CSC

CSC Summer School in HIGH-PERFORMANCE COMPUTING

June 26 – July 4, 2018 • Espoo, Finland

Lecture Notes

All material (C) 2011-2018 by CSC – IT Center for Science Ltd. and the authors.
This work is licensed under a **Creative Commons Attribution-NonCommercial-ShareAlike** 3.0 Unported License, **http://creativecommons.org/licenses/by-nc-sa/3.0/**

	Tue 26.6.	Wed 27.6.		Thu 28.6.	
7.30-9		Breakfast		Breakfast	
9-10	Transports from Helsinki city	Introduction to Fortran	Introduction to C	Introduction to Fortran	Introduction to C
10-11	Registration				
11-12	Briefing session				
12-13	Lunch	Lunch		Lunch	
13-14	Introduction to supercomputing & supercomputers	Introduction to Fortran	Introduction to C	Introduction to Fortran	Introduction to C
14-15					
15-16	Working with Unix				
16-17	Version control with git	Frisbee golf		Fast Scoop	
17.30-18	Dinner	Dinner		Exercises	
18-19	Version control with git	Exercises		BBQ evening	
19-20	Get-together				
20-21			Lakeside saunas		
21-22	Snack	Snack			

	Fri 29.6.	Sat 30.6.		Sun 1.7.	
7.30-9	Breakfast	Breakfast		Breakfast	
9-10	OpenMP I	Free time		MPI II: collective communication & user-defined communicators	
10-11					
11-12					
12-13	Lunch	Lunch		Lunch	
13-14	OpenMP II	MPI I: getting started & point-to-point communication		Debugging	
14-15					
15-16				MPI III: Non-blocking communication	
16-17	Ultimate				
17.30-18	Dinner	Dinner		Dinner	
18-19	Exercises	Exercises		Exercises	
19-20	Free time				Trekking in Nuuksio
20-21		Lightning talks			
21-22	Snack	Snack		Snack	

	Mon 2.7.	Tue 3.7.		Wed 4.7.
7.30-9	Breakfast	Breakfast		Breakfast
9-10	MPI IV: Derived datatypes	Parallel I/O		Tools for data analysis & visualization
10-11				
11-12				
12-13	Lunch	Lunch		Lunch
13-14	Hybrid MPI+OpenMP programming	Application performance		HPC application design
14-15				
15-16				
16-17	Speedminton	Canoeing trip		Transports to Helsinki city
17.30-18	Dinner			
18-19	Exercises			
19-20		Farewell dinner & party		
20-21			Lakeside saunas	
21-22	Snack	Snack		

Jussi Enkovaara
Joonas Nättilä

INTRODUCTION TO SUPERCOMPUTING AND SUPERCOMPUTERS

Introduction to supercomputing

What is high-performance computing?

- Utilising computer power that is much larger than available in typical desktop computer
- Performance of HPC system (i.e. supercomputer) is often measured in floating point operations per second (flop/s)
  - For software, other measures can be more meaningful
- Currently, the most powerful system reaches ~100 x 10^{15} flop/s (100 Pflop / s)

Top 500 list

WHAT ARE SUPERCOMPUTERS USED FOR?

Materials science

- New materials
  - Design of meta-materials
  - Hydrogen storage
- New methods for catalysis
  - Industrial processes
  - Air and water purification
- Design of devices from first principles

Life sciences

- Next-generation sequencing techniques
- Identifying genomic variants associated with common complex diseases
- Understanding the natural development of diseases
- Medical imaging and diagnostics
- Simulated surgeries
- Predicting protein folding

Earth sciences

- Long term climate modeling
  - Coupling atmospheric, ocean and land models
  - Understanding and predicting the climate change
- High-resolution weather prediction
  - Predicting extreme weather conditions
  - District-scale forecasts
  - Ensembles
- Whole-Earth seismological models

Artificial intelligence

- Machine learning
  - deep neural networks
- Large scale data analysis
- Interpreting experimental data
- Prediction of material properties

Utilizing HPC in scientific research

- Goal for this school: everyone is able to write and modify HPC applications!

WHAT ARE SUPERCOMPUTERS MADE OF?

CPU frequency development

- Power consumption of CPU: ~f^3

Stanford CPU DB

Parallel processing

- Modern (super)computers rely on parallel processing
- **Multiple** CPU cores
  - #1 system has ~10 000 000 cores
- Vectorization
  - Single instruction can process multiple data (SIMD)
- Pipelining
  - Core executes different parts of instructions in parallel
- Accelerators
  - GPUs

Anatomy of supercomputer

- Modern supercomputers consist of nodes which are connected with high-speed network
  - Latency ~1 µs, bandwidth ~10 GB / s
- A node can contain several multicore CPUS
- Additionally, node can contain one or more accelerators
- Memory within the node is directly usable by all CPU cores

Nod

Supercomputer autopsy – Sisu.csc.fi

From laptop to Tier-0

- The most fundamental difference between a small university cluster and Tier-0 supercomputer is the number of nodes
  - The interconnect in high end systems is often also more capable

Cloud computing

- Cloud infrastructure is run on top of normal HPC system:
  - Shared memory nodes connected by network
- User obtains **virtual** machines
- Infrastructure as a service (IaaS)
  - User has full freedom (and responsibility) of operating system and the whole software environment
- Platform as a service (PaaS)
  - User develops and runs software within the provided environment

Cloud computing and HPC

- Suitability of cloud computing for HPC depends heavily on application
  - Single node performance is often ok
- Virtualization adds overhead especially for the networking
  - Some providers offer high-speed interconnects with a higher price
- Moving data out from the cloud can be expensive
- Currently, cloud computing is not very cost-effective solution for most large scale HPC simulations

PARALLEL COMPUTING CONCEPTS

Computing in parallel

- Parallel computing
  - A problem is split into smaller subtasks
  - Multiple subtasks are processed *simultaneously* using multiple cores

Types of parallel problems

- Tightly coupled
  - Lots of interaction between subtasks
  - Weather simulation
  - Low latency, high speed interconnect is essential
- Embarrassingly parallel
  - Very little (or no) interaction between subtasks
  - Sequence alignment queries for multiple independent sequences in bioinformatics

Exposing parallelism

- Data parallelism
  - Data is distributed across cores
  - Each core performs simultaneously (nearly) identical operations with different data
  - Cores may need to interact with each other, e.g. exchange information about data on domain boundaries

Exposing parallelism

- Task farm (master / worker)

- Master sends tasks to workers and receives results
- There are normally more tasks than workers, and tasks are assigned dynamically

Exposing parallelism

- MapReduce is special task farm like approach suitable especially for large scale data analysis
- Two types of workers
  - Map: given an input data, emit list of (key, value) pairs
  - Reduce: combine the values for a key
- User works only with serial code for Map and Reduce operations
- MapReduce framework takes care of parallelization and data distribution

Parallel scaling

- Strong parallel scaling
  - Constant problem size
  - Execution time decreases in proportion to the increase in the number of cores

- Weak parallel scaling
  - Increasing problem size
  - Execution time remains constant when number of cores increases in proportion to the problem size

What limits parallel scaling

- Load imbalance
  - Distribution of workload to different cores varies
- Parallel overheads
  - Additional operations which are not present in serial calculation
  - Synchronization, redundant computations, communications
- Amdahl's law: the fraction of non-parallelizable parts establishes the limit on how many cores can be harnessed

PARALLEL PROGRAMMING

Programming languages

- The de facto standard programming languages in HPC are (still!) C/C++ and Fortran
- Higher level languages like Python and Julia are gaining popularity
  - Often computationally intensive parts are still written in C/C++ or Fortran
- For some applications there are high-level frameworks with interfaces to multiple languages
  - TensorFlow for deep learning
  - Spark for MapReduce

Parallel programming models

- Parallel execution is based on threads or processes (or both) which run at the same time on different CPU cores
- Processes
  - Interaction is based on exchanging messages between processes
  - MPI (Message passing interface)
- Threads
  - Interaction is based on shared memory, i.e. each thread can access directly other threads data
  - OpenMP, pthreads

Parallel programming models

MPI: Processes

- Independent execution units
- MPI launches N processes at application startup
- Works over multiple nodes

OpenMP: Threads

- Threads share *memory space*
- Threads are created and destroyed (*parallel regions*)
- Limited to a single node

FUTURE OF HIGH-PERFORMANCE COMPUTING

Exascale challenge

- Performance of supercomputers has increased exponentially for a long time
- However, there are unprecedented challenges in reaching exascale (1 x 10^{18} flop/s)
  - Power consumption: scaling current #1 energy efficient system would still require ~60 MW
  - Fault tolerance: with current technology exascale system experiences hardware failure every few hours
  - Application scalability: how to program for 100 000 000 cores?

