as $s^{[k+1]}$.

Updating formula:

$$s^{[k+1]} = s^{[k]} - \frac{f(s^{[k]})}{f'(s^{[k]})}$$
$$s = s - \frac{s^2 - \pi}{2s} = \frac{1}{2} \left(s + \frac{\pi}{s} \right)$$

Demo...

Goals:

- Develop our own version of `sqrt` function.
- Start simple and add complexity in stages.
- Illustrate some Python programming.
- Illustrate use of git to track our development

HPSC 101 — Lecture 4

This lecture:

- Continue demo: computing square roots
- More about Python, IPython
- More about git, Unix

Installing new software on VM

Some useful additions...

```
$ sudo apt-get install xxdiff  
(venv) $ pip install sympy  
  
(venv) $ pip install pytest
```

Demo

Demo

Python and git (*30 minutes*)

- Fork the `hpsc_2023` repository into your own Github account
- Create `mycubrt()` (be careful with negative x values here). This file must sit in `code/python folder`
- Create unit tests and run the tests using nose. Make sure all the tests pass.
- Add and commit this work
- Push this work to your Github repository

HPSC101 — Lecture 5

This lecture:

- Python concepts and objects
- Data types, lists, tuples
- Modules
- Demo — plotting and IPython notebook

Python is an **object oriented** general-purpose language

Advantages:

- Can be used interactively from a Python shell (similar to Matlab)
- Can also write scripts to execute from Unix shell
- Little overhead to start programming
- Powerful modern language
- Many **modules** are available for specialized work
- Good graphics and visualization modules
- Easy to combine with other languages (e.g. Fortran)
- Open source and runs on all platforms

Python

Disadvantage: Can be slow to do certain things,
such as looping over arrays.

Code is [interpreted](#) rather than compiled

Need to use suitable modules (e.g. NumPy) for speed.

Can easily create custom modules from compiled code written
in Fortran, C, etc.

Can also use extensions such as [Cython](#) that makes it easier to
mix Python with C code that will be compiled.

Python is often used for high-level scripts that e.g., download
data from the web, run a set of experiments, collate and plot
results.

Object-oriented language

Nearly everything in Python is an **object** of some **class**.

The class description tells what data the object holds (**attributes**) and what operations (**methods** or functions) are defined to interact with the object.

Object-oriented language

Nearly everything in Python is an **object** of some **class**.

The class description tells what data the object holds (**attributes**) and what operations (**methods** or functions) are defined to interact with the object.

Every “**variable**” is really just a pointer to some object. You can reset it to point to some other object at will.

So variables don’t have “type” (e.g. integer, float, string).
(But the objects they currently point to do.)

Object-oriented language

```
>>> x = 3.4
>>> print id(x), type(x)      # id() returns memory
8645588 <type 'float'>      address

>>> x = 5
>>> print id(x), type(x)
8401752 <type 'int'>

>>> x = [4,5,6]
>>> print id(x), type(x)
1819752 <type 'list'>

>>> x = [7,8,9]
>>> print id(x), type(x)
1843808 <type 'list'>
```

Object-oriented language

```
>>> x = [7,8,9]
>>> print id(x), type(x)
1843808 <type 'list'>

>>> x.append(10)
>>> x
[7, 8, 9, 10]
>>> print id(x), type(x)
1843808 <type 'list'>
```

Note: Object of type 'list' has a method 'append' that **changes** the object.

A list is a **mutable object**.

Object-oriented language — gotcha

```
>>> x = [1,2,3]
>>> print id(x), x
1845768 [1, 2, 3]
```

```
>>> y = x
>>> print id(y), y
1845768 [1, 2, 3]
```

```
>>> y.append(27)
>>> y
[1, 2, 3, 27]
```

```
>>> x
[1, 2, 3, 27]
```

Note: x and y point to the same object!

Making a copy

```
>>> x = [1,2,3]
>>> print id(x), x
1845768 [1, 2, 3]

>>> y = list(x)      # creates new list object
>>> print id(y), y
1846488 [1, 2, 3]

>>> y.append(27)

>>> y
[1, 2, 3, 27]

>>> x
[1, 2, 3]
```

integers and floats are immutable

If `type(x)` in [int, float], then setting `y = x` creates a new object `y` pointing to a new location.

```
>>> x = 3.4  
>>> print id(x), x  
8645588 3.4
```

```
>>> y = x  
>>> print id(y), y  
8645588 3.4
```

```
>>> y = y+1  
  
>>> print id(y), y  
8463377 4.4
```

```
>>> print id(x), x  
8645588 3.4
```

Lists

The **elements of a list** can be **any objects**
(need not be same type):

```
>>> L = [3, 4.5, 'abc', [1,2]]
```

Indexing starts at 0:

```
>>> L[0]  
3
```

```
>>> L[2]  
'abc'
```

```
>>> L[3]  
[1, 2]
```

```
>>> L[3][0] # element 0 of L[3]  
1
```

Lists

Lists have several built-in methods, e.g. append, insert, sort, pop, reverse, remove, etc.

```
>>> L = [3, 4.5, 'abc', [1,2]]
```

```
>>> L2 = L.pop(2)
```

```
>>> L2  
'abc'
```

```
>>> L  
[3, 4.5, [1, 2]]
```

Note: L still points to the same object, but it has changed.

In IPython: Type L. followed by Tab to see all attributes and methods.

Lists and tuples

```
>>> L = [3, 4.5, 'abc']
>>> L[0] = 'xy'
>>> L
['xy', 4.5, 'abc']
```

A **tuple** is like a list but is **immutable**:

```
>>> T = (3, 4.5, 'abc')
>>> T[0]
3
>>> T[0] = 'xy'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support
           item assignment
```

Python modules

When you start Python it has a few basic built-in types and functions.

To do something fancier you will probably **import** modules.

Example: to use square root function:

```
>>> from numpy import sqrt  
>>> sqrt(2.)  
1.4142135623730951
```

Python modules

When type `import modname`, Python looks on its **search path** for the file `modname.py`.

You can add more directories using the Unix environment variable **PYTHONPATH**.

Or, in Python, using the **sys** module:

```
>>> import sys  
>>> sys.path    # returns list of directories  
['', '/usr/bin', ....]  
  
>>> sys.path.append('newdirectory')
```

The empty string “” in the search path means it looks first in the current directory.

Python modules

Different ways to import:

```
>>> from numpy import sqrt  
>>> sqrt(2.)  
1.4142135623730951
```

```
>>> from numpy import *  
>>> sqrt(2.)  
1.4142135623730951
```

```
>>> import numpy  
>>> numpy.sqrt(2.)  
1.4142135623730951
```

```
>>> import numpy as np  
>>> np.sqrt(2.)  
1.4142135623730951
```

Graphics and Visualization

Many tools are available for plotting numerical results.

Some open source Python options:

- matplotlib for 1d plots and
2d plots (e.g. pseudocolor, contour, quiver)
- Mayavi for 3d plots (curves, surfaces, vector fields)

Graphics and Visualization

Open source packages developed by National Labs...

- VisIt
- ParaView

Harder to get going, but designed for large-scale 3d plots,
distributed data, adaptive mesh refinement results, etc.:

Each have stand-alone GUI and also Python scripting
capabilities.

Based on VTK (Visualization Tool Kit).

HPSC 101 — Lecture 6

This lecture:

- NumPy arrays and functions
- “Pythonic” ways to do things
- Python: main programs and private variables
- Timing Python execution

Lists aren't good as numerical arrays

Lists in Python are quite general, can have arbitrary objects as elements.

Addition and scalar multiplication are defined for lists, but not what we want for numerical computation, e.g.

Multiplication repeats:

```
>>> x = [2., 3.]  
>>> 2*x  
[2.0, 3.0, 2.0, 3.0]
```

Addition concatenates:

```
>>> y = [5., 6.]  
>>> x+y  
[2.0, 3.0, 5.0, 6.0]
```

NumPy module

Instead, use NumPy arrays:

```
>>> import numpy as np  
  
>>> x = np.array([2., 3.])  
>>> 2*x  
array([ 4.,  6.])
```

Try x^*y where both x and y are arrays of the same size.

Other operations also apply component-wise:

```
>>> np.sqrt(x) * np.cos(x) * x**3  
array([-4.708164, -46.29736719])
```

Note: $*$ is component-wise multiply

NumPy arrays

Unlike lists, **all elements** of an `np.array` have the **same type**

```
>>> np.array([1, 2, 3])      # all integers  
array([1, 2, 3])
```

```
>>> np.array([1, 2, 3.])     # one float  
array([ 1.,  2.,  3.])      # they're all floats!
```

Can explicitly state desired data type:

```
>>> x = np.array([1, 2, 3], dtype=complex)  
>>> print x  
[ 1.+0.j,  2.+0.j,  3.+0.j]
```

```
>>> (x + 1.j) * 2.j  
array([-2.+2.j, -2.+4.j, -2.+6.j])
```

NumPy arrays for vectors and matrices

```
>>> A = np.array([[1., 2], [3, 4], [5, 6]])  
>>> A  
array([[ 1.,  2.],  
       [ 3.,  4.],  
       [ 5.,  6.]])  
  
>>> A.shape  
(3, 2)  
  
>>> A.T  
array([[ 1.,  3.,  5.],  
       [ 2.,  4.,  6.]])  
  
>>> x = np.array([1., 1.])  
>>> x.T  
array([ 1.,  1.])
```

NumPy arrays for vectors and matrices

```
>>> A  
array([[ 1.,  2.],  
       [ 3.,  4.],  
       [ 5.,  6.]])  
  
>>> x  
array([ 1.,  1.])  
  
>>> np.dot(A, x)      # matrix-vector product  
array([ 3.,  7., 11.])  
  
>>> np.dot(A.T, A)    # matrix-matrix product  
array([[ 35.,  44.],  
       [ 44.,  56.]])
```

NumPy matrices for vectors and matrices

For Linear algebra, may instead want to use `numpy.matrix`:

```
>>> A = np.matrix( [[1.,2], [3,4], [5,6]] )
>>> A
matrix([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
```

Or, Matlab style (as a string that is converted):

```
>>> A = np.matrix("1.,2; 3,4; 5,6")
>>> A
matrix([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
```

NumPy matrices for vectors and matrices

Note: vectors are handled as matrices with 1 row or column:

```
>>> x = np.matrix("4.;5.")  
>>> x  
matrix([[ 4.],  
       [ 5.]])  
>>> x.T  
matrix([[ 4.,  5.]])  
>>> A*x  
matrix([[ 14.],  
       [ 32.],  
       [ 50.]])
```

But note that indexing into `x` requires two indices:

```
>>> print x[0,0], x[1,0]  
4.0 5.0  
Try x[:, :]
```

Which to use, array or matrix?

For linear algebra matrix may be easier (and more like Matlab),
but vectors need two subscripts!

For most other uses, arrays more natural, e.g.

```
>>> x = np.linspace(0., 3., 100)    # 100 points  
>>> y = x**5 - 2.*sqrt(x)*cos(x)    # 100 values  
>>> plot(x,y)
```

`np.linspace` returns an `array`, which is what is needed here.

We will always use arrays.

See http://www.scipy.org/NumPy_for_Matlab_Users

Rank of an array

The **rank** of an array is the number of subscripts it takes:

```
>>> A = np.ones((4, 4))
```

```
>>> A
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

```
>>> np.rank(A)
2
```

Warning: This is not the rank of the matrix in the linear algebra sense (dimension of the column space)!

Rank of an array

Scalars have rank 0:

```
>>> z = np.array(7.)  
>>> z  
array(7.0)
```

NumPy arrays of any dimension are supported, e.g. rank 3:

```
>>> T = np.ones((2,2,2))  
>>> T  
array([[[ 1.,  1.],  
       [ 1.,  1.]],  
  
       [[ 1.,  1.],  
       [ 1.,  1.]]])  
>>> T[0,0,0]  
1.0
```

Linear algebra with NumPy

```
>>> A = np.array([[1., 2.], [3, 4]])  
>>> A  
array([[ 1.,  2.],  
       [ 3.,  4.]])  
  
>>> b = np.dot(A, np.array([8., 9.]))  
>>> b  
array([ 26.,  60.])
```

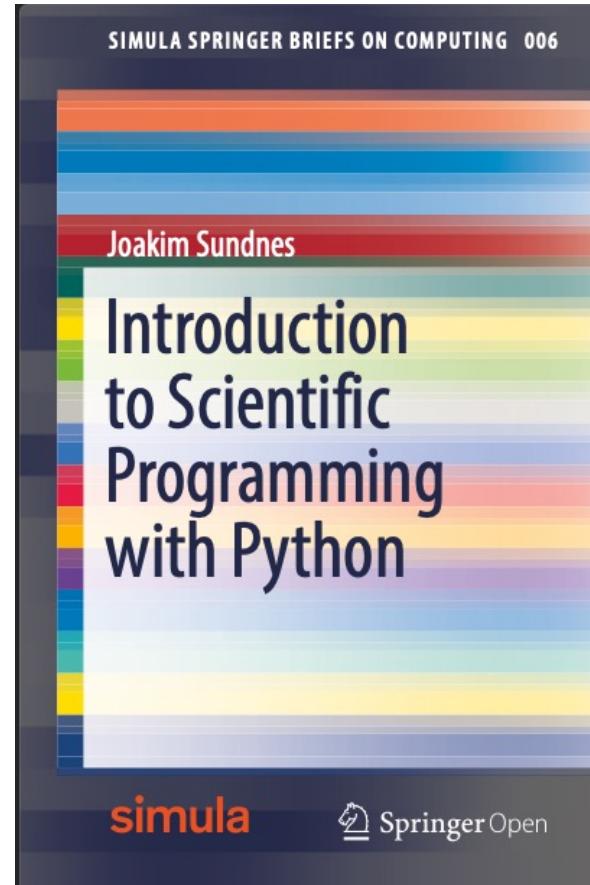
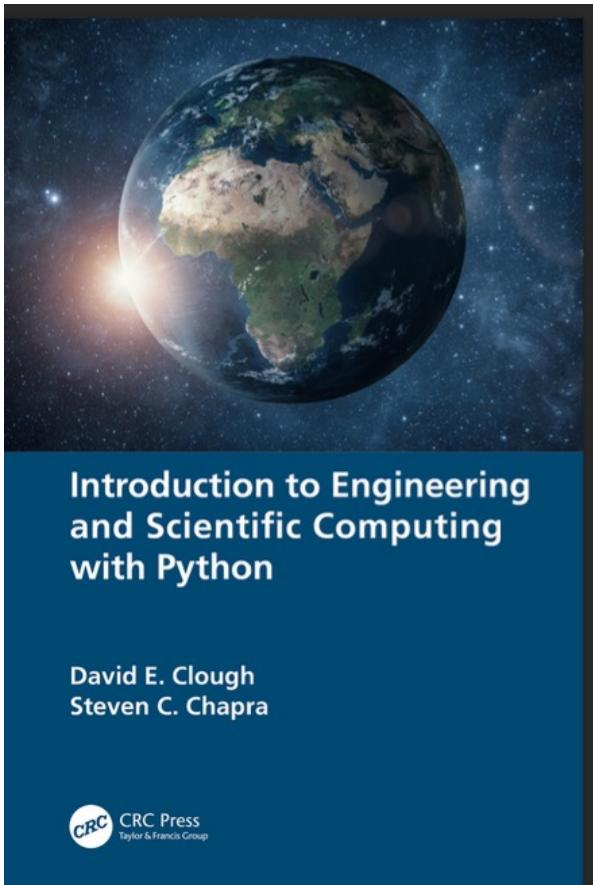
Now solve $Ax = b$:

```
>>> from numpy.linalg import solve  
>>> solve(A,b)  
array([ 8.,  9.])
```

Eigenvalues

```
>>> from numpy.linalg import eig  
  
>>> eig(A) # returns a tuple (evals, evecs)  
  
(array([-0.37228132,  5.37228132]),  
 array([[ -0.82456484, -0.41597356],  
        [ 0.56576746, -0.90937671]]))  
  
>>> evals, evecs = eig(A) # unpacks tuple  
  
>>> evals  
array([-0.37228132,  5.37228132])  
  
>>> evecs  
array([[ -0.82456484, -0.41597356],  
        [ 0.56576746, -0.90937671]])
```

Reference



[https://link.springer.com
/book/10.1007/978-3-
030-50356-7](https://link.springer.com/book/10.1007/978-3-030-50356-7)

Pythonic ways

- f-string format (since Python 3.6)

```
print("At iteration number %s, s= %20.15f" %(k,s))  
print(f"At iteration number {k}, s= {s}")  
print(f"At iteration number {k}, s= {s:10.15f}")
```

```
t = 1.234567  
print(f"Default output gives t = {t}.")  
print(f"We can set the precision: t = {t:.2}.")  
print(f"Or control the number of decimals: t = {t:.2f}.")
```

Default output gives t = 1.234567.
We can set the precision: t = 1.2.
Or control the number of decimals: t = 1.23.

- Zip function

```
for low, high in zip(A_low, A_high):  
    print(low, high)
```

Python ways - continued

- **List slicing**

```
>>> a = [2, 3.5, 8, 10]
>>> a[2:] # from index 2 to end of list [8, 10]
>>> a[1:3] # from index 1 up to, but not incl., index 3 [3.5, 8]
>>> a[:3] # from start up to, but not incl., index 3 [2, 3.5, 8]
>>> a[1:-1] # from index 1 to next last element [3.5, 8]
>>> a[:] # the whole list [2, 3.5, 8, 10]
```

b = a[:] will make a copy of the entire list a, and any subsequent changes to b will not change a

- **Membership**

```
List1=[1,2,3,"hello"]
2 in List1 - True
4 in List1 - False
"hello" in List1 - True
```

Quadrature (numerical integration)

Estimate $\int_0^2 x^2 dx = \frac{8}{3}$:

```
>>> from scipy.integrate import quad  
  
>>> def f(x):  
...     return x**2  
...  
>>> quad(f, 0., 2.)  
(2.666666666666667, 2.960594732333751e-14)
```

returns (value, error estimate).

Other keyword arguments to set error tolerance, for example.

Lambda functions

In the last example, f is so simple we might want to just include its definition directly in the call to `quad`.

We can do this with a **lambda function**:

```
>>> f = lambda x: x**2  
>>> f(4)  
16
```

This defines the same f as before. But instead we could do:

```
>>> quad(lambda x: x**2, 0., 2.)  
(2.666666666666667, 2.960594732333751e-14)
```

“Main program” in a Python module

Python modules often end with a section that looks like:

```
if __name__ == "__main__":  
  
    # some code
```

This code is **not** executed if the file is imported as a module, only if it is run as a script, e.g. by...

```
$ python filename.py  
  
>>> execfile("filename.py")  
  
In[1]: run filename.py
```

Exercise

- Write a function to calculate the exponential using the exponential series
 - The function should take the number of terms of the series as an argument, use default of 100
 - Check the convergence of the series
- Plot $y=\exp(x)$ for 1000 points between 0 and 100, using the built-in numpy `exp()` function and `your_exp()` function
- Compare the execution time of the two functions using the `timeit` command of Ipython
- All work should be done in Jupyter online and the notebook saved back into your local machine and pushed into your fork

HPSC 101 — Lecture 7 – part 2

This lecture:

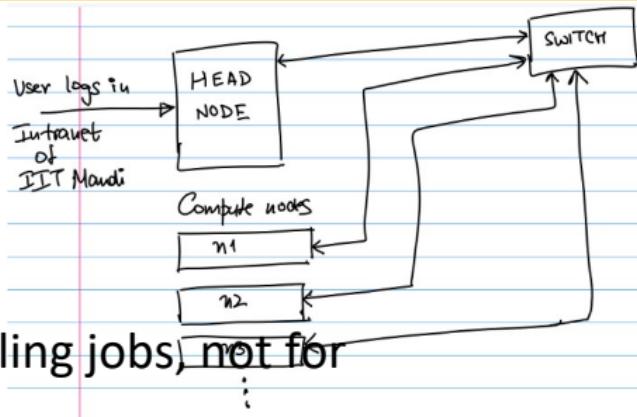
- HPC introduction
- Python on HPC
- Hands-on
- Exercise

HPC - Introduction

HPC topology

Head node:

Suited to login and scheduling jobs, ~~not for~~
computation



Compute nodes:

Suited for heavy computation, typically has 2 processors per node (machine), each processor has up to 12 cores, 64+GB RAM, 2TB local storage

HPC IIT Mandi (website tour)

- .[https://sites.google.com/iit
mandi.ac.in/hpc-iit-mandi/](https://sites.google.com/iitmandi.ac.in/hpc-iit-mandi/)

- .CPUHPC (10.8.1.19): CPU-based parallelism

- .GPUHPC (10.8.1.20): GPU-based parallelism

- .10G connectivity between nodes

- .Filesystems

- Home: 10GB

- Working dir (wd): 2TB

- .Software: basic software + *Singularity*

- .Example PBS scripts

- .Queue details

- .Resources

Applications

Engineering

- Fluid flow and heat transfer (Open FOAM, Fluidity, ANSYS Fluent)
- Solid mechanics (ANSYS)
- Deep learning (Python modules)

Physics

- Molecular dynamics (in-house codes)

Chemistry

- Computational chemistry (Gromacs)

Biology

- Gene sequencing

IIT Mandi HPC cluster: 2884 processing cores; 300+ users • 169 nodes; 2884 cores

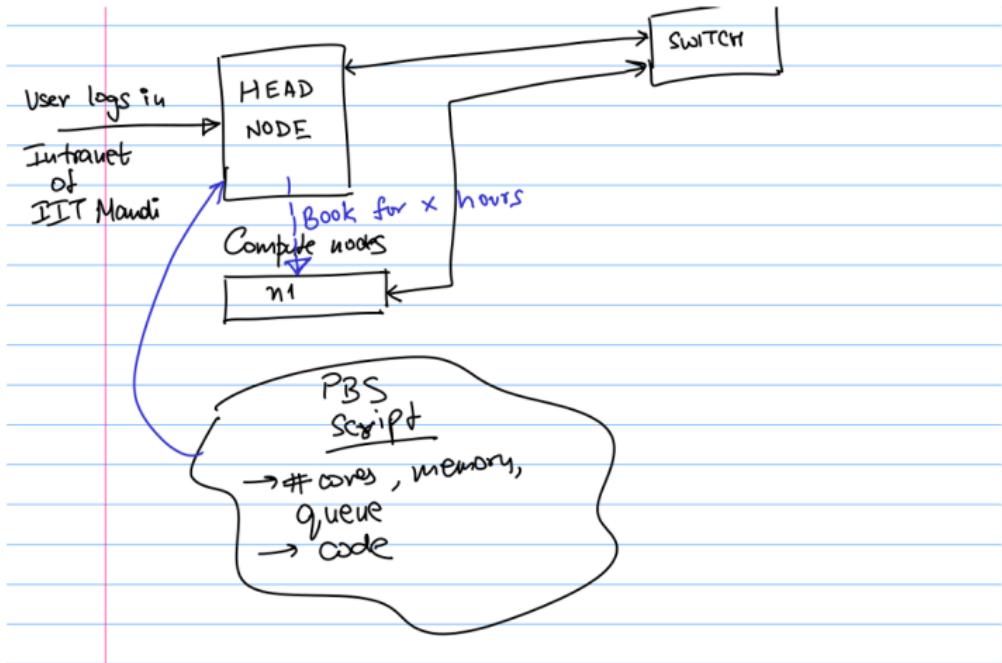
- Intel Xeon processors
- 11.5 TB memory
- 986 TB storage
- 33 Nvidia graphical processing units (GPUs)
- 10Gb/s ethernet connectivity
- Cooling system

PBS: portable batch system

Queuing system for job submission which takes care of the number of jobs and excessive usage

Clusters

Filesystems



Hands-on

- Session screenshot to be recorded
- Logging in to the cluster(s) and directory structure
- Prepare virtualenv and install packages using pip
- PBS script for running a Python code
- Launching the PBS script using qsub
- Output and error files

Commands

```
scl --list - list all scl (Red Hat) packages available  
scl enable rh-python36 bash (exit to exit from the scl)  
pip3 install --user --upgrade --proxy=http://10.8.0.1:8080 pip  
pip3 install --user --upgrade --proxy=http://10.8.0.1:8080 virtualenv  
virtualenv ~/virtualenvs/testenv  
source ./virtualenvs/testenv/bin/activate  
pip3 install --upgrade --proxy=http://10.8.0.1:8080 pip  
pip3 install --proxy=http://10.8.0.1:8080 numpy ipython jupyter  
pip3 list - to check installed packages
```

```
qsub <pbs_script.sh>
```

PBS script – detailed discussion

- PBS directives used in the script
- man qsub
 - discuss environment variables
- <https://docs.adaptivecomputing.com/torque/4-0-2/Content/topics/commands/qsub.htm>

Qstat

qstat -an

-a - all jobs

-n - display nodes allocated to jobs

qstat -q

qstat <jobid> -f - full detail

qdel jobid

qdel all

Interactive PBS – Ipython

`Qsub -I <pbs_script.sh>`

Exercise 1

1. Login to the cluster
2. Copy the mysqrt.py file to wd
3. Copy a sample serial PBS script from the HPC website and edit it to calculate the square root of 2.0 with the debug mode on. The output should not be redirected into a file. See where it goes.
4. Launch the script on CPUHPC. Also, try with a different queue.
5. Launch the script on GPUHPC. Take care that the queues are different on the two clusters.
6. Also, try using the interactive mode in qsub
7. Try qstat options, and pbsnodes to check node mapping.
8. Copy the output file and error files back to your PC
9. Added challenge: can you write a bash conditional in your PBS script that runs the code only if the job is sent to n121. Use `-I nodes=n121.cluster.iitmandi.ac.in` to test it.

Exercise 2 - advanced

- Copy the mysqrt.py module to the HPC home
- Write a python script called root.py that calculates the sqrt of number using mysqrt module and prints it. The number should be passed to the script through bash using sys.argv variable
- Write a bash loop that calls root.py for the first 1000 even numbers, i.e. 2,4,6,...
- The results should be appended into a file in the following format:
 - Sqrt of 2 is 1.414xxx, time taken xx seconds
 - Sqrt of 4 is 2.0, time taken xx seconds
- Run the bash loop over the HPC cluster in the serial queue

HPSC 101 — Lecture 7

This lecture:

- Python debugging demo
- Compiled languages
- Introduction to Fortran 90 syntax
- Declaring variables, loops, booleans

Compiled vs. interpreted language

Not so much a feature of language syntax as of how language is converted into machine instructions.

Many languages use elements of both.

Interpreter:

- Takes commands one at a time, converts into machine code, and executes.
- Allows interactive programming at a shell prompt, as in Python or Matlab.
- Can't take advantage of optimising over a entire program — does not know what instructions are coming next.
- Must translate each command while running the code, possibly many times over in a loop.

Compiled language

The program must be written in 1 or more files ([source code](#)).

These files are input data for the [compiler](#), which is a computer program that analyzes the source code and converts it into [object code](#).

Compiled language

The program must be written in 1 or more files ([source code](#)).

These files are input data for the [compiler](#), which is a computer program that analyzes the source code and converts it into [object code](#).

The object code is then passed to a [linker](#) or [loader](#) that turns one or more objects into an [executable](#).

Compiled language

The program must be written in 1 or more files ([source code](#)).

These files are input data for the [compiler](#), which is a computer program that analyzes the source code and converts it into [object code](#).

The object code is then passed to a [linker](#) or [loader](#) that turns one or more objects into an [executable](#).

Why two steps?

Object code contains [symbols](#) such as variables that may be defined in other objects. Linker resolves the symbols and converts them into addresses in memory.

Compiled language

The program must be written in 1 or more files ([source code](#)).

These files are input data for the [compiler](#), which is a computer program that analyzes the source code and converts it into [object code](#).

The object code is then passed to a [linker](#) or [loader](#) that turns one or more objects into an [executable](#).

Why two steps?

Object code contains [symbols](#) such as variables that may be defined in other objects. Linker resolves the symbols and converts them into addresses in memory.

Often large programs consist of many separate files and/or library routines — don't want to re-compile them all when only one is changed. (Later we'll use [Makefiles](#).)

Fortran history

Prior to Fortran, programs were often written in machine code or assembly language.

FORTAN = FORmula TRANslator

Fortran history

Prior to Fortran, programs were often written in machine code or assembly language.

FORTAN = FORmula TRANslator

Fortran I: 1954–57, followed by Fortran II, III, IV, Fortran 66.

Major changes in Fortran 77, which is still widely used.

Fortran history

Prior to Fortran, programs were often written in machine code or assembly language.

FORTRAN = FORmula TRANslator

Fortran I: 1954–57, followed by Fortran II, III, IV, Fortran 66.

Major changes in Fortran 77, which is still widely used.

“I don’t know what the language of the year 2000 will look like, but I know it will be called Fortran.”

– Tony Hoare, 1982

Fortran history

Major changes again from Fortran 77 to Fortran 90.

Fortran 95: minor changes.

Fortran 2003, 2008: not fully implemented by most compilers.

We will use Fortran 90/95.

gfortran — GNU open source compiler

Several commercial compilers also available.

Fortran syntax

Big differences between Fortran 77 and Fortran 90/95.

Fortran 77 still widely used:

- Legacy codes (written long ago, millions of lines...)
- Faster for some things.

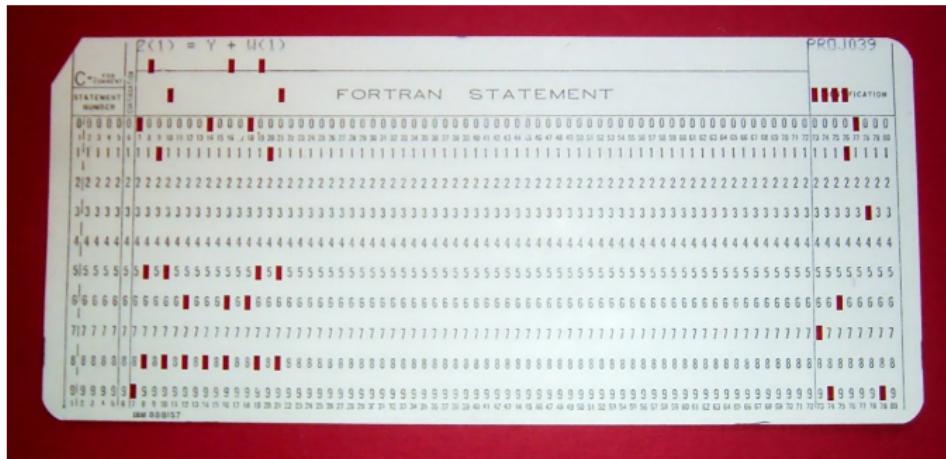
Note: In general adding more high-level programming features to a language makes it harder for compiler to optimize into fast-running code.

Fortran syntax

One big difference: Fortran 77 (and prior versions) required **fixed format** of lines:

Executable statements must start in column 7 or greater,

Only the first 72 columns are used, the rest ignored!



<http://en.wikipedia.org/wiki/File:FortranCardPROJ039.agr.jpg>

Punch cards and decks



<http://en.wikipedia.org/wiki/File:PunchCardDecks.agr.jpg>

Paper tape



http://en.wikipedia.org/wiki/Punched_tape

Fortran syntax

Fortran 90: free format.

Indentation is optional (but highly recommended).

[gfortran](#) will compile Fortran 77 or 90/95.

Use file extension .f for fixed format (column 7 ...)

Use file extension .f90 for free format.

Simple Fortran program

```
! example1.f90
program example1
    implicit none
    real (kind=8) :: x,y,z

    x = 3.d0
    y = 1.d-1
    z = x + y
    print *, "z = ", z
end program example1
```

Notes:

- Indentation optional (but make it readable!)
- First **declaration of variables** then **executable statements**
- **implicit none** means all variables must be declared

Simple Fortran program

```
! example1.f90
program example1
    implicit none
    real (kind=8) :: x,y,z

    x = 3.d0
    y = 1.d-1
    z = x + y
    print *, "z = ", z
end program example1
```

More notes:

- (kind = 8) means 8-bytes used for storage,
- 3.d0 means 3×10^0 in double precision (8 bytes)
- 2.d-1 means $2 \times 10^{-1} = 0.2$

Simple Fortran program

```
! example1.f90
program example1
    implicit none
    real (kind=8) :: x,y,z

    x = 3.d0
    y = 2.d-1
    z = x + y
    print *, "z = ", z
end program example1
```

More notes:

- `print *, ...`: The `*` means no special format specified
As a result all available digits of `z` will be printed.
- Later will see how to specify print format.

Compiling and running Fortran

Suppose `example1.f90` contains this program.

Then:

```
$ gfortran example1.f90
```

compiles and links and creates an **executable** named `a.out`

To run the code after compiling it:

```
$ ./a.out
z =      3.200000000000000
```

The command `./a.out` executes this file (in the current directory).

Compiling and running Fortran

Can give executable a different name with `-o` flag:

```
$ gfortran example1.f90 -o example1.exe
$ ./example1.exe
z =      3.20000000000000
```

Can separate compile and link steps:

```
$ gfortran -c example1.f90 # creates example1.o

$ gfortran example1.o -o example1.exe
$ ./example1.exe
z =      3.20000000000000
```

This creates and then uses the object code `example1.o`.

Compile-time errors

Introduce an error in the code: (zz instead of z)

```
program example1
    implicit none
    real (kind=8) :: x,y,z
    x = 3.d0
    y = 2.d-1
    zz = x + y
    print *, "z = ", z
end program example1
```

This gives an error when compiling:

```
$ gfortran example1.f90
example1.f90:11.6:
zz = x + y
1
Error: Symbol 'zz' at (1) has no IMPLICIT type
```

Without the “implicit none”

Introduce an error in the code: (`zz` instead of `z`)

```
program example1
    real (kind=8) :: x,y,z
    x = 3.d0
    y = 2.d-1
    zz = x + y
    print *, "z = ", z
end program example1
```

This compiles fine and gives the result:

```
$ gfortran example1.f90
$ ./a.out
z = -3.626667641771191E-038
```

Or some other **random nonsense** since `z` was never set.

Fortran types

Variables refer to particular storage location(s), must declare variable to be of a particular type and this won't change.

The statement

```
implicit none
```

means all variables must be explicitly declared.

Otherwise you can use a variable without prior declaration and the type will depend on what letter the name starts with.

Default:

- integer if starts with i, j, k, l, m, n
- real (kind=4) otherwise (single precision)

Many older Fortran codes use this convention!

Much safer to use implicit none for clarity,
and to help avoid typos.

Fortran arrays and loops

```
! loop1.f90
program loop1
    implicit none
    integer, parameter :: n = 10000
    real (kind=8), dimension(n) :: x, y
    integer :: i

    do i=1,n
        x(i) = 3.d0 * i
    enddo

    do i=1,n
        y(i) = 2.d0 * x(i)
    enddo

    print *, "Last y computed: ", y(n)
end program loop1
```

Fortran arrays and loops

```
program loop1
    implicit none
    integer, parameter :: n = 10000
    real (kind=8), dimension(n) :: x, y
    integer :: i
```

Comments:

- `integer, parameter` means this value will not be changed.
- `dimension(n) :: x, y` means these are arrays of length `n`.

Fortran arrays and loops

```
do i=1,n  
    x(i) = 3.d0 * i  
enddo
```

Comments:

- $x(i)$ means i 'th element of array.
- Instead of `enddo`, can also use labels...