Quantum computing

- Quantum computers can solve certain types of problems exponentially faster than classical computers
- General purpose quantum computer is still far away
- For optimisation problems, D-Wave computer based on quantum annealing is already commercially available

CASE STUDY: HEAT EQUATION

Heat equation

- Partial differential equation that describes the variation of temperature in a given region over time

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u$$

  - Temperature variation: u(x, y, z, t)
  - Thermal diffusivity constant: α

Numerical solution

- Discretize: Finite difference Laplacian in two dimensions

$$\nabla^2 u(i,j) \rightarrow \frac{u(i-1,j) - 2u(i,j) + u(i+1,j)}{(\Delta x)^2} + \frac{u(i,j-1) - 2u(i,j) + u(i,j+1)}{(\Delta y)^2}$$

Temperature field u(i,j)

Time evolution

- Explicit time evolution with time step Δt

$$u^{m+1}(i,j) = u^m(i,j) + \Delta t\, \alpha\, \nabla^2 u^m(i,j)$$

- Note: algorithm is stable only when

$$\Delta t < \frac{1}{2\,\alpha} \frac{(\Delta x \Delta y)^2}{(\Delta x)^2 + (\Delta y)^2}$$

- Given the initial condition ($u(t{=}0) = u^0$) one can follow the time evolution of the temperature field

Solving heat equation in parallel

- Temperature at each grid point can be updated independently
- Domain decomposition

- Straightforward in shared memory computer

Solving heat equation in parallel

- In distributed memory computers, each core can access only its own memory
- Information about neighbouring domains is stored in "ghost layers"

- Before each update cycle, CPU cores communicate boundary data: halo exchange

Serial code structure

Parallel code structure

Joonas Nättilä
Seija Sirkiä

WORKING WITH COMPUTING PLATFORMS & VERSION CONTROL

Working with computing platforms

INTRODUCTION TO UNIX

Basic use of Unix-like systems

- To be able to follow the upcoming content, please make sure that you have the relevant basic Unix skills. In particular, check that
  - you know how to start and/or use a command line interface (that is, console, shell, terminal)
  - you know how and when to use at least these basic commands: `ls`, `cp`, `mv`, `rm`
  - you can use some text editor, such as `nano`, `vim`, `emacs`

"Do one thing and do it well"

- This is the core principle of the Unix philosophy
- In accordance to it, all input and output of the different tools should be single flat files...
- ... so that they can be combined through the use of pipes
- Example:

```
$ ls -l | grep key | less
```

Basic skills for working with supercomputers

- Logging in to, and copying files from and to the supercomputer, i.e. a remote host: `ssh` and `scp`
- Basic use of version control
- Make
- Setting up the environment to compile and run programs
  - Module system (before compiling)
  - Batch queue system (for running)

Logging in and moving files

- The `ssh` command is used for logging into a remote host
  - You will be asked for your password to establish a connection
  - Additionaly, ssh can be used to run a command on the host

```
$ ssh [user@]host
```

- The `scp` command is used for secured copying of files and folders across different hosts on a network
  - You will be asked for your password here as well

```
$ scp [user@]host:source target # remote->local
$ scp source [user@]host:target # local->remote
```

Hands-on: ssh and scp

- Create a local text file, for example, "`name.txt`" with your first name as the contents
- Copy your text file to Sisu using `scp`

```
$ scp local.txt trng99@sisu.csc.fi:~/
```

- Login to Sisu using `ssh`, check that file is there, and modify it: for example, add your last name to it
- Copy the modified file back to your local computer, and check that the file has changed

MAKE

Compiling and linking: a reminder

- A *compiler* turns a *source code* file into an *object* file of machine code that the processor can understand
- A linker combines several compiled object files into one *executable* file
- Together, compiling and linking can be called building

Make

Target program

- Large programs are usually separated into several files and can have complicated dependency structures
- Compilation and linking becomes complicated and prone to errors

Make

- **make** is a tool for build automation
- The user writes a **Makefile** with *rules* for generating *targets*
  - a rule is made of *prerequisites* (source files) and *recipes*
- When a source file is changed **make** is able to track the dependencies and regenerate the targets by applying the given rule
  - Changes are detected by comparing the time stamps of the files

Simple Makefile example

- Comments start with #
- Syntax for targets is **target: dep1 dep2**
- Recipies are written after target and *indented with tabulator!*
- The first target in the file is the one that gets generated by default

```
# Compile hello
hello: hello.c
→gcc -o hello hello.c
```

```
$ make hello
gcc -o hello hello.c
$
```

MODULES AND BATCH JOB SYSTEM

Module environment on Sisu

- Supercomputers usually have a large number of users and different users need different development environments and applications
- *Environment modules* provide a convenient way to dynamically change the user's environment so that different compiler suites and application versions can be used more easily

Most common Module commands in Sisu

Load module mod in shell environment	`module load mod`
Remove module mod from environment	`module unload mod`
Get information about module mod	`module show mod`
List all available modules	`module avail`
List loaded modules	`module list`
Get information about a module or, if no argument is given, about the module sub-commands	`module help mod`
To switch loaded mod1 to mod2	`module switch mod1 mod2`

Batch queue system

- Manages the workflow for supercomputers
  - High overall resource utilization
  - CSC systems use SLURM
- To run your program on a supercomputer you need to
  - Describe how you want to run the job (in a batch job file)
  - Add a command that launches your program, in Sisu this is **aprun** (we put this in the batch job file)
    - Taito uses **srun** as job launcher
  - Submit your job to a queue

Batch job file on Sisu

- Example of a batch job file (e.g. my_batch_job_file.sh)

```
#!/bin/bash                          | Invokes bash shell
#SBATCH -J my_job_name               | Defines a name for your job
#SBATCH -o %J.out                    | Standard output file
#SBATCH -e %J.err                    | Standard error file
#SBATCH -N 1                         | Number of nodes to use (each has 24 cores)
#SBATCH -t 1:00:00                   | Maximum time to run the program (hh:mm:ss)
#SBATCH -p test                      | Batch job partition (initially use 'test')
aprun -n 24 path_to_your_program_file
```

- For full list of options run the command below (on CLI):

```
$ sbatch -h                          | sbatch options
$ man aprun                          | aprun manual
```

Running batch jobs

- Submit your batch job script to the queue using sbatch
  $ sbatch my_job.sh
- You can follow the status of your jobs with squeue:
  $ squeue -u my_username
- If something goes wrong, you can cancel your job with scancel:
  $ scancel jobid
  (here the jobid is the numeric id of the job)

VERSION CONTROL WITH GIT

Hands up - is this familiar?

Solution: Version control

- A version control system (VCS) is a kind of a "time machine"
  - Keeps track of changes made to files
- And a project manager
  - Keeps track of multiple versions of the code
- Imagine it as something that sits inside your folder and helps you with your files

Advantages of using version control

- You can go back to the code as it was at any given point in time with single command
- You can merge unrelated changes made by different people
- You can manually decide what to do with conflicting regions and some of the work is done for you by the version control system
  - e.g. it does what it can and shows you the places where it needs manual intervention

Distributed vs. Centralized

- Centralized
- Distributed

Remote repositories

- Modern Version control systems are distributed
  - Git (and Mercurial,…)
  - You may also encounter older non-distributed VCSs like svn
- Git is usually paired up with web-based repository
  - GitHub (140k matches in Google Scholar)
  - Bitbucket (12k)
  - GitLab (2k), SourceForge (160k, however…) etc.
  - Your own server
- Most modern and widely used combo: Git + Github

Social coding with git+Github

- Interact with collaborators
  - Scales from 1 to 1000+ contributors (may include your supervisor too!)
- Open projects enable other people to join too
- Or have a direct influence on people who contribute by running a closed project

Social coding with git+Github

- These days Github account also serves as your "code CV"

Initializing repositories

- First we need to create a new remote repository in GitHub
- Next we "clone" (=download) it
  - `git clone` https://github.com/user/repo.git

Forking & cloning repositories

- Sometimes you want to copy repository by somebody else
- This is called "forking" and can be done in GitHub
- Forking a repository allows you to freely experiment with changes without affecting the original project.
- Once forked, we can again clone the repository to ourselves

Forking & cloning repositories

- Let's fork & clone our first repository
  - Fork https://github.com/csc-training/summerschool

  - `git clone https://github.com/user/summerschool`

Git Workflow

Git Workflow

1. New file changes are pulled from the remote repository
   - Files are now up-to-date (in respect to the remote repository)
2. Something is modified
3. Modifications are added for commit
   - Think of this as putting your file changes into a package
4. Changes are committed to the local repository
5. Commits are pushed to the remote repository

Git Workflow

- In every point, you can check the current status of the local repository
  - `git status`
  - What is changed? What is staged for commit?
- Sometimes it is useful to drop local changes
  - `git stash`
  - Restores the local repository to the latest commit by stashing your changes (they can be retrieved later on)
- These are super useful, hence, use them often!

Advanced features - Branching

- By default everything is happening at "master" branch
- You can freely create new branches
  - `git branch new_branch_name`
  - `git checkout new_branch_name`
- `git pull` is actually `git fetch && git merge` that fetches remote master branch and merges it into local master branch
- Allows to work on some aspect of the project and then fall back to the master branch if needed
  - Changes made to the new branch are stored
  - Files can be restored to the default master branch version

Advanced features - Pull requests

- In bigger projects it is typical not to push directly but to send a "pull request" from your fork or branch
  - Typically done in the www-interface of GitHub
- Pull request is usually a fix or an improvement containing multiple commits
- Allows to review and modify changes

Summary

- If you are not using git, start using it!
- Remember the standard workflow: pull, modify, add, commit, push
- Explore GitHub, there are quite a many cool repositories!
- Check https://help.github.com/ - it has an answer to everything git-related!

Version control git commands

- List of common commands
  - Download repository: `git clone https://…`
  - Update changes from remote repo: `git pull`
  - Check the status of repository: `git status`
  - Add new file for staging: `git add file1`
  - Commit changes to local repository: `git commit -m "X"`
  - Update local repo to remote repo: `git push`
  - Delete a file from repository: `git rm file1`
  - Documentation: `git help [cmd]`

Jussi Enkovaara
Pekka Manninen

FORTRAN 95/2003

Overview

Wednesday

09:00-09:45	Getting started with Fortran
09:45-10:15	Exercises
10:15-10:30	Coffee break
10:30-11:00	Fortran arrays
11:15-12:00	Exercises
12:00-13:00	Lunch break
13:00-13:30	Fortran arrays (continued)
13:30-14:00	Exercises
14:00-14:15	Coffee break
14:15-15:00	Procedures & modules
15:00-15:45	Exercises

Thursday

09:00-09:45	Input/Output
09:45-10:30	Exercises
10:30-10:45	Coffee break
10:45-11:15	Derived types
11:15-12:00	Exercises
12:00-13:00	Lunch break
13:00-13:45	Further useful features
13:45-14:15	Exercises
14:15-14:30	Coffee break
14:30-15:00	Exercises
15:00-15:20	Overview of advanced features
15:20-15:45	Exercise review, Q & A

Getting started with Fortran

Outline

- Fortran as a language
  - look and feel
  - from code to program
- Variables and operators
  - declare, assign
  - arrays
- Control structures
  - branches
  - loops

FORTRAN AS A LANGUAGE

Why learn Fortran?

- Well suited for **numerical computations**
  - Likely over 50% of scientific applications are written in Fortran
- **Fast** code (compilers can optimize well)
- Handy **array data types**
- **Clarity** of code
- **Portability** of code
- Optimized **numerical libraries** available

Short history of Fortran

- John W. Backus et al (1954): The IBM Mathematical **For**mula **Tran**slating System
- Early years development: Fortran II (1958), Fortran IV (1961), Fortran 66 & Basic Fortran (1966)
- Fortran 77 (1978)
- Fortran 90 (1991) a major revision and Fortran 95 (1997) a minor revision to it

Short history of Fortran

- Fortran 2003 (2004): a major revision, adding e.g. object-oriented features, C-bindings
  - ”Fortran 95/2003” is the current *de facto* standard
- The latest standard is Fortran 2008 (2010): a minor revision
  - Most notable addition: Fortran coarray syntax
  - Compiler support nearly complete
  - The next revision: Fortran 2015

Compiling and linking

Transition from code to a program

- Compile and link in one go, execute the binary

```
gfortran main.f90 -o foo
./foo
```

- In more complex cases (multiple sources)
  - Compile each source code file (.f90) into an object file (.o)

```
gfortran -c main.f90
gfortran -c sub.f90
```

  - Link object files into a binary and execute the binary

```
gfortran -o foo main.o sub.o
./foo

# after some modifications in "sub.f90"
gfortran -c sub.f90
gfortran -o foo main.o sub.o
./foo
```

Look and feel

```
program square_root_example
   ! comments start with an exclamation point.
   ! you will find data type declarations, couple arithmetic operations
   ! and an interface that will ask a value for these computations.
   implicit none
   real :: x, y
   intrinsic sqrt ! fortran standard provides many commonly used functions

   ! command line interface. ask a number and read it in
   write (*,*) 'give a value (number) for x:'
   read (*,*) x

   y = x**2+1   ! power function and addition arithmetic

   write (*,*) 'given value for x:', x
   write (*,*) 'computed value of x**2 + 1:', y
   ! print the square root of the argument y to screen
   write (*,*) 'computed value of sqrt(x**2 + 1):', sqrt(y)
end program square_root_example
```

Source code remarks

- Free format source code, but
  - A variable name can be no longer than 31 characters containing only letters, digits or underscore, and must start with a letter
  - Maximum row length is 132 characters
- No distinction between lower and uppercase characters
  - Character strings are case sensitive
- Line break is the statement separator
  - If a line is ended with an ampersand (&), the line continues onto the next line (max. 39 continuation lines allowed)
  - Semicolon (;) is the separator between statements on a single line

VARIABLES

Variables

```
integer :: n0

real :: a, b
real :: r1=0.0

complex :: c
complex :: imag_number=(0.1, 1.0)

character(len=80) :: place
character(len=80) :: name='james bond'

logical :: test0 = .true.
logical :: test1 = .false.

real, parameter :: pi=3.14159
```

Variables must be *declared* at the beginning of the program or procedure where they are used

the *intrinsic* data types in Fortran are **integer**, **real**, **complex**, **character** and **logical**

They can also be given a value at declaration (not recommended)

Constants are defined with the PARAMETER clause – they cannot be altered after their declaration

Operators

Arithmetic operators

```
real :: x,y
integer :: i = 10
x = 2.0**(-i)  !power function and negation     precedence: first
x = x*real(i)  !multiplication and type change  precedence: second
x = x/2.0      !division                        precedence: second
i = i+1        !addition                        precedence: third
i = i-1        !subtraction                     precedence: third
```

Relational operators

```
<  or .lt.   !less than
<= or .le.   !less than or equal to
== or .eq.   !equal to
/= or .ne.   !not equal to
>  or .gt.   !greater than
>= or .ge.   !greater than or equal to
```

Logical operators

```
.not.     !logical negation              precedence: first
.and.     !logical conjunction           precedence: second
.or.      !logical inclusive disjunction precedence: third
```

Arrays

```
integer, parameter :: m = 100, n = 500
integer :: idx(m)
real :: vector(0:n-1)
real :: matrix(m, n)
character (len=80) :: screen (24)

! or

integer, dimension(m) :: idx
real, dimension(0:n-1) :: vector
real, dimension(m, n) :: matrix
character(len=80), dimension(24) :: screen
```

By default, Fortran indexing starts from 1

CONTROL STRUCTURES

Conditionals (if-else)

- Conditionals allow the program to execute different code based on some condition(s)

```
if (condition) then
    ! do something
else if (condition2) then
    ! .. or maybe alternative something else
else
    ! .. or at least this
end if
```

- Condition can be anything from a simple comparison to a complex combination using logical operators

Conditionals example

```
program placetest
  implicit none
  logical :: in_square1, in_square2
  real :: x, y
  write(*,*) 'give point coordinates x and y'
  read (*,*) x, y
  in_square1 = (x >= 0. .and. x <= 2. .and. y >= 0. .and. y <= 2.)
  in_square2 = (x >= 1. .and. x <= 3. .and. y >= 1. .and. y <= 3.)
  if (in_square1 .and. in_square2) then      ! inside both
    write(*,*) 'point within both squares'
  else if (in_square1) then                  ! inside square 1 only
    write(*,*) 'point inside square 1'
  else if (in_square2) then                  ! inside square 2 only
    write(*,*) 'point inside square 2'
  else                                       ! both are .false.
    write(*,*) 'point outside both squares'
  end if
end program placetest
```