```
do 100 i=1,n  
    x(i) = 3.d0 * i  
100      continue
```

The number 100 is arbitrary. Useful for long loops.
Often seen in older codes.

Fortran if-then-else

```
! ifelse1.f90

program ifelse1
    implicit none
    real(kind=8) :: x
    integer :: i

    i = 3

    if (i<=2) then
        print *, "i is less or equal to 2"
    else if (i/=5) then
        print *, "i is greater than 2, not equal to 5"
    else
        print *, "i is equal to 5"
    endif
end program ifelse1
```

Fortran if-then-else

Booleans: .true. .false.

Comparisons:

< or .lt. <= or .le.

> or .gt. >= or .ge.

== or .eq. /= or .ne.

Examples:

```
if ((i >= 5) .and. (i < 12)) then
```

```
if (((i .lt. 5) .or. (i .ge. 12)) .and. &
(i .ne. 20)) then
```

Note: & is the Fortran continuation character.

Statement continues on next line.

Fortran if-then-else

```
! boolean1.f90
program boolean1
    implicit none
    integer :: i,k
    logical :: ever_zero

    ever_zero = .false.
    do i=1,10
        k = 3*i - 1
        ever_zero = (ever_zero .or. (k == 0))
    enddo

    if (ever_zero) then
        print *, "3*i - 1 takes the value 0 for some i"
    else
        print *, "3*i - 1 is never 0 for i tested"
    endif
end program boolean1
```

HPSC 101 — Lecture 8

This lecture:

- Fortran subroutines and functions
- Arrays
- Dynamic memory

Fortran functions and subroutines

For now, assume we have a single file `filename.f90` that contains the main program and also any functions or subroutines needed.

Later we will see how to split into separate files.

Will also discuss use of [modules](#).

Fortran functions and subroutines

For now, assume we have a single file `filename.f90` that contains the main program and also any functions or subroutines needed.

Later we will see how to split into separate files.

Will also discuss use of **modules**.

Functions take some input arguments and return a single value.

Usage: $y = f(x)$ or $z = g(x, y)$

Should be declared as **external** with the type of value returned:

```
real(kind=8), external :: f
```

Fortran functions

```
1 ! $UWHPSC/codes/fortran/fcn1.f90
2
3 program fcn1
4     implicit none
5     real(kind=8) :: y,z
6     real(kind=8), external :: f
7
8     y = 2.
9     z = f(y)
10    print *, "z = ",z
11 end program fcn1
12
13 real(kind=8) function f(x)
14     implicit none
15     real(kind=8), intent(in) :: x
16     f = x**2
17 end function f
```

Prints out: z = 4.000000000000000

Fortran subroutines

Subroutines have arguments, each of which might be for input or output or both.

Usage: call sub1(x,y,z,a,b)

Can specify the **intent** of each argument, e.g.

```
real(kind=8), intent(in) :: x,y  
real(kind=8), intent(out) :: z  
real(kind=8), intent(inout) :: a,b
```

specifies that x, y are passed in and not modified,
z may not have a value coming in but will be set by sub1,
a, b are passed in and may be modified.

After this call, z, a, b may all have changed.

Fortran subroutines

```
1 ! $UWHPSC/codes/fortran/sub1.f90
2
3 program sub1
4     implicit none
5     real(kind=8) :: y,z
6
7     y = 2.
8     call fsub(y,z)
9     print *, "z = ",z
10    end program sub1
11
12 subroutine fsub(x,f)
13     implicit none
14     real(kind=8), intent(in) :: x
15     real(kind=8), intent(out) :: f
16     f = x**2
17 end subroutine fsub
```

Fortran subroutines

A version that takes an array as input and squares each value:

```
1 ! $UWHPSC/codes/fortran/sub2.f90
2
3 program sub2
4   implicit none
5   real(kind=8), dimension(3) :: y,z
6   integer n
7
8   y = (/2., 3., 4./)
9   n = size(y)
10  call fsub(y,n,z)
11  print *, "z = ",z
12 end program sub2
13
14 subroutine fsub(x,n,f)
15 ! compute f(x) = x**2 for all elements of the array x
16 ! of length n.
17 implicit none
18 integer, intent(in) :: n
19 real(kind=8), dimension(n), intent(in) :: x
20 real(kind=8), dimension(n), intent(out) :: f
21 f = x**2
22 end subroutine fsub
```

Array operations in Fortran

Fortran 90 supports some operations on arrays...

```
! $UWHPSC/codes/fortran/vectorops.f90
program vectorops
    implicit none
    real(kind=8), dimension(3) :: x, y

    x = (/10.,20.,30./)          ! initialize
    y = (/100.,400.,900./)

    print *, "x = "
    print *, x

    print *, "x**2 + y = "
    print *, x**2 + y           ! componentwise
```

Array operations in Fortran

```
! $UWHPSC/codes/fortran/vectorops.f90
! continued...

print *, "x*y = "
print *, x*y           ! = (x(1)y(1), x(2)y(2), ...)

print *, "sqrt(y) = "
print *, sqrt(y)          ! componentwise

print *, "dot_product(x,y) = "
print *, dot_product(x,y)    ! scalar product

end program vectorops
```

Array operations in Fortran — Matrices

```
! $UWHPSC/codes/fortran/arrayops.f90
program arrayops
    implicit none
    real(kind=8), dimension(3,2) :: a
    ...
    ! create a as 3x2 array:
    A = reshape((/1,2,3,4,5,6/), (/3,2/))
```

Note:

- Fortran is case insensitive: `A = a`
- Reshape fills array by **columns**, so

$$A = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}.$$

Array operations in Fortran — Matrices

```
! $UWHPSC/codes/fortran/arrayops.f90 (continued)
real(kind=8), dimension(3,2) :: a
real(kind=8), dimension(2,3) :: b
real(kind=8), dimension(3,3) :: c
integer :: i

print *, "a = "
do i=1,3
    print *, a(i,:)
    ! i'th row
enddo

b = transpose(a)
    ! 2x3 array

c = matmul(a,b)
    ! 3x3 matrix product
```

Array operations in Fortran — Matrices

```
! $UWHPSC/codes/fortran/arrayops.f90 (continued)
real(kind=8), dimension(3,2) :: a
real(kind=8), dimension(2) :: x
real(kind=8), dimension(3) :: y

x = (/5, 6/)
y = matmul(a,x)      ! matrix-vector product
print *, "x = ", x
print *, "y = ", y
```

Linear systems in Fortran

There is no equivalent of the Matlab backslash operator for solving a linear system $Ax = b$ ($b = A \backslash b$)

Must call a library subroutine to solve a system.

Later we will see how to use [LAPACK](#) routines
for this.

Note: Under the hood, Matlab calls LAPACK too!

So does NumPy.

Array storage

Rank 1 arrays have a single index, for example:

```
real(kind=8) :: x(3)  
real(kind=8), dimension(3) :: x
```

are equivalent ways to define `x` with elements
`x(1)`, `x(2)`, `x(3)`.

You can also specify a different starting index:

```
real(kind=8) :: x(0:2), y(4:6), z(-2:0)
```

These are all arrays of length 3 and this would be a valid assignment:

```
y(5) = z(-2)
```

Multi-dimensional array storage

Memory can be thought of linear, indexed by a single address.

A one-dimensional array of length N will generally occupy N consecutive memory locations: $8N$ bytes for floats.

A two-dimensional array (e.g. matrix) of size $m \times n$ will require mn memory locations.

Multi-dimensional array storage

Memory can be thought of linear, indexed by a single address.

A one-dimensional array of length N will generally occupy N consecutive memory locations: $8N$ bytes for floats.

A two-dimensional array (e.g. matrix) of size $m \times n$ will require mn memory locations.

Might be stored **by rows**, e.g. first row, followed by second row, etc.

This what's done in **Python or C**, as suggested by notation:

```
A = [[10, 20, 30], [40, 50, 60]]
```

Multi-dimensional array storage

Memory can be thought of linear, indexed by a single address.

A one-dimensional array of length N will generally occupy N consecutive memory locations: $8N$ bytes for floats.

A two-dimensional array (e.g. matrix) of size $m \times n$ will require mn memory locations.

Might be stored **by rows**, e.g. first row, followed by second row, etc.

This what's done in **Python or C**, as suggested by notation:

```
A = [[10, 20, 30], [40, 50, 60]]
```

Or, could be stored **by columns**, as done in **Fortran!**

Multi-dimensional array storage

$$A_{\text{Py}} = \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{bmatrix} \quad A_{\text{Fort}} = \begin{bmatrix} 10 & 30 & 50 \\ 20 & 40 & 60 \end{bmatrix}$$

```
Apy = reshape(array([10,20,30,40,50,60]), (2,3))  
Afort = reshape((/10,20,30,40,50,60/), (/2,3/))
```

Suppose the array storage starts at memory location 3401.

In Python or Fortran, the elements will be stored in the order:

loc 3401	Apy[0,0] = 10	Afort(1,1) = 10
loc 3402	Apy[0,1] = 20	Afort(2,1) = 20
loc 3403	Apy[0,2] = 30	Afort(1,2) = 30
loc 3404	Apy[1,0] = 40	Afort(2,2) = 40
loc 3405	Apy[1,1] = 50	Afort(1,3) = 50
loc 3406	Apy[1,2] = 60	Afort(2,3) = 60

Aside on np.reshape

The `np.reshape` method can go through data in either order:

```
>>> v = linspace(10,60,6)
```

```
>>> v
array([ 10.,  20.,  30.,  40.,  50.,  60.])

>>> reshape(v, (2,3))      # order='C' by default
array([[ 10.,  20.,  30.],
       [ 40.,  50.,  60.]])
```



```
>>> reshape(v, (2,3), order='F')
array([[ 10.,  30.,  50.],
       [ 20.,  40.,  60.]])
```

Aside on np.reshape

The `np.reshape` method can go through data in either order:

```
>>> A  
array([[ 10.,  20.,  30.],  
       [ 40.,  50.,  60.]])  
  
>>> A.reshape(3,2)      # order='C' by default  
array([[ 10.,  20.],  
       [ 30.,  40.],  
       [ 50.,  60.]])  
  
>>> A.reshape((3,2),order='F')  
array([[ 10.,  50.],  
       [ 40.,  30.],  
       [ 20.,  60.]])
```

Note: `reshape` can be called as function or method of A...

```
>>> reshape(A, (3,2), order='F')
```

Aside on np.flatten

The `np.flatten` method converts an N-dim array to a 1-dimensional one:

```
>>> A = np.array([[10.,20,30],[40,50,60]])  
>>> A  
array([[ 10.,   20.,   30.],  
       [ 40.,   50.,   60.]])  
  
>>> A.flatten()      # Default is 'C'  
array([ 10.,   20.,   30.,   40.,   50.,   60.])  
  
>>> A.flatten('F') # Fortran ordering  
array([ 10.,   40.,   20.,   50.,   30.,   60.])
```

Memory management for arrays

Often a program needs to be written to handle arrays whose size is not known until the program is running.

Fortran 77 approaches:

- Allocate arrays large enough for any application,
- Use “work arrays” that are partitioned into pieces.

We will look at some examples from LAPACK since you will probably see this in other software!

Memory management for arrays

Often a program needs to be written to handle arrays whose size is not known until the program is running.

Fortran 77 approaches:

- Allocate arrays large enough for any application,
- Use “work arrays” that are partitioned into pieces.

We will look at some examples from LAPACK since you will probably see this in other software!

The good news:

Fortran 90 allows dynamic memory allocation.

Memory allocation

```
real(kind=8) dimension(:), allocatable :: x
real(kind=8) dimension(:,:), allocatable :: a

allocate(x(10))
allocate(a(30,10))

! use arrays...

! then clean up:
deallocate(x)
deallocate(a)
```

Memory allocation

If you might run out of memory, use optional argument `stat` to return the status...

```
real(kind=8), dimension(:,:,:), allocatable :: a  
  
allocate(a(30000,10000), stat=alloc_error)  
  
if (alloc_error /= 0) then  
    print *, "Insufficient memory"  
    stop  
endif
```

Passing arrays to subroutines — code with bug

```
1 ! $CLASSHG/codes/fortran/arraypassing1.f90
2
3 program arraypassing1
4
5     implicit none
6     real(kind=8) :: x,y
7     integer :: i,j
8
9     x = 1.
10    y = 2.
11    i = 3
12    j = 4
13    call setvals(x)
14    print *, "x = ",x
15    print *, "y = ",y
16    print *, "i = ",i
17    print *, "j = ",j
18
19 end program arraypassing1
20
21 subroutine setvals(a)
22     ! subroutine that sets values in an array a of Length 3.
23     implicit none
24     real(kind=8), intent(inout) :: a(3)
25     integer i
26     do i = 1,3
27         a(i) = 5.
28     enddo
29 end subroutine setvals
```

Note: x is a scalar, setvals dummy argument a is an array.

Passing arrays to subroutines

The `call setvals(x)` statement passes the **address** where `x` is stored.

In the subroutine, the array `a` of length 3 is assumed to start at this address. So next $3 \times 8 = 24$ bytes are assumed to be elements of `a(1:3)`.

In fact these 24 bytes are occupied by

- x. (8 bytes),
- y. (8 bytes),
- i. (4 bytes),
- j. (4 bytes).

So setting `a(1:3)` changes all these variables!

Passing arrays to subroutines

This produces:

```
x =      5.000000000000000
y =      5.000000000000000
i =      1075052544
j =          0
```

Nasty!!

- The storage location of `x` and the next 2 storage locations were all set to the floating point value `5.0e0`
- This messed up the values originally stored in `y`, `i`, `j`.
- Integers are stored differently than floats. Two integers take up 8 bytes, the same as one float, so the assignment `a(3) = 5.` overwrites both `i` and `j`.
- The first half of the float `5.`, when interpreted as an integer, is huge.

Passing arrays to subroutines — another bug

```
1 ! $CLASSHG/codes/fortran/arraypassing2.f90
2
3 program arraypassing2
4
5     implicit none
6     real(kind=8) :: x,y
7     integer :: i,j
8
9     x = 1.
10    y = 2.
11    i = 3
12    j = 4
13    call setvals(x)
14    print *, "x = ",x
15    print *, "y = ",y
16    print *, "i = ",i
17    print *, "j = ",j
18
19 end program arraypassing2
20
21 subroutine setvals(a)
22     ! subroutine that sets values in an array a of length 1000.
23     implicit none
24     real(kind=8), intent(inout) :: a(1000)
25     integer i
26     do i = 1,1000
27         a(i) = 5.
28     enddo
29 end subroutine setvals
```

Note: We now try to set 1000 elements in memory!

Passing arrays to subroutines

This compiles fine, but running it gives:

Segmentation fault

This means that the program tried to change a value of memory it was not allowed to.

Only a small amount of memory is devoted to the variables declared.

The memory we tried to access might be where the program itself is stored, or something related to another program that's running.

Segmentation faults

Debugging segmentation faults can be difficult.

Tip: Compile using `-fbounds-check` flag of gfortran.

This catches **some** cases when you try to access an array out of bounds.

But not the case just shown! The variable was passed to a subroutine that doesn't know how long the array should be.

For a case where this helps, see
`$UWHPSC/codes/fortran/segfault1.f90`

HPSC 101 — Lecture 9

This lecture:

- Multi-file Fortran codes
- Makefiles

Reading:

[Software Carpentry lectures on Make](#)

Dependency checking

Makefiles give a way to recompile only the parts of the code that have changed.

Also used for checking dependencies in other build systems, e.g. creating figures, running latex, bibtex, etc. to construct a manuscript.

Dependency checking

Makefiles give a way to recompile only the parts of the code that have changed.

Also used for checking dependencies in other build systems, e.g. creating figures, running latex, bibtex, etc. to construct a manuscript.

More modern build systems are available, e.g. [SCons](#), which allows expressing dependencies and build commands in Python.

But [make](#) (or [gmake](#)) are still widely used.

Fortran code with 3 units

```
1 ! $UWHPSC/codes/fortran/multifile1/fullcode.f90
2
3 program demo
4     print *, "In main program"
5     call sub1()
6     call sub2()
7 end program demo
8
9 subroutine sub1()
10    print *, "In sub1"
11 end subroutine sub1
12
13 subroutine sub2()
14    print *, "In sub2"
15 end subroutine sub2
```

Split code into 3 separate files...

```
1 ! $UWHPSC/codes/fortran/multifile1/main.f90
2
3 program demo
4     print *, "In main program"
5     call sub1()
6     call sub2()
7 end program demo
```

```
1 ! $UWHPSC/codes/fortran/multifile1/sub1.f90
2
3 subroutine sub1()
4     print *, "In sub1"
5 end subroutine sub1
```

```
1 ! $UWHPSC/codes/fortran/multifile1/sub2.f90
2
3 subroutine sub2()
4     print *, "In sub2"
5 end subroutine sub2
```

Splitting Fortran codes into files

Compile all three and link together into single executable:

```
$ gfortran main.f90 sub1.f90 sub2.f90 \
-o main.exe
```

Run the executable:

```
$ ./main.exe
In main program
In sub1
In sub2
```

Splitting Fortran codes into files

Can split into separate compile....

```
$ gfortran -c main.f90 sub1.f90 sub2.f90  
  
$ ls *.o  
main.o  sub1.o  sub2.o
```

... and link steps:

```
$ gfortran main.o sub1.o sub2.o -o main.exe  
  
$ ./main.exe > output.txt
```

Note: Redirected output to a text file.

Splitting Fortran codes into files

Advantage: If we modify sub2.f90 to print "Now in sub2" we only need to recompile this piece:

```
$ gfortran -c sub2.f90
```

```
$ gfortran main.o sub1.o sub2.o -o main.exe
```

```
$ ./main.exe
In main program
In sub1
Now in sub2
```

When working on a big code (e.g. 100,000 lines split between 200 subroutines) this can make a big difference!

Splitting Fortran codes into files

Advantage: If we modify sub2.f90 to print "Now in sub2" we only need to recompile this piece:

```
$ gfortran -c sub2.f90
```

```
$ gfortran main.o sub1.o sub2.o -o main.exe
```

```
$ ./main.exe
In main program
In sub1
Now in sub2
```

When working on a big code (e.g. 100,000 lines split between 200 subroutines) this can make a big difference!

Use of **Makefiles** greatly simplifies this.

Makefiles

A common way of automating software builds and other complex tasks with dependencies.

A Makefile is itself a program in a special language.

```
1 # $UWHPSC/codes/fortran/multifile1/Makefile
2
3 output.txt: main.exe
4     ./main.exe > output.txt
5
6 main.exe: main.o sub1.o sub2.o
7     gfortran main.o sub1.o sub2.o -o main.exe
8
9 main.o: main.f90
10    gfortran -c main.f90
11 sub1.o: sub1.f90
12    gfortran -c sub1.f90
13 sub2.o: sub2.f90
14    gfortran -c sub2.f90
```

Makefiles

```
$ cd $UWHPSC/codes/fortran/multifile1  
$ rm -f *.o *.exe    # remove old versions  
  
$ make main.exe  
gfortran -c main.f90  
gfortran -c sub1.f90  
gfortran -c sub2.f90  
gfortran main.o sub1.o sub2.o -o main.exe
```

Uses commands for making `main.exe`.

note: First had to make all the `.o` files.
Then executed the rule to make `main.exe`

Structure of a Makefile

Typical element in the simple Makefile:

```
target: dependencies
<TAB> command(s) to make target
```

Important to use tab character, not spaces!!

Warning: Some editors replace tabs with spaces!

Typing “make target” means:

- ① Make sure all the dependencies are up to date (those that are also targets)
- ② If target is **older** than any dependency, **recreate** it using the specified commands.

Structure of a Makefile

Typical element in the simple Makefile:

```
target: dependencies
<TAB> command(s) to make target
```

Important to use tab character, not spaces!!

Warning: Some editors replace tabs with spaces!

Typing “make target” means:

- ① Make sure all the dependencies are up to date (those that are also targets)
- ② If target is **older** than any dependency, **recreate** it using the specified commands.

These rules are applied **recursively**!

Make examples

```
$ rm -f *.o *.exe
```

```
$ make sub1.o  
gfortran -c sub1.f90
```

```
$ make main.o  
gfortran -c main.f90
```

```
$ make main.exe  
gfortran -c sub2.f90  
gfortran main.o sub1.o sub2.o -o main.exe
```

Note: Last make required compiling sub2.f90
but **not** sub1.f90 or main.f90.

Age of dependencies

The last modification time of the file is used.

```
$ ls -l sub1.*  
-rw-r--r-- 1 rjl staff 111 Apr 18 16:05 sub1.f90  
-rw-r--r-- 1 rjl staff 936 Apr 18 16:56 sub1.o
```

```
$ make sub1.o  
make: 'sub1.o' is up to date.
```

```
$ touch sub1.f90; ls -l sub1.f90  
-rw-r--r-- 1 rjl staff 111 Apr 18 17:10 sub1.f90
```

```
$ make main.exe  
gfortran -c sub1.f90  
gfortran main.o sub1.o sub2.o -o main.exe
```

Makefiles

First version of Makefile has 3 rules that are very similar

```
1 # $UWHPSC/codes/fortran/multifile1/Makefile
2
3 output.txt: main.exe
4     ./main.exe > output.txt
5
6 main.exe: main.o sub1.o sub2.o
7     gfortran main.o sub1.o sub2.o -o main.exe
8
9 main.o: main.f90
10    gfortran -c main.f90
11 sub1.o: sub1.f90
12    gfortran -c sub1.f90
13 sub2.o: sub2.f90
14    gfortran -c sub2.f90
```

Replace these with a **pattern** rule...

Implicit rules

General rule to make the .o file from .f90 file:

```
1 # $UWHPSC/codes/fortran/multifile1/Makefile2
2
3 output.txt: main.exe
4     ./main.exe > output.txt
5
6 main.exe: main.o sub1.o sub2.o
7     gfortran main.o sub1.o sub2.o -o main.exe
8
9 %.o : %.f90
10    gfortran -c $<
```

Making `main.exe` requires `main.o` `sub1.o` `sub2.o`
to be up to date.

Rather than a rule to make each one separately,
the implicit rule (lines 9-10) is used for all three.

Specifying a different makefile

To use a makefile with a different name than `Makefile`:

```
$ make sub1.o -f Makefile2  
gfortran -c sub1.f90
```

The rules in `Makefile2` will be used.

The directory `$UWHPSC/codes/fortran/multifile1`
contains several sample makefiles.

See [class notes: Makefiles](#) for a summary.

Implicit rules

We have to repeat the list of .o files twice:

```
1 # $UWHPSC/codes/fortran/multifile1/Makefile2
2
3 output.txt: main.exe
4     ./main.exe > output.txt
5
6 main.exe: main.o sub1.o sub2.o
7     gfortran main.o sub1.o sub2.o -o main.exe
8
9 %.o : %.f90
10    gfortran -c $<
```

Simplify and reduce errors by defining a macro.

Makefile variables or macros

```
1 # $UWHPSC/codes/fortran/multifile1/Makefile3
2
3 OBJECTS = main.o sub1.o sub2.o
4
5 output.txt: main.exe
6     ./main.exe > output.txt
7
8 main.exe: $(OBJECTS)
9     gfortran $(OBJECTS) -o main.exe
10
11 %.o : %.f90
12     gfortran -c $<
```

By convention, all-caps names are used for Makefile macros.

Note that to use `OBJECTS` we must write `$ (OBJECTS)`.

Makefile variables

```
1 # $UWHPSC/codes/fortran/multifile1/Makefile4
2
3 FC = gfortran
4 FFLAGS = -O3
5 LFLAGS =
6 OBJECTS = main.o sub1.o sub2.o
7
8 output.txt: main.exe
9     ./main.exe > output.txt
10
11 main.exe: $(OBJECTS)
12     $(FC) $(LFLAGS) $(OBJECTS) -o main.exe
13
14 %.o : %.f90
15     $(FC) $(FFLAGS) -c $<
```

Here we have added for the name of the Fortran command and for compile flags and linking flags.

Makefile variables

```
$ rm -f *.o *.exe  
$ make -f Makefile4  
  
gfortran -O3 -c main.f90  
gfortran -O3 -c sub1.f90  
gfortran -O3 -c sub2.f90  
gfortran -O3 main.o sub1.o sub2.o -o main.exe  
.main.exe > output.txt
```

Can specify variables on command line:

```
$ rm -f *.o *.exe  
$ make main.exe FFLAGS=-g -f Makefile4  
  
gfortran -g -c main.f90  
gfortran -g -c sub1.f90  
gfortran -g -c sub2.f90  
gfortran -g main.o sub1.o sub2.o -o main.exe
```

Phony targets — don't create files

```
1 # $UWHPSC/codes/fortran/multifile1/Makefile5
2
3 OBJECTS = main.o sub1.o sub2.o
4 .PHONY: clean
5
6 output.txt: main.exe
7     ./main.exe > output.txt
8
9 main.exe: $(OBJECTS)
10    gfortran $(OBJECTS) -o main.exe
11
12 %.o : %.f90
13    gfortran -c $<
14
15 clean:
16    rm -f $(OBJECTS) main.exe
```

Note: No dependencies, so always do commands

```
$ make clean -f Makefile5
rm -f main.o sub1.o sub2.o main.exe
```

Common Makefile error

Using spaces instead of tab...

If we did this in the `clean` commands, we'd get:

```
$ make clean -f Makefile5
```

```
Makefile5:14: *** missing separator. Stop.
```

make help

```
1 # $UWHPSC/codes/fortran/multifile1/Makefile6
2
3 OBJECTS = main.o sub1.o sub2.o
4 .PHONY: clean help
5
6 output.txt: main.exe
7     ./main.exe > output.txt
8
9 main.exe: $(OBJECTS)
10    gfortran $(OBJECTS) -o main.exe
11
12 %.o : %.f90
13    gfortran -c $<
14
15 clean:
16    rm -f $(OBJECTS) main.exe
17
18 help:
19    @echo "Valid targets:"
20    @echo "  main.exe"
21    @echo "  main.o"
22    @echo "  sub1.o"
23    @echo "  sub2.o"
24    @echo "  clean: removes .o and .exe files"
```

`echo` means print out the string.

`@echo` means print out the string but don't print the command.

Fancier things are possible...

```
1 # $UWHPSC/codes/fortran/multifile1/Makefile7
2
3 SOURCES = $(wildcard *.f90)
4 OBJECTS = $(subst .f90,.o,$(SOURCES))
5
6 .PHONY: test
7
8 test:
9     @echo "Sources are: " $(SOURCES)
10    @echo "Objects are: " $(OBJECTS)
```

This gives:

```
$ make test -f Makefile6
Sources are: fullcode.f90 main.f90 sub1.f90 sub2.f90
Objects are: fullcode.o main.o sub1.o sub2.o
```

Note this found `fullcode.f90` too!

HPSC 101 — Lecture 10

Outline:

- Fortran modules
- Newton's method example

Fortran modules

General structure of a module:

```
module <MODULE-NAME>
    ! Declare variables
contains
    ! Define subroutines or functions
end module <MODULE-NAME>
```

A program/subroutine/function can **use** this module:

```
program <NAME>
    use <MODULE-NAME>
    ! Declare variables
    ! Executable statements
end program <NAME>
```

Fortran modules

Can also specify a list of what variables/subroutines/functions from module to be used.

Similar to `from numpy import linspace`
rather than `from numpy import *`

```
program <NAME>
    use <MODULE-NAME>, only: <LIST OF SYMBOLS>
        ! Declare variables
        ! Executable statements
end program <NAME>
```

Makes it easier to see which variables come from each module.

Fortran module example

```
1 ! $UWHPSC/codes/fortran/multifile2/sub1m.f90
2
3 module sub1m
4
5 contains
6
7 subroutine sub1()
8     print *, "In sub1"
9 end subroutine sub1
10
11 end module sub1m
```

```
1 ! $UWHPSC/codes/fortran/multifile2/main.f90
2
3 program demo
4     use sub1m, only: sub1
5     print *, "In main program"
6     call sub1()
7 end program demo
```

Fortran modules

Some uses:

- Can define **global variables** in modules to be used in several different routines.

In Fortran 77 this had to be done with **common blocks** — much less elegant.

- Subroutine/function **interface information** is generated to aid in checking that proper arguments are passed.

It's often best to put all subroutines and functions in modules for this reason.

- Can define new **data types** to be used in several routines. (“derived types” rather than “intrinsic types”)

Compiling Fortran modules

If sub1m.f90 is a module, then compiling it creates sub1m.o and also sub1m.mod:

```
$ gfortran -c sub1m.f90
```

```
$ ls  
main.f90      sub1m.f90      sub1m.mod      sub1m.o
```

the module must be compiled before any subroutine or program that uses it!