Loops

- Three loop formats available in Fortran
  - integer counter (fixed number of iterations)
  - condition controlled (do until condition is false)
  - explicit exit statement

```
do {control clause}
    ! execute something again and again until stopped
end do

! where the control clause (optional) is either of the form
!    i=init_value, max_value, increment
! or a condition to execute while true
!    while (condition)
```

Loops example

```
integer :: i, stepsize, numberofpoints
integer, parameter :: max_points=100000
real :: x_coodinate(max_points), x, totalsum

! a do-loop with an integer counter (count controlled)
stepsize = 2
do i = 1, max_points, stepsize
  x_coordinate(i) = i*stepsize*0.05
end do
! condition controlled loop (do while)
totalsum = 0.0
read(*,*) x
do while (x > 0)
  totalsum = totalsum + x
  read(*,*) x
end do
```

Loops example

```
real :: x, totalsum, eps
totalsum = 0.0

! do loop without loop control
do
  read(*,*) x
  if (x < 0) then
    exit     ! = exit the loop
  else if (x > upperlimit) then
    cycle    ! = do not execute any further statements, but
             !   instead cycle back to the beginning of the loop
  end if
  totalsum = totalsum + x
end do
```

Labels example

```
program gcd
  ! computes the greatest common divisor, Euclidean algorithm
  implicit none
  integer :: m, n, t
  write(*,*)' give positive integers m and n :'
  read(*,*) m, n
  write(*,*)'m:', m,' n:', n
  positive_check: if (m > 0 .and. n > 0) then
    main_algorithm: do while (n /= 0)
      t = mod(m,n)
      m = n
      n = t
    end do main_algorithm
    write(*,*) 'greatest common divisor: ',m
  else
    write(*,*) 'negative value entered'
  end if positive_check
end program gcd
```

Labels can be given to control structures and used in conjunction with e.g. exit and cycle statements

Select case

```
integer :: i
logical :: is_prime,
   test_prime_number

...

select case (i)
  case (2,3,5,7)
    is_prime = .true.
  case (1,4,6,8:10)
    is_prime = .false.
  case default
    is_prime=test_prime_number(i)
end select

...
```

- `select case` statements matches the entries of a list against the case index
  - Only one found match is allowed
  - Usually arguments are character strings or integers
  - `default` branch if no match found

Summary

- Fortran is – despite its long history – a modern programming language designed especially for scientific computing
  - Versatile, quite easy to learn, powerful
- In our first encounter, we discussed
  - Variables, data types, operators
  - Control structures: loops and conditionals

Fortran arrays

Outline

- Array syntax & array sections
- Dynamic memory allocation
- Array intrinsic functions
- Pointers to arrays

Introduction to Fortran arrays

- Fortran is a very versatile programming language for handling arrays
  - especially multi-dimensional arrays
- Arrays refer to a data type (built-in or derived), but also have one or more dimensions specified in the variable declaration
  - Fortran supports up to 15 dimensions

Recall: declaring arrays

```
integer, parameter :: m = 100, n = 500
integer :: idx(m)
real :: vector(0:n-1)
real :: matrix(m, n)
character (len = 80) :: screen(24)

! or equivalently

integer, dimension(m) :: idx
real, dimension(0:n-1) :: vector
real, dimension(m, n) :: matrix
character(len=80), dimension(24) :: screen
```

By default, indexing starts from 1

Array syntax

- Arrays can be accessed element-by-element

```
do j = 0, 10
   vector(j) = 0
   idx(j) = j
end do
```

- Arrays can be treated also with *array syntax*

```
vector = 0
! or vector(:) = 0

vector(0:10) = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
! or
idx(0:10) = [ (j, j = 0, 10) ] ! "implied do"
! the [...] syntax is equivalent to (/ ... /)
```

Array syntax

- Array syntax allows for less explicit do loops

```
integer :: m = 100, n = 200
real :: a(m,n) , x(n), y(m)
integer :: i , j

y = 0.0
outer_loop : do j = 1, n
  inner_loop : do  i = 1, m
    y(i) = y(i) + a(i,j) * x(j)
  end do inner_loop
end do outer_loop
```

```
integer :: m = 100, n = 200
real :: a(m,n) , x(n), y(m)
integer :: j

y = 0.0
outer_loop : do j = 1, n
    y(:) = y(:) + a(:,j) * x(j)
end do outer_loop
```

Array sections

- With Fortran array syntax we can access a part of an array in an intuitive way: *array sections* ("slices" in some other languages)

```
Sub_Vector( 3 : N )  = 0
Every_Third( : : 3 ) = 1
Diag_Block( i-1 : i+1, j-2 : j+2 ) = k

real :: a( 1000, 1000)
a(2:500, 3:300:3) = 4.0
```

(start : end : stride)

- When copying array sections, both left and right hand sides has to have conforming dimensions

```
lhs(1:2, 0:9) = rhs(-2:0, 20:29)
! but here is an error :
lhs(1:2, 0:9) = rhs(-2:0, 20:29)
```

Dynamic memory allocation

- Memory allocation is *static* if the array dimensions have been declared at compile time
- If the sizes of an array depends on the input to program, its memory should be *allocated* at runtime
  - memory allocation becomes *dynamic*

Dynamic memory allocation

```
integer :: alloc_stat
integer, allocatable :: idx(:)
real, allocatable :: mat(:,:)

...
read (*,*) m, n
allocate (idx(0:m-1))

allocate (mat(m,n), stat=alloc_stat)
if (alloc_stat /= 0) stop
...
deallocate (mat)
```

The shapes (but not sizes) of the arrays need to be known upon declaration

You can specify the start index, by default indexing starts from 1

An optional error code whether the allocation was successful

Destroys the allocation (and the contents of the array)

Memory allocation with automatic arrays

- "Automatic arrays" can be seen in older codes
  - Not recommended

```
subroutine calculate(m, n)
  integer :: m, n ! intended dimensions
  integer :: idx(0:m-1)       ! an automatic array
  real :: mat(m,n)            ! an automatic array
! no explicit allocate – but no checks upon failure either
  ...
  call do_something(m, n, idx, mat)
  ...
! no explicit deallocate - memory gets reclaimed automatically
end subroutine calculate
```

Array intrinsic functions

- Built-in functions can apply various operations on a whole array, not just array elements
  - As a result either another array or a scalar value is returned
- A selection of just some part of the array through *masking* is possible

Array intrinsic functions examples

- `size(array [,dim])` returns the number of elements in the array
- `shape(array)` returns an integer vector containing the size of the array with respect to each of its dimension
- `count(l_array [,dim])` returns the count of elements which are .true. in the logical l_array
- `sum(array[, dim][, mask])` returns the sum of the elements

Array intrinsic functions

- `any(l_array [, dim])` returns a scalar value of .true. if any value in logical l_array is .true.
- `all(l_array [, dim])` returns a scalar value of .true. if all values in logical l_array are .true.
- `minval /maxval (array [,dim] [, mask])` return the minimum/maximum value in a given array
- `minloc/maxloc (array [, mask])` return a vector of location(s) where the minimum/maximum value(s) are found

Array intrinsic functions: example

```
integer :: j
integer, parameter :: m = 10, n = 20
real :: x(m,n), v(n)

call random_number(x)

print *, size(x), size(v) ! prints m * n, n
print *, shape(x)         ! prints m, n
print *, size(shape(x))   ! prints 2
print *, count(x >= 0)
print *, sum(x, dim=2, mask=x < 0.5)  ! the result is a vector

v(1:n) = [ (j, j=1,n) ]
print *, any(v > -1 .and. v < 1)
print *, all(x >= 0, dim=1)
print *, minval(v), maxval(v)
print *, minloc(v), maxloc(v)
```

Array intrinsic functions

- `reshape(array,shape)` returns a reconstructed array with different shape than in the input array

```
integer :: m, n
real :: a(m, n), v(m*n)
…
! convert a (m-by-n matrix) into v (a m x n vector) without loops
v = reshape(a, shape(v))
```

Array intrinsic functions

- Some array functions manipulate vectors/matrices effectively
  - `dot_product(v,w)` returns a dot product of two vectors
  - `matmul(a,b)` returns matrix multiply of two matrices
  - `transpose(a)` returns transposed of the input matrix

Array intrinsic functions

- Array control statements `forall` and `where` are commonly used in the context of manipulating arrays
- They can provide a masked assignment of values using effective vector operations

```
integer :: j, ix(5)
ix(:) = (/ (j, j=1,size(ix)) /)
where (ix == 0) ix = -9999
where (ix < 0)
    ix = -ix
elsewhere
    ix = 0
end where
```

```
integer :: j
real :: a(100,100), b(100), c(100)
! fill in diagonal matrix
forall (j=1:100) a(j,j) = b(j)
! fill in lower bi-diagonal matrix
forall (j=2:100) a(j,j-1) = c(j)
```

Pointers to arrays

- The **`pointer`** attribute enables to create array (or scalar) *aliasing variables*
- Pointer variables are usually employed to *refer* to another array or array section
- A pointer variable can also be a sole variable itself, used together with **`allocate`** for dynamic memory allocation
  - This is not a recommended practice
- C programmers: a "pointer" is a completely different construct in Fortran

Pointers to arrays

- A pointer array can refer to an already allocated memory region, declared as a `target`

```
integer, pointer :: p_x(:)
integer, target :: x(1000)
...
p_x => x
p_x => x ( 2 : 300 )
p_x => x ( 1 : 1000 : 5 )
...
p_x(1) = 0
nullify(p_x)
```

The pointer array can point to a whole array or a part of it, no need to reallocate

This would change the value of x(1) too

Now p_x points to nothing

Pointers to arrays

```
real, pointer :: p_mat(:,:) => null()
real, target  :: mat(100,200)

p_mat => mat
if ( associated (p_mat) ) &
  print *, 'points to something'

nullify(p_mat)
if (.not. associated (p_mat) ) &
  print *, 'points to nothing'
```

- The **`associated`** function can be used to check whether a `pointer` is associated with a target
  - Returns `.true.` if associated with a target, `.false.` if not
  - For an uninitialized `pointer` variables, the return value is undefined

Summary

- *Arrays* make Fortran language a very versatile vehicle for computationally intensive program development
- Using its *array syntax*, vectors and matrices can be initialized and used in a very intuitive way
- *Dynamic memory allocation* enables sizing of arrays according to particular needs
- *Array intrinsic functions* further simplify coding effort and improve code readability

Procedures and modules

Outline

- Structured programming
- Modules
- Procedures: functions and subroutines
- Interfaces

Structured programming

- Structured programming based on program sub-units (*functions*, *subroutines* and *modules*) enables
  - Testing and debugging separately
  - Re-use of code
  - Improved readability
  - Re-occurring tasks
- The key to success is in well defined data structures and scoping, which lead to clean procedure interfaces

Modular programming

- Modularity means dividing a program into minimally dependent *modules*
  - Enables division of the program into smaller self-contained units
- Fortran modules enable
  - Global definitions of procedures, variables and constants
  - Compilation-time *error checking*
  - Hiding *implementation details*
  - *Grouping* routines and data structures
  - Defining *generic procedures* and custom operators

What are procedures?

- Procedure is block of code that can be called from other code.
- Calling code passes data to procedure via arguments
- Fortran has two types of procedures: *subroutines* and *functions*
- Subroutines pass data back via arguments

```
call mySubroutine(arg1_in, arg2_in, arg_out)
```

- Functions return a value

```
value = myFunction(arg1, arg2)
```

Procedure declarations

Subroutine

Declaration:

```
subroutine sub(arg1, arg2,...)
  [declarations]
  [statements]
end subroutine sub
```

Use as

```
call sub(arg1, arg2,...)
```

Function

Declaration:

```
[type] function func(arg1,
    arg2,...) [result(val)]
  [declarations]
  [statements]
end function func
```

Use as

```
res = func(arg1, arg2,...)
```

Procedure declarations: example

```
subroutine dist(x, y, d)
  implicit none
  real :: x, y, d
  d = sqrt(x**2 + y**2)
end subroutine dist

program do_something
  ...
  call dist(x, y, r)
  ...
```

```
real function dist(x, y)
  implicit none
  real :: x, y
  dist = sqrt(x**2 + y**2)
end function dist

program do_something
  ...
  r = dist(x, y)
  ...
```

Procedure arguments

- Fortran passes call arguments *by reference*
  - Only the memory addresses of the arguments are passed to the called procedure
  - Any change to the value of an argument changes the value at the calling program
  - The `INTENT` keyword can be used to specify how argument is used

INTENT keyword

```
subroutine foo(x, y, z)
  implicit none
  real, intent(in)       :: x
  real, intent(inout) :: y
  real, intent(out)   :: z

  x = 10     ! compilation error
  y = 10     ! correct
  z = y * x  ! correct
end subroutine foo
```

- Declares how formal argument is intended to be used for transferring a value
  - **`IN`**: the value of the argument is read-only i.e. cannot be changed
  - **`OUT`**: the value of the argument must be provided
  - **`INOUT`** (the default)
- Compiler uses `INTENT` for error checking and optimization
- Improves readability of code

Local variables in procedures

- Local variables can be declared in the procedure

```
subroutine foo(x, y)
  implicit none
  real, intent(in) :: x
  real, intent(out) :: y

  integer :: i ! Local variable
  ...
```

- Local variables are not visible outside the procedure
- By default, local variables do not retain their values through successive calls of the procedure

Local variables in procedures

- If local variable is given **SAVE** attribute, its value is retained through successive calls
- Initialization in declaration is done only in the first call to procedure, and implicit SAVE is applied

```
subroutine foo1(x)
  …
  integer :: i
  i = 0
  i = i + 1
```

```
subroutine foo2(x)
  …
  integer :: i = 0
  i = i + 1
```

- In **foo1** variable **i** starts always from 0 and gets value 1
- In **foo2** variable **i** gets values 1, 2, 3, … in each successive call

Should I use a subroutine or a function?

- The main difference is that functions can be used in expressions:
  `r = dist(x1, y1) + dist(x2, y2)`
- Recommendation as a good programming practice
  - Use functions for computing value based on input
  - Use subroutines for performing operation that changes some of the inputs

PROCEDURE TYPES, MODULES

Procedure types

- There are four procedure types in Fortran: *intrinsic, external*, *internal* and *module* procedures
- Procedure types differ in
  - Scoping, i.e. what data and other procedures a procedure can access
  - Interface type, explicit or implicit
- Compiler can check the argument types of the at compile time only if the interface is explicit!

Implicit and explicit procedure types

- The interfaces of the intrinsic, internal and module procedures are *explicit*
- The interfaces of the external procedures, such as many library subroutines, are *implicit*. You can write an explicit interface to those, though.
- Intrinsic procedures are the procedures defined by the programming language itself, such as `intrinsic sin`

Module procedures and variables

Declaration

```
module check
  implicit none
  integer, parameter :: &
    longint = selected_int_kind(8)
contains
  function check_this(x) result(z)
    integer(longint):: x, z
    ...
  end function
end module check
```

Module procedures are declared after the CONTAINS statement

Usage

Procedures defined in modules can be referred to in any other program unit with the USE clause

```
program testprog
  use check
  implicit none
  integer(kind=longint) :: x,test
  test = check_this(x)
end program testprog
```

a good habit

```
use check, only: longint
```

Visibility of module objects

- Variables and procedures in *modules* can be `PRIVATE` or `PUBLIC`
  - `PUBLIC` = visible for all program units using the module (the default)
  - `PRIVATE` will hide the objects from other program units

```
real :: x, y
private :: x
public :: y
! or
real, private :: x
real, public :: y
```

Internal procedures

- Each program unit (program/subroutine/function) may contain *internal procedures*

```
subroutine my_subroutine
  ...
  call my_internal_subroutine
  ...
contains
  subroutine my_internal_subroutine
    ...
  end subroutine my_internal_subroutine
end subroutine my_subroutine
```

Internal procedures

- Declared at the end of a program unit after the **`contains`** statement
  - Nested `contains` statements are not allowed
- Variable scoping:
  - Parent unit's variables and objects are accessible
  - Parent unit's variables are overlapped by local variables with the same name
- Can be called only from declaring program unit
- Often used for "small and local, convenience" procedures

Internal procedures: example

```
subroutine parent()
  implicit none
  integer :: i,j
  i = 1; j = 1
  call child()
  ! after subroutine call i = 2 and j = 1

contains

  subroutine child()
    implicit none
    integer :: j
    i = i + 1    ! variable i is from the scope of parent
    j = 0        ! variable j has local scope
  end subroutine child

end subroutine parent
```

External procedures

- Declared in a separate program unit
  - Referred to with the **EXTERNAL** keyword
  - Compiled separately and linked to the final executable
- Avoid using them within a program, module procedures provide much better compile time error checking
- External procedures are often needed when using
  - procedures written with different programming language
  - library routines (e.g. BLAS and LAPACK libraries)
  - old F77 subroutines