```
$ rm -f sub1m.mod  
$ gfortran main.f90 sub1m.f90  
main.f90:5.13:
```

```
use sub1m  
1
```

```
Fatal Error: Can't open module file 'sub1m.mod'  
for reading at (1): No such file or directory
```

Another module example

```
1 ! $UWHPSC/codes/fortran/circles/circle_mod.f90
2
3 module circle_mod
4
5     implicit none
6     real(kind=8), parameter :: pi = 3.141592653589793d0
7
8 contains
9
10    real(kind=8) function area(r)
11        real(kind=8), intent(in) :: r
12        area = pi * r**2
13    end function area
14
15    real(kind=8) function circumference(r)
16        real(kind=8), intent(in) :: r
17        circumference = 2.d0 * pi * r
18    end function circumference
19
20 end module circle_mod
```

Another module example

```
1 ! $UWHPSC/codes/fortran/circles/main.f90
2
3 program main
4
5     use circle_mod, only: pi, area
6     implicit none
7     real(kind=8) :: a
8
9     ! print parameter pi defined in module:
10    print *, 'pi = ', pi
11
12    ! test the area function from module:
13    a = area(2.d0)
14    print *, 'area for a circle of radius 2: ', a
15
16 end program main
```

Running this gives:

```
pi =      3.14159265358979
area for a circle of radius 2:      12.5663706143
```

Module variables

```
1 ! $UWHPSC/codes/fortran/circles/circle_mod.f90
2 ! Version where pi is a module variable.
3
4 module circle_mod
5
6     implicit none
7     real(kind=8) :: pi
8     save
9
10 contains
11
12     real(kind=8) function area(r)
13         real(kind=8), intent(in) :: r
14         area = pi * r**2
15     end function area
16
17     real(kind=8) function circumference(r)
18         real(kind=8), intent(in) :: r
19         circumference = 2.d0 * pi * r
20     end function circumference
21
22 end module circle_mod
```

Module variables

```
1 ! $UWHPSC/codes/fortran/circles/main.f90
2
3 program main
4
5     use circle_mod, only: pi, area
6     implicit none
7     real(kind=8) :: a
8
9     call initialize()    ! sets pi
10
11    ! print module variable pi:
12    print *, 'pi = ', pi
13
14    ! test the area function from module:
15    a = area(2.d0)
16    print *, 'area for a circle of radius 2: ', a
17
18 end program main
```

Module variables

The module variable `pi` should be initialized in a program unit that is called only once.

It can be initialized to full machine precision using

$$\pi = \arccos(-1)$$

```
1 ! $UWHPSC/codes/fortran/circles/initialize.f90
2
3 subroutine initialize()
4
5     ! Set the value of pi used elsewhere.
6     use circle_mod, only: pi
7     pi = acos(-1.d0)
8
9 end subroutine initialize
```

Makefile

```
1 # $UWHPSC/codes/fortran/circles2/Makefile
2
3 OBJECTS = circle_mod.o \
4           main.o \
5           initialize.o
6
7 MODULES = circle_mod.mod
8
9 .PHONY: clean
10
11 output.txt: main.exe
12         ./main.exe > output.txt
13
14 main.exe: $(MODULES) $(OBJECTS)
15         gfortran $(OBJECTS) -o main.exe
16
17 %.o: %.f90
18         gfortran -c $<
19
20 %.mod: %.f90
21         gfortran -c $<
22
23 clean:
24         rm -f $(OBJECTS) $(MODULES) main.exe
```

Fortran subroutines

A version that takes an array as input and squares each value:

```
1 ! $UWHPSC/codes/fortran/sub2.f90
2
3 program sub2
4   implicit none
5   real(kind=8), dimension(3) :: y,z
6   integer n
7
8   y = (/2., 3., 4./)
9   n = size(y)
10  call fsub(y,n,z)
11  print *, "z = ",z
12 end program sub2
13
14 subroutine fsub(x,n,f)
15 ! compute f(x) = x**2 for all elements of the array x
16 ! of length n.
17 implicit none
18 integer, intent(in) :: n
19 real(kind=8), dimension(n), intent(in) :: x
20 real(kind=8), dimension(n), intent(out) :: f
21 f = x**2
22 end subroutine fsub
```

Module version — creates an interface

Now do not need to pass the value `n` into the subroutine.

```
1 ! $UWHPSC/codes/fortran/sub3.f90
2
3 module sub3module
4
5 contains
6
7 subroutine fsub(x,f)
8 ! compute f(x) = x**2 for all elements of the array x.
9 implicit none
10 real(kind=8), dimension(:,), intent(in) :: x
11 real(kind=8), dimension(size(x)), intent(out) :: f
12 f = x**2
13 end subroutine fsub
14
15 end module sub3module
16
17 !-----
18
19 program sub3
20   use sub3module
21   implicit none
22   real(kind=8), dimension(3) :: y,z
23
24   y = (/2., 3., 4./)
25   call fsub(y,z)
26   print *, "z = ", z
27 end program sub3
```

Module for Newton's method

See the [class notes: Fortran example for Newton's method](#)

Parallel programming – motivation and basics

Computer architecture

How fast are computers?

Kilo = thousand (10^3)

Mega = million (10^6)

Giga = billion (10^9)

Tera = trillion (10^{12})

Peta = 10^{15}

Exa = 10^{18}

Processor speeds usually measured in Gigahertz these days.

Hertz means “machine cycles per second”.

One operation may take a few cycles.

So a 1 GHz processor (10^9 cycles per second) can do

> 100,000,000 floating point operations per second

(> 100 Megaflops).

The Cray-1 computer



- World's first “supercomputer”
- Sold to Los Alamos, NCAR, etc. starting in 1976
- Price: up to \$8.8 million

The Cray-1 computer



- World's first “supercomputer”
- Sold to Los Alamos, NCAR, etc. starting in 1976
- Price: up to \$8.8 million
- Speed: 80-100 Mflops
- Memory: 8MB

Overview

High Performance Computing (HPC) often means heavy-duty computing on clusters or supercomputers with 100s of thousands of cores.

“World’s fastest computer”

#1. Sunway TaihuLight (National Supercomputing Centre, China)
10,649,600 ≈ 125 Petaflops



See <http://top500.org> for current list.

How fast are computers?

Not long ago counting **flops** was the best way to measure performance for scientific computing.

How fast are computers?

Not long ago counting **flops** was the best way to measure performance for scientific computing.

Example: Computing matrix-matrix product $C = AB$.

If A and B are $n \times n$ then so is C .

Each element c_{ij} is the inner product of
 i th row of A with j th column of B .

Requires n multiplications and $n - 1$ additions to compute c_{ij} .

How fast are computers?

Not long ago counting **flops** was the best way to measure performance for scientific computing.

Example: Computing matrix-matrix product $C = AB$.

If A and B are $n \times n$ then so is C .

Each element c_{ij} is the inner product of
 i th row of A with j th column of B .

Requires n multiplications and $n - 1$ additions to compute c_{ij} .

n^2 elements in $C \Rightarrow$ Requires $O(n^3)$ floating point ops total.

Note: $n = 10,000 \Rightarrow n^3 = 10^{12}$

($> 1,000$ seconds on 1 GHz processor)

How fast are computers?

Not long ago counting **flops** was the best way to measure performance for scientific computing.

Example: Computing matrix-matrix product $C = AB$.

If A and B are $n \times n$ then so is C .

Each element c_{ij} is the inner product of
ith row of A with jth column of B .

Requires n multiplications and $n - 1$ additions to compute c_{ij} .

n^2 elements in $C \Rightarrow$ Requires $O(n^3)$ floating point ops total.

Note: $n = 10,000 \Rightarrow n^3 = 10^{12}$

(> 1,000 seconds on 1 GHz processor)

**But these days, the bottle neck is often
getting data to and from the processor!**

Note that each element of A , B is used n times.

Memory Hierarchy

(Main) Memory: “Fast” memory that is hopefully large enough to contain all the programs and data currently running.

(But not nearly fast enough to keep up with CPU.)

Typically 2 – 8 GB.

Recall $\text{GB} = \text{gigabyte} = 10^9 \text{ bytes} = 8 \times 10^9 \text{ bits}$.

For example, 1GB holds a single $10,000 \times 10,000$ matrix of floating point values (8 bytes each),
or 125 matrices that are each 1000×1000 .

Hard Drive: Slower memory that contains data (including photos, video, music, etc.) and all programs you might want to use.

Typically 500 GB – 2 TB. (Slower but cheaper.)

32-bit vs. 64-bit architecture

Each byte in memory has an address, which is an integer.
On 32-bit machines, registers can only store

$$2^{32} = 4294967296 \approx 4 \text{ billion distinct addresses}$$

\Rightarrow at most 4GB of memory can be addressed.

32-bit vs. 64-bit architecture

Each byte in memory has an address, which is an integer.
On 32-bit machines, registers can only store

$$2^{32} = 4294967296 \approx 4 \text{ billion distinct addresses}$$

\Rightarrow at most 4GB of memory can be addressed.

Newer machines often have more, leading to the need for 64-bit architectures

$$2^{64} = 1.84 \times 10^{19} \text{ distinct addresses}$$

\Rightarrow could address an exabyte of memory.

32-bit vs. 64-bit architecture

Each byte in memory has an address, which is an integer.
On 32-bit machines, registers can only store

$$2^{32} = 4294967296 \approx 4 \text{ billion distinct addresses}$$

\Rightarrow at most 4GB of memory can be addressed.

Newer machines often have more, leading to the need for 64-bit architectures (8 bytes for addresses).

$$2^{64} = 1.84 \times 10^{19} \text{ distinct addresses}$$

\Rightarrow could address an exabyte of memory.

Note: Integers might still be stored in 4 bytes, for example.

Floats might be either `real (kind=4)` or `real (kind=8)`.

CPU and registers

CPU — central processor unit

Executes instructions such as **add** or **multiply**.

Takes data from **registers**, performs operations, stores back to registers.

Transferring between registers and processor is **very fast**.

Different types of registers, e.g.

- Integer, floating point
- instruction registers
- address registers

Generally a **very small number of registers**.

Data and instructions must be transferred between other memory and registers as needed.

Memory Hierarchy

Between registers and memory there are 2 or 3 levels of **cache**, each larger but slower.

Registers: access time 1 cycle

L1 cache: a few cycles

L2 cache: ~ 10 cycles

(Main) Memory: ~ 250 cycles

Hard drive: 1000s of cycles

Terminology

Latency refers to amount of time it takes to complete a given unit of work.

Throughput refers to the amount of work that can be completed per unit time.

Terminology

Latency refers to amount of time it takes to complete a given unit of work.

Throughput refers to the amount of work that can be completed per unit time.

Exploit parallelism to hide latency and increase throughput.

Even a “single core” machine has lots of things going on at once.

For example:

- Pipelined operations
- Executing / fetching / storing
- Prefetching future instructions
- Prefetching data into cache

5-stage instruction pipeline for RISC machine

Instr No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

IF = Instruction Fetch,

ID = Instruction Decode,

EX = Execute,

MEM = Memory access,

WB = Register write back.

http://en.wikipedia.org/wiki/Instruction_pipeline

Reducing memory latency

Reduce memory fetches by **reusing data in cache as much as possible**. **Requires temporal locality**.

Very simple Python example: if `len(x)` much larger than cache size,

```
z = 0.; w = 0.  
for i in range(len(x)):  
    z = z + x[i]  
for i in range(len(x)):  
    w = w + 3. * x[i]
```

should be rewritten as

```
for i in range(len(x)):  
    z = z + x[i]  
    w = w + 3. * x[i]
```

Reducing memory latency

Reduce memory fetches by reusing data in cache as much as possible. Requires temporal locality.

Very simple Python example: if `len(x)` much larger than cache size,

```
z = 0.; w = 0.  
for i in range(len(x)):  
    z = z + x[i]  
for i in range(len(x)):  
    w = w + 3. * x[i]
```

should be rewritten as

```
for i in range(len(x)):  
    z = z + x[i]  
    w = w + 3. * x[i]
```

Note: Both are bad in Python, use e.g. `z = np.sum(x); w = 3*z`

HPSC 101 — Lecture 11

Outline:

- Cache considerations
- Fortran optimization

Reading:

- S. Goedecker and A. Hoisie, “Performance Optimization of Numerically Intensive Codes”, SIAM, 2001. **ebook edition:**
[http://pubs.siam.org/doi/book/10.1137/1.
9780898718218](http://pubs.siam.org/doi/book/10.1137/1.9780898718218)

Cache lines

When data is brought into cache, more than 1 value is fetched at a time.

A **cache line** typically holds 64 or 128 consecutive bytes (8 or 16 floats).

L1 Cache might hold 1000 cache lines.

Cache miss occurs if the the value you need next is not in cache.

Cache lines

When data is brought into cache, more than 1 value is fetched at a time.

A **cache line** typically holds 64 or 128 consecutive bytes (8 or 16 floats).

L1 Cache might hold 1000 cache lines.

Cache miss occurs if the the value you need next is not in cache.

Another cache line will be brought from higher up the hierarchy, and may displace some variables in cache.

Those cache lines will first have to be written back to memory.

Bottom line: Good to do lots of work on each set of data while in cache, before it has to be written back.

Organize algorithm for **Temporal locality**.

Spatial locality

Also good to organize algorithm so data that is **consecutive in memory** is used together when possible.

If data you need is scattered through memory, many cache lines will be needed and will contain data you don't need.

This is called **spatial locality**.

Multi-dimensional array storage

$$A = \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{bmatrix}$$

```
Apy = reshape(array([10,20,30,40,50,60]), (2,3))  
Afort = reshape((/10,20,30,40,50,60/), (/2,3/))
```

Suppose the array storage starts at memory location 3401.

In Python or Fortran, the elements will be stored in the order:

loc 3401	Apy[0,0] = 10	Afort(1,1) = 10
loc 3402	Apy[0,1] = 20	Afort(2,1) = 20
loc 3403	Apy[0,2] = 30	Afort(1,2) = 30
loc 3404	Apy[1,0] = 40	Afort(2,2) = 40
loc 3405	Apy[1,1] = 50	Afort(1,3) = 50
loc 3406	Apy[1,2] = 60	Afort(2,3) = 60

Memory layout of 1000×3 array in Fortran

Memory location or offset of each array element:

1	1001	2001
2	1002	2002
3	1003	2003
4	1004	2004
5	1005	2005
:	:	:
999	1999	2999
1000	2000	3000

Looping over elements by column steps through memory sequentially (stride = 1)

Looping over elements by row does not.

stride = 1000 in this case since we jump ahead 1000 locations in memory with each step.

Spatial locality

Suppose A is $n \times n$ matrix,

D is $n \times n$ diagonal matrix with diagonal elements d_i .

Compute product $B = DA$ with elements $b_{ij} = d_i a_{ij}$.

Which is better in Python?? Same number of flops!

```
for i in range(n):
    for j in range(n):
        b[i,j] = d[i] * a[i,j]
```

or

```
for j in range(n):
    for i in range(n):
        b[i,j] = d[i] * a[i,j]
```

Spatial locality

Suppose A is $n \times n$ matrix,

D is $n \times n$ diagonal matrix with diagonal elements d_i .

Compute product $B = DA$ with elements $b_{ij} = d_i a_{ij}$.

Which is better in Python?? Same number of flops!

```
for i in range(n):
    for j in range(n):
        b[i,j] = d[i] * a[i,j]
```

or

```
for j in range(n):
    for i in range(n):
        b[i,j] = d[i] * a[i,j]
```

Answer: First one faster in Python (but loops still slow!)

Spatial locality

Compute product $B = DA$ with elements $b_{ij} = d_i a_{ij}$.

Which is better in Fortran?? Same number of flops!

```
do i=1,n  
    do j=1,n  
        b(i,j) = d(i) * a(i,j)  
    enddo; enddo
```

or

```
do j=1,n  
    do i=1,n  
        b(i,j) = d(i) * a(i,j)  
    enddo; enddo
```

Spatial locality

Compute product $B = DA$ with elements $b_{ij} = d_i a_{ij}$.

Which is better in Fortran?? Same number of flops!

```
do i=1,n  
    do j=1,n  
        b(i,j) = d(i) * a(i,j)  
    enddo; enddo
```

or

```
do j=1,n  
    do i=1,n  
        b(i,j) = d(i) * a(i,j)  
    enddo; enddo
```

Answer: Second one faster in Fortran!

Array ordering — which loop is faster?

```
integer, parameter :: m = 4097, n = 10000
real(kind=8), dimension(m,n) :: a

do i = 1,m
    do j=1,n
        a(i,j) = 0.d0
    enddo
enddo

do j = 1,n
    do i=1,m
        a(i,j) = 0.d0
    enddo
enddo
```

Array ordering — which loop is faster?

```
integer, parameter :: m = 4097, n = 10000
real(kind=8), dimension(m,n) :: a

do i = 1,m
    do j=1,n
        a(i,j) = 0.d0
    enddo
enddo

do j = 1,n
    do i=1,m
        a(i,j) = 0.d0
    enddo
enddo
```

First: 1.02 seconds, Second: 0.10 seconds

Much worse if m is high power of 2

```
integer, parameter :: m = 4096, n = 10000
real(kind=8), dimension(m,n) :: a

do i = 1,m
    do j=1,n
        a(i,j) = 0.d0
    enddo
enddo

do j = 1,n
    do i=1,m
        a(i,j) = 0.d0
    enddo
enddo
```

First: 2.4 seconds, Second: 0.19 seconds

More about cache

Simplified model of one level direct mapped cache.

32-bit memory address: 4.3×10^9 addresses

Suppose cache holds $512 = 2^9$ cache lines (9-bit address)

A given memory location cannot go anywhere in cache.

9 low order bits of memory address determine cache address.

More about cache

Simplified model of one level direct mapped cache.

32-bit memory address: 4.3×10^9 addresses

Suppose cache holds $512 = 2^9$ cache lines (9-bit address)

A given memory location cannot go anywhere in cache.

9 low order bits of memory address determine cache address.

For a memory fetch:

- Determine cache address, check if this holds desired words from memory.
- If so, use it.
- If not, check “dirty bit” to see if has been modified since load.
- If so, write to memory before loading new cache line.

Cache collisions

Return to example where matrix has $4096 = 2^{12}$ rows.

Cache line holds 64 bytes = 8 floats. $4096/8 = 512$ cache lines per column of matrix.

Loading one column of matrix will fill up cache lines $0, 1, 2, \dots, 511$.

Second column will go back to cache line 0.

But all elements in cache have been used before this happens,

Prefetching can be done by optimizing compiler.

Cache collisions

Return to example where matrix has $4096 = 2^{12}$ rows.

Cache line holds 64 bytes = 8 floats. $4096/8 = 512$ cache lines per column of matrix.

Loading one column of matrix will fill up cache lines $0, 1, 2, \dots, 511$.

Second column will go back to cache line 0.

But all elements in cache have been used before this happens,

Prefetching can be done by optimizing compiler.

Worse — Going across the rows:

The first 8 elements of column 1 go to cache line 0.

The first 8 elements of column 2 **also map to cache line 0**.

Similarly for all columns. The rest of cache stays empty.

More about cache

If cache holds more lines:

1024 lines \Rightarrow

first 8 bytes of column 1 go to cache line 0,
first 8 bytes of column 2 go to cache line 512,
first 8 bytes of column 3 go to cache line 0,
first 8 bytes of column 4 go to cache line 512.

Still only using 1/512 of cache.

More about cache

If cache holds more lines:

1024 lines \Rightarrow

first 8 bytes of column 1 go to cache line 0,
first 8 bytes of column 2 go to cache line 512,
first 8 bytes of column 3 go to cache line 0,
first 8 bytes of column 4 go to cache line 512.

Still only using 1/512 of cache.

In practice cache is often **set associative**: small number of cache addresses for each memory address.

Padding

Matrix dimensions that are high powers of 2 should usually be avoided.

Even though natural for some algorithms such as FFTs

May be worth declaring larger arrays and only using part of it.

Code optimization

Basic considerations like memory layout should always be kept in mind.

However:

- Also important to consider programmer time.
- Writing readable code is very important in getting program correct.

Code optimization

Basic considerations like memory layout should always be kept in mind.

However:

- Also important to consider programmer time.
- Writing readable code is very important in getting program correct.
- Some optimizations not worth spending time on.
- Often best to first get code working properly and then determine whether optimization is necessary.

“Premature optimization is the root of all evil” (Don Knuth)

Code optimization

Basic considerations like memory layout should always be kept in mind.

However:

- Also important to consider programmer time.
- Writing readable code is very important in getting program correct.
- Some optimizations not worth spending time on.
- Often best to first get code working properly and then determine whether optimization is necessary.

“Premature optimization is the root of all evil” (Don Knuth)

- If so, determine which parts of code need to be improved and spend effort on these sections. (Tools such as [gprof](#))
- Use optimized software such as BLAS, LAPACK.

HPSC — Lecture 12

Outline:

- More about computer arithmetic
- Fortran optimization and compiler flags
- Parallel computing

Reading:

- Optimization flags: <http://gcc.gnu.org/onlinedocs/gcc-3.4.5/gcc/Optimize-Options.html>

Excess precision

Different small values reported when evaluating $f(x)$ for x very close to root.

Try compiling with gfortran flag `-ffloat-store`.

This forces variables to be written out of registers to cache before reusing.

Sometimes registers have more precision than other memory to try to get a bit better accuracy.

Sometimes nice, but can destroy reproducibility.

Floating point real numbers

Base 10 scientific notation:

$$0.2345e-18 = 0.2345 \times 10^{-18} = 0.000000000000000000002345$$

Mantissa: 0.2345, Exponent: -18

Floating point real numbers

Base 10 scientific notation:

$$0.2345e-18 = 0.2345 \times 10^{-18} = 0.0000000000000000000002345$$

Mantissa: 0.2345, Exponent: -18

Binary floating point numbers:

Example: Mantissa: 0.101101, Exponent: -11011 means:

$$\begin{aligned}0.101101 &= 1(2^{-1}) + 0(2^{-2}) + 1(2^{-3}) + 1(2^{-4}) + 0(2^{-5}) + 1(2^{-6}) \\&= 0.703125 \text{ (base 10)}\end{aligned}$$

$$-11011 = -1(2^4) + 1(2^3) + 0(2^2) + 1(2^1) + 1(2^0) = -27 \text{ (base 10)}$$

So the number is

$$0.703125 \times 2^{-27} \approx 5.2386894822120667 \times 10^{-9}$$

Floating point real numbers

Fortran:

real (kind=4): 4 bytes

This used to be standard single precision real

real (kind=8): 8 bytes

This used to be called double precision real

Python float datatype is 8 bytes.

8 bytes = 64 bits,

53 bits for mantissa and 11 bits for exponent (64 bits = 8 bytes).

We can store 52 binary bits of precision.

$2^{-52} \approx 2.2 \times 10^{-16} \Rightarrow$ roughly 15 digits of precision.

Floating point real numbers (8 bytes)

Since $2^{-52} \approx 2.2 \times 10^{-16}$

this corresponds to roughly 15 digits of precision.

We can hope to get at most 15 correct digits in computations.

For example:

```
>>> from numpy import pi  
>>> pi  
3.1415926535897931
```

```
>>> 1000 * pi  
3141.5926535897929
```

Note: storage and arithmetic is done in base 2
Converted to base 10 only when printed!

Absolute and relative error

Let \hat{z} = exact answer to some problem,
 z^* = computed answer using some algorithm.

Absolute error: $|z^* - \hat{z}|$

Relative error: $\frac{|z^* - \hat{z}|}{|\hat{z}|}$

If $|\hat{z}| \approx 1$ these are roughly the same.

But in general relative error is a better measure of
how many correct digits in the answer:

Relative error $\approx 10^{-k} \Rightarrow \approx k$ correct digits.

Precision of floating point

If x a real number then $f\ell(x)$ represents the closest floating point number.

Unless overflow or underflow occurs, this generally has relative error

$$\left| \frac{f\ell(x) - x}{x} \right| \leq E_m$$

where E_m is Machine epsilon.

$E_m \approx 10^{-k} \Rightarrow$ about k correct digits.

8-byte double precision: $E_m \approx 2.22 \times 10^{-16}$.

Machine epsilon (for 8 byte reals)

```
>>> y = 1. + 3.e-16
```

```
>>> y
```

```
1.0000000000000002
```

```
>>> y - 1.
```

```
2.2204460492503131e-16
```

Machine epsilon is the distance between 1.0 and the next largest number that can be represented: $2^{-52} \approx 2.2204 \times 10^{-16}$

```
>>> y = 1 + 1e-16
```

```
>>> y
```

```
1.0
```

```
>>> y == 1
```

```
True
```

Catastrophic cancellation of nearly equal numbers

We generally don't need 16 digits in our solutions
But often need that many digits to get reliable results.

```
>>> from numpy import pi
>>> pi
3.1415926535897931

>>> y = pi * 1.e-10
>>> y
3.1415926535897934e-10

>>> z = 1. + y
>>> z
1.0000000003141594    # 15 digits correct in z

>>> z - 1.
3.141593651889707e-10  # only 6 or 7 digits right!
```

Sample compiler optimizations

See: [http://gcc.gnu.org/onlinedocs/gcc-3.4.5/gcc/
Optimize-Options.html](http://gcc.gnu.org/onlinedocs/gcc-3.4.5/gcc/Optimize-Options.html)

for a list of many gcc optimization flags.

Often -O2 or -O3 flag is used to include many common optimizations.

Global common subexpression elimination

-fgcse (or -O2, -O3) optimization flag will replace:

```
do i=1,n  
    y(i) = 2.d0 * x(i) * pi  
enddo
```

by machine code equivalent of...

```
pi2 = 2.d0 * pi  
do i=1,n  
    y(i) = pi2 * x(i)  
enddo
```

Global common subexpression elimination

-fgcse (or -O2, -O3) optimization flag will replace:

```
do i=1,n  
    y(i) = 2.d0 * x(i) * pi  
enddo
```

by machine code equivalent of...

```
pi2 = 2.d0 * pi  
do i=1,n  
    y(i) = pi2 * x(i)  
enddo
```

Note: May give slightly different results because computer arithmetic is non-commutative!

Sample compiler optimization – inlining functions

-finline-functions (or -O3) optimization flag will replace function calls by the corresponding code inline:

E.g., in .../newton/newton.f90, replace

```
! evaluate function and its derivative:  
fx = f(x)  
fxprime = fp(x)
```

by machine code equivalent of...

```
fx = x**2 - 4.d0  
fxprime = 2.d0*x
```

Sample compiler optimization – inlining functions

-finline-functions (or -O3) optimization flag will replace function calls by the corresponding code inline:

E.g., in .../newton/newton.f90, replace

```
! evaluate function and its derivative:  
fx = f(x)  
fxprime = fp(x)
```

by machine code equivalent of...

```
fx = x**2 - 4.d0  
fxprime = 2.d0*x
```

Overhead of function call is avoided. Can make a big difference if $f(x)$ is evaluated in a loop over large array.

Manual code optimization

Often it is necessary to rethink the algorithm in order to optimize code.

“Premature optimization is the root of all evil” (Don Knuth)

Once code is working, determine which parts of code need to be improved and spend effort on these sections.

Use tools such as [gprof](#) to identify bottlenecks.

Block matrix multiply

Compute $C = AB$. Can partition into blocks:

$$\begin{array}{cc|c} C_{11} & C_{12} & | \\ C_{21} & C_{22} & | \end{array} = \begin{array}{cc|c} A_{11} & A_{12} & | \\ A_{21} & A_{22} & | \end{array} \begin{array}{cc|c} B_{11} & B_{12} & | \\ B_{21} & B_{22} & | \end{array}$$

where

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j}$$

When blocks A_{11} and B_{11} are in cache can compute the $A_{11}B_{11}$ part of $C_{11} = A_{11}B_{11} + A_{12}B_{21}$

Block matrix multiply

Compute $C = AB$. Can partition into blocks:

$$\begin{array}{cc|cc} C_{11} & C_{12} & A_{11} & A_{12} \\ C_{21} & C_{22} & A_{21} & A_{22} \end{array} = \begin{array}{cc|cc} B_{11} & B_{12} & B_{21} & B_{22} \end{array}$$

where

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j}$$

When blocks A_{11} and B_{11} are in cache can compute the $A_{11}B_{11}$ part of $C_{11} = A_{11}B_{11} + A_{12}B_{21}$

Might next bring in B_{12} and compute the $A_{11}B_{12}$ part of

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

Matrix transpose

```
do j=1,n  
    do i=1,n  
        b(j,i) = a(i,j)  
    enddo  
enddo
```

Accessing a by column but b by row.

Switching loop order \Rightarrow accessing a by row!

Better to do by blocks

$$\begin{matrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{matrix}^T = \begin{matrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{matrix}^T = \begin{matrix} A_{11}^T & A_{21}^T \\ A_{12}^T & A_{22}^T \end{matrix}^T$$

Matrix transpose

Suppose stride s divides n . Then can rewrite as:

Strip mining:

```
do jj=1,n,s  
    do j=jj,jj+s-1  
        do ii=1,n,s  
            do i=ii,ii+s-1  
                b(j,i) = a(i,j)
```

Loop reordering:

```
do jj=1,n,s  
    do ii=1,n,s  
        do j=jj,jj+s-1  
            do i=ii,ii+s-1  
                b(j,i) = a(i,j)
```

Loops over blocks in outer loops, within block in inner loops.

CPU time vs. throughput

a, b, beach 1000×1000 matrices. Compare multiply, add

Compare time of $c = \text{matmul}(a, b)$ vs. $c = a+b$.

Compare megaflops per second: $1e-6 * \text{nflops} / (t_2 - t_1)$.

Add: CPU time (sec): 0.00687200

rate: 145.52 megaflop/sec

Multiply: CPU time (sec): 2.38393500 slower

rate: 838.53 megaflop/sec higher

CPU time vs. throughput

a, b each 1000×1000 matrices. Compare multiply, add

Compare time of $c = \text{matmul}(a, b)$ vs. $c = a+b$.

Compare megaflops per second: $1e-6 * \text{nflops} / (t_2 - t_1)$.

Add: CPU time (sec): 0.00687200

rate: 145.52 megaflop/sec

Multiply: CPU time (sec): 2.38393500 slower

rate: 838.53 megaflop/sec higher

For addition: $\text{nflops} = n^2 = O(n^2)$

For multiplication: $\text{nflops} = (2n-1) * n^2 = O(n^3)$,

More flops, but each element is used n times,

\Rightarrow More flops per memory access \Rightarrow higher rate.

Parallel Computing

- Basic concepts
- Shared vs. distributed memory
- OpenMP (shared)
- MPI (shared or distributed)

Some general references

Some books...

P.S. Pacheco, *An Introduction to Parallel Programming*, Elsevier, 2011.

T. Rauber and G. Ruenger, *Parallel Programming For Multicore and Cluster Systems*, Springer, 2010.

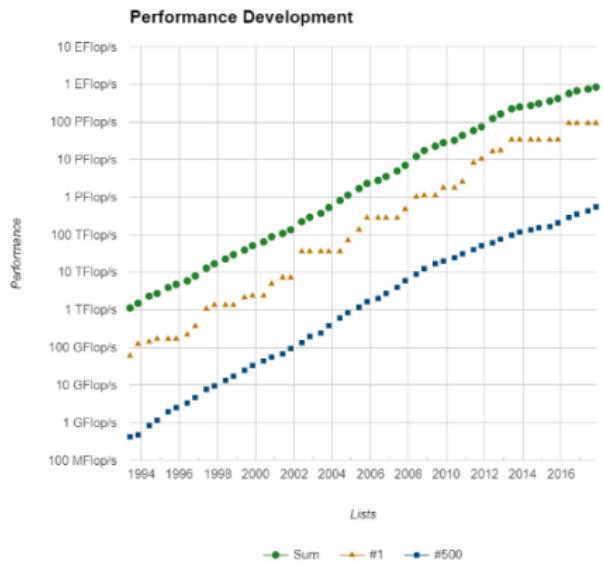
C. Lin and L. Snyder, *Principles of Parallel Programming*, 2008.

L. R. Scott, T. Clark, B. Bagheri, *Scientific Parallel Computing*, Princeton University Press, 2005.

Increasing speed

Moore's Law: Processor speed doubles every 18 months.
⇒ factor of 1024 in 15 years.

Going forward: Number of cores doubles every 18 months.



Top: Total computing power of top 500 computers

Middle: #1 computer

Bottom: #500 computer

<http://www.top500.org>

Parallel processing

Shared memory:

All processors have access to the same memory.

Multicore chip: separate L1 caches, L2 might be shared.

Parallel processing

Shared memory:

All processors have access to the same memory.

Multicore chip: separate L1 caches, L2 might be shared.

Distributed memory:

Each processor has its own memory and caches.

Transferring data between processors is slow.

E.g., clusters of computers, supercomputers

Parallel processing

Shared memory:

All processors have access to the same memory.

Multicore chip: separate L1 caches, L2 might be shared.

Distributed memory:

Each processor has its own memory and caches.

Transferring data between processors is slow.

E.g., clusters of computers, supercomputers

General purpose GPU computing: (Graphical Processor Unit)

Parallel processing

Shared memory:

All processors have access to the same memory.

Multicore chip: separate L1 caches, L2 might be shared.

Distributed memory:

Each processor has its own memory and caches.

Transferring data between processors is slow.

E.g., clusters of computers, supercomputers

General purpose GPU computing: (Graphical Processor Unit)

Hybrid: Often clusters of multicore/GPU machines!

Multi-thread computing

For example, multi-threaded program on dual-core computer.

Thread:

A thread of control: program code, program counter, call stack, small amount of thread-specific data (registers, L1 cache).

Shared memory and file system.

Threads may be spawned and destroyed as computation proceeds.

Languages like [OpenMP](#).

POSIX Threads

Portable Operating System Interface

Standardized C language threads programming interface

For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995).

Implementations adhering to this standard are referred to as **POSIX threads, or Pthreads.**

Multi-thread computing

Some issues:

Limited to modest number of cores when memory is shared.

Multiple threads have access to same data — convenient and fast.

Contention: But, need to make sure they don't conflict (e.g. two threads should not write to same location at same time).

Dependencies, synchronization: Need to make sure some operations are done in proper order!

May need **cache coherence:** If Thread 1 changes x in its private cache, other threads might need to see changed value.

Multi-process computing

A **process** is a thread that also has its own private address space.

Multiple processes are often running on a single computer (e.g. different independent programs).

For distributed memory parallel computers, a single computation must be tackled with multiple processes because of memory layout.

Larger cost in creating and destroying processes.

Greater latency in sharing data.

Multi-process computing

A **process** is a thread that also has its own private address space.

Multiple processes are often running on a single computer (e.g. different independent programs).

For distributed memory parallel computers, a single computation must be tackled with multiple processes because of memory layout.

Larger cost in creating and destroying processes.

Greater latency in sharing data.

Processes communicate by **passing messages**.

Languages like **MPI** — Message Passing Interface.

Multi-process computing with distributed memory

Some issues:

Often more complicated to program.

High cost of data communication between processes.

Want to maximize processing on local data relative to communication with other processes.

Often need to partition problem domain into subdomains,
(e.g. domain decomposition for PDEs)

Generally requires coarse grain parallelism.

Lecture 13

Outline:

- Amdahl's law
- Speed up, strong and weak scaling
- OpenMP

Amdahl's Law

Typically only part of a computation can be parallelized.

Suppose 50% of the computation is inherently sequential,
and the other 50% can be parallelized.

Question: How much faster could the computation potentially run on many processors?

Amdahl's Law

Typically only part of a computation can be parallelized.

Suppose 50% of the computation is inherently sequential,
and the other 50% can be parallelized.

Question: How much faster could the computation potentially run on many processors?

Answer: At most a factor of 2, no matter how many processors.

The sequential part is taking half the time and that time is still required even if the parallel part is reduced to zero time.

Amdahl's Law

Suppose 10% of the computation is inherently sequential,
and the other 90% can be parallelized.

Question: How much faster could the computation potentially run on many processors?

Amdahl's Law

Suppose 10% of the computation is inherently sequential,
and the other 90% can be parallelized.

Question: How much faster could the computation potentially run on many processors?

Answer: At most a factor of 10, no matter how many processors.

The sequential part is taking 1/10 of the time and that time is still required even if the parallel part is reduced to zero time.

Amdahl's Law

Suppose $1/S$ of the computation is inherently sequential,
and the other $(1 - 1/S)$ can be parallelized.

Then can gain at most a factor of S , no matter how many
processors.

Amdahl's Law

Suppose $1/S$ of the computation is inherently sequential,
and the other $(1 - 1/S)$ can be parallelized.

Then we can gain at most a factor of S , no matter how many
processors.

If T_s is the time required on a sequential machine and we run
on P processors, then the time required will be (at least):

$$T_P = (1/S)T_s + (1 - 1/S)T_s/P$$

Amdahl's Law

Suppose $1/S$ of the computation is inherently sequential,
and the other $(1 - 1/S)$ can be parallelized.

Then we can gain at most a factor of S , no matter how many
processors.

If T_S is the time required on a sequential machine and we run
on P processors, then the time required will be (at least):

$$T_P = (1/S)T_S + (1 - 1/S)T_S/P$$

Note that

$$T_P \rightarrow (1/S)T_S \quad \text{as} \quad P \rightarrow \infty$$

Amdahl's Law

Suppose $1/S$ of the computation is inherently sequential \Rightarrow

$$T_P = (1/S)T_S + (1 - 1/S)T_S/P$$

Example: If 5% of the computation is inherently sequential ($S = 20$), then the reduction in time is:

P	T_P
1	T_S
2	$0.525T_S$
4	$0.288T_S$
32	$0.080T_S$
128	$0.057T_S$
1024	$0.051T_S$

Speedup

The ratio T_s/T_p of time on a sequential machine to time running in parallel is the **speedup**.

This is generally less than P for P processors.

Perhaps much less.

Amdahl's Law plus overhead costs of starting processes/threads, communication, etc.

Speedup

The ratio T_s/T_p of time on a sequential machine to time running in parallel is the **speedup**.

This is generally less than P for P processors.

Perhaps much less.

Amdahl's Law plus overhead costs of starting processes/threads, communication, etc.

Caveat: May (rarely) see speedup greater than P ...

For example, if data doesn't all fit in one cache
but does fit in the combined caches of multiple processors.

Scaling

Some algorithms **scale** better than others as the number of processors increases.

Typically interested on how well algorithms work for large problems requiring lots of time, e.g.

- Particle methods for n particles,
- algorithms for solving systems of n equations,
- algorithms for solving PDEs on $n \times n \times n$ grid in 3D,

Scaling

Some algorithms **scale** better than others as the number of processors increases.

Typically interested on how well algorithms work for large problems requiring lots of time, e.g.

- Particle methods for n particles,
- algorithms for solving systems of n equations,
- algorithms for solving PDEs on $n \times n \times n$ grid in 3D,

For large n , there **may** be lots of inherent parallelism.

But this depends on many factors:

- dependencies between calculations,
- communication as well as flops,
- nature of problem and algorithm chosen.

Scaling

Typically interested on how well algorithms work for large problems requiring lots of time.

Strong scaling: How does the algorithm perform as the number of processors P increases for a **fixed problem size n ?**

Any algorithm will eventually break down (consider $P > n$)

Scaling

Typically interested on how well algorithms work for large problems requiring lots of time.

Strong scaling: How does the algorithm perform as the number of processors P increases for a **fixed problem size n ?**

Any algorithm will eventually break down (consider $P > n$)

Weak scaling: How does the algorithm perform when the problem size increases with the number of processors?

E.g. If we double the number of processors can we solve a problem “twice as large” in the same time?

Weak scaling

What does “twice as large” mean?

Depends on how algorithm complexity scales with n .

Example: Solving $n \times n$ linear system with Gaussian elimination requires $O(n^3)$ flops.

Doubling n requires 8 times as many operations.

Weak scaling

What does “twice as large” mean?

Depends on how algorithm complexity scales with n .

Example: Solving $n \times n$ linear system with Gaussian elimination requires $O(n^3)$ flops.

Doubling n requires 8 times as many operations.

Problem is “twice as large” if we increase n by a factor of $2^{1/3} \approx 1.26$, e.g. from 100×100 to 126×126 .

Weak scaling

What does “twice as large” mean?

Depends on how algorithm complexity scales with n .

Example: Solving $n \times n$ linear system with Gaussian elimination requires $O(n^3)$ flops.

Doubling n requires 8 times as many operations.

Problem is “twice as large” if we increase n by a factor of $2^{1/3} \approx 1.26$, e.g. from 100×100 to 126×126 .

(Or may be better to count memory accesses!)

Weak scaling

Solving steady state heat equation on $n \times n \times n$ grid.

n^3 grid points \Rightarrow linear system with this many unknowns.

If we used Gaussian elimination (**very bad idea!**) we would require $\sim (n^3)^3 = n^9$ flops.

Doubling n would require $2^9 = 512$ times more flops.

Weak scaling

Solving steady state heat equation on $n \times n \times n$ grid.

n^3 grid points \Rightarrow linear system with this many unknowns.

If we used Gaussian elimination (**very bad idea!**) we would require $\sim (n^3)^3 = n^9$ flops.

Doubling n would require $2^9 = 512$ times more flops.

Good iterative methods can do the job in $O(n^3) \log_2(n)$ work or less. (e.g. multigrid).

Weak scaling

Solving steady state heat equation on $n \times n \times n$ grid.

n^3 grid points \Rightarrow linear system with this many unknowns.

If we used Gaussian elimination (**very bad idea!**) we would require $\sim (n^3)^3 = n^9$ flops.

Doubling n would require $2^9 = 512$ times more flops.

Good iterative methods can do the job in $O(n^3) \log_2(n)$ work or less. (e.g. multigrid).

Developing better algorithms is as important as better hardware!!

Speedup for problems like steady state heat equation

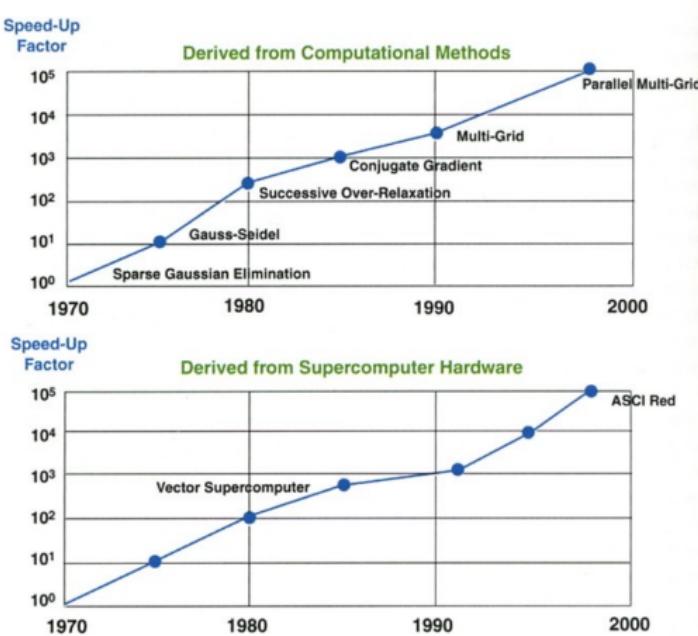


Fig. 2 Comparison of the contributions of mathematical algorithms and computer hardware.

Source: SIAM Review 43(2001), p. 168.

OpenMP

“Open Specifications for MultiProcessing”

Standard for shared memory parallel programming.
For shared memory computers, such as multi-core.

Can be used with Fortran (77/90/95/2003), C and C++.

Complete specifications at <http://www.openmp.org>

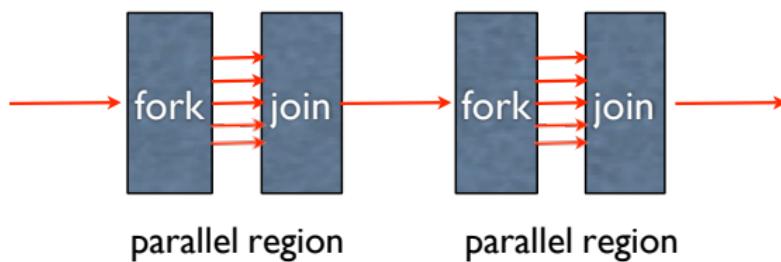
OpenMP References

- <http://www.openmp.org>
- <http://www.openmp.org/wp/resources/>
- B. Chapman, G. Jost, R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, MIT Press, 2007.
- R. Chandra, L. Dagum, et. al., *Parallel Programming in OpenMP*, Academic Press, 2001.

OpenMP — Basic Idea

Explicit programmer control of parallelization using fork-join model of parallel execution

- all OpenMP programs begin as single process, the master thread, which executes until a parallel region construct encountered
- FORK: master thread creates team of parallel threads
- JOIN: When threads complete statements in parallel region construct they synchronize and terminate, leaving only the master thread.



OpenMP — Basic Idea

- Rule of thumb: One thread per processor (or core),
- User inserts **compiler directives** telling compiler how statements are to be executed
 - which parts are parallel
 - how to assign code in **parallel regions** to threads
 - what data is **private** (local) to threads
- Compiler generates explicit threaded code

OpenMP — Basic Idea

- Rule of thumb: One thread per processor (or core),
- User inserts **compiler directives** telling compiler how statements are to be executed
 - which parts are parallel
 - how to assign code in **parallel regions** to threads
 - what data is **private** (local) to threads
- Compiler generates explicit threaded code
- **Dependencies** in parallel parts require **synchronization** between threads
- User's job to **remove dependencies** in parallel parts or use **synchronization**. (Tools exist to look for **race conditions**.)

OpenMP compiler directives

Uses [compiler directives](#) that start with `!$` (pragmas in C.)

These look like comments to standard Fortran but are recognized when compiled with the flag [-fopenmp](#).

[OpenMP statements:](#)

Ordinary Fortran statements conditionally compiled:

```
!$ print *, "Compiled with -fopenmp"
```

OpenMP compiler directives, e.g.

```
!$omp parallel do
```

Calls to OpenMP library routines:

```
use omp_lib      ! need this module
!$ call omp_set_num_threads(2)
```

OpenMP directives

```
!$omp directive [clause ...]
    if (scalar_expression)
    private (list)
    shared (list)
    default (shared | none)
    firstprivate (list)
    reduction (operator: list)
    copyin (list)
    num_threads (integer-expression)
```

A few OpenMP directives

```
!$omp parallel [clause]
    ! block of code
 !$omp end parallel
```

```
!$omp parallel do [clause]
    ! do loop
 !$omp end parallel do
```

```
!$omp barrier
    ! wait until all threads arrive
```

Several others we'll see later...

OpenMP

API also provides for (but implementation may not support):

- Nested parallelism (parallel constructs inside other parallel constructs)
- Dynamically altering number of threads in different parallel regions

The standard says nothing about parallel I/O.

OpenMP provides "relaxed-consistency" view of memory.

Threads can cache their data and are not required to maintain exact consistency with real memory all the time.

`!$omp flush`

can be used as a **memory fence** at a point where all threads must have consistent view of memory.

OpenMP test code

```
program test
    use omp_lib
    integer :: thread_num

    ! Specify number of threads to use:
    !$ call omp_set_num_threads(2)

    print *, "Testing openmp ..."

    !$omp parallel
    !$omp critical
    !$ thread_num = omp_get_thread_num()
    !$ print *, "This thread = ", thread_num
    !$omp end critical
    !$omp end parallel
end program test
```

OpenMP test code output

Compiled with OpenMP:

```
$ gfortran -fopenmp test.f90  
$ ./a.out
```

Testing openmp ...

This thread = 0

This thread = 1

(or threads might print in the other order!)

Compiled without OpenMP:

```
$ gfortran test.f90  
$ ./a.out  
Testing openmp ...
```

OpenMP test code

```
! Specify number of threads to use:  
!$ call omp_set_num_threads(2)
```

Can specify more threads than processors, but they won't execute in parallel.

The number of threads is determined by (in order):

- Evaluation of **if** clause of a directive
(if evaluates to zero or false \Rightarrow serial execution)
- Setting the **num_threads** clause
- the **omp_set_num_threads()** library subroutine
- the **OMP_NUM_THREADS** environment variable
- Implementation default

OpenMP test code

```
!$omp parallel
 !$omp critical
 !$ thread_num = omp_get_thread_num()
 !$ print *, "This thread = ", thread_num
 !$omp end critical
 !$omp end parallel
```

The `!$omp parallel` block spawns two threads and each one works independently, doing all instructions in block.

Threads are destroyed at `!$omp end parallel`.

However, the statements are also in a `!$omp critical` block, which indicates that this section of the code can be executed by only one thread at a time, so in fact they are not done in parallel.

So why do this? The function `omp_get_thread_num()` returns a unique number for each thread and we want to print both of these.

OpenMP test code

Incorrect code without critical section:

```
!$omp parallel  
!$ thread_num = omp_get_thread_num()  
!$ print *, "This thread = ", thread_num  
!$omp end parallel
```

Why not do these in parallel?

1. If the prints are done simultaneously they may come out **garbled** (characters of one interspersed in the other).
2. `thread_num` is a **shared variable**. If this were not in a critical section, the following would be possible:

Thread 0 executes function, sets `thread_num=0`

Thread 1 executes function, sets `thread_num=1`

Thread 0 executes print statement: "This thread = 1"

Thread 1 executes print statement: "This thread = 1"

There is a **data race** or **race condition**.

OpenMP test code

Could change to add a **private** clause:

```
!$omp parallel private(thread_num)  
  
!$ thread_num = omp_get_thread_num()  
  
!$omp critical  
!$ print *, "This thread = ", thread_num  
!$omp end critical  
!$omp end parallel
```

Then each thread has it's own version of the `thread_num` variable.

HPSC 101 — Lecture 14

Outline:

- OpenMP:
- Parallel do loops, reductions

Reading:

- codes/openmp

OpenMP parallel do loops

```
!$omp parallel do
do i=1,n
    ! do stuff for each i
    enddo
 !$omp end parallel do      ! OPTIONAL
```

indicates that the do loop can be done in parallel.

Requires:

- what's done for each value of i is independent of others
- Different values of i can be done in any order.

The iteration variable `i` is private to the thread: each thread has its own version.

By default, all other variables are shared between threads unless specified otherwise.

Need to be careful that threads use shared variables properly.

OpenMP parallel do loops

This code fills a vector y with function values that take a bit of time to compute:

```
! fragment of codes/openmp/yeval.f90

dx = 1.d0 / (n+1.d0)

 !$omp parallel do private(x)
 do i=1,n
   x = i*dx
   y(i) = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.d0)
 enddo
```

Elapsed time for $n = 10^8$, without OpenMP: about 3.0 sec.

Elapsed time using OpenMP on 2 processors: about 1.9 sec.

Memory stack

Note: Parallel threads use stack and you may need to increase the limit (e.g. on the VM):

```
$ gfortran -fopenmp yeval.f90
```

```
$ ./a.out
```

```
Segmentation fault
```

```
$ ulimit -s
```

```
8192
```

```
$ ulimit -s unlimited
```

```
$ ./a.out
```

```
Using OpenMP with 2 threads
```

```
Filled vector y of length 100000000
```

Memory stack

Note: Parallel threads use stack and you may need to increase the limit (e.g. on the VM):

```
$ gfortran -fopenmp yeval.f90
```

```
$ ./a.out
```

```
Segmentation fault
```

```
$ ulimit -s
```

```
8192
```

```
$ ulimit -s unlimited
```

```
$ ./a.out
```

```
Using OpenMP with 2 threads
```

```
Filled vector y of length 100000000
```

On Mac, there's a hard limit `ulimit -s hard or`

`gfortran -fopenmp -Wl,-stack_size -Wl,3f000000 yeval.f90`

Memory: Heap and Stack

Memory devoted to data for a program is generally split up:

Heap: Dynamically allocated memory — memory allocator looks for free block of memory, keeps track of free list, does garbage collection, etc.

Stack: Block of memory where space is allocated on “top” of the stack as needed and “popped” off the stack when no longer needed. **Last in – first out (LIFO).**

Fast relative to heap allocation.

Natural way to allocate storage for nested subroutine or function calls: If A calls B calls C, then when the variables used by C are popped off the stack, we’re back to the variables of B.

Memory: Heap and Stack

Memory devoted to data for a program is generally split up:

Heap: Dynamically allocated memory — memory allocator looks for free block of memory, keeps track of free list, does garbage collection, etc.

Stack: Block of memory where space is allocated on “top” of the stack as needed and “popped” off the stack when no longer needed. **Last in – first out (LIFO).**

Fast relative to heap allocation.

Natural way to allocate storage for nested subroutine or function calls: If A calls B calls C, then when the variables used by C are popped off the stack, we’re back to the variables of B.

Private variables for threads also put on stack, popped off when parallel block ends.

OpenMP parallel do loops

This code is **not correct**:

```
!$omp parallel do
do i=1,n
    x = i*dx
    y(i) = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.d0)
enddo
```

OpenMP parallel do loops

This code is **not correct**:

```
!$omp parallel do
do i=1,n
    x = i*dx
    y(i) = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.d0)
enddo
```

By default, `x` is a shared variable.

Might happen that:

Processor 0 sets `x` properly for one value of `i`,
Processor 1 sets `x` properly for another value of `i`,
Processor 0 uses `x` but is now incorrect.

OpenMP parallel do loops

Correct version:

```
!$omp parallel do private(x)
do i=1,n
    x = i*dx
    y(i) = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.d0)
enddo
```

Now each thread has its own version of `x`.

Iteration counter `i` is private by default.

OpenMP parallel do loops

Correct version:

```
!$omp parallel do private(x)
do i=1,n
    x = i*dx
    y(i) = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.d0)
enddo
```

Now each thread has its own version of `x`.

Iteration counter `i` is private by default.

Note that `dx`, `n`, `y` are shared by default. OK because:

`dx`, `n` are used but not changed,
`y` is changed, but independently for each `i`

OpenMP parallel do loops

Incorrect code:

```
dx = 1.d0 / (n+1.d0)
 !$omp parallel do private(x,dx)
do i=1,n
    x = i*dx
    y(i) = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.d0)
enddo
```

Specifying `dx` private won't work here.

This will create a private variable `dx` for each thread but it will be **uninitialized**.

Will run but give garbage.

OpenMP parallel do loops

Could fix with:

```
dx = 1.d0 / (n+1.d0)
!$omp parallel do firstprivate(dx)
do i=1,n
    x = i*dx
    y(i) = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.d0)
enddo
```

The **firstprivate** clause creates private variables and initializes to the value from the master thread prior to the loop.

OpenMP parallel do loops

Could fix with:

```
dx = 1.d0 / (n+1.d0)
!$omp parallel do firstprivate(dx)
do i=1,n
    x = i*dx
    y(i) = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.d0)
enddo
```

The **firstprivate** clause creates private variables and initializes to the value from the master thread prior to the loop.

There is also a **lastprivate** clause to indicate that the last value computed by a thread (for $i = n$) should be copied to the master thread's copy for continued execution.

OpenMP parallel do loops

```
! from codes/openmp/private1.f90

n = 7
y = 2.d0
!$omp parallel do firstprivate(y) lastprivate(y)
do i=1,n
    y = y + 10.d0
    x(i) = y
    !omp critical
    print *, "i = ",i," x(i) = ",x(i)
    !omp end critical
enddo
print *, "At end, y = ",y
```

Run with 2 threads: The 7 values of i will be split up, perhaps

$i = 1, 2, 3, 4$ executed by thread 0,

$i = 5, 6, 7$ executed by thread 1.

Thread 0's private y will be updated 4 times, $2 \rightarrow 12 \rightarrow 22 \rightarrow 32 \rightarrow 42$

Thread 1's private y will be updated 3 times, $2 \rightarrow 12 \rightarrow 22 \rightarrow 32$

OpenMP parallel do loops

```
! from codes/openmp/private1.f90

n = 7
y = 2.d0
 !$omp parallel do firstprivate(y) lastprivate(y)
do i=1,n
    y = y + 10.d0
    x(i) = y
    !$omp critical
    print *, "i = ", i, " x(i) = ", x(i)
    !$omp end critical
enddo
print *, "At end, y = ", y
```

might produce:

i =	1	x(i) =	12.000000000000000
i =	5	x(i) =	12.000000000000000
i =	2	x(i) =	22.000000000000000
i =	6	x(i) =	22.000000000000000
i =	3	x(i) =	32.000000000000000
i =	7	x(i) =	32.000000000000000
i =	4	x(i) =	42.000000000000000
At end, y =			32.000000000000000

Order might be different but final y will be from $i = 7$.

OpenMP parallel do loops — changing default

Default is that loop iterator is private, other variables shared.
Can change this, e.g.

```
!$omp parallel do default(private) shared(x,z) &
 !$omp firstprivate(y) lastprivate(y)
do i=1,n
etc.
```

With this change, only `x` and `z` are shared.

Note continuation character `&` and continuation line.

OpenMP synchronization

```
!$omp parallel do
do i=1,n
    ! do stuff for each i
enddo
!$omp end parallel do    ! OPTIONAL

        ! master thread continues execution
```

There is an **implicit barrier** at the end of the loop.

The master thread will not continue until all threads have finished with their subset of 1, 2, ..., n.

Except if ended by:

```
!$omp end parallel do nowait
```

Conditional clause

Loop overhead may not be worthwhile for short loops.
(Multi-thread version may run slower than sequential)

Can use conditional clause:

```
$omp parallel do if (n > 1000)
do i=1,n
    ! do stuff
enddo
```

If $n \leq 1000$ then no threads are created,
master thread executes loop sequentially.

Nested loops

```
!$omp parallel do private(i)
do j=1,m
    do i=1,n
        a(i,j) = 0.d0
    enddo
enddo
```

The loop on j is split up between threads.

The thread handling $j=1$ does the entire loop on i ,
sets $a(1,1)$, $a(2,1)$, ..., $a(n,1)$.

Nested loops

```
!$omp parallel do private(i)
do j=1,m
    do i=1,n
        a(i,j) = 0.d0
    enddo
enddo
```

The loop on j is split up between threads.

The thread handling $j=1$ does the entire loop on i ,
sets $a(1,1)$, $a(2,1)$, ..., $a(n,1)$.

Note: The loop iterator i must be declared **private!**

j is private by default, i is shared by default.

Nested loops

Which is better? (assume $m \approx n$)

```
!$omp parallel do private(i)
do j=1,m
    do i=1,n
        a(i,j) = 0.d0
    enddo
enddo
```

or

```
do j=1,m
    !$omp parallel do
    do i=1,n
        a(i,j) = 0.d0
    enddo
enddo
```

Nested loops

Which is better? (assume $m \approx n$)

```
!$omp parallel do private(i)
do j=1,m
    do i=1,n
        a(i,j) = 0.d0
    enddo
enddo
```

or

```
do j=1,m
    !$omp parallel do
    do i=1,n
        a(i,j) = 0.d0
    enddo
enddo
```

The first has less overhead: Threads created only once.

The second has more overhead: Threads created m times.

Nested loops

But have to make sure loop can be parallelized!

Incorrect code for replicating first column:

```
!$omp parallel do private(j)
do i=2,n
    do j=1,m
        a(i,j) = a(i-1,j)
    enddo
enddo
```

Corrected: (j 's can be done in any order, i 's cannot)

```
!$omp parallel do private(i)
do j=1,m
    do i=2,n
        a(i,j) = a(i-1,j)
    enddo
enddo
```

Reductions

Incorrect code for computing $\|x\|_1 = \sum_i |x_i|$:

```
norm = 0.d0
 !$omp parallel do
 do i=1,n
     norm = norm + abs(x(i))
 enddo
```

There is a **race condition**: each thread is updating same shared variable `norm`.

Correct code:

```
 !$omp parallel do reduction(+: norm)
 do i=1,n
     norm = norm + abs(x(i))
 enddo
```

A **reduction** reduces an array of numbers to a single value.

Reductions

A more complicated way to do this:

```
norm = 0.d0
 !$omp parallel private(mysum) shared(norm)
 mysum = 0
 !$omp do
 do i=1,n
     mysum = mysum + abs(x(i))
 enddo

 !$omp critical
 norm = norm + mysum
 !$omp end critical
 !$omp end parallel
```

Some other reductions

Can do reductions using +, -, *, min, max, .and., .or., some others

General form:

```
!$omp parallel do reduction(operator : list)
```

Example with max:

```
y = -1.d300 ! very negative value
 !$omp parallel do reduction(max: y)
 do i=1,n
     y = max(y,x(i))
 enddo
 print *, 'max of x = ',y
```

Some other reductions

General form:

```
!$omp parallel do reduction(operator : list)
```

Example with .or.:

```
logical anyzero
```

```
! set x...
anyzero = .false.
```

```
!$omp parallel do reduction(.or.: anyzero)
do i=1,n
    anyzero = anyzero .or. (x(i) == 0.d0)
enddo
print *, 'anyzero = ', anyzero
```

Prints T if any $x(i)$ is zero, F otherwise.

Exercise

1. Implement the Monte Carlo Pi code (as given in the midsem exam) in Fortran
2. Parallelise the code using OpenMP

HPSC 101 — Lecture 15

Outline:

- OpenMP:
- Nested loops, reductions
- Monte Carlo Pi - exercise

Reading:

- codes/openmp

Nested loops

```
!$omp parallel do private(i)
do j=1,m
    do i=1,n
        a(i,j) = 0.d0
    enddo
enddo
```

The loop on j is split up between threads.

The thread handling $j=1$ does the entire loop on i ,
sets $a(1,1)$, $a(2,1)$, ..., $a(n,1)$.

Nested loops

```
!$omp parallel do private(i)
do j=1,m
    do i=1,n
        a(i,j) = 0.d0
    enddo
enddo
```

The loop on j is split up between threads.

The thread handling $j=1$ does the entire loop on i ,
sets $a(1,1)$, $a(2,1)$, ..., $a(n,1)$.

Note: The loop iterator i must be declared **private!**

j is private by default, i is shared by default.

Nested loops

Which is better? (assume $m \approx n$)

```
!$omp parallel do private(i)
do j=1,m
    do i=1,n
        a(i,j) = 0.d0
    enddo
enddo
```

or

```
do j=1,m
    !$omp parallel do
    do i=1,n
        a(i,j) = 0.d0
    enddo
enddo
```

Nested loops

Which is better? (assume $m \approx n$)

```
!$omp parallel do private(i)
do j=1,m
    do i=1,n
        a(i,j) = 0.d0
    enddo
enddo
```

or

```
do j=1,m
    !$omp parallel do
    do i=1,n
        a(i,j) = 0.d0
    enddo
enddo
```

The first has less overhead: Threads created only once.

The second has more overhead: Threads created m times.

Nested loops

But have to make sure loop can be parallelized!

Incorrect code for replicating first column:

```
!$omp parallel do private(j)
do i=2,n
    do j=1,m
        a(i,j) = a(i-1,j)
    enddo
enddo
```

Corrected: (j 's can be done in any order, i 's cannot)

```
!$omp parallel do private(i)
do j=1,m
    do i=2,n
        a(i,j) = a(i-1,j)
    enddo
enddo
```

Reductions

Incorrect code for computing $\|x\|_1 = \sum_i |x_i|$:

```
norm = 0.d0
 !$omp parallel do
 do i=1,n
     norm = norm + abs(x(i))
 enddo
```

There is a **race condition**: each thread is updating same shared variable `norm`.

Correct code:

```
 !$omp parallel do reduction(+: norm)
 do i=1,n
     norm = norm + abs(x(i))
 enddo
```

A **reduction** reduces an array of numbers to a single value.

Reductions

A more complicated way to do this:

```
norm = 0.d0
 !$omp parallel private(mysum) shared(norm)
 mysum = 0
 !$omp do
 do i=1,n
     mysum = mysum + abs(x(i))
 enddo

 !$omp critical
 norm = norm + mysum
 !$omp end critical
 !$omp end parallel
```

Some other reductions

Can do reductions using +, -, *, min, max, .and., .or., some others

General form:

```
!$omp parallel do reduction(operator : list)
```

Example with max:

```
y = -1.d300 ! very negative value
 !$omp parallel do reduction(max: y)
 do i=1,n
     y = max(y,x(i))
 enddo
 print *, 'max of x = ',y
```

Some other reductions

General form:

```
!$omp parallel do reduction(operator : list)
```

Example with .or.:

```
logical anyzero
```

```
! set x...
anyzero = .false.
```

```
!$omp parallel do reduction(.or.: anyzero)
do i=1,n
    anyzero = anyzero .or. (x(i) == 0.d0)
enddo
print *, 'anyzero = ', anyzero
```

Prints T if any $x(i)$ is zero, F otherwise.

Timing fortran codes

Outline:

- Timing Fortran codes

Codes:

- codes/fortran/timings.f90
- codes/openmp/timings.f90

Determining CPU and execution time

Unix `time` command, e.g.

```
$ time ./a.out  
<output from code>
```

```
real      0m5.279s  
user      0m1.915s  
sys       0m0.006s
```

Means the elapsed (wall clock) time was 5.279 seconds, CPU time dedicated to your code was \approx 1.915 seconds. System time \approx 0.006 seconds.

Determining CPU and execution time

Unix `time` command, e.g.