Interfaces

- For external procedures, interfaces determine the type and properties of arguments and return values
- Defined by an **INTERFACE** block:

```
interface
   interface-body
end interface
```

- The ***interface-body*** matches the subprogram header
  - position, rank and type of arguments
  - return value type and rank (for functions)

Interfaces

```
interface
  subroutine not_dangerous(a, b, c)
    integer :: a, b, c
  end subroutine not_dangerous
end interface

integer :: x, y, z
x=1; y=1; z=1
! call external subroutine without
! an interface
call dangerous(x,y,z)
! call external subroutine with
! an interface
call not_dangerous(x,y,z)
```

- Wrong calling arguments to **external** procedures may lead to errors during the executable linking phase or even when the executable is being run
- It is highly recommended to construct **interface** blocks for any external procedures used

Interfaces: example

```
! lu decomposition from lapack
interface
  subroutine dgetrf(m, n, a, lda, ipiv, info)
    integer :: info, lda, m, n
    integer:: ipiv(*)
    double precision :: a(lda,*)
  end subroutine dgetrf
end interface
! euclidean norm from blas
interface
  function dnrm2(n, x, incx)
    integer :: n, incx
    double precision :: x(*)
    double precision :: dnrm2
  end function dnrm2
end interface
```

Global data and global variables

- Global variables can be accessed from any program unit
- Module variables with the `save` attribute provide controllable way to define and use global variables

```
module commons
  integer, parameter :: r = 0.42
  integer, save :: n, ntot
  real, save :: abstol, reltol
end module commons
```

  - Explicit interface: type checking, limited scope
- Generally, use of global variables is not recommended

Summary

- Procedural programming makes the code more readable and easier to develop
  - Procedures encapsulate some piece of work that makes sense and may be worth re-using elsewhere
- Fortran uses *functions* and *subroutines*
  - Values of procedure arguments may be changed upon calling the procedure
- Fortran *modules* are used for modular programming and data encapsulation

Outline

- Input/output (I/O) formatting
- Internal I/O
- File I/O
  - File opening and closing
  - Writing and reading to/from a file
  - Formatted and unformatted (binary) files
  - Stream I/O

Input/Output formatting

- To prettify output and to make it human readable, use `format` descriptors in connection with the `write` statement
- Although less often used nowadays, it can also be used with `read` to input data at fixed line positions and using predefined field lengths
- Used through `format` statements, `character` variable or embedded in `read/write` fmt keyword

Output formatting

Data type	Format descriptors	Examples

Integer	`iw, iw.m`	`write(*,'(i5)') j`
		`write(*,'(i5.3)') j`
		`write(*,'(i0)') j`

Real (decimal and exponential forms, auto-scaling)	`fw.d`
	`ew.d`
	`gw.d`
		`write(*,'(f7.4)') r`
		`write(*,'(e12.3)') r`
		`write(*,'(g20.13)') r`

Character	`a, aw`	`write(*,'(a)') c`
Logical	`lw`	`write(*,'(l2)') l`

w=width of the output field, **d**=number of digits to the right of decimal point, **m**=minimum number of characters to be used.
Variables: Integer :: J, Real :: R, Character :: C, Logical :: L

Output formatting: miscellaneous

- With complex numbers provide format for both real and imaginary parts:

```
complex :: z
write (*,'(f6.3,2x,f6.3)') z  ! real & imaginary parts
```

- Line break and whitespace:

```
write (*,'(f6.3,/,f6.3)') x, y ! linebreak between x,y
write (*,'(i3,2x,f6.3)') i, x ! 2 spaces between i & x
```

- It is possible that an edit descriptor will be repeated a specified number of times

```
write (*,'(5i8)') ivec(1:5)
write (*,'(4(i5,2x,f8.3))') (ivec(j),zvec(j),j=1,4)
```

The I0 and G0 format descriptors

- Dynamic sizing of **`real`** and **`integer`** valued output
  - **`I0`** appeared in F03 and G0 was introduced in F08
- Output fields are left justified with all the unnecessary leading blanks (and precision for **`real`** valued variables) removed

```
integer :: i = 12345
real (kind=4) :: sp = 1.23e0
real (kind=8) :: dp = 1.234567890d0
write(*,fmt='("<i=",i0,", reals=",2(g0,1x),">")') i,sp,dp
```

- Output is `<i=12345, reals=1.230000 1.234567890000000 >`

Handling character strings

- Fortran provides several intrinsic functions for handling character strings, such as
  - `trim(string)` - removes blank spaces from the end of string
  - `adjustl(string)/adjustr(string)` - moves blank spaces from the beginning/end of the string to the end/beginning of it
  - `len(string)` - length of a string
  - `index(string, substring)` - returns the starting position of a substring within a string

Internal I/O

- Often it is necessary to filter out data from a given character string
  - Or to pack values into a character string
- Fortran internal I/O with `read` & `write` becomes now handy
- Actual files are not involved at all

Internal I/O: examples

```
character(len=13) :: cl1
character(len=60) :: cl2
integer :: njobs, istep

! extract a number from character string
cl1 = 'time step# 10'
read(cl1,fmt='(10x,i3)') istep


! write data to a character string
njobs = 2014
write(cl2,'(a,i0)') 'the number of jobs completed = ', njobs
```

Opening and closing files: basic concepts

- Writing to or reading from a file is similar to writing onto a terminal screen or reading from a keyboard
- Differences
  - File must be opened with an OPEN statement, in which the unit number and (optionally) the file name are given
  - Subsequent writes (or reads) must to refer to the given unit number
  - File should be closed at the end

Opening and closing a file

- The syntax is (the brackets [ ] indicate optional keywords or arguments)

```
open([unit=]iu, file='name' [, options])
close([unit=]iu [, options])
```

- For example

```
open(10, file= 'output.dat', status='new')
close(unit=10, status='keep')
```

Opening and closing a file

- The first parameter is the unit number
- The keyword `unit=` can be omitted
- The unit numbers 0, 5 and 6 are predefined
  - 0 is output for standard (system) error messages
  - 5 is for standard (user) input
  - 6 is for standard (user) output
  - These units are opened by default and should not be re-opened nor closed by the user

Opening and closing a file

- The *default input/output unit* can be referred with a star:

```
write(*, ...)
read(*, ...)
```

  - Note that these are not necessarily the same as the *stdout* and *stdin* unit numbers 6 and 5
- If the file name is omitted in the OPEN, the file name will based on unit number being opened, e.g. for `unit=12` this usually means the filename 'fort.12' (on UNIX-systems)

File opening options

`status`: existence of a file
  - `'old'`, `'new'`, `'replace'`, `'scratch'`, `'unknown'`

`position`: offset, where to start writing
  - `'append'`

`action`: file operation mode
  - `'write'`, `'read'`, `'readwrite'`

`form`: text or binary file
  - `'formatted'`, `'unformatted'`

File opening options

`access`: direct or sequential file access
  - `'direct'`, `'sequential'`, `'stream'`,

`iostat`: error indicator, (output) integer
  - non-zero only upon an error

`err`: the fortran label number to jump upon an error

`recl`: record length, (input) integer
  - for direct access files only
  - warning (check): may be in bytes *or* words

File opening: file properties

- Use the `inquire` statement to find out information about
  - file existence
  - file unit open status
  - various file attributes
- The syntax has two forms, one based on file name, the other for unit number

```
inquire(file='name', options ...)
inquire(unit=iu, options ...)
```

File opening: file properties

`exist`	does file exist? (logical)
`opened`	is file / unit opened? (logical)
`form`	`'formatted'` or `'unformatted'` (char)
`access`	`'sequential'` or `'direct'` or `'stream'` (char)
`action`	`'read'`, `'write'`, `'readwrite'` (char)
`recl`	record length (integer)
`size`	file size in bytes (integer)

File opening: file properties example

```
! check the existence of a file
logical :: file_exist
inquire(file='foo.dat', exist=file_exist)
if (.not. file_exist) then
  write(*,*) 'the file does not exist'
else
  ! do something with the file foo.dat
endif
```

File writing and reading

- Writing to and reading from a file is done by giving the corresponding unit number (iu) as a parameter :

```
write(iu,*) str
write(unit=iu, fmt=*) str
read(iu,*) str
read(unit=iu, fmt=*) str
```

The star format (*) indicates list-directed output (i.e. programmer does not choose the input/output styles)

- Formats and other options can be used as needed
- If keyword 'unit' is used, also the keyword 'fmt' must be used
  - Note: `'fmt'` is applicable to formatted, text files only

Formatted vs. unformatted files

- Text or *formatted* files are
  - Human readable
  - Portable i.e. machine independent
- Binary or *unformatted* files are
  - Machine readable only, generally not portable
  - Much *faster to access* than formatted files
  - Suitable for large amount of data due to *reduced file sizes*
  - Internal data representation used for numbers, thus no number conversion, no rounding of errors compared to formatted data

Unformatted I/O

- Write to a sequential binary file

```
real :: rval
character(len=60) :: string
open(10, file='foo.dat', form='unformatted')
write(10) rval
write(10) string
close(10)
```

- No `format` descriptors allowed
- Reading similarly

```
read(10) rval
read(10) string
```

Stream I/O

- A binary file write adds extra record delimiters (hidden from programmer) to the beginning and end of records
- In Fortran 2003 using access method 'stream' avoids this and implements a C-like approach
- It is recommended to use stream I/O
- Create a stream (binary) file

```
real :: dbheader(20), dbdata(300)
open(10,file='my_database.dat', access='stream')
write(10) dbheader
write(10) dbdata
close(10)
```

- Reading similarly

Command line input

- In many cases, it is convenient to give parameters for the program directly during program launch
  - Instead of using a parser, reading from an input file etc.
- Fortran 2003 provides a way for this
  - `command_argument_count()`
  - `get_command_argument(integer i, character arg(i))`

Command line input

- Example: reading in two integer values from the command line
- The (full) program should be launched as (e.g.)
  `% ./a.out 100 100`

```
subroutine read_command_line(height, width)
  integer, intent(out) :: height, width
  character(len=10) :: args(2)
  integer :: n_args, i
  n_args = command_argument_count()
  if ( n_args /= 2 ) then
    write(*,*) ' Usage : ./exe height width'
    call abort()
  end if
  do i = 1, 2
    call get_command_argument(i,args(i))
    args(i) = trim(adjustl(args(i)))
  end do
  read(args(1),*) height
  read(args(2),*) width
end subroutine read_command_line
```

Summary

- Input/Output formatting
- Internal I/O
- Files: communication between a program and the outside world
  - Opening and closing a file
  - Data reading & writing
- Use unformatted (stream) I/O for all except text files

Derived data types

Built-in data types in Fortran

- Fortran has built-in data types for
  - integers (`integer`)
  - floating point numbers (`real`)
  - complex numbers (`complex`)
  - truth values (`logical`)
  - variable length character strings (`character(:)`)
- Each of these built-in types may be declared as multi-dimensional arrays

Precision of built-in numeric types

- Fortran standard does not specify the precision of numeric types, i.e. how many bits are used for representing the number
  - Default is often 32 bits (`real` has then 7 significant digits)
- The numerical precision can be controlled through the `kind` parameter, e.g.

```
use iso_fortran_env, only : REAL64, INT16
...
real(kind=REAL64) :: double_precision_number
integer(kind=INT16) :: short_integer_number
```

What is a derived data type?

- Derived data type is a data structure composed of built-in data types and possibly other derived data types
  - Equivalent to structs in C programming language
- Derived type is defined in the variable declaration section of programming unit
  - Not visible to other programming units
  - Unless defined in a module and used via the use clause, which is most often the preferred way

What is a derived data type?

- For real-world applications, using only intrinsic types is often insufficient
- It is beneficial to group the data together as larger objects
  - Code becomes easier to read and maintain
  - Cleaner interfaces
  - Encapsulation of data
- Variables used in the same context should be grouped together using modules and derived data types

Derived type declaration

- Type declaration

```
type particletype
   real :: x
   real :: y
   real :: z
   integer :: charge
end type particletype
```

- Declaring a variable with a type

```
type(particletype) :: proton, water(300)
```

- Elements are accessed with **%** operator

```
write(*,*) proton % charge
```

Derived type initialization

- Data type initialization

```
water(1) = particletype(0.75, 0.19, 0.0, 1)
water(2) = particletype(0.0, -0.38, 0.0, 8)
water(3) = particletype(-0.75, 0.19, 0.0, 1)

! or elementwise
water(1) % x = 0.75
water(1) % y = 0.19
water(1) % z = 0.0
```

Nested derived types

- Derived types can contain other derived types as components

```
type moleculetype
   type(particletype), allocatable :: atoms(:)
   real :: mu
end type moleculetype
...
type solvent
   type(moleculetype), allocatable :: fluid(:)
   complex :: epsilon
end type solvent
```

- Access as

```
beverage % fluid(1) % atoms(1) % x = 0.75
```

Data structures: memory layout

Array of Structures

```
type point
real :: x, y, z
end type point
type(point), allocatable :: points

allocate(points(N))
```

Structure of Arrays

```
type point
real, allocatable :: x(:)
real, allocatable :: y(:)
real, allocatable :: z(:)
end type point
type(point) :: points

allocate(points%x(N), &
         points%y(N), &
         points%z(N))
```

Data structures: memory layout

Array of Structures

```
integer :: i, j
real :: dist(4,4)
do i = 1, 4
  do j = i, 4
dist(i,j) = sqrt( &
  (points(i)%x-points(j)%x)**2 +
  (points(i)%y-points(j)%y)**2 +
  (points(i)%z-points(j)%z)**2)
  end do
end do
```

Memory layout

Structure of Arrays

```
integer :: i, j
real :: dist(4,4)
do i = 1, 4
  do j = i, 4
dist(i,j) = sqrt( &
  (points%x(i)-points%x(j))**2 +
  (points%y(i)-points%y(j))**2 +
  (points%z(i)-points%z(j))**2)
  end do
end do
```

Memory layout

Summary

- Derived data types enables grouping of data to form logical objects
- A Fortran program becomes more readable and modular with sensible use of derived data types
- Handling of complex data structures such as linked lists or binary trees becomes more manageable with use of derived types
- Enables the use of object oriented programming concepts

Some further useful features in Fortran

Outline

- Generic procedures
- Environment variables
- Executing commands

Generic procedures

- Procedures which perform similar actions but for different data types can be defined as *generic procedures*
- Procedures are called using the *generic name* and compiler uses the correct procedure based on the argument number, type and dimensions
  - Generic name is defined in an `interface` section
- Can improve code readability and maintainability
  - Overuse can have the opposite effect

Generic procedures example

```
module swapmod
  implicit none
  interface swap
    module procedure swap_real, swap_char
  end interface
contains
  subroutine swap_real(a, b)
    real, intent(inout) :: a, b
    real :: temp
    temp = a; a = b; b = temp
  end subroutine
  subroutine swap_char(a, b)
    character, intent(inout) :: a, b
    character :: temp
    temp = a; a = b; b = temp
  end subroutine
end module swapmod
```

```
program switch
  use swapmod
  implicit none
  character :: n,s
  real :: x,y
  n = 'j'
  s = 's'
  x=10
  y=20
  print *,x,y
  print *,n,s
  call swap(n,s)
  call swap(x,y)
  print *,x,y
  print *,n,s
end program
```

Overloading operators

- Procedures can also be assigned to operators
  - For example *, +, -, ==, <, >
  - Own operator names can be used too, for example `.dot.`
- Especially useful together with user-defined datatypes

Overloading operators

- Instead of a procedure name, use `operator` or `assignment` keyword
  - Operator: 1 or 2 input arguments with `intent(in)`
  - Assignment: 2 input arguments with `intent(inout)` or `intent(out)` and `intent(in)`

```
interface operator(+)
  module procedure base_sum
end interface operator

function base_sum(x,y) result(z)
  type(base), intent(in) :: x,y
  type(base) :: z
  z % val = x % val + y % val
end function
```

Operator overloading example

```
module operator_demo
  implicit none
  type pair
    real :: a, b
  end type
  interface operator (<)
    module procedure is_smaller
  end interface
contains
  logical function is_smaller(x, y)
    type(pair), intent(inout) :: x, y
    if (x%a < y%a .and. x%b < y%b) then
      is_smaller = .true.
    else
      is_smaller = .false.
    end if
  end subroutine
end module swapmod
```

```
program switch
  use operator_demo
  implicit none
  type(pair) :: p1, p2

  p1%a = 1.0; p1%b = 2.0
  p2%a = 3.0; p2%b = 4.0

  if (p1 < p2) then
   print *, 'p1 is smaller'
  else
   print *, 'p1 not smaller'
  end

end program
```