```
$ time ./a.out  
<output from code>
```

```
real      0m5.279s  
user      0m1.915s  
sys       0m0.006s
```

Means the elapsed (wall clock) time was 5.279 seconds, CPU time dedicated to your code was \approx 1.915 seconds. System time \approx 0.006 seconds.

Doesn't allow examining parts of code, not always very accurate.

Note that timing small codes can be deceptive

Fortran timing utilities

`system_clock`: elapsed time between 2 calls.

`cpu_time`: CPU time used between 2 calls.

See timings code

Exercise

1. Implement the Monte Carlo Pi code (as given in the midsem exam) in Fortran
 1. Input data from command line
2. Parallelise the code using OpenMP
3. Strong and weak scaling of the code
4. Test execution on HPC, perform scaling on HPC

HPSC 101 — Lecture 16

Outline:

- OpenMP:
- loop dependencies
- threadsafe and pure subroutines and functions
- other directives, beyond "parallel do"

Reading:

- codes/openmp

Dependencies in loops

```
do i=1,n  
    z(i) = x(i) + y(i)  
    w(i) = cos(z(i))  
enddo
```

There is a **data dependence** between the two statements in this loop.

The value $w(i)$ cannot be computed before $z(i)$.

However, this *can* be parallelized with a **parallel do** since the same thread will always execute both statements in the right order for each i .

Matrix-matrix multiplication

```
!$omp parallel do private(i,k)
do j=1,n
    do i=1,n
        c(i,j) = 0.d0
        do k=1,n
            c(i,j) = c(i,j) + a(i,k)*b(k,j)
        enddo
    enddo
enddo
```

This works since $c(i,j)$ is only modified by thread handling column j .

Loop-Carried Dependencies

```
x = 1.d0      ! initialize all elements to 1
x(1) = 5.d0

do i=2,n
    x(i) = x(i-1)
enddo
```

There is a **loop-carried data dependence** in this loop.

The assignment for `i=3` must not be done before `i=2` or it may get the wrong value.

Loop-Carried Dependencies

Example: Solve ODE initial value problem

$$\begin{aligned}y'(t) &= 2y(t), \\y(0) &= 1\end{aligned}$$

with Euler's method

$$y(t + \Delta t) \approx y(t) + \Delta t \cdot y'(t) = y(t) + \Delta t(2y(t)),$$

to approximate $y(t) = e^{2t}$ for $0 \leq t \leq 5$:

```
y(1) = 1.d0
dt = 0.001d0      ! time step
n = 5000          ! number of steps to reach t=5
do i=2,n
    y(i) = y(i-1) + dt*2.d0*y(i-1)
enddo
```

Cannot parallelize.

Loop-Carried Dependencies

```
y = 0.d0
do i=1,10
    if (i==3) y = 1.d0
    x(i) = y
enddo
```

There is a **loop-carried data dependence** in this loop.

In serial execution: only first two elements of `x` are 0.d0.

With `!$omp parallel do`:

later index (e.g. `i=6`) **might** be executed before `i=3`.

Thread-safe functions

Consider this code:

```
!$omp parallel do
do i=1,n
    y(i) = myfcn(x(i))
enddo
```

Does this give the same results as the serial version?

Thread-safe functions

Consider this code:

```
!$omp parallel do
do i=1,n
    y(i) = myfcn(x(i))
enddo
```

Does this give the same results as the serial version?

Maybe not... it depends on what the function does!

If this gives the same results regardless of the order threads call for different values of i , then the function is **thread safe**.

Thread-safe functions

A thread-safe function:

```
function myfcn(x)
    real(kind=8), intent(in) :: x
    real(kind=8), intent(out) :: myfcn
        real(kind=8) :: z ! local variable
        z = exp(x)
        myfcn = z*cos(x)
end function myfcn
```

Executing this function for one value of x is completely independent of execution for other values of x .

Note that each call creates a new local value z on the call stack, so z is private to the thread executing the function.

Non-Thread-safe functions

Suppose `z`, `count` are global variables defined in module `globals.f90`.

Then this function is **not thread-safe**:

```
function myfcn(x)
    real(kind=8), intent(in) :: x
    real(kind=8), intent(out) :: myfcn
    use globals
    count = count+1 ! counts times called
    z = exp(x)
    myfcn = z*cos(x) + count
end function myfcn
```

The value of `count` seen when calling `y(i) = myfcn(x(i))` will depend on the order of execution of different values of `i`.

Moreover, `z` might be modified by another thread between when it is computed and when it is used.

Aside on global variables in Fortran

```
module globals
    implicit none
    save
    integer :: count
    real(kind=8) :: z
end module globals
```

The `save` command says that values of these variables should be saved from one use to the next.

Fortran 77 and before: Instead used `common blocks`:

```
common /globals/ z, count
```

can be included in any file where `z` and `count` should be available. (**Also not thread safe!**)

Non-Thread-safe functions

Beware of input or output...

Suppose unit 20 has been opened for reading in the main program, value on line i should be used in calculating $y(i) \dots$

This function is **not thread-safe**:

```
function myfcn(x)
    real(kind=8), intent(in) :: x
    real(kind=8), intent(out) :: myfcn
    real(kind=8) :: z

    read(20,*) z
    myfcn = z*cos(x)
end function myfcn
```

Will work in serial mode but if threads execute in different order,
will give wrong results.

Pure subroutines and functions

A subroutine can be declared **pure** if it:

- Does not alter global variables,
- Does not do I/O,
- Does not declare local variables with the `save` attribute, such as `real, save :: z`
- For functions, does not alter any input arguments.

Example:

```
pure subroutine f(x,y)
    implicit none
    real(kind=8), intent(in) :: x
    real(kind=8), intent(inout) :: y
    y = x**2 + y
end subroutine f
```

Good idea even for sequential codes: Allows some compiler optimizations.

Forall statement in Fortran 90

In place of

```
do i=1,n  
    x(i) = 2.d0*i  
end do
```

can write

```
forall (i=1:n)  
    x(i) = 2.d0*i  
end forall
```

Tells compiler that the statements can execute in any order.

Also may lead to compiler optimization even on serial computer.

Forall statement in Fortran 90

Nested loops can be written with `forall`:

```
forall (i=1:n, j=1:n)
    a(i,j) = 2.d0*i*j
end forall
```

Tells compiler that it could reorder loops at will to optimize cache usage, for example.

Can also include **masks**:

```
forall (i=1:n, j=1:n, b(i,j).ne.0.d0)
    a(i,j) = 1.d0 / b(i,j)
end forall
```

OpenMP — beyond parallel loops

The directive `!$omp parallel` is used to create a number of threads that will each execute the same code...

```
!$omp parallel  
    ! some code  
!$omp end parallel
```

The code will be executed `nthreads` times, once by each thread.

SPMD: Single program, multiple data

OpenMP — beyond parallel loops

The directive `!$omp parallel` is used to create a number of threads that will each execute the same code...

```
!$omp parallel  
    ! some code  
 !$omp end parallel
```

The code will be executed `nthreads` times, once by each thread.

SPMD: Single program, multiple data

Terminology note:

SIMD: Single instruction, multiple data

refers to hardware (vector machines) that apply same arithmetic operation to a vector of values in lock-step.

SPMD is a software term — need not be in lock step.

OpenMP parallel with do loops

Note: This code...

```
!$omp parallel
    do i=1,10
        print *, "i = ",i
    enddo
 !$omp end parallel
```

... is not the same as:

```
!$omp parallel do
    do i=1,10
        print *, "i = ",i
    enddo
 !$omp end parallel do
```

OpenMP parallel with do loops

Note: This code...

```
!$omp parallel
    do i=1,10
        print *, "i = ",i
    enddo
 !$omp end parallel
```

The entire do loop ($i=1,2,\dots,10$) will be executed by each thread!
With 2 threads, 20 lines will be printed.

... is not the same as:

```
!$omp parallel do
    do i=1,10
        print *, "i = ",i
    enddo
 !$omp end parallel do
```

which will only print 10 lines!

OpenMP parallel with do loops

```
!$omp parallel do
    do i=1,10
        print *, "i = ",i
    enddo
 !$omp end parallel do
```

could also be written as:

```
!$omp parallel
 !$omp do
    do i=1,10
        print *, "i = ",i
    enddo
 !$omp end do
 !$omp end parallel
```

More generally, if `!$omp do` is inside a parallel block, then the loop is split between threads rather than done in total by each

OpenMP parallel with do loops

The !\$omp do directive is useful for...

```
!$omp parallel
```

```
! some code executed by every thread
```

```
!$omp do  
do i=1,n  
    ! loop to be split between threads  
    enddo  
!$omp end do
```

```
! more code executed by every thread
```

```
!$omp end parallel
```

Some other useful directives...

Execution of part of code by a single thread:

```
!$omp parallel  
! some code executed by every thread  
  
!$omp single  
! code executed by only one thread  
!$omp end single  
  
!$omp end parallel
```

Can also use !\$omp master to force execution by master thread.

Example: Initializing or printing out a shared variable.

Some other useful directives...

barriers:

```
!$omp parallel
```

```
! some code executed by every thread
```

```
!$omp barrier
```

```
! some code executed by every thread
```

```
!$omp end parallel
```

Every thread will stop at barrier until all threads have reached this point.

Make sure all threads reach barrier or code will hang!

Some other useful directives...

barriers:

```
!$omp parallel
```

```
! some code executed by every thread
```

```
!$omp barrier
```

```
! some code executed by every thread
```

```
!$omp end parallel
```

Every thread will stop at barrier until all threads have reached this point.

Make sure all threads reach barrier or code will hang!

Implied barriers after some blocks, e.g. !\$omp do
or !\$omp single.

Some other useful directives...

Sections:

```
!$omp parallel num_threads 2
```

```
!$omp sections
```

```
!$omp section
```

```
    ! code executed by only one thread
```

```
!$omp section
```

```
    ! code executed by a different thread
```

```
!$omp end sections    !! with implied barrier !!
```

```
!$omp end parallel
```

Example: Read in two large data files simultaneously.

From \$UWHPSC/codes/openmp/demo2.f90

```
8      integer, parameter :: n = 100000
9      real(kind=8), dimension(n) :: x,y,z

14     !$omp parallel ! spawn two threads
15     !$omp sections ! split up work between them
16
17     !$omp section
18     x = 1.d0 ! one thread initializes x array
19
20     !$omp section
21     y = 1.d0 ! another thread initializes y array
22
23     !$omp end sections
24     !$omp barrier ! not needed, implied at end of sections
25
26     !$omp single ! only want to print once:
27     print *, "Done initializing x and y"
28     !$omp end single nowait ! ok for other thread to continue
29
30     !$omp do ! split work between threads:
31     do i=1,n
32       z(i) = x(i) + y(i)
33     enddo
34
35     !$omp end parallel
```

HPSC Lecture 17

Outline:

- Fine grain vs. coarse grain parallelism
- Manually splitting loops between threads
- Examples with bugs

Reading:

- /codes/openmp
- <https://computing.llnl.gov/tutorials/openMP/>

Fine vs. coarse grain parallelism

Fine grain: Parallelize at the level of individual loops, splitting work for each loop between threads.

Coarse grain: Split problem up into large pieces and have each thread deal with one piece.

May need to synchronize or share information at some points.

Fine vs. coarse grain parallelism

Fine grain: Parallelize at the level of individual loops, splitting work for each loop between threads.

Coarse grain: Split problem up into large pieces and have each thread deal with one piece.

May need to synchronize or share information at some points.

More similar to what must be done in MPI.

Fine vs. coarse grain parallelism

Fine grain: Parallelize at the level of individual loops, splitting work for each loop between threads.

Coarse grain: Split problem up into large pieces and have each thread deal with one piece.

May need to synchronize or share information at some points.

More similar to what must be done in MPI.

Domain Decomposition: Splitting up a problem on a large domain (e.g. three-dimensional grid) into pieces that are handled separately (with suitable coupling).

Solution of independent ODEs by Euler's method

Solve $u'_i(t) = c_i u_i(t)$ for $t \geq 0$

with initial condition $u_i(0) = \eta_i$. Decoupled system of ODEs
for $i = 1, 2, \dots, n$

Solution of independent ODEs by Euler's method

Solve $u'_i(t) = c_i u_i(t)$ for $t \geq 0$

with initial condition $u_i(0) = \eta_i$. Decoupled system of ODEs
for $i = 1, 2, \dots, n$

Exact solution: $u_i(t) = e^{c_i t} \eta_i$.

Solution of independent ODEs by Euler's method

Solve $u'_i(t) = c_i u_i(t)$ for $t \geq 0$

with initial condition $u_i(0) = \eta_i$. Decoupled system of ODEs
for $i = 1, 2, \dots, n$

Exact solution: $u_i(t) = e^{c_i t} \eta_i$.

Euler method: $u_i(t + \Delta t) \approx u_i(t) + \Delta t c_i u_i(t) = (1 + c_i \Delta t) u_i(t)$.

Implement this for large number of time steps for large n .

Solution of independent ODEs by Euler's method

Solve $u'_i(t) = c_i u_i(t)$ for $t \geq 0$

with initial condition $u_i(0) = \eta_i$. Decoupled system of ODEs
for $i = 1, 2, \dots, n$

Exact solution: $u_i(t) = e^{c_i t} \eta_i$.

Euler method: $u_i(t + \Delta t) \approx u_i(t) + \Delta t c_i u_i(t) = (1 + c_i \Delta t) u_i(t)$.

Implement this for large number of time steps for large n .

For each i time stepping can't be easily made parallel.

But for large n , this problem is **embarrassingly parallel**:

Problem for each i is completely decoupled from problem for any other i . Could solve them all simultaneously with no communication needed.

Fine grain solution with parallel do loops

```
!$omp parallel do
do i=1,n
    u(i) = eta(i)
enddo

do m=1,nsteps
    !$omp parallel do
    do i=1,n
        u(i) = (1.d0 + dt*c(i))*u(i)
    enddo
enddo
```

Note that threads are forked $nsteps+1$ times.

Requires shared memory:

don't know which thread will handle each i .

Fine grain solution with parallel do loops

Might try to fork threads only once via: Wrong!

```
!$omp parallel private(m)
 !$omp do
 do i=1,n
     u(i) = eta(i)
 enddo

do m=1,nsteps
    !$omp do
    do i=1,n
        u(i) = (1.d0 + dt*c(i))*u(i)
    enddo
 enddo
 !$omp end parallel
```

Fine grain solution with parallel do loops

Might try to fork threads only once via: Wrong!

```
!$omp parallel private(m)
 !$omp do
 do i=1,n
     u(i) = eta(i)
 enddo

 do m=1,nsteps
     !$omp do
     do i=1,n
         u(i) = (1.d0 + dt*c(i))*u(i)
     enddo
 enddo
 !$omp end parallel
```

Error: the loop on m will be done independently by each thread.
(Actually works in this case but not good coding.)

Fine grain solution with parallel do loops

Can rearrange loops:

```
!$omp parallel private(m)
 !$omp do
 do i=1,n
     u(i) = eta(i)
 enddo

 !$omp do
 do i=1,n
     do m=1,nsteps
         u(i) = (1.d0 + dt*c(i))*u(i)
     enddo
 enddo
 !$omp end parallel
```

Fine grain solution with parallel do loops

Can rearrange loops:

```
!$omp parallel private(m)
 !$omp do
 do i=1,n
     u(i) = eta(i)
 enddo

 !$omp do
 do i=1,n
     do m=1,nsteps
         u(i) = (1.d0 + dt*c(i))*u(i)
     enddo
 enddo
 !$omp end parallel
```

Only works because ODEs are decoupled — can take all time steps on $u_1(t)$ without interacting with $u_2(t)$, for example.

Coarse grain solution of ODEs

Split up $i = 1, 2, \dots, n$ into n threads disjoint sets.

A set goes from $i=i_{\text{start}}$ to $i=i_{\text{end}}$

These **private values** are different for each thread.

Each thread handles 1 set for the entire problem.

```
!$omp parallel private(istart,iend,i,m)  
istart = ??  
iend = ??  
do i=istart,iend  
    u(i) = éta(i)  
enddo  
do i=istart,iend  
    do m=1,nsteps  
        u(i) = (1.d0 + dt*c(i))*u(i)  
        enddo  
    enddo  
!$omp end parallel
```

Threads are forked only once,
Each thread only needs subset of data.

Setting `istart` and `iend`

Example: If $n=100$ and $nthreads = 2$, we would want:

Thread 0: $i_{start} = 1$ and $i_{end} = 50$,

Thread 1: $i_{start} = 51$ and $i_{end} = 100$.

If $nthreads$ divides n evenly...

```
points_per_thread = n / nthreads
 !$omp parallel private(thread_num, istart, iend, i)
     thread_num = 0      ! needed in serial mode
     !$ thread_num = omp_get_thread_num()

     istart = thread_num * points_per_thread + 1
     iend = (thread_num+1) * points_per_thread

     do i=istart,iend
         ! work on thread's part of array
         enddo
     ...
 !$omp end parallel
```

Setting `istart` and `iend` more generally

Example: If $n=101$ and $nthreads = 2$, we would want:

Thread 0: $istart= 1$ and $iend= 51$,

Thread 1: $istart=52$ and $iend=101$.

If $nthreads$ might not divide n evenly...

```
points_per_thread = (n + nthreads - 1) / nthreads
!$omp parallel private(thread_num, istart, iend, i)

    thread_num = 0      ! needed in serial mode
    !$ thread_num = omp_get_thread_num()

    istart = thread_num * points_per_thread + 1
    iend = min((thread_num+1)*points_per_thread, n)

    do i=istart,iend
        ! work on thread's part of array
        enddo
    ...

!$omp end parallel
```

Example: Normalizing a vector

Given a vector (1-dimensional array) x , Compute the normalized vector $x / \|x\|_1$, with $\|x\|_1 = \text{sum}(|x|)$

Fine-grain: Using parallel do loops.

```
norm = 0.d0
 !$omp parallel do reduction(+: norm)
do i=1,n
    norm = norm + abs(x(i))
enddo

 !$omp parallel do
do i=1,n
    x(i) = x(i) / norm
enddo
```

Note: Must finish computing `norm` before using for any `x(i)`, so we are using the **implicit barrier** after the first loop.

Example: Normalizing a vector

Another **fine-grain approach**, forking threads only once:

```
! from /codes/openmp/normalize1.f90

norm = 0.d0
 !$omp parallel private(i)

 !$omp do reduction(+: norm)
 do i=1,n
     norm = norm + abs(x(i))
 enddo
 !$omp barrier ! not needed (implicit)

 !$omp do
 do i=1,n
     x(i) = x(i) / norm
 enddo
 !$omp end parallel
```

Example: Normalizing a vector

Compute the normalized vector $x / \|x\|_1$

Coarse grain version:

Assign blocks of i values to each thread. Threads must:

- Compute thread's contribution to $\|x\|_1$,

$$\text{norm_thread} = \sum_{i=\text{start}}^{\text{iend}} |x_i|,$$

- Collaborate to compute total value $\|x\|_1$:

$$\|x\|_1 = \sum_{\text{threads}} \text{norm_thread}$$

- Loop over $i = \text{istart}, \dots, \text{iend}$ to divide x_i by $\|x\|_1$.

Example: Normalizing a vector

```
! from /codes/openmp/normalize2.f90

    norm = 0.d0
 !$omp parallel private(i,norm_thread, &
 !$omp           istart,iend,thread_num)
 !$ thread num = omp_get_thread_num()
 istart = thread num * points_per_thread + 1
 iend = min((thread_num+1) * points_per_thread, n)

norm_thread = 0.d0
do i=istart,iend
    norm_thread = norm_thread + abs(x(i))
enddo

! update global norm with value from each thread:
 !$omp critical
     norm = norm + norm_thread
 !$omp end critical

 !$omp barrier !! needed here

do i=istart,iend
    y(i) = x(i) / norm
enddo

 !$omp end parallel
```

Example: Normalizing a vector — parallel block

```
norm_thread = 0.d0
do i=istart,iend
    norm_thread = norm_thread + abs(x(i))
enddo

! update global norm with value from each thread
!$omp critical
    norm = norm + norm_thread
!$omp end critical

!$omp barrier !! needed here

do i=istart,iend
    y(i) = x(i) / norm
enddo
```

Normalizing a vector — possible bugs

1. Not declaring proper variables `private`

Normalizing a vector — possible bugs

1. Not declaring proper variables `private`
2. Setting `norm = 0.d0` inside parallel block.

Ok if it's in a `omp single` block. Otherwise second thread might set to zero after first thread has updated by `norm_thread`.

Normalizing a vector — possible bugs

1. Not declaring proper variables `private`
2. Setting `norm = 0.d0` inside parallel block.

Ok if it's in a `omp single` block. Otherwise second thread might set to zero after first thread has updated by `norm_thread`.

3. Not using `omp critical` block to update global `norm`.
Data race.

Normalizing a vector — possible bugs

1. Not declaring proper variables `private`
2. Setting `norm = 0.d0` inside parallel block.

Ok if it's in a `omp single` block. Otherwise second thread might set to zero after first thread has updated by `norm_thread`.

3. Not using `omp critical` block to update global `norm`.
Data race.
4. Not having a barrier between updating `norm` and using it.
First thread may use `norm` before other threads have added their contributions.

Normalizing a vector — possible bugs

1. Not declaring proper variables `private`
2. Setting `norm = 0.d0` inside parallel block.

Ok if it's in a `omp single` block. Otherwise second thread might set to zero after first thread has updated by `norm_thread`.

3. Not using `omp critical` block to update global `norm`.
Data race.

4. Not having a barrier between updating `norm` and using it.
First thread may use `norm` before other threads have added their contributions.

None of these bugs would give compile or run-time errors!
Just wrong results (sometimes).

OpenMP example with shared exit criterion

Solve $u'_i(t) = c_i u_i(t)$ for $t \geq 0$
with initial condition $u_i(0) = \eta_i$.

Exact solution: $u_i(t) = e^{c_i t} \eta_i$.

Euler method: $u_i(t + \Delta t) \approx u_i(t) + \Delta t c_i u_i(t) = (1 + c_i \Delta t) u_i(t)$.

New condition: Stop time stepping when any of the $u_i(t)$ values exceeds 100.

(Will certainly happen as long as $c_j > 0$ for some j .)

OpenMP example with shared exit criterion

Stop time stepping when any of the $u_i(t)$ values exceeds 100.

Idea:

Each time step, compute $u_{\max} = \text{maximum value of } u_i \text{ over all } i$ and exit the time-stepping if $u_{\max} > 100$.

OpenMP example with shared exit criterion

Stop time stepping when any of the $u_i(t)$ values exceeds 100.

Idea:

Each time step, compute $umax = \text{maximum value of } u_i \text{ over all } i$ and exit the time-stepping if $umax > 100$.

Each thread has a private variable $umax_thread$ for the maximum value of u_i for its values of i . Updated for each i .

OpenMP example with shared exit criterion

Stop time stepping when any of the $u_i(t)$ values exceeds 100.

Idea:

Each time step, compute $umax = \text{maximum value of } u_i \text{ over all } i$ and exit the time-stepping if $umax > 100$.

Each thread has a private variable $umax_thread$ for the maximum value of u_i for its values of i . Updated for each i .

Each thread updates shared $umax$ based on its $umax_thread$.
This needs to be done in **critical section**.

OpenMP example with shared exit criterion

Stop time stepping when any of the $u_i(t)$ values exceeds 100.

Idea:

Each time step, compute $umax = \text{maximum value of } u_i \text{ over all } i$ and exit the time-stepping if $umax > 100$.

Each thread has a private variable $umax_thread$ for the maximum value of u_i for its values of i . Updated for each i .

Each thread updates shared $umax$ based on its $umax_thread$.

This needs to be done in **critical section**.

Also need two **barriers** to make sure all threads are in sync at certain points.

OpenMP example with shared exit criterion

Stop time stepping when any of the $u_i(t)$ values exceeds 100.

Idea:

Each time step, compute $umax = \text{maximum value of } u_i \text{ over all } i$ and exit the time-stepping if $umax > 100$.

Each thread has a private variable $umax_thread$ for the maximum value of u_i for its values of i . Updated for each i .

Each thread updates shared $umax$ based on its $umax_thread$.

This needs to be done in **critical section**.

Also need two **barriers** to make sure all threads are in sync at certain points.

Study code in /codes/openmp/umax1.f90.

OpenMP example with shared exit criterion

```
!$omp parallel private(i,m,umax_thread, &
!$omp                      istart,iend,thread_num)
!$ thread num = omp_get_thread_num()
istart = thread num-* points_per_thread + 1
iend = min((thread_num+1) * points_per_thread, n)

do m=1,nsteps
    umax_thread = 0.d0
    !$omp single
        umax = 0.d0
    !$omp end single
    do i=istart,iend
        u(i) = (1.d0 + c(i)*dt) * u(i)
        umax_thread = max(umax_thread, u(i))
    enddo

    !$omp critical
        umax = max(umax, umax_thread)
    !$omp end critical
    !$omp barrier
    if (umax > 100) exit
    !$omp barrier
enddo
 !$omp end parallel
```

do loop in parallel block:

```
do m=1,nsteps
    umax_thread = 0.d0
    !$omp single
        umax = 0.d0
    !$omp end single
    do i=istart,iend
        u(i) = (1.d0 + c(i)*dt) * u(i)
        umax_thread = max(umax_thread, u(i))
    enddo
    !$omp critical
        umax = max(umax, umax_thread)
    !$omp end critical
    !$omp barrier
    if (umax > 100) exit
    !$omp barrier
enddo
```

OpenMP example with shared exit criterion

If there were **no** barriers, the following could happen:

Thread 0 executes critical section first, setting `umax` to 0.5.

Thread 0 checks if `umax > 100`. False, starts next iteration.

Thread 1 executes critical section, updating `umax` to 110.

Thread 1 checks if `umax > 100`. True, so it exits.

Thread 0 next sets `umax` to 0.4.

Thread 0 might never reach `umax > 100`. **Runs forever.**

OpenMP example with shared exit criterion

If there were **no** barriers, the following could happen:

Thread 0 executes critical section first, setting `umax` to 0.5.

Thread 0 checks if `umax > 100`. False, starts next iteration.

Thread 1 executes critical section, updating `umax` to 110.

Thread 1 checks if `umax > 100`. True, so it exits.

Thread 0 next sets `umax` to 0.4.

Thread 0 might never reach `umax > 100`. **Runs forever.**

With only first barrier, the following could happen:

`umax < 100` in iteration m .

Thread 1 checks if `umax > 100`. Go to iteration $m + 1$.

Thread 1 does iteration on i and sets `umax > 100`,

Stops at first barrier.

Thread 0 (iteration m) checks if `umax > 100`. True, **Exits.**

Thread 0 never reaches first barrier again, **code hangs.**

HPSC — Lecture 18

Outline:

- MPI concepts
- Communicators, broadcast, reduce

Reading:

- /codes/mpi

MPI — Message Passing Interface

OpenMP can only be used on **shared memory** systems with a single address space used by all threads.

Distributed memory systems require a different approach.

e.g. clusters of computers, supercomputers, heterogeneous networks.

MPI — Message Passing Interface

OpenMP can only be used on **shared memory** systems with a single address space used by all threads.

Distributed memory systems require a different approach.

e.g. clusters of computers, supercomputers, heterogeneous networks.

Message Passing:

SPMD model: All processors execute same program, but with different data.

Program manages memory by placing data in processes.

Data that must be shared is explicitly sent between processes.

MPI References

There are several implementations of MPI available.

The VM has Open MPI installed, see www.open-mpi.org.

The Argonne National Lab version [MPICH](#) is also widely used.

See also the [MPI Standard](#)

Standard reference book:

W. Gropp, E. Lusk, A. Skjellum, *Using MPI*, Second Edition,
MIT Press, 1999. [link](#)

[Bill Gropp's tutorials](#)

MPI — Simple example

```
program test1
    use mpi
    implicit none
    integer :: ierr, numprocs, proc_num,
    call mpi_init(ierr)
    call mpi_comm_size(MPI_COMM_WORLD, numprocs, ierr)
    call mpi_comm_rank(MPI_COMM_WORLD, proc_num, ierr)
    print *, 'Hello from Process ', proc_num, &
              ' of ', numprocs, ' processes'
    call mpi_finalize(ierr)
end program test1
```

Always need to: use mpi,

Start with mpi_init,

End with mpi_finalize.

Compiling and running MPI code (Fortran)

Try this test:

```
$ cd /codes/mpi  
$ mpif90 test1.f90  
$ mpiexec -n 4 a.out
```

You should see output like:

```
Hello from Process number 1 of 4 processes  
Hello from Process number 3 of 4 processes  
Hello from Process number 0 of 4 processes  
Hello from Process number 2 of 4 processes
```

Note: Number of processors is specified with `mpiexec`.

MPI Communicators

All communication takes place in **groups of processes**.

Communication takes place in some **context**.

A group and a context are combined in a **communicator**.

MPI_COMM_WORLD is a communicator provided by default that includes **all processors**.

MPI Communicators

All communication takes place in **groups of processes**.

Communication takes place in some **context**.

A group and a context are combined in a **communicator**.

MPI_COMM_WORLD is a communicator provided by default that includes **all processors**.

MPI_COMM_SIZE(**comm**, **numprocs**, **ierr**) returns the number of processors in communicator **comm**.

MPI_COMM_RANK(**comm**, **proc_num**, **ierr**) returns the rank of this processor in communicator **comm**.

mpi module

The `mpi` module includes:

Subroutines such as `mpi_init`, `mpi_comm_size`,
`mpi_comm_rank`, ...

Global variables such as

`MPI_COMM_WORLD`: a communicator,

`MPI_INTEGER`: used to specify the type of data being sent

`MPI_SUM`: used to specify a type of reduction

Remember: Fortran is **case insensitive**:

`mpi_init` is the same as `MPI_INIT`.

MPI functions

There are 125 MPI functions.

Can write many program with these 8:

- MPI_INIT(ierr) Initialize
- MPI_FINALIZE(ierr) Finalize
- MPI_COMM_SIZE(...) Number of processors
- MPI_COMM_RANK(...) Rank of this processor
- MPI_SEND(...) Send a message
- MPI_RECV(...) Receive a message
- MPI_BCAST(...) Broadcast to other processors
- MPI_REDUCE(...) Reduction operation

Example: Approximate π

Use $\pi = 4 \int_0^1 \frac{1}{1+x^2} dx$

$$\approx 4\Delta x \sum_{i=1}^n \frac{1}{1+x_i^2} \quad (\text{midpoint rule})$$

where $\Delta x = 1/n$ and $x_i = (i - 1/2)\Delta x$.

Fortran:

```
dx = 1.d0 / n
pisum = 0.d0
do i=1,n
    x = (i-0.5d0) * dx
    pisum = pisum + 1.d0 / (1.d0 + x**2)
enddo
pi = 4.d0 * dx * pisum
```

Approximate π using OpenMP parallel do

```
n = 1000
dx = 1.d0 / n
pisum = 0.d0
!$omp parallel do reduction(+: pisum) &
           private(x)
do i=1,n
    x = (i-0.5d0) * dx
    pisum = pisum + 1.d0 / (1.d0 + x**2)
enddo
pi = 4.d0 * dx * pisum
```

Approximate π using OpenMP — parallel chunks

```
n = 1000
points_per_thread = (n + nthreads - 1) / nthreads
pisum = 0.d0

 !$omp parallel private(i,pisum_thread,x, &
 !$omp                      istart,iend,thread_num)

 !$ thread num = omp_get_thread num()
 istart = Thread num * points_per_thread + 1
 iend = min((thread_num+1) * points_per_thread, n)

 pisum_thread = 0.d0
 do i=istart,iend
     x = (i-0.5d0)*dx
     pisum_thread = pisum_thread + &
                     1.d0 / (1.d0 + x**2)
 enddo

 !$omp critical
     pisum = pisum + pisum_thread
 !$omp end critical
 !$omp end parallel

 pi = 4.d0 * dx * pisum
```

Approximate π using MPI

```
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, proc_num, ierr)

if (proc_num == 0) n = 1000
! Broadcast to all processes:
call MPI_BCAST(n, 1, MPI_INTEGER, 0, &
               MPI_COMM_WORLD, ierr)

dx = 1.d0/n

points_per_proc = (n + numprocs - 1)/numprocs
istart = proc_num * points_per_proc + 1
iend = min((proc_num + 1)*points_per_proc, n)

pisum_proc = 0.d0
do i=istart,iend
    x = (i-0.5d0)*dx
    pisum_proc = pisum_proc + 1.d0 / (1.d0 + x**2)
enddo
call MPI_REDUCE(pisum_proc,pisum,1, &
                MPI_DOUBLE PRECISION,MPI_SUM,0, &
                MPI_COMM_WORLD,ierr)

if (proc_num == 0) then
    pi = 4.d0 * dx * pisum
endif
```

MPI Broadcast

Broadcast a value from Process **root** to all other processes.

General form:

```
call MPI_BCAST(start, count, &
               datatype, root, &
               comm, ierr)
```

where:

- start: starting address (variable, array element)
- count: number of elements to broadcast
- datatype: type of each element
- root: process doing the broadcast
- comm: communicator

MPI Broadcast Examples

```
call MPI_BCAST(start, count, &
               datatype, root, &
               comm, ierr)
```

Broadcast 1 double precision value:

```
call MPI_BCAST(x, 1, &
               MPI_DOUBLE_PRECISION, 0, &
               MPI_COMM_WORLD, ierr)}
```

Broadcast jth column of a matrix (contiguous in memory):

```
real(kind=8), dimension(nrows, ncols) :: a
...
call MPI_BCAST(a(1,j), nrows, &
               MPI_DOUBLE_PRECISION, 0, &
               MPI_COMM_WORLD, ierr)
```

MPI Broadcast Examples

Broadcast *i*th row of a matrix (not contiguous!):

```
real(kind=8), dimension(nrows, ncols) :: a
real(kind=8), dimension(ncols) :: buffer
...
do j=1,ncols
    buffer(j) = a(i,j)
enddo

call MPI_BCAST(buffer, ncols, &
               MPI_DOUBLE_PRECISION, 0, &
               MPI_COMM_WORLD, ierr)
```

MPI Broadcast Examples

Broadcast *i*th row of a matrix (not contiguous!):

```
real(kind=8), dimension(nrows, ncols) :: a
real(kind=8), dimension(ncols) :: buffer
...
do j=1,ncols
    buffer(j) = a(i,j)
enddo

call MPI_BCAST(buffer, ncols, &
               MPI_DOUBLE_PRECISION, 0, &
               MPI_COMM_WORLD, ierr)
```

Can instead create a strided datatype with
MPI_TYPE_VECTOR.

MPI Reduce

Collect values from all processes and reduce to a scalar.

General form:

```
call MPI_REDUCE(sendbuf, recvbuf, count, &
                 datatype, op, root, &
                 comm, ierr)
```

where:

- **sendbuf:** source address
- **recvbuf:** result address
- **count:** number of elements to send / receive
- **datatype:** type of each element
- **op:** reduction operation
- **root:** process receiving and reducing
- **comm:** communicator

MPI Reduce

A few possible reduction operations op:

- MPI_SUM: add together
- MPI_PROD: multiply together
- MPI_MAX: take maximum
- MPI_MIN: take minimum
- MPI_LAND: logical and
- MPI_LOR: logical or

MPI Reduce

Examples: Compute $\|x\|_\infty = \max_i |x_i|$ for a distributed vector:

```
xnorm_proc = 0.d0
do i=istart,iend
    xnorm_proc = max(xnorm_proc, abs(x(i)))
enddo

call MPI_REDUCE(xnorm_proc, xnorm, 1, &
                MPI_DOUBLE_PRECISION, MPI_MAX, 0, &
                MPI_COMM_WORLD, ierr)

if (proc_num == 0) print "norm of x = ", xnorm
```

Note: Do not need an `MPI_BARRIER` before or after the Reduce.

Processors do not exit from `MPI_REDUCE` until all have called the subroutine.

MPI Reduce

This code is wrong:

```
if (proc_num /= 0) then
    call MPI_REDUCE(xnorm_proc, xnorm, 1, &
                    MPI_DOUBLE_PRECISION,MPI_SUM,0, &
                    MPI_COMM_WORLD, ierr)
    print "Done with Reduce: ", proc_num
endif
if (proc_num == 0) print "norm of x = ", xnorm
```

With more than one process, the Reduce statement is called by all but one.

None of them will ever print the “Done with Reduce” statement or continue to run. (**Code hangs.**)

MPI Reduce

This code is wrong:

```
if (proc_num /= 0) then
    call MPI_REDUCE(xnorm_proc, xnorm, 1, &
                    MPI_DOUBLE_PRECISION,MPI_SUM,0, &
                    MPI_COMM_WORLD, ierr)
    print "Done with Reduce: ", proc_num
endif
if (proc_num == 0) print "norm of x = ", xnorm
```

With more than one process, the Reduce statement is called by all but one.

None of them will ever print the “Done with Reduce” statement or continue to run. (**Code hangs.**)

If only processors 1, 2, ... should participate in Reduce, need a different **communicator** than `MPI_COMM_WORLD`.

MPI Reduce for vectors

Compute: $\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^m |a_{ij}|$ for an $m \times n$ matrix A .

Suppose there are m processes and the i th process has a vector `arow(1:n)` containing the i th row of A .

Use `MPI_REDUCE` to sum

the first element of each row vector into `colsum(1)`,
second element of each row vector into `colsum(2)`, etc.

```
real(kind=8) :: arow(n), arow_abs(n), colsum(n)
...
arow_abs = abs(arow)
call MPI_REDUCE(arow_abs(1), colsum, n, &
                MPI_DOUBLE PRECISION, MPI_SUM, 0, &
                MPI_COMM_WORLD, ierr)
if (proc_num == 0) then
    anorm = maxval(colsum)
    print "1-norm of A = ", anorm
endif
```

HPSC – Lecture 19

Outline:

- Review MPI, reduce and bcast
- MPI send and receive
- Master–Worker paradigm

References:

- `$/codes/mpi`
- [MPI Standard](#)
- [OpenMPI](#)

MPI — Simple example

```
program test1
    use mpi
    implicit none
    integer :: ierr, numprocs, proc_num,
    call mpi_init(ierr)
    call mpi_comm_size(MPI_COMM_WORLD, numprocs, ierr)
    call mpi_comm_rank(MPI_COMM_WORLD, proc_num, ierr)
    print *, 'Hello from Process ', proc_num, &
              ' of ', numprocs, ' processes'
    call mpi_finalize(ierr)
end program test1
```

Always need to: use mpi,

Start with mpi_init,

End with mpi_finalize.

Compiling and running MPI code (Fortran)

Try this test:

```
$ cd $UWHPSC/codes/mpi  
$ mpif90 test1.f90  
$ mpiexec -n 4 a.out
```

You should see output like:

```
Hello from Process number 1 of 4 processes  
Hello from Process number 3 of 4 processes  
Hello from Process number 0 of 4 processes  
Hello from Process number 2 of 4 processes
```

Note: Number of processors is specified with `mpiexec`.

MPI Communicators

All communication takes place in **groups of processes**.

Communication takes place in some **context**.

A group and a context are combined in a **communicator**.

MPI_COMM_WORLD is a communicator provided by default that includes **all processors**.

MPI Communicators

All communication takes place in **groups of processes**.

Communication takes place in some **context**.

A group and a context are combined in a **communicator**.

MPI_COMM_WORLD is a communicator provided by default that includes **all processors**.

MPI_COMM_SIZE(**comm**, **numprocs**, **ierr**) returns the number of processors in communicator **comm**.

MPI_COMM_RANK(**comm**, **proc_num**, **ierr**) returns the rank of this processor in communicator **comm**.

mpi module

The `mpi` module includes:

Subroutines such as `mpi_init`, `mpi_comm_size`,
`mpi_comm_rank`, ...

Global variables such as

`MPI_COMM_WORLD`: a communicator,

`MPI_INTEGER`: used to specify the type of data being sent

`MPI_SUM`: used to specify a type of reduction

Remember: Fortran is **case insensitive**:

`mpi_init` is the same as `MPI_INIT`.

MPI functions

There are 125 MPI functions.

Can write many program with these 8:

- MPI_INIT(ierr) Initialize
- MPI_FINALIZE(ierr) Finalize
- MPI_COMM_SIZE(...) Number of processors
- MPI_COMM_RANK(...) Rank of this processor
- MPI_SEND(...) Send a message
- MPI_RECV(...) Receive a message
- MPI_BCAST(...) Broadcast to other processors
- MPI_REDUCE(...) Reduction operation

MPI Reduce

Examples: Compute $\|x\|_\infty = \max_i |x_i|$ for a distributed vector:
(each process has some subset of x elements)

```
xnorm_proc = 0.d0
! set istart and iend for each process
do i=istart,iend
    xnorm_proc = max(xnorm_proc, abs(x(i)))
enddo

call MPI_REDUCE(xnorm_proc, xnorm, 1, &
                MPI_DOUBLE_PRECISION, MPI_MAX, 0, &
                MPI_COMM_WORLD,ierr)

if (proc_num == 0) print "norm of x = ", xnorm
```

Processors do not exit from MPI_REDUCE until all have called the subroutine.

Normalize the vector x : Replace x by $x/\|x\|_\infty$

```
! compute xnorm_proc on each process as before.

call MPI_REDUCE(xnorm_proc, xnorm, 1, &
                 MPI_DOUBLE_PRECISION,MPI_MAX, 0, &
                 MPI_COMM_WORLD,ierr)
! only Process 0 has the value of xnorm

call MPI_BCAST(xnorm, 1, &
                 MPI_DOUBLE_PRECISION, 0, &
                 MPI_COMM_WORLD,ierr)
! now every process has the value of xnorm

do i=istart,iend
    x(i) = x(i) / xnorm
enddo
```

MPI AllReduce

To make a reduction available to *all* processes:

```
call MPI_REDUCE(xnorm_proc, xnorm, 1, &
                 MPI_DOUBLE_PRECISION, MPI_MAX, 0, &
                 MPI_COMM_WORLD, ierr)
! only Process 0 has the value of xnorm

call MPI_BCAST(xnorm, 1, &
               MPI_DOUBLE_PRECISION, 0, &
               MPI_COMM_WORLD, ierr)
```

One-step alternative: simpler and perhaps more efficient...

```
call MPI_ALLREDUCE(xnorm_proc, xnorm, 1, &
                   MPI_DOUBLE_PRECISION, MPI_MAX, &
                   MPI_COMM_WORLD, ierr)
```

Remember — no shared memory

Suppose all of vector x is stored on memory of Process 0,

We want to normalize x (using more than one processor),

and replace x by normalized version in memory of Process 0.

Remember — no shared memory

Suppose all of vector x is stored on memory of Process 0,

We want to normalize x (using more than one processor),

and replace x by normalized version in memory of Process 0.

We would have to:

- Send parts of x to other processes,
- Compute `xnorm_proc` on each process,
- Use `MPI_ALLREDUCE` to combine into `xnorm` and broadcast to all processes,
- Normalize part of x on each process,
- Send each part of normalized x back to Process 0.

Remember — no shared memory

Suppose all of vector x is stored on memory of Process 0,

We want to normalize x (using more than one processor),

and replace x by normalized version in memory of Process 0.

We would have to:

- Send parts of x to other processes,
- Compute `xnorm_proc` on each process,
- Use `MPI_ALLREDUCE` to combine into `xnorm` and broadcast to all processes,
- Normalize part of x on each process,
- Send each part of normalized x back to Process 0.

Communication cost will probably make this much slower than just normalizing all of x on Process 0!

Remember — no shared memory

Might be worthwhile if much more work is required for each element of x .

Suppose all of vector x is stored on memory of Process 0,

Want to solve an expensive differential equation
with different initial conditions given by elements of x ,

and then collect all results on Process 0.

Remember — no shared memory

Might be worthwhile if much more work is required for each element of x .

Suppose all of vector x is stored on memory of Process 0,

Want to solve an expensive differential equation
with different initial conditions given by elements of x ,

and then collect all results on Process 0.

Master–Worker paradigm:

- Process 0 sends different chunks of x to Process 1, 2, . . .
- Each process grinds away to solve differential equations
- Each process sends results back to Process 0.

MPI Send and Receive

`MPI_BCAST` sends from one process to all processes.

Often want to send selectively from Process i to Process j .

Use `MPI_SEND` and `MPI_RECV`.

MPI Send and Receive

`MPI_BCAST` sends from one process to all processes.

Often want to send selectively from Process i to Process j .

Use `MPI_SEND` and `MPI_RECV`.

Need a way to `tag` messages so they can be identified.

The parameter `tag` is an integer that can be matched to identify a message.

Tag can also be used to provide information about what is being sent, for example if a Master process sends rows of a matrix to other processes, the `tag` might be the row number.

MPI Send

Send value(s) from this Process to Process dest.

General form:

```
call MPI_SEND(start, count, &  
            datatype, dest, &  
            tag, comm, ierr)
```

where:

- start: starting address (variable, array element)
- count: number of elements to send
- datatype: type of each element
- dest: destination process
- tag: identifier tag (integer between 0 and 32767)
- comm: communicator

MPI Receive

Receive value(s) from Process source with label tag.

General form:

```
call MPI_RECV(start, count, &
              datatype, source, &
              tag, comm, status, ierr)
```

where:

- **source:** source process
- **tag:** identifier tag (integer between 0 and 32767)
- **comm:** communicator
- **status:** integer array of length `MPI_STATUS_SIZE`.

MPI Receive

Receive value(s) from Process **source** with label **tag**.

General form:

```
call MPI_RECV(start, count, &
              datatype, source, &
              tag, comm, status, ierr)
```

where:

- **source:** source process
- **tag:** identifier tag (integer between 0 and 32767)
- **comm:** communicator
- **status:** integer array of length **MPI_STATUS_SIZE**.

source could be **MPI_ANY_SOURCE** to match any source.

tag could be **MPI_ANY_TAG** to match any tag.

MPI Send and Receive — simple example

```
if (proc_num == 4) then
    i = 55
    call MPI_SEND(i, 1, MPI_INTEGER, 3, 21, &
                  MPI_COMM_WORLD, ierr)
    endif
if (proc_num == 3) then
    call MPI_RECV(j, 1, MPI_INTEGER, 4, 21, &
                  MPI_COMM_WORLD, status, ierr)
    print *, "j = ", j
    endif
```

Processor 3 will print j = 55

MPI Send and Receive — simple example

```
if (proc_num == 4) then
    i = 55
    call MPI_SEND(i, 1, MPI_INTEGER, 3, 21, &
                  MPI_COMM_WORLD, ierr)
    endif
if (proc_num == 3) then
    call MPI_RECV(j, 1, MPI_INTEGER, 4, 21, &
                  MPI_COMM_WORLD, status, ierr)
    print *, "j = ", j
    endif
```

Processor 3 will print $j = 55$

The tag is 21. (Arbitrary integer between 0 and 32767)

MPI Send and Receive — simple example

```
if (proc_num == 4) then
    i = 55
    call MPI_SEND(i, 1, MPI_INTEGER, 3, 21, &
                  MPI_COMM_WORLD, ierr)
    endif
if (proc_num == 3) then
    call MPI_RECV(j, 1, MPI_INTEGER, 4, 21, &
                  MPI_COMM_WORLD, status, ierr)
    print *, "j = ", j
    endif
```

Processor 3 will print $j = 55$

The tag is 21. (Arbitrary integer between 0 and 32767)

Blocking Receive: Processor 3 won't return from `MPI_RECV` until message is received.

MPI Send and Receive — simple example

```
if (proc_num == 4) then
    i = 55
    call MPI_SEND(i, 1, MPI_INTEGER, 3, 21, &
                  MPI_COMM_WORLD, ierr)
    endif
if (proc_num == 3) then
    call MPI_RECV(j, 1, MPI_INTEGER, 4, 21, &
                  MPI_COMM_WORLD, status, ierr)
    print *, "j = ", j
    endif
```

Processor 3 will print $j = 55$

The tag is 21. (Arbitrary integer between 0 and 32767)

Blocking Receive: Processor 3 won't return from `MPI_RECV` until message is received.

Run-time error if `num_procs <= 4` (Procs are 0,1,2,3)

Send/Receive example

Pass value of `i` from Processor 0 to 1 to 2 ... to `num_procs-1`

```
if (proc_num == 0) then
    i = 55
    call MPI_SEND(i, 1, MPI_INTEGER, 1, 21, &
                  MPI_COMM_WORLD, ierr)
endif

else if (proc_num < num_procs - 1) then
    call MPI_RECV(i, 1, MPI_INTEGER, proc_num-1, 21, &
                  MPI_COMM_WORLD, status,-ierr)
    call MPI_SEND(i, 1, MPI_INTEGER, proc_num+1, 21, &
                  MPI_COMM_WORLD, ierr)

else if (proc_num == num_procs - 1) then
    call MPI_RECV(i, 1, MPI_INTEGER, proc_num-1, 21, &
                  MPI_COMM_WORLD, status,-ierr)
    print *, "i = ", i
endif
```

MPI Receive

Receive value(s) from Process source with label tag.

General form:

```
call MPI_RECV(start, count, &
              datatype, source, &
              tag, comm, status, ierr)
```

where:

- **source:** source process
- **tag:** identifier tag (integer between 0 and 32767)
- **comm:** communicator
- **status:** integer array of length `MPI_STATUS_SIZE`.

MPI Receive

Receive value(s) from Process **source** with label **tag**.

General form:

```
call MPI_RECV(start, count, &
              datatype, source, &
              tag, comm, status, ierr)
```

where:

- **source**: source process
- **tag**: identifier tag (integer between 0 and 32767)
- **comm**: communicator
- **status**: integer array of length **MPI_STATUS_SIZE**.

source could be **MPI_ANY_SOURCE** to match any source.

tag could be **MPI_ANY_TAG** to match any tag.

MPI Receive — status argument

```
call MPI_RECV(start, count, &  
             datatype, source, &  
             tag, comm, status, ierr)
```

Elements of the `status` array give additional useful information about the message received.

In particular,

`status(MPI_SOURCE)` is the **source** of the message,
May be needed if `source = MPI_ANY_SOURCE`.

`status(MPI_TAG)` is the **tag** of the message received,
May be needed if `tag = MPI_ANY_TAG`.

Another Send/Receive example

Master (Processor 0) sends j th column to Worker Processor j ,
gets back 1-norm to store in $\text{anorm}(j), j = 1, \dots, \text{ncols}$

```
! code for Master (Processor 0):
if (proc_num == 0) then
    do j=1,ncols
        call MPI_SEND(a(1,j), nrows, MPI_DOUBLE PRECISION, &
                      j, j, MPI_COMM_WORLD, ierr)
    enddo
    do j=1,ncols
        call MPI_RECV(colnorm, 1, MPI_DOUBLE PRECISION, &
                      MPI_ANY_SOURCE, MPI_ANY_TAG, &
                      MPI_COMM_WORLD, status, ierr)
        jj = status(MPI_TAG)
        anorm(jj) = colnorm
    enddo
endif
```

Note: Master may receive back in any order!

`MPI_ANY_SOURCE` will match first to arrive.

The tag is used to tell which column's norm has arrived (jj).

Send and Receive example — worker code

Master (Processor 0) sends j th column to Worker Processor j ,
gets back 1-norm to store in $\text{anorm}(j), j = 1, \dots, \text{ncols}$

```
! code for Workers (Processors 1, 2, ...):
if (proc_num /= 0) then
    call MPI_RECV(colvect, nrows, MPI_DOUBLE_PRECISION, &
                  0, MPI_ANY_TAG, &
                  MPI_COMM_WORLD, status, ierr)
    j = status(MPI_TAG)      ! this is the column number
                           ! (should agree with proc_num)

    colnorm = 0.d0
    do i=1,nrows
        colnorm = colnorm + abs(colvect(i))
    enddo

    call MPI_SEND(colnorm, 1, MPI_DOUBLE_PRECISION, &
                  0, j, MPI_COMM_WORLD, ierr)
endif
```

Note: Sends back to Process 0 with tag j .

Send **may** be blocking

```
if (proc_num == 4) then
    i = 55
    call MPI_SEND(i, 1, MPI_INTEGER, 3, 21, &
                  MPI_COMM_WORLD, ierr)
    call MPI_RECV(j, 1, MPI_INTEGER, 3, 22, &
                  MPI_COMM_WORLD, status, ierr)
endif

if (proc_num == 3) then
    j = 66
    call MPI_SEND(j, 1, MPI_INTEGER, 4, 22, &
                  MPI_COMM_WORLD, ierr)
    call MPI_RECV(i, 1, MPI_INTEGER, 4, 21, &
                  MPI_COMM_WORLD, status, ierr)
endif
```

Both processors *might* get stuck in MPI_SEND!

May depend on size of data and send buffer.

Blocking send: MPI_SSEND. See [documentation](#)

Send **may** be blocking

```
if (proc_num == 4) then
    i = 55
    call MPI_SEND(i, 1, MPI_INTEGER, 3, 21, &
                  MPI_COMM_WORLD, ierr)
    call MPI_RECV(j, 1, MPI_INTEGER, 3, 22, &
                  MPI_COMM_WORLD, status, ierr)
endif

if (proc_num == 3) then
    j = 66
    call MPI_SEND(j, 1, MPI_INTEGER, 4, 22, &
                  MPI_COMM_WORLD, ierr)
    call MPI_RECV(i, 1, MPI_INTEGER, 4, 21, &
                  MPI_COMM_WORLD, status, ierr)
endif
```

Both processors *might* get stuck in MPI_SEND!

May depend on size of data and send buffer.

Blocking send: MPI_SSEND. See [documentation](#)

There are also non-blocking sends and receives:

MPI_ISEND, MPI_IRECV

Non-blocking receive

```
call MPI_IRecv(start, count, datatype, &  
               source, tag, comm, request, ierror)
```

Additional argument: `request`.

Program continues after initiating receive,

Can later check if it has finished with

```
call MPI_Test(request, flag, status, ierror)
```

`flag` is logical output variable.

Or can later wait for it to finish with

```
call MPI_Wait(request, status, ierror)
```

HPSC – Lecture 20

Outline:

- Heat equation and discretization
- Iterative methods

Sample codes:

- [/codes/openmp/jacobi1d_omp1.f90](#)
- [/codes/openmp/jacobi1d_omp2.f90](#)

Heat Equation / Diffusion Equation

Partial differential equation (PDE) for $u(x, t)$
in one space dimension and time.

u represents temperature in a 1-dimensional metal rod.

Or concentration of a chemical diffusing in a tube of water.

Heat Equation / Diffusion Equation

Partial differential equation (PDE) for $u(x, t)$
in one space dimension and time.

u represents temperature in a 1-dimensional metal rod.

Or concentration of a chemical diffusing in a tube of water.

The PDE is

$$ut(x, t) = Du_{xx}(x, t) + f(x, t)$$

where subscripts represent partial derivatives,

D = diffusion coefficient (assumed constant in space & time),

$f(x, t)$ = source term (heat or chemical being added/removed).

Heat Equation / Diffusion Equation

Partial differential equation (PDE) for $u(x, t)$
in one space dimension and time.

u represents temperature in a 1-dimensional metal rod.

Or concentration of a chemical diffusing in a tube of water.

The PDE is

$$ut(x, t) = Du_{xx}(x, t) + f(x, t)$$

where subscripts represent partial derivatives,

D = diffusion coefficient (assumed constant in space & time),

$f(x, t)$ = source term (heat or chemical being added/removed).

Also need initial conditions $u(x, 0)$

and boundary conditions $u(x_1, t), u(x_2, t)$.

Steady state diffusion

If $f(x, t) = f(x)$ does not depend on time and if the boundary conditions don't depend on time, then $u(x, t)$ will converge towards steady state distribution satisfying

$$0 = Du_{xx}(x) + f(x)$$

(by setting $u_t = 0$.)

This is now an [ordinary differential equation \(ODE\)](#) for $u(x)$.

Steady state diffusion

If $f(x, t) = f(x)$ does not depend on time and if the boundary conditions don't depend on time, then $u(x, t)$ will converge towards steady state distribution satisfying

$$0 = Du_{xx}(x) + f(x)$$

(by setting $u_t = 0$.)

This is now an [ordinary differential equation \(ODE\)](#) for $u(x)$.

We can solve this on an interval, say $0 \leq x \leq 1$ with

[Boundary conditions:](#)

$$u(0) = \alpha, \quad u(1) = \beta.$$

Steady state diffusion

More generally: Take $D = 1$ or absorb in f ,

$$u_{xx}(x) = -f(x) \quad \text{for } 0 \leq x \leq 1,$$

Boundary conditions:

$$u(0) = \alpha, \quad u(1) = \beta.$$

Can be solved exactly if we can integrate f twice and use boundary conditions to choose the two constants of integration.

Steady state diffusion

More generally: Take $D = 1$ or absorb in f ,

$$u_{xx}(x) = -f(x) \quad \text{for } 0 \leq x \leq 1,$$

Boundary conditions:

$$u(0) = \alpha, \quad u(1) = \beta.$$

Can be solved exactly if we can integrate f twice and use boundary conditions to choose the two constants of integration.

Example: $\alpha = 20$, $\beta = 60$, $f(x) = 0$ (no heat source)

Solution: $u(x) = \alpha + x(\beta - \alpha) \Rightarrow u''(x) = 0$.

No heat source \Rightarrow linear variation in steady state ($u_{xx} = 0$).

Steady state diffusion

More generally: Take $D = 1$ or absorb in f ,

$$u_{xx}(x) = -f(x) \quad \text{for } 0 \leq x \leq 1,$$

Boundary conditions:

$$u(0) = \alpha, \quad u(1) = \beta.$$

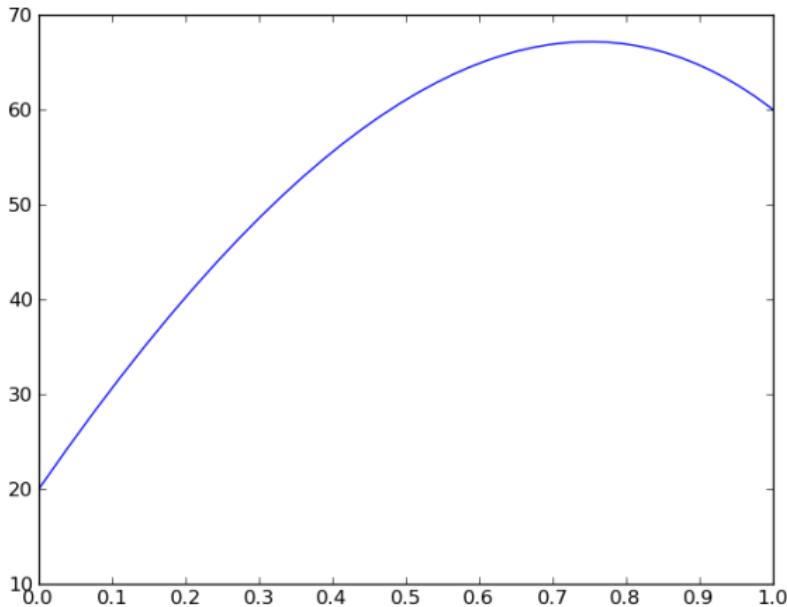
Can be solved exactly if we can integrate f twice and use boundary conditions to choose the two constants of integration.

More interesting example:

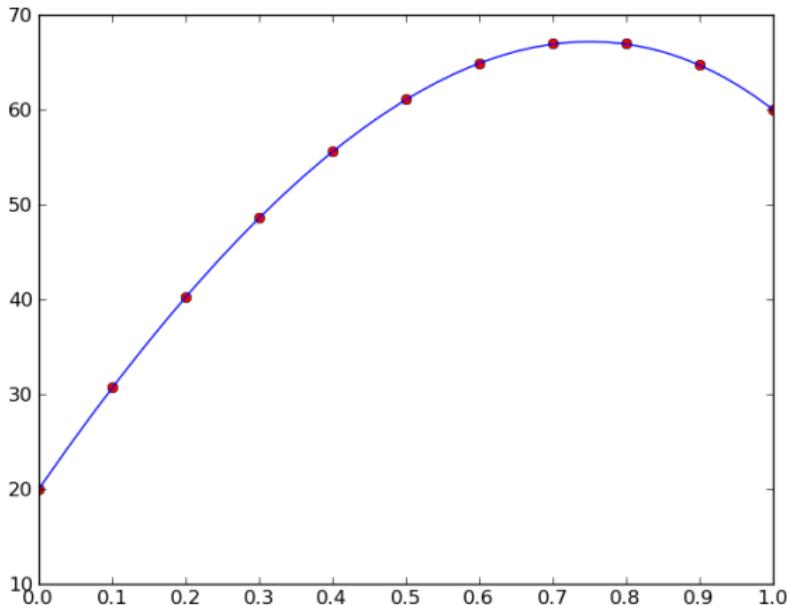
Example: $\alpha = 20$, $\beta = 60$, $f(x) = 100e^x$,

Solution: $u(x) = (100e - 60)x + 120 - 100e^x$.

Steady state diffusion



Steady state diffusion



For more complicated equations, [numerical methods](#) must generally be used, giving approximations at discrete points.

Finite difference method

Define grid points $x_i = i\Delta x$ in interval $0 \leq x \leq 1$, where

$$\Delta x = \frac{1}{n+1}$$

So $x_0 = 0$, $x_{n+1} = 1$, and the n grid points x_1, x_2, \dots, x_n are equally spaced inside the interval.

Finite difference method

Define grid points $x_i = i\Delta x$ in interval $0 \leq x \leq 1$, where

$$\Delta x = \frac{1}{n+1}$$

So $x_0 = 0$, $x_{n+1} = 1$, and the n grid points x_1, x_2, \dots, x_n are equally spaced inside the interval.

Let $U_i \approx u(x_i)$ denote approximate solution.

We know $U_0 = \alpha$ and $U_{n+1} = \beta$ from boundary conditions.

Finite difference method

Define grid points $x_i = i\Delta x$ in interval $0 \leq x \leq 1$, where

$$\Delta x = \frac{1}{n+1}$$

So $x_0 = 0$, $x_{n+1} = 1$, and the n grid points x_1, x_2, \dots, x_n are equally spaced inside the interval.

Let $U_i \approx u(x_i)$ denote approximate solution.

We know $U_0 = \alpha$ and $U_{n+1} = \beta$ from boundary conditions.

Idea: Replace differential equation for $u(x)$ by system of n algebraic equations for U_i values ($i = 1, 2, \dots, n$).

Finite difference method

$$U_i \approx u(x_i)$$

$$u_x(x_{i+1/2}) \approx \frac{U_{i+1} - U_i}{\Delta x}$$

$$u_x(x_{i-1/2}) \approx \frac{U_i - U_{i-1}}{\Delta x}$$

Finite difference method

$$U_i \approx u(x_i)$$

$$u_x(x_{i+1/2}) \approx \frac{U_{i+1} - U_i}{\Delta x}$$

$$u_x(x_{i-1/2}) \approx \frac{U_i - U_{i-1}}{\Delta x}$$

So we can approximate second derivative at x_i by:

$$\begin{aligned} u_{xx}(x_i) &\approx \frac{1}{\Delta x} \left(\frac{U_{i+1} - U_i}{\Delta x} - \frac{U_i - U_{i-1}}{\Delta x} \right) \\ &= \frac{1}{\Delta x^2} (U_{i-1} - 2U_i + U_{i+1}) \end{aligned}$$

Finite difference method

$$U_i \approx u(x_i)$$

$$u_x(x_{i+1/2}) \approx \frac{U_{i+1} - U_i}{\Delta x}$$

$$u_x(x_{i-1/2}) \approx \frac{U_i - U_{i-1}}{\Delta x}$$

So we can approximate second derivative at x_i by:

$$\begin{aligned} u_{xx}(x_i) &\approx \frac{1}{\Delta x^2} \left(\frac{U_{i+1} - U_i}{\Delta x} - \frac{U_i - U_{i-1}}{\Delta x} \right) \\ &= \frac{1}{\Delta x^2} (U_{i-1} - 2U_i + U_{i+1}) \end{aligned}$$

This gives coupled system of n linear equations:

$$\frac{1}{\Delta x^2} (U_{i-1} - 2U_i + U_{i+1}) = -f(x_i)$$

for $i = 1, 2, \dots, n$. With $U_0 = \alpha$ and $U_{n+1} = \beta$.

Tridiagonal linear system

$$\alpha - 2U_1 + U_2 = -\Delta x^2 f(x_1) \quad (i=1)$$

$$U_1 - 2U_2 + U_3 = -\Delta x^2 f(x_2) \quad (i=2)$$

Etc.

For $n = 5$:

$$\begin{bmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \end{bmatrix} = -\Delta x^2 \begin{bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ f(x_4) \\ f(x_5) \end{bmatrix} - \begin{bmatrix} \alpha \\ 0 \\ 0 \\ 0 \\ \beta \end{bmatrix}$$

Tridiagonal linear system

$$a - 2U_1 + U_2 = -\Delta x^2 f(x_1) \quad (i=1)$$

$$U_1 - 2U_2 + U_3 = -\Delta x^2 f(x_2) \quad (i=2)$$

Etc.

For $n = 5$:

$$\begin{bmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \\ U_5 \end{bmatrix} = -\Delta x^2 \begin{bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ f(x_4) \\ f(x_5) \end{bmatrix} - \begin{bmatrix} \alpha \\ 0 \\ 0 \\ 0 \\ \beta \end{bmatrix}$$

General $n \times n$ system requires $O(n^3)$ flops to solve.

Tridiagonal $n \times n$ system requires $O(n)$ flops to solve.

Could use LAPACK routine [dgtsv](#).

Heat equation in 2 dimensions

One-dimensional equation generalizes to

$$ut(x, y, t) = D(u_{xx}(x, y, t) + u_{yy}(x, y, t)) + f(x, y, t)$$

on some domain in the x - y plane, with initial and boundary conditions.

We will only consider rectangle $0 \leq x \leq 1$, $0 \leq y \leq 1$.

Heat equation in 2 dimensions

One-dimensional equation generalizes to

$$ut(x, y, t) = D(u_{xx}(x, y, t) + u_{yy}(x, y, t)) + f(x, y, t)$$

on some domain in the x - y plane, with initial and boundary conditions.

We will only consider rectangle $0 \leq x \leq 1$, $0 \leq y \leq 1$.

Steady state problem (with $D = 1$):

$$u_{xx}(x, y) + u_{yy}(x, y) = -f(x, y)$$

This is a PDE in two spatial variables. [\(Poisson Problem\)](#)

Heat equation in 2 dimensions

One-dimensional equation generalizes to

$$ut(x, y, t) = D(u_{xx}(x, y, t) + u_{yy}(x, y, t)) + f(x, y, t)$$

on some domain in the x - y plane, with initial and boundary conditions.

We will only consider rectangle $0 \leq x \leq 1$, $0 \leq y \leq 1$.

Steady state problem (with $D = 1$):

$$u_{xx}(x, y) + u_{yy}(x, y) = -f(x, y)$$

This is a PDE in two spatial variables. [\(Poisson Problem\)](#)

Laplace's equation if $f(x, y) \equiv 0$.

$\nabla^2 = (\partial_x^2 + \partial_y^2)$ is the Laplacian operator.

Finite difference equations for 2D Poisson problem

Let $U_{ij} \approx u(x_i, y_j)$.

Replace differential equation

$$u_{xx}(x, y) + u_{yy}(x, y) = -f(x, y)$$

by algebraic equations

$$\begin{aligned} & \frac{1}{\Delta x^2} (U_{i-1,j} - 2U_{i,j} + U_{i+1,j}) \\ & + \frac{1}{\Delta y^2} (U_{i,j-1} - 2U_{i,j} + U_{i,j+1}) = -f(x_i, y_j) \end{aligned}$$

Finite difference equations for 2D Poisson problem

Let $U_{ij} \approx u(x_i, y_j)$.

Replace differential equation

$$u_{xx}(x, y) + u_{yy}(x, y) = -f(x, y)$$

by algebraic equations

$$\begin{aligned} & \frac{1}{\Delta x^2} (U_{i-1,j} - 2U_{i,j} + U_{i+1,j}) \\ & + \frac{1}{\Delta y^2} (U_{i,j-1} - 2U_{i,j} + U_{i,j+1}) = -f(x_i, y_j) \end{aligned}$$

If $\Delta x = \Delta y = h$:

$$\frac{1}{h^2} (U_{i-1,j} + U_{i+1,j} + U_{i,j-1} + U_{i,j+1} - 4U_{i,j}) = -f(x_i, y_j).$$

Finite difference equations for 2D Poisson problem

$$\frac{1}{h^2} (U_{i-1,j} + U_{i+1,j} + U_{i,j-1} + U_{i,j+1} - 4U_{i,j}) = -f(x_i, y_j).$$

On $n \times n$ grid ($\Delta x = \Delta y = 1/(n+1)$) this gives a linear system of n^2 equations in n^2 unknowns.

The above equation must be satisfied for $i = 1, 2, \dots, n$ and $j = 1, 2, \dots, n$.

Matrix is $n^2 \times n^2$,

e.g. on 100 by 100 grid, matrix is $10,000 \times 10,000$.

Contains $(10,000)^2 = 100,000,000$ elements.

Finite difference equations for 2D Poisson problem

$$\frac{1}{h^2} (U_{i-1,j} + U_{i+1,j} + U_{i,j-1} + U_{i,j+1} - 4U_{i,j}) = -f(x_i, y_j).$$

On $n \times n$ grid ($\Delta x = \Delta y = 1/(n+1)$) this gives a linear system of n^2 equations in n^2 unknowns.

The above equation must be satisfied for $i = 1, 2, \dots, n$ and $j = 1, 2, \dots, n$.

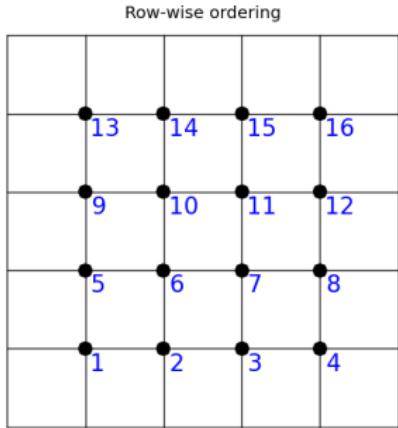
Matrix is $n^2 \times n^2$,

e.g. on 100 by 100 grid, matrix is $10,000 \times 10,000$.

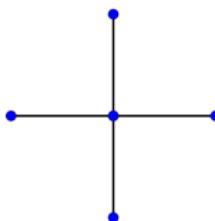
Contains $(10,000)^2 = 100,000,000$ elements.

Matrix is **sparse**: each row has at most 5 nonzeros out of n^2 elements! But structure is no longer tridiagonal.

Finite difference equations for 2D Poisson problem



stencil



Matrix has block tridiagonal structure:

$$A = \frac{1}{h^2} \begin{bmatrix} T & I & & \\ I & T & I & \\ & I & T & I \\ & & I & T \end{bmatrix} \quad T = \begin{bmatrix} -4 & 1 & & \\ 1 & -4 & 1 & \\ & 1 & -4 & 1 \\ & & 1 & -4 \end{bmatrix}$$

Iterative methods

Back to one space dimension first...

Coupled system of n linear equations:

$$(U_{i-1} - 2U_i + U_{i+1}) = -\Delta x^2 f(x_i)$$

for $i = 1, 2, \dots, n$. With $U_0 = \alpha$ and $U_{n+1} = \beta$.

Iterative method starts with initial guess $U^{[0]}$ to solution and then improves $U^{[k]}$ to get $U^{[k+1]}$ for $k = 0, 1, \dots$

Note: Generally does not involve modifying matrix A .

Do not have to store matrix A at all, only know about stencil.

Jacobi iteration

$$(U_{i-1} - 2U_i + U_{i+1}) = -\Delta x^2 f(x_i)$$

Solve for U_i :

$$U_i = \frac{1}{2} (U_{i-1} + U_{i+1} + \Delta x^2 f(x_i)).$$

Note: With no heat source, $f(x) = 0$,
the temperature at each point is average of neighbors.

Jacobi iteration

$$(U_{i-1} - 2U_i + U_{i+1}) = -\Delta x^2 f(x_i)$$

Solve for U_i :

$$U_i = \frac{1}{2} (U_{i-1} + U_{i+1} + \Delta x^2 f(x_i)).$$

Note: With no heat source, $f(x) = 0$,
the temperature at each point is average of neighbors.

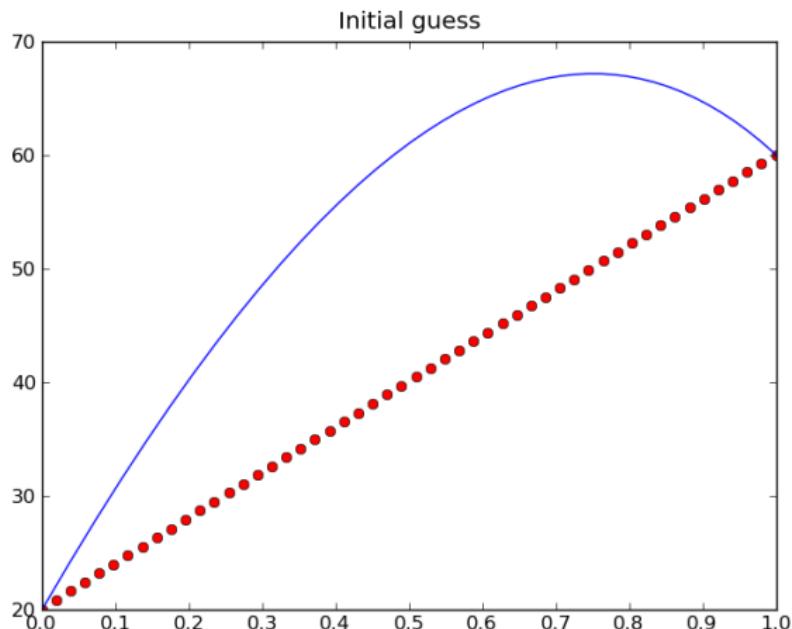
Suppose $U^{[k]}$ is an approximation to solution. Set

$$U_i^{[k+1]} = \frac{1}{2} (U_{i-1}^{[k]} + U_{i+1}^{[k]} + \Delta x^2 f(x_i)) \quad \square \quad \text{for } i = 1, 2, \dots, n.$$

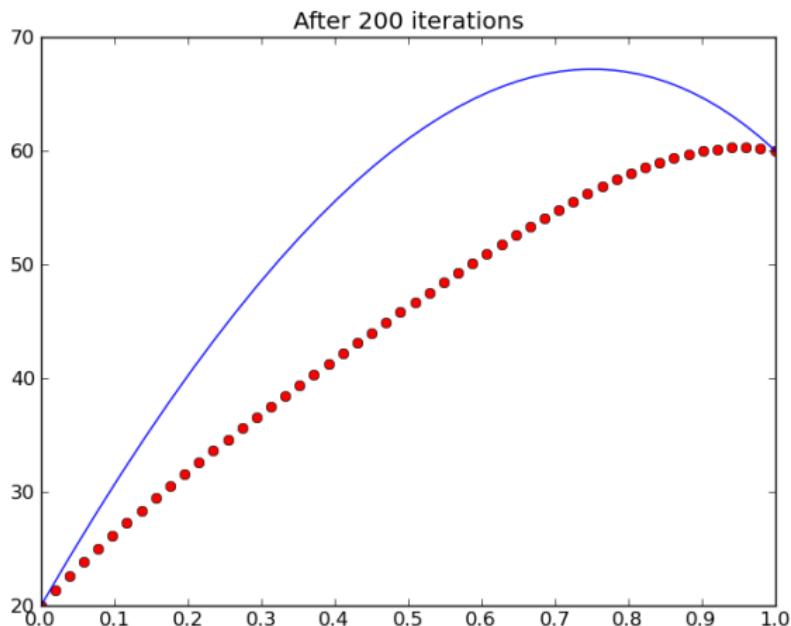
Repeat for $k = 0, 1, 2, \dots$ until convergence.

Can be shown to converge (eventually... **very slow!**)

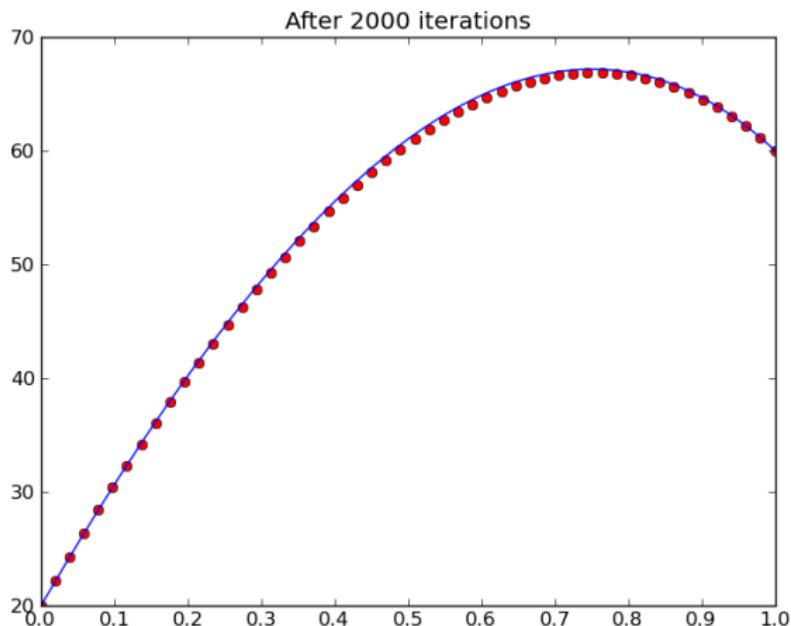
Slow convergence of Jacobi



Slow convergence of Jacobi



Slow convergence of Jacobi



Iterative methods

Jacobi iteration is about the worst possible iterative method.

But it's very simple, and useful as a test for parallelization.

Better iterative methods:

- Gauss-Seidel
- Successive Over-Relaxation (SOR)
- Conjugate gradients
- Preconditioned conjugate gradients
- Multigrid

Iterative methods – initialization

```
! allocate storage for boundary points too:  
allocate(x(0:n+1), u(0:n+1), f(0:n+1))  
  
dx = 1.d0 / (n+1.d0)  
  
 !$omp parallel do  
do i=0,n+1  
    ! grid points:  
    x(i) = i*dx  
    ! source term:  
    f(i) = 100.*exp(x(i))  
    ! initial guess (linear function):  
    u(i) = alpha + x(i)*(beta-alpha)  
enddo
```

Jacobi iteration in Fortran

```
uold = u ! starting values before updating
do iter=1,maxiter
    dumax = 0.d0
    do i=1,n
        u(i) = 0.5d0*(uold(i-1) + uold(i+1) + dx**2*f(i))
        dumax = max(dumax, abs(u(i)-uold(i)))
    enddo
    ! check for convergence:
    if (dumax .lt. tol) exit
    uold = u ! for next iteration
enddo
```

Note: we must use old value at $i - 1$ for Jacobi.

Otherwise we get the **Gauss-Seidel** method.

$$u(i) = 0.5d0 * (u(i-1) + u(i+1) + dx^{**}2 * f(i))$$

Jacobi iteration in Fortran

```
uold = u ! starting values before updating
do iter=1,maxiter
    dumax = 0.d0
    do i=1,n
        u(i) = 0.5d0*(uold(i-1) + uold(i+1) + dx**2*f(i))
        dumax = max(dumax, abs(u(i)-uold(i)))
    enddo
    ! check for convergence:
    if (dumax .lt. tol) exit
    uold = u ! for next iteration
enddo
```

Note: we must use old value at $i - 1$ for Jacobi.

Otherwise we get the **Gauss-Seidel** method.

$$u(i) = 0.5d0*(u(i-1) + u(i+1) + dx**2*f(i))$$

This actually converges faster!

Jacobi with OpenMP parallel do (fine grain)

See: [/codes/openmp/jacobi1d_omp1.f90](#)

```
uold = u ! starting values before updating
do iter=1,maxiter
    dumax = 0.d0
    !$omp parallel do reduction(max : dumax)
    do i=1,n
        u(i) = 0.5d0*(uold(i-1) + uold(i+1) + dx**2*f(i))
        dumax = max(dumax, abs(u(i)-uold(i)))
    enddo
    ! check for convergence:
    if (dumax .lt. tol) exit
    !$omp parallel do
    do i=1,n
        uold(i) = u(i) ! for next iteration
    enddo
enddo
```

Note: Forking threads twice each iteration.

Jacobi with OpenMP – coarse grain

General Approach:

- Fork threads only once at start of program.
- Each thread is responsible for some portion of the arrays, from $i=i_{\text{start}}$ to $i=i_{\text{end}}$.
- Each iteration, must copy u to u_{old} , update u , check for convergence.
- Convergence check requires coordination between threads to get global $dumax$.
- Print out final result after leaving parallel block

See code in the repository or the notes:

[codes/openmp/jacobi1d_omp2.f90](#)

Jacobi with MPI

Each process is responsible for some portion of the arrays,
from $i=i_{\text{start}}$ to $i=i_{\text{end}}$.

No shared memory: each process only has part of array.

Updating formula:

$$u(i) = 0.5d0 * (u_{\text{old}}(i-1) + u_{\text{old}}(i+1) + dx^{**2} * f(i))$$

Need to exchange values at boundaries:

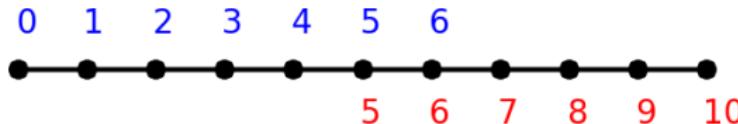
Updating at $i=i_{\text{start}}$ requires $u_{\text{old}}(i_{\text{start}-1})$

Updating at $i=i_{\text{end}}$ requires $u_{\text{old}}(i_{\text{start}+1})$

Example with $n = 9$ interior points (plus boundaries):

Process 0 has $i_{\text{start}} = 1, i_{\text{end}} = 5$

Process 1 has $i_{\text{start}} = 6, i_{\text{end}} = 9$



Jacobi with MPI — Sending to neighbors

```
call mpi_comm_rank(MPI_COMM_WORLD, me, ierr)
...
do iter = 1, maxiter
    uold = u

    if (me > 0) then
        ! Send left endpoint value to "left"
        call mpi_isend(uold(istart), 1, MPI_DOUBLE PRECI-
                        me -1, 1, MPI_COMM_WORLD, req1, ierr)
    end if

    if (me < ntasks-1) then
        ! Send right endpoint value to "right"
        call mpi_isend(uold(iend), 1, MPI_DOUBLE PRECISI-
                        me +1, 2, MPI_COMM_WORLD, req2, ierr)
    end if
end do
```

Note: Non-blocking `mpi_isend` is used,

Different tags (1 and 2) for left-going, right-going messages.

Jacobi with MPI — Receiving from neighbors

Note: `uold(istart)` from `me+1` goes into `uold(iend+1)`:
`uold(iend)` from `me-1` goes into `uold(istart-1)`:

```
do iter = 1, maxiter
    ! mpi_send's from previous slide
    if (me < ntasks-1) then
        ! Receive right endpoint value
        call mpi_recv(uold(iend+1), 1, MPI_DOUBLE_PRECIS
                      me + 1, 1, MPI_COMM_WORLD, mpistatus, ierr)
    end if

    if (me > 0) then
        ! Receive left endpoint value
        call mpi_recv(uold(istart-1), 1, MPI_DOUBLE_PRECIS
                      me - 1, 2, MPI_COMM_WORLD, mpistatus, ierr)
    end if

    ! Apply Jacobi iteration on my section of array
    do i = istart, iend
        u(i) = 0.5d0*(uold(i-1) + uold(i+1) + dx**2*f(i))
        dumax_task = max(dumax_task, abs(u(i) - uold(i)))
    end do
end do
```

Jacobi with MPI

Other issues:

- Convergence check requires coordination between processes to get global `dumax`.
Use `MPI_ALLREDUCE` so all process check same value.
- Part of final result must be printed by each process
(into common file `heatsoln.txt`), in proper order.

See code in the repository or the notes:

[/codes/mpi/jacobi1d_mpi.f90](#)

Jacobi with MPI — Writing solution in order

Want to write table of values $x(i), u(i)$ in `heatsoln.txt`.

Need them to be in proper order, so Process 0 must write to this file first, then Process 1, etc.

Jacobi with MPI — Writing solution in order

Want to write table of values $x(i), u(i)$ in `heatsoln.txt`.

Need them to be in proper order, so Process 0 must write to this file first, then Process 1, etc.

Approach:

Each process me waits for a message from $me-1$ indicating that it has finished writing its part. (Contents not important.)

Each process must open the file (without clobbering values already there), write to it, then close the file.

Jacobi with MPI — Writing solution in order

Want to write table of values $x(i), u(i)$ in `heatsoln.txt`.

Need them to be in proper order, so Process 0 must write to this file first, then Process 1, etc.

Approach:

Each process me waits for a message from $me-1$ indicating that it has finished writing its part. (Contents not important.)

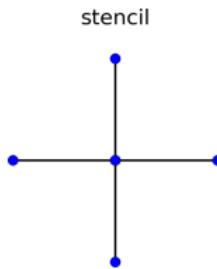
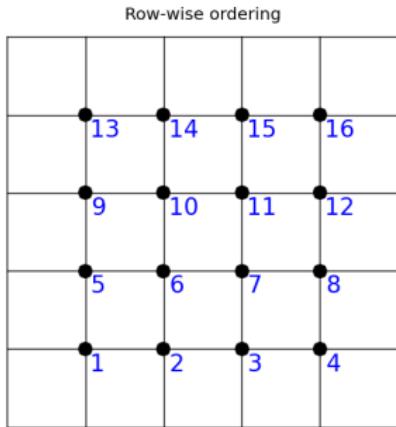
Each process must open the file (without clobbering values already there), write to it, then close the file.

Assumes all processes share a file system!

On cluster or supercomputer, need to either:

- send all results to single process for writing, or
- write distributed files that may need to be combined later
(some visualization tools handle distributed data!)

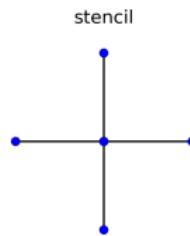
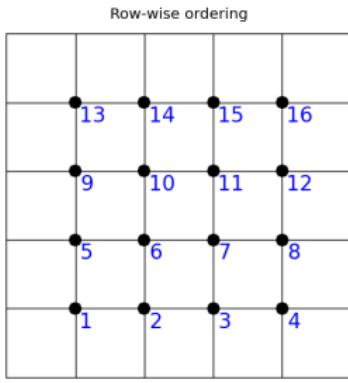
Jacobi in 2D



Updating point 7 for example (u_{32}):

$$U_{32}^{[k+1]} = \frac{1}{4}(U_{22}^{[k]} + U_{42}^{[k]} + U_{21}^{[k]} + U_{41}^{[k]} + h^2 f_{32})$$

Jacobi in 2D using MPI



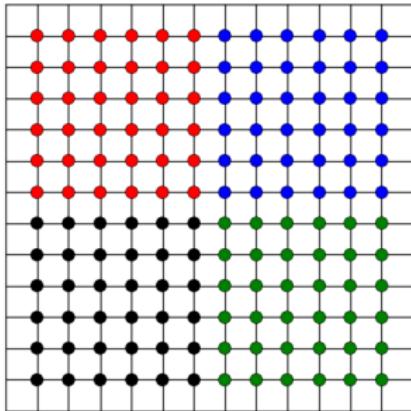
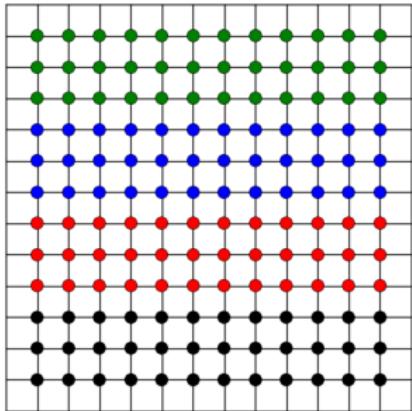
With two processes: Could partition unknown into
Process 0 takes grid points 1–8
Process 1 takes grid points 9–16

Each time step:
Process 0 sends top boundary (5–8) to Process 1,
Process 1 sends bottom boundary (9–12) to Process 0.

Jacobi in 2D using MPI

With more grid points and processes...

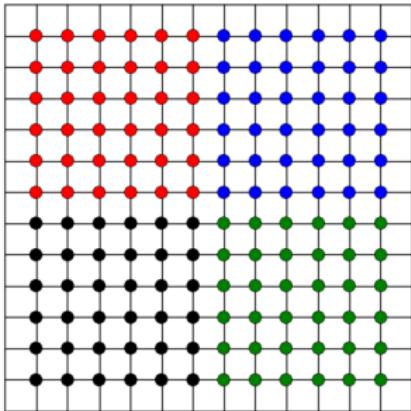
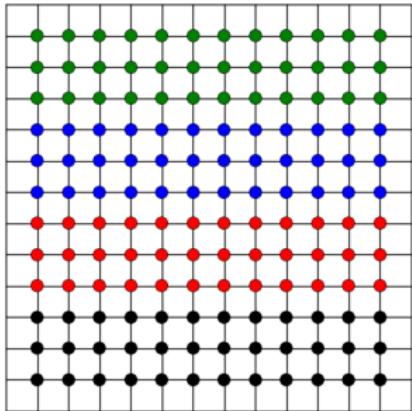
Could partition several different ways, e.g. with 4 processes:



Jacobi in 2D using MPI

With more grid points and processes...

Could partition several different ways, e.g. with 4 processes:



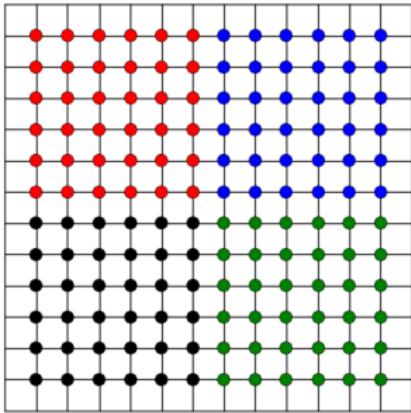
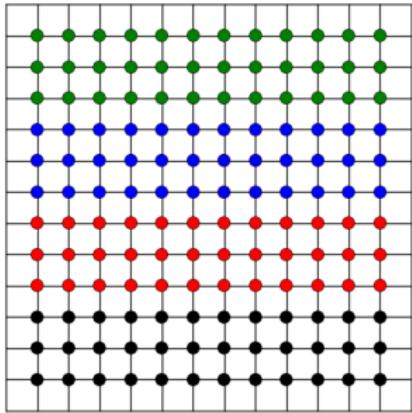
The partition on the right requires less communication.

With m^2 processes on grid with n^2 points:

$2(m^2 - 1)n$ boundary points on left,

$4(m - 1)n$ boundary points on right.

Jacobi in 2D using MPI



For partition on left: Natural to number processes 0,1,2,3 and pass boundary data from Process k to $k \pm 1$.

For $m \times m$ array of processors as on right: How do we figure out the neighboring process numbers?

Creating a communicator for Cartesian blocks

```
integer dims(2)
logical isperiodic(2), reorder

ndim = 2           ! 2d grid of processes
dims(1) = 4        ! for 4x6 grid of processes
dims(2) = 6
isperiodic(1) = .false.    ! periodic in x?
isperiodic(2) = .false.    ! periodic in y?
reorder = .true.         ! optimize ordering

call MPI_CART_CREATE(MPI_COMM_WORLD, ndim, &
                     dims, isperiodic, reorder, comm2d, ierr)
```

Create communicator [comm2d](#). See also:

[MPI_CART_CREATE](#), [MPI_CART_SHIFT](#), [MPI_CART_COORDS](#).

HPSC – Lecture 21

Outline:

- Monte Carlo methods
- Random number generators
- Monte Carlo integrators
- Random walk solution of Poisson problem

Monte Carlo Methods



Computational methods that use random (or pseudo-random) sampling to obtain numerical approximations.

Originally developed in 1940's at Los Alamos for neutron diffusion problems.

Monte Carlo methods

Examples:

- Approximate a definite integral by sampling the integrand at random points (rather than on a regular grid, as with Trapezoid or Simpson).
- Random walk solution to a Poisson problem
- Given a probability distribution of inputs to some problem, estimate probability distribution of output.

Sensitivity analysis

Uncertainty quantification

- Simulate processes that have random data or forcing.

Classical quadrature

Midpoint rule in 1 dimension:

$$\int_a^b f(x) dx \approx h \sum_{i=1}^n f(x_i)$$

There are n terms in sum and error is $\mathcal{O}(h^2) = \mathcal{O}(1/n^2)$

Midpoint rule in 2 dimensions:

$$\int_{a_2}^{b_2} \int_{a_1}^{b_1} g(x_1, x_2) dx_1 dx_2 \approx h^2 \sum_{j=1}^n \sum_{i=1}^n g(x_1^{[i]}, x_2^{[j]})$$

There are $N = n^2$ terms in sum and accuracy is
 $\mathcal{O}(h^2) = \mathcal{O}(1/n^2) = \mathcal{O}(1/N)$

Classical quadrature

Midpoint rule in 20 dimensions:

$$\int_{a_{20}}^{b_{20}} \cdots \int_{a_2}^{b_2} \int_{a_1}^{b_1} g(x_1, x_2, \dots, x_{20}) dx_1 dx_2 \cdots dx_{20}$$
$$\approx h^{20} \sum_{k=1}^n \cdots \sum_{j=1}^n \sum_{i=1}^n g(x_1^{[i]}, x_2^{[j]}, \dots, x_{20}^{[k]})$$

There are $N = n^{20}$ terms in sum and accuracy is
 $O(h^2) = O(1/n^2) = O((1/N)^{1/10})$

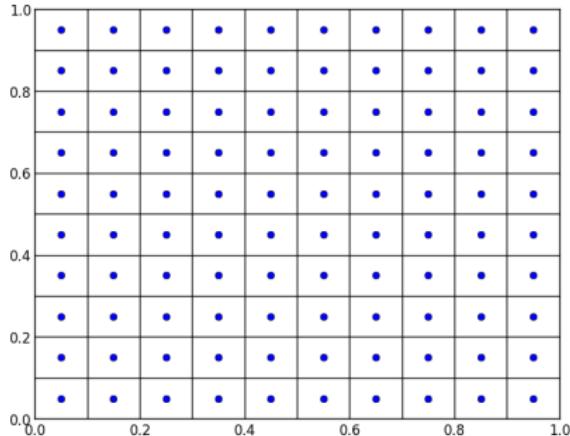
Note: with only $n = 10$ points in each direction, $N = 10^{20}$.

On 1 GFlop computer, would take 10^{11} seconds > 3000 years to compute sum and get accuracy $\approx 1/n^2 = 0.01$.

Also each evaluation of g might be expensive!

Classical quadrature

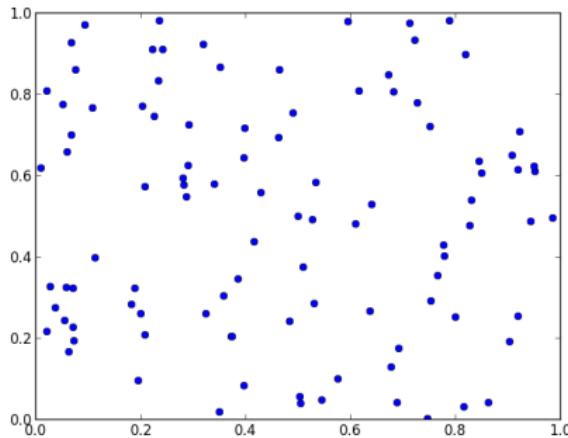
$N = 100$ points in two space dimensions for Midpoint:



$$\int_{a_2}^{b_2} \int_{a_1}^{b_1} g(x_1, x_2) dx_1 dx_2 \approx h^2 \sum_{j=1}^n \sum_{i=1}^n g(x_1^{[i]}, x_2^{[j]})$$

Monte Carlo integration

$N = 100$ random points in the same 2-dimensional region:



$$\int_{a_2}^{b_2} \int_{a_1}^{b_1} g(x_1, x_2) dx_1 dx_2 \approx \frac{V}{N} \sum_{k=1}^N g(x_1^{[k]}, x_2^{[k]})$$

$V = (b_2 - a_2)(b_1 - a_1)$ is volume.

Monte Carlo integration

Accuracy: With N random points, error is $O(1/\sqrt{N})$

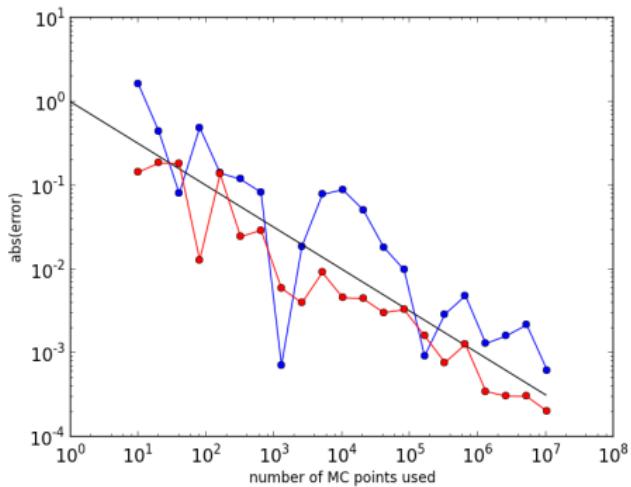
This is true independent of the number of dimensions!

In 20 dimensions, if g is smooth then can expect error ≈ 0.01 with $N = 10000$. (vs. $N = 10^{20}$ for Midpoint.)

Log-log plot of errors with Monte Carlo

Black line: $1/\sqrt{N} = N^{-1/2}$.

Note that $E(N) = C/\sqrt{N} \implies \log(E(N)) = \log(C) - \frac{1}{2}\log(N)$



Red points: For an integral in 2 dimensions

Blue points: For an integral in 20 dimensions

Pseudo-Random number generators

Hard to generate a truly random number on the computer.

Instead generally use pseudo-random number generators that produce a sequence of numbers by some deterministic formula, but designed so that numbers generated are approximately distributed according to desired distribution.

Pseudo-Random number generators

Hard to generate a truly random number on the computer.

Instead generally use pseudo-random number generators that produce a sequence of numbers by some deterministic formula, but designed so that numbers generated are approximately distributed according to desired distribution.

Linear congruential generator:

$$X_{n+1} = aX_n + c \bmod m$$

e.g. from *Numerical Recipes*:

$$a = 1664525, c = 1013904223, m = 2^{32}$$

Requires a **seed** X_0 to get started.

Pseudo-Random number generators

In Python: Plot of 100 random points was generated using...

```
from numpy.random import RandomState
random_generator = RandomState(seed=55)
r = random_generator.uniform(0., 1., size=200)
plot(r[::2], r[1::2], 'bo')
```

Pseudo-Random number generators

In Python: Plot of 100 random points was generated using...

```
from numpy.random import RandomState  
random_generator = RandomState(seed=55)  
r = random_generator.uniform(0., 1., size=200)  
plot(r[::2], r[1::2], 'bo')
```

Initializing with `seed=None` will use a “random” seed.

Specifying a seed makes it possible to reproduce the same results later.

Pseudo-Random number generators

In Fortran:

```
integer, dimension(:), allocatable :: seed

! determine how many seeds needed:
call random_seed(size = nseed)
allocate(seed(nseed) )

seed = ...           ! array of integers

call random_seed(put = seed)
deallocate(seed)
```

Pseudo-Random number generators

To reduce to a single seed1:

```
if (seed1 == 0) then
    ! randomize the seed: not repeatable
    call system_clock(count = clock)
    seed1 = clock
endif

do i=1,nseed
    seed(i) = seed1 + 37*(i-1)
enddo
```

Pseudo-Random number generators

To generate n random numbers, uniformly distributed in $[0,1]$:

```
real(kind=4), allocatable :: r(:)
allocate(r(n))
call random_number(r)

r_ab = a + r*(b-a)      ! uniform in [a,b]
```

Pseudo-Random number generators

To generate n random numbers, uniformly distributed in $[0,1]$:

```
real(kind=4), allocatable :: r(:)
allocate(r(n))
call random_number(r)

r_ab = a + r*(b-a)      ! uniform in [a,b]
```

Note: More efficient in general to call `random_number` once for array of length n rather than n times in succession, but same sequence of numbers will be generated.

State at end of one call is used at start of next call!

Pseudo-Random number generators in parallel

With OpenMP...

State is changed whenever any thread calls `random_number`.

Different threads share same global state.

(Should be thread safe, but can't generate in parallel.)

```
real(kind=4) :: r, x(100)

!$omp parallel do private(r)
do i=1,100
    call random_number(r)
    x(i) = r
enddo
```

Should produce same set of random numbers but may not end up in same order!

Pseudo-Random number generators in parallel

With MPI... (Processes cannot share the state)

If each process initializes with same seed, then each process will generate the same sequence of random numbers

```
call random_number(r)
```

Will produce the same `r` on each process.

Pseudo-Random number generators in parallel

With MPI... (Processes cannot share the state)

If each process initializes with same seed, then each process will generate the same sequence of random numbers

```
call random_number(r)
```

Will produce the same `r` on each process.

This might not be what you want, e.g. if splitting up Monte Carlo integration between processes — want each to sample a different set of points on each process.

Pseudo-Random number generators in parallel

With MPI... (Processes cannot share the state)

If each process initializes with same seed, then each process will generate the same sequence of random numbers

```
call random_number(r)
```

Will produce the same `r` on each process.

This might not be what you want, e.g. if splitting up Monte Carlo integration between processes — want each to sample a different set of points on each process.

Would need to seed differently on each Process, e.g.

```
seed(i) = seed1 + 37*(i-1) + 97*proc_num
```

Monte Carlo solution of Poisson problem

Suppose we want to compute an approximate solution to

$$u_{xx} + u_{yy} = 0 \quad \text{with } u \text{ given on boundary}$$

at a **single point** (x_0, y_0) .

Finite difference approach: Discretize domain and solve linear system for approximations U_{ij} at **all** points on grid.

Monte Carlo solution of Poisson problem

Suppose we want to compute an approximate solution to

$$u_{xx} + u_{yy} = 0 \quad \text{with } u \text{ given on boundary}$$

at a **single point** (x_0, y_0) .

Finite difference approach: Discretize domain and solve linear system for approximations U_{ij} at **all** points on grid.

Instead can take a **random walk** starting at (x_0, y_0) and evaluate u at the first **boundary point** the walk reaches.

Do this N times and average all the values obtained.

Monte Carlo solution of Poisson problem

Suppose we want to compute an approximate solution to

$$u_{xx} + u_{yy} = 0 \quad \text{with } u \text{ given on boundary}$$

at a **single point** (x_0, y_0) .

Finite difference approach: Discretize domain and solve linear system for approximations U_{ij} at **all** points on grid.

Instead can take a **random walk** starting at (x_0, y_0) and evaluate u at the first **boundary point** the walk reaches.

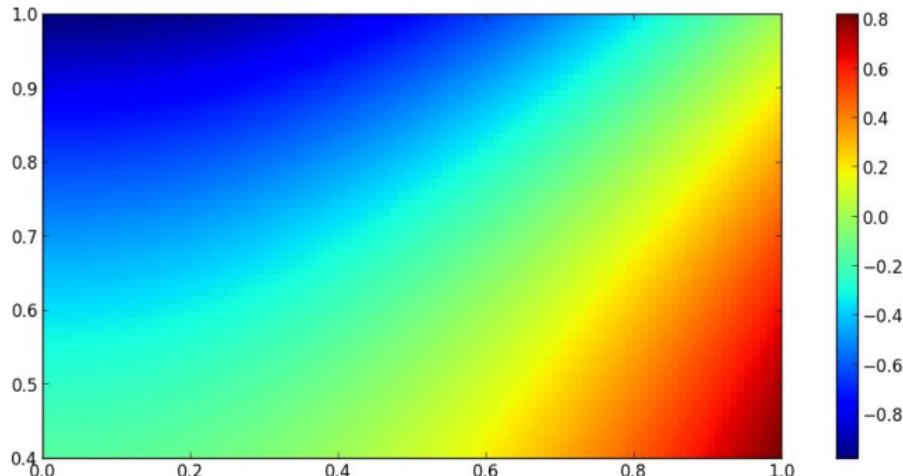
Do this N times and average all the values obtained.

This average converges to $u(x_0, y_0)$ with rate $1/\sqrt{N}$

Monte Carlo solution of Poisson problem

$u_{xx} + u_{yy} = 0$ with solution $u(x, y) = x^2 - y^2$.

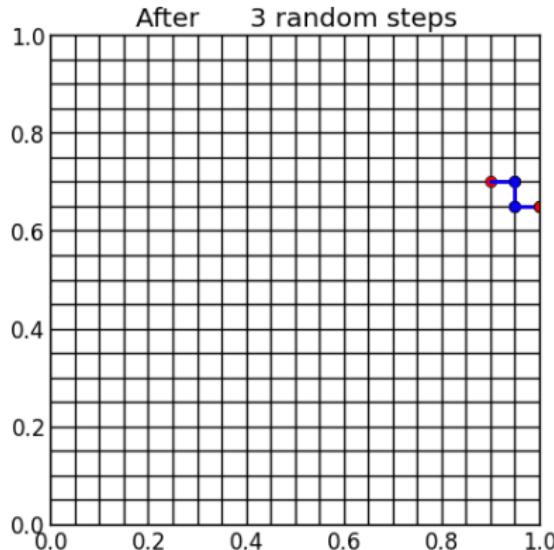
Estimate solution at $(x_0, y_0) = (0.9, 0.7)$ where $u(x_0, y_0) = 0.32$.



Random walk on a lattice

$u_{xx} + u_{yy} = 0$ with solution $u(x, y) = x^2 - y^2$.

Estimate solution at $(x_0, y_0) = (0.9, 0.7)$ where $u(x_0, y_0) = 0.32$.

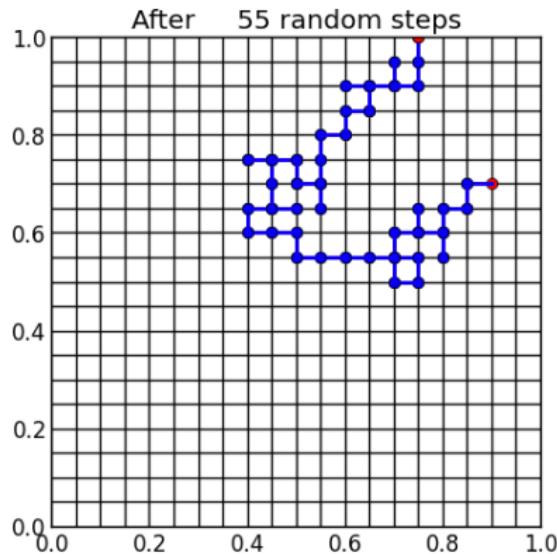


Hit boundary where $u = 0.577500$

Random walk on a lattice

$u_{xx} + u_{yy} = 0$ with solution $u(x, y) = x^2 - y^2$.

Estimate solution at $(x_0, y_0) = (0.9, 0.7)$ where $u(x_0, y_0) = 0.32$.



Hit boundary where $u = -0.437500$

Random walk on a lattice

Strategy:

Start at (x_0, y_0) .

Each step, move to one of 4 neighbors, choosing with equal probability.

If $0 \leq r \leq 1$ is a uniformly distributed random number then decide based on:

$0 \leq r < 0.25 \Rightarrow$ move left

$0.25 \leq r < 0.5 \Rightarrow$ move right

$0.5 \leq r < 0.75 \Rightarrow$ move down

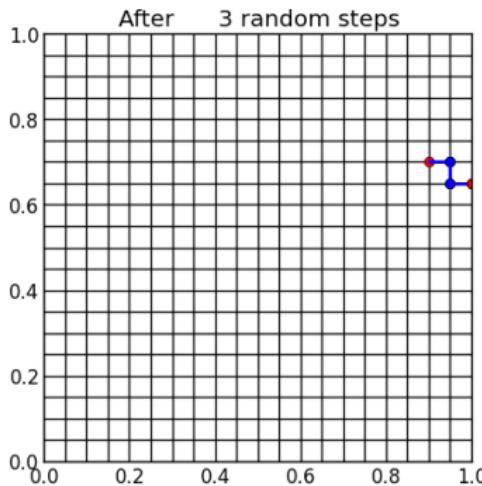
$0.75 \leq r \leq 1.0 \Rightarrow$ move down

Random walk on a lattice

Why does this work? Let E_{ij} be expected value of boundary value reached if starting at grid point (i, j) .

$$\text{Then } E_{ij} = \frac{1}{4}(E_{i-1,j} + E_{i+1,j} + E_{i,j-1} + E_{i,j+1})$$

The same equation as finite difference method for Poisson!



Hit boundary where $u = 0.577500$

Discuss Python, Fortran and parallelised MPI codes

HPSC – Lecture 22

Outline:

- Python plus Fortran: f2py
- LAPACK and the BLAS
- Python compilers
- File writing – ASCII vs binary
- Summary

Sample codes:

- /codes/f2py

f2py — combining Fortran and Python

Often want to use
Fortran for intensive computations,
Python to provide nice user interface, plot results,
automate a series of runs with different parameters,
do convergence tests as grid size is refined, etc.

Can write data files to disk from Fortran, read into Python,
This is what we've done for plotting in homeworks.

Sometimes nice to call Fortran directly from Python.
e.g. LAPACK is used under the hood in NumPy.

f2py provides a [wrapper](#) for Fortran code.

f2py — combining Fortran and Python

Basic idea:

`fortrancode.f90` contains a function or subroutine, e.g.
`function f1(x)` that returns a single value.

```
$ f2py -m mymodulename -c fortrancode.f90
```

This creates a binary file `mymodulename.so` that can be used as a Python module.

```
>>> from mymodulename import f1  
>>> y = f1(3.)
```

f2py — function example

/codes/f2py/fcn1.f90

```
function f1(x)
    real(kind=8), intent(in) :: x
    real(kind=8) :: f1
    f1 = exp(x)
end function f1
```

Then we can do...

```
$ f2py -m fcn1 -c fcn1.f90
$ python
>>> import fcn1
>>> fcn1.f1(1.)
2.7182818284590451
```

f2py — subroutine example

/codes/f2py/sub1.f90

```
subroutine mysub(a,b,c,d)
    real (kind=8), intent(in) :: a,b
    real (kind=8), intent(out) :: c,d
    c = a+b
    d = a-b
end subroutine mysub
```

Then we can do...

```
$ f2py -m sub1 -c sub1.f90
$ python
>>> import sub1
>>> y = sub1.mysub(3., 5.)
>>> print y
(8.0, -2.0)
```

Note: Tuple (c, d) is returned by the Python function.

f2py — Jacobi iteration

/codes/f2py/jacobi1.f90

```
subroutine iterate(u0,iters,f,u,n)
```

Takes input array `u0` of length `n` and right hand side array `f` and produces `u` by taking `iters` iterations of Jacobi.

/codes/f2py/plot_jacobi_iterates.py

```
# Set u = initial guess; f = rhs
for nn in range(nplots):
    u = jacobi1.iterate(u, iters_per_plot, f)
    plt.plot(x, u, 'o-')
    plt.draw()
    time.sleep(.5)
```

Other wrappers...

- **Cython**: Allows writing C code embedded in Python.

<http://www.cython.org/>

- **Jython**: For Java.

<http://www.jython.org/>

- **swig**: Connects C and C++ to many other languages

<http://www.swig.org/>

Mathematical Software

It is best to **use high-quality software** as much as possible, for several reasons:

- It will take **less time** to figure out how to use the software than to write your own version. (Assuming it's well documented!)
- Good general software has been **extensively tested** on a wide variety of problems.
- Often general software is much **more sophisticated** than what you might write yourself, for example it may provide error estimates automatically, or it may be **optimized** to run fast.

Software sources

- Netlib: <http://www.netlib.org>
- NIST Guide to Available Mathematical Software:
<http://gams.nist.gov/>
- Trilinos: <http://trilinos.sandia.gov/>
- DOE ACTS: <http://acts.nerc.gov/>
- PETSc nonlinear solvers:
<http://www.mcs.anl.gov/petsc/petsc-as/>
- Many others!

LAPACK — www.netlib.org/lapack/

Many routines for linear algebra
using non-iterative methods.

Typical name: XYYZZZ

X is precision

YY is type of matrix, e.g. GE (general), BD (bidiagonal),

ZZZ is type of operation, e.g. SV (solve system),

EV (eigenvalues, vectors), SVD (singular values, vectors)

LAPACK — www.netlib.org/lapack/

Many routines for linear algebra.

Typical name: XYYZZZ

X is precision

YY is type of matrix, e.g. GE (general), BD (bidiagonal),

ZZZ is type of operation, e.g. SV (solve system),

EV (eigenvalues, vectors), SVD (singular values, vectors)

Examples:

DGESV can be used to solve a general $n \times n$ linear system in double precision.

DGTSV can be used to solve a general $n \times n$ tridiagonal linear system in double precision.

Installing LAPACK

On Virtual Machine or other Debian or Ubuntu Linux:

```
$ sudo apt-get install liblapack-dev
```

This will include BLAS (but not optimized for your system).

Alternatively can download tar files and compile.

See complete documentation at

<http://www.netlib.org/lapack/>

The BLAS

Basic Linear Algebra Subroutines

Core routines used by LAPACK (Linear Algebra Package) and elsewhere.

Generally optimized for particular machine architectures, cache hierarchy.

Can create optimized BLAS using

ATLAS (Automatically Tuned Linear Algebra Software)

See notes and <http://www.netlib.org/blas/faq.html>

- Level 1: Scalar and vector operations
- Level 2: Matrix-vector operations
- Level 3: Matrix-matrix operations

The BLAS

Subroutine names start with:

- S: single precision
- D: double precision
- C: single precision complex
- Z: double precision complex

Examples:

- SAXPY: single precision replacement of y by $ax + y$.
- DDOT: dot product of two vectors
- DGEMV: matrix-vector multiply, general matrices
- DGEMM: matrix-matrix multiply, general matrices
- DSYMM: matrix-matrix multiply, symmetric matrices

Using libraries

If `program.f90` uses BLAS routines...

```
$ gfortran -c program.f90  
$ gfortran program.o -lblas
```

or can combine as

```
$ gfortran program.f90 -lblas
```

When linking together `.o` files, will look for a file called `libblas.a` (probably in `/usr/lib`).

This is a archived static library.

Using libraries

If `program.f90` uses BLAS routines...

```
$ gfortran -c program.f90  
$ gfortran program.o -lblas
```

or can combine as

```
$ gfortran program.f90 -lblas
```

When linking together `.o` files, will look for a file called `libblas.a` (probably in `/usr/lib`).

This is a archived static library.

Can specify different library location using
`-L/path/to/library`.

Making blas library

Download <http://www.netlib.org/blas/blas.tgz>.

Put this in desired location, e.g. \$HOME/lapack/blas.tgz

```
$ cd $HOME/lapack
$ tar -zxf blas.tgz      # creates BLAS subdirectory
$ cd BLAS
$ gfortran -O3 -c *.f
$ ar cr libblas.a *.o  # creates libblas.a
```

To use this library:

```
$ gfortran program.f90 -lblas \
-L$HOME/lapack/BLAS
```

Note: Non-optimized Fortran 77 versions.

Better approach would be to use [ATLAS](#).

Creating LAPACK library

Can be done from source at

<http://www.netlib.org/lapack/>

but somewhat more difficult.

Individual routines and dependencies can be obtained from netlib, e.g. the double precision versions from:

<http://www.netlib.org/lapack/double>

Download .tgz file and untar into directory where you want to use them, or make a library of just these files.

Memory management for arrays

Often a program needs to be written to handle arrays whose size is not known until the program is running.

Fortran 77 approaches:

- Allocate arrays large enough for any application,
- Use “work arrays” that are partitioned into pieces.

We will look at some examples from LAPACK since you will probably see this in other software!

Memory management for arrays

Often a program needs to be written to handle arrays whose size is not known until the program is running.

Fortran 77 approaches:

- Allocate arrays large enough for any application,
- Use “work arrays” that are partitioned into pieces.

We will look at some examples from LAPACK since you will probably see this in other software!

The good news:

Fortran 90 allows dynamic memory allocation.

DGESV — Solves a general linear system

<http://www.netlib.org/lapack/double/dgesv.f>

```
SUBROUTINE DGESV( N, NRHS, A, LDA, IPIV,  
& B, LDB, INFO )
```

N = size of system (square $N \times N$)

A = matrix on input, L,U factors on output,
dimension(LDA,K) with LDA, K >= N

LDA = leading dimension of A
(number of rows in declaration of A)

Example:

```
real(kind=8) dimension(100,500) :: a  
! fill a(1:20, 1:20) with 20x20 matrix  
n = 20  
lda = 100
```

DGESV — Solves a general linear system

Example:

```
real(kind=8), dimension(100,500) :: a
real(kind=8), dimension(200,400) :: b
integer, dimension(600) :: ipiv
! fill a(1:20, 1:20) with 20x20 matrix
! b(1:20, 1:3) with 3 right hand sides

n = 20; nrhs = 3; lda = 100; ldb = 200

call dgesv(n, nrhs, a, lda, ipiv, b, ldb, info)
```

What is passed to dgesv is start_address, the address of first element of a. (Matrix is stored by columns)

Whenever a(i,j) appears in code, address is:

$$\text{address} = \text{start_address} + (j-1)*\text{lda} + (i-1)$$

DGESV — Solves a general linear system

```
SUBROUTINE DGESV( N, NRHS, A, LDA, IPIV,  
& B, LDB, INFO )
```

NRHS = number of right hand sides

B = matrix whose columns are right hand side(s) on input
solution vector(s) on output.

LDB = leading dimension of B.

INFO = integer returning 0 if successful.

A = matrix on input, L,U factors on output,

IPIV = Returns pivot vector (permutation of rows)
integer, dimension (N)
Row I was interchanged with row IPIV(I).

Gaussian elimination as factorization

If A is nonsingular it can be factored as

$$PA = LU$$

where

P is a permutation matrix (rows of identity permuted),

L is lower triangular with 1's on diagonal,

U is upper triangular.

After returning from `dgesv`:

A contains L and U (without the diagonal of L),

`IPIV` gives ordering of rows in P .

Gaussian elimination as factorization

Example:

$$A = \begin{bmatrix} 2 & 1 & 3 \\ 4 & 3 & 6 \\ 2 & 3 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & 1 & 3 \\ 4 & 3 & 6 \\ 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ 1/2 & -1/3 & 1 \end{bmatrix} \begin{bmatrix} 4 & 3 & 6 \\ 0 & 1.5 & 1 \\ 0 & 0 & 1/3 \end{bmatrix}$$

$$\text{IPIV} = (2,3,1)$$

and A ends up as

$$\begin{bmatrix} 4 & 3 & 6 \\ 1/2 & 1.5 & 1 \\ 1/2 & -1/3 & 1/3 \end{bmatrix}$$

dgesv examples

See /codes/lapack/random.

Sample codes that solve the linear system $Ax = b$ with a random $n \times n$ matrix A , where the value n is run-time input.

`randomsys1.f90` is with static array allocation.

`randomsys2.f90` is with dynamic array allocation.

dgesv examples

See /codes/lapack/random.

Sample codes that solve the linear system $Ax = b$ with a random $n \times n$ matrix A , where the value n is run-time input.

`randomsys1.f90` is with static array allocation.

`randomsys2.f90` is with dynamic array allocation.

`randomsys3.f90` also estimates **condition number** of A .

$$\kappa(A) = \|A\| \|A^{-1}\|$$

Can bound relative error in solution in terms of relative error in data using this:

$$Ax^* = b^* \text{ and } Ax \approx \tilde{b} \Rightarrow \frac{\|\tilde{x} - x^*\|}{\|x^*\|} \leq \kappa(A) \frac{\|b - b^*\|}{\|b^*\|}$$

Just-in-time compilers for Python

Standard implementation of Python as interpreted language.

Importing `mymodule.py` creates `mymodule.pyc`, which is Bytecode (portable code or pcode):

One-byte operators with operands,
Interpreted by software at runtime.

Runs much slower than **compiled code** that is machine-specific instructions.

Just-in-time compilers for Python

Standard implementation of Python as interpreted language.

Importing `mymodule.py` creates `mymodule.pyc`, which is Bytecode (portable code or pcode):

One-byte operators with operands,
Interpreted by software at runtime.

Runs much slower than compiled code that is machine-specific instructions.

Just-in -time (JIT) compilation: Converts bytecode at runtime into native machine code.

Can sometimes run faster than pre-compiled code.

Just-in-time compilers for Python

Examples:

- [PyPy](#) — alternative implementation of Python
- [numba](#) — compiles decorated code to [LLVM](#) (formerly Low Level Virtual Machine, compiler infrastructure)

Included in the [Anaconda Python distribution](#)

Numba — autojit decorator

```
In [1]: def loopsum(n):
    x = 0
    for i in range(n):
        x = x + i
```

```
In [2]: %timeit loopsum(10000)
```

```
1000 loops, best of 3: 495 us per loop
```

Numba — autojit decorator

```
In [1]: def loopsum(n):
    x = 0
    for i in range(n):
        x = x + i
```

```
In [2]: %timeit loopsum(10000)
```

```
1000 loops, best of 3: 495 us per loop
```

```
In [3]: from numba import autojit
```

```
In [4]: @autojit
def loopsum2(n):
    x = 0
    for i in range(n):
        x = x + i
```

```
In [5]: %timeit loopsum2(10000)
```

```
1000000 loops, best of 3: 1.5 us per loop
```

ASCII vs. binary output

Often need to write out a large array of floats with full precision.

For example, one solution value on 3d grid ...

```
do i=1,n  
  do j=1,n  
    do k=1,n  
      write(21,'(e24.16)') u(i,j,k)  
    enddo; enddo; enddo
```

How much disk space does this take?

ASCII vs. binary output

Often need to write out a large array of floats with full precision.

For example, one solution value on 3d grid ...

```
do i=1,n  
  do j=1,n  
    do k=1,n  
      write(21,'(e24.16)') u(i,j,k)  
    enddo; enddo; enddo
```

How much disk space does this take?

A single number such as $0.4000000000000000E+01$
has 24 ASCII characters \Rightarrow 24 bytes per value.

Total $24n^3$ bytes. E.g. $100 \times 100 \times 100$ grid: $n = 100 \Rightarrow 24 \text{ MB}$.

ASCII vs. binary output

Often need to write out a large array of floats with full precision.

For example, one solution value on 3d grid ...

```
do i=1,n  
  do j=1,n  
    do k=1,n  
      write(21,'(e24.16)') u(i,j,k)  
    enddo; enddo; enddo
```

How much disk space does this take?

A single number such as $0.4000000000000000E+01$
has 24 ASCII characters \Rightarrow 24 bytes per value.

Total $24n^3$ bytes. E.g. $100 \times 100 \times 100$ grid: $n = 100 \Rightarrow 24 \text{ MB}$.

Note: In memory storing one 8-byte float takes only 8 bytes.
 $(n = 100 \Rightarrow 8\text{MB})$ **ASCII takes 3 \times the space.**

ASCII vs. binary output

Often need to write out a large array of floats with full precision.

For example, one solution value on 3d grid ...

```
do i=1,n  
  do j=1,n  
    do k=1,n  
      write(21,'(e24.16)') u(i,j,k)  
    enddo; enddo; enddo
```

How much disk space does this take?

A single number such as $0.4000000000000000E+01$
has 24 ASCII characters \Rightarrow 24 bytes per value.

Total $24n^3$ bytes. E.g. $100 \times 100 \times 100$ grid: $n = 100 \Rightarrow 24 \text{ MB}$.

Note: In memory storing one 8-byte float takes only 8 bytes.

$(n = 100 \Rightarrow 8\text{MB})$ ASCII takes 3x the space.

Also takes additional time to convert to ASCII,

$\approx 10\times$ slower to write ASCII than dumping binary.

Binary output in Fortran

Can use **unformatted** write in Fortran:

```
! /codes/io/binwrite.f90
open(unit=20, file="u.bin", form="unformatted", &
      status="unknown", access="stream")
do j=1,100
    do i=1,500
        u(i,j) = real(m*(j-1) + i, kind=8)
    enddo
enddo
write(20) u    ! writes entire array in binary
close(20)
```

Ascii is 3x larger in size

The resulting binary file `u.bin` cannot be edited directly. But we can read it into Python...

Reading binary data files in Python

To recover u array of dimension $m \times n$ in Python:

```
# /codes/io/binread.py

import numpy as np

file = open('u.bin', 'rb')
uvec = np.fromfile(file, dtype=np.float64)

m,n = np.loadtxt('mn.txt', dtype=int)

# now use Fortran ordering to fill u by columns:
u = uvec.reshape((m,n),order='F')
```

Other options for binary data

Binary formats that contain a lot of [metadata](#)...

Hierarchical Data Format: [HDF](#), [HDF4](#), [HDF5](#)

HDF5 file structure includes two major types of object:

- [Datasets](#): multidimensional arrays of a homogenous type
- [Groups](#): container structures for datasets and other groups

See also: [h5py](#), [PyTables](#)

Other options for binary data

Binary formats that contain a lot of [metadata](#)...

[Hierarchical Data Format](#): [HDF](#), [HDF4](#), [HDF5](#)

HDF5 file structure includes two major types of object:

- [Datasets](#): multidimensional arrays of a homogenous type
- [Groups](#): container structures for datasets and other groups

See also: [h5py](#), [PyTables](#)

[NetCDF](#) (Network Common Data Form): Built on top of HDF5.

See also [ncdump](#), [netcdf4-python](#)

Summary, take away messages...

- Version control — git

Use for all your projects, collaborations, ...

Consider contributing to open source projects

Submit a pull request

Summary, take away messages...

- Version control — git

Use for all your projects, collaborations, ...

Consider contributing to open source projects

Submit a pull request

- Python, NumPy, SciPy, matplotlib, IPython

Quickly trying out new ideas, optimize later

Graphics and visualization

Scripting to guide big computations

Combining codes from different languages

Many capabilities not seen in class, e.g.

Manipulating text files, regular expressions,
building web interfaces

Summary, take away messages...

- Fortran 90

- Compiled language

- Tightly constrained but can run very fast

- Native multi-dimensional arrays

Summary, take away messages...

- Fortran 90

- Compiled language

- Tightly constrained but can run very fast

- Native multi-dimensional arrays

- Makefiles

- Dependency checking

- Often used for building software

Summary, take away messages...

- Fortran 90

- Compiled language

- Tightly constrained but can run very fast

- Native multi-dimensional arrays

- Makefiles

- Dependency checking

- Often used for building software

- Debugging code

- Unit tests, pytest

- Print statements, pdb, gdb

Summary, take away messages...

- Fortran 90

- Compiled language

- Tightly constrained but can run very fast

- Native multi-dimensional arrays

- Makefiles

- Dependency checking

- Often used for building software

- Debugging code

- Unit tests, pytest

- Print statements, pdb, gdb

- Memory hierarchy, cache considerations

- Consider layout of arrays in memory

- Aim for spatial and temporal locality

Summary, take away messages...

- Parallel computing

Increasingly necessary for all computing

Amdahl's law —

 inherently sequential code limits parallelization

Weak vs. strong scaling

Fine grain vs. coarse grain parallelism

Load balancing

Summary, take away messages...

- Parallel computing

Increasingly necessary for all computing

Amdahl's law —

 inherently sequential code limits parallelization

Weak vs. strong scaling

Fine grain vs. coarse grain parallelism

Load balancing

- OpenMP

Assumes shared memory

Often very easy to add to existing codes

Need to worry about shared/private variables,
race conditions

Summary, take away messages...

- MPI — Message Passing Interface
 - Always assumes distributed memory
 - Sharing data requires message passing
 - SPMD: Single Program Multiple Data
 - Entire program run by each process
 - But different processes may take different branches

Summary, take away messages...

- MPI — Message Passing Interface
 - Always assumes distributed memory
 - Sharing data requires message passing
 - SPMD: Single Program Multiple Data
 - Entire program run by each process
 - But different processes may take different branches
- Computer arithmetic
 - Floating point number representation, 4 byte vs. 8 byte
 - IEEE standards
 - Reproducibility still difficult in parallel
 - Relative error and precision possible

Summary, take away messages...

- Linear algebra

LAPACK, BLAS — optimized code

Iterative methods for large sparse system

Poisson problems: $u_{xx} = f(x)$

Two-dimensional Poisson problem $u_{xx} + u_{yy} = f(x, y)$

Summary, take away messages...

- **Linear algebra**

LAPACK, BLAS — optimized code

Iterative methods for large sparse system

Poisson problems: $u_{xx} = f(x) \Rightarrow$ tridiagonal

Two-dimensional Poisson problem $u_{xx} + u_{yy} = f(x, y)$

- **Quadrature methods / numerical integration**

Midpoint, Trapezoid, Simpson Rules

Monte Carlo methods in high dimensions

Summary, take away messages...

- **Linear algebra**

Matrix norms and condition number of $Ax = b$

LAPACK, BLAS — optimized code

Iterative methods for large sparse system

Poisson problems: $u_{xx} = f(x) \Rightarrow$ tridiagonal

Two-dimensional Poisson problem $u_{xx} + u_{yy} = f(x, y)$

- **Quadrature methods / numerical integration**

Midpoint, Trapezoid, Simpson Rules

Monte Carlo methods in high dimensions

- **Monte Carlo methods**

Pseudo Random Number Generation

Use of seed for reproducibility

Random walks

Happy Computing!