Environment variables

- Besides command line arguments, environment variables are a common way to modify program behaviour
- Fortran has a standardized method for accessing values of environment variables

Environment variables

- Besides command line arguments, environment variables are a common way to modify program behaviour

```
character(len=256) :: enval
integer:: len,stat
! extract hostname
call get_environment_variable('HOSTNAME',enval,len,stat)
if (stat == 0) write (*,'(a,a)') 'host=', enval(1:len)

! extract user
call get_environment_variable('USER',enval,len,stat)
if (stat == 0) write (*,'(a,a)') 'user=', enval(1:len)
```

Executing commands

- Invoking external programs from within a program would be useful occasionally
  - No source nor library API available for a useful program
  - Perl/Python/etc parsing scripts
- Fortran (2008) has a standardized method for invoking an external command

Executing commands

- Execute a command line
  `call execute_command_line(command[,wait][,exitstat] [,cmdstat][,cmdmsg])`
  - **`command`** is a character string containing the command to be invoked
  - **`wait`** is logical value indicating if command termination is to be waited (**`.true.`**, the default) or if the command is to be executed asynchronously (**`.false.`**) (optional)
  - **`exitstat`** is an integer value containing the return value of the command if **`wait=.true.`** (optional)
  - **`cmdstat`** is an integer value. It is assigned a value of zero if **`command`** executed successfully. For other return codes, see docs (optional)
  - **`cmdmsg`** is a character string containing explanatory message for positive values of **`cmdstat`** (optional)

Executing commands: example

```
program execcommand
  implicit none
  integer :: estat, cstat

  ! execute a unix command
  call execute_command_line('ls -al', .true., estat, cstat)
  if (estat==0) write (*,'(a)') 'command completed successfully'
end program execcommand
```

Summary

- Generic procedures
  - Same procedure name for multiple argument types
- Operator overloading
  - Can use operators for user-defined types
- Reading environment variables
- Executing system commands

Advanced features in Fortran

Outline

- Language interoperability
- Object oriented programming
- Fortran coarrays

Language interoperability issues

- The problem of language interoperability has been present for a long time
  - Before Fortran 2003 the standard did not specify the calling convention in detail
- Interoperability is becoming even more important than before
  - Steering of computation with scripting languages
  - Utilizing libraries that are written using other programming languages

Before Fortran 2003

- Compilers often, but not always
  - Passed all procedure arguments by-reference (i.e. by-address)
  - Referred to functions and subroutines in lowercase letters and by adding an additional underscore after, e.g. **Func** becomes **func_** to be found by the linker
  - Passed function return values via stack
  - Passed **character** strings by-reference, with an additional hidden length argument, passed by value.
    - Note: Fortran **character** strings are not null-character terminated

Portability

- The traditional way to have interoperability with C requires *a priori* knowledge of lowercase and underscore policy used by the compiler
- Complex cases, such as passing **character** strings or passing arguments by value, are generally very error prone and may lead to catastrophic errors at runtime
- Often a separate layer of C functions was needed for interoperability

The iso_c_binding module

- Fortran 2003 intrinsic module **iso_c_binding** is used with

  use, intrinsic :: iso_c_binding
- Module contains
  - Access to named constants that represent **kind** type parameters of data representations compatible with C types
  - The derived types **c_ptr** and **c_funptr** corresponding to C pointer and C function pointer types, respectively
  - Useful procedures: **c_loc**, **c_funloc**, **c_f_pointer**, **c_associated**, **c_f_funpointer**, **c_sizeof** (f08)

Interoperability with Fortran 2003: example

```
function call_c_func(data, nx, ny) result(stat)
    use, intrinsic :: ISO_C_BINDING
    implicit none
    real(dp), dimension(:,:), intent(in) :: data
    integer, intent(in) :: nx, ny
    integer :: stat

    interface
       ! C-definition is:  int func(double *data, const int nx, const int ny)
       function c_func(data, nx, ny) bind(C,name="func") result(stat)
         use, intrinsic :: ISO_C_BINDING
         implicit none
         real(kind=C_DOUBLE) :: data(*)
         integer(kind=C_INT), value, intent(IN) :: nx, ny
         integer(kind=C_INT) :: stat
       end function c_func
    end interface

    stat = c_func(data, nx, ny)

function call_c_func
```

Object oriented programming

- Object Oriented Programming (OOP) is programming paradigm where
  - data and functionality are wrapped inside of an "object"
  - objects provide methods which operate on (the data of) the object
  - method is a procedure that is tied to the data of an object
  - objects can be inherited from other "parent" objects
- Object oriented programming in Fortran facilitated with type extensions and type- and object-bound procedures

Fortran coarrays

- Parallel processing as part of Fortran language standard
  - Only small changes required to convert existing Fortran code to support a robust and potentially efficient parallelism
- A Partitioned Global Address Space (PGAS) paradigm
  - Parallelism implemented over “distributed shared memory”
- Integrated into Fortran 2008 standard
  - Compiler support is still incomplete (Cray: excellent, Intel: moderate, GNU: experimental)

Further advance features

- Abstract interfaces and procedure pointers
  - Declaring several procedures with the same interface
- Asynchronous I/O
- Submodules
  - Implementation of module can be split into multiple files

Summary

- Fortran is a modern language that continues to evolve
  - Support for object-oriented programming
  - Complex, generic data structures can be implemented in modern Fortran
  - Coarrays provide parallel programming support in the language itself
- Fortran remains as one of the major languages for scientific computation

Sami Ilvonen
Mika Jalava
Joonas Nättilä
Seija Sirkiä

INTRODUCTION TO C/C++

Introduction to C/C++ – overview

Wednesday

9.00-9.45	Getting started with C
9.45-10.15	Exercises
10.15-10.30	Coffee break
10.30-11.15	Control structures and loops
11.15-12.00	Exercises
12.00-13.00	Lunch break
13.00-13.30	Data structures and functions
13.30-14.15	Exercises
14.15-14.30	Coffee break
14.30-15.00	Dynamic arrays
15.00-15.30	Exercises

Thursday

9.00-9.30	Preprocessing, libraries
9.30-10.15	Exercises
10.15-10.30	Coffee break
10.30-11.00	File I/O
11.00-11.45	Exercises
11.45-12.00	Q&A
12.00-13.00	Lunch break
13.00-13.45	Towards C++
13.45-14.15	Exercises
14.15-14.30	Coffee break
14.30-15.15	Standard template library
15.15-15.30	Exercises

CSC

Introduction to C

What is C?

- C is a *general-purpose* programming language initially developed by Dennis Ritchie between 1969 and 1973 at AT&T Bell Labs
- It is an imperative procedural language intended for system software
  - Strong ties with UNIX operating system
- It has influenced many other programming languages
  - C++, C#, ObjC, Java, JavaScript, Go, csh, ...

Why C?

- It's popular and widely used
  - Available on almost all platforms
  - Lots of libraries for different tasks
- Provides a relatively low-level (or mid-level) access to the operating system and hardware
  - System-level programming, embedded systems
  - Can lead to better performance

Standards

- First standard by ANSI in 1989, known as "ANSI C" or C89
  - Still the best choice when portability is important!
  - ISO adopted the same standard in 1990 (C90)
- Next revision in 1999, C99
  - New datatypes, complex numbers, variable length arrays,...
- Current standard is C11
  - Improved Unicode support, atomic operations, multi-threading, bounds-checked functions, ...

On this course we will use mainly ANSI C/C90 or C99

GETTING STARTED

Look and feel

```c
/* Computing the factorial of an specified (by the user) number */
#include <stdio.h>
int fact(int n);
int main(void){
   int current;
   printf("Enter some POSITIVE integer (non-positive to finish): ");
   scanf("%d", &current);
   while (current > 0) {
      printf("Factorial of %d is %d\n", current, fact(current));
      printf("Enter a POSITIVE integer (non-positive to finish): ");
      scanf("%d", &current);
   }
}

/* This function returns the factorial of the specified integer (n) */
int fact(int n) {
 ...
```

Basic syntax

- Code is case sensitive
- Statements terminate with **;**
- {} enclose blocks
- There are two ways to comment :

```c
/* Single or
Multiple lines */

// Single line
```

```c
/* example function */
float foo(int bar) {
   int i, c=0;
   float x;

   x = 0.1;
   for (i=0; i<bar; i++) {
      x += i*3.14 + c;
      c = i + 2;
   }
   return x;
}
```

Basic syntax

- Free format
  - whitespace, newlines, layout etc. do not matter ...

to the computer !

```
/* same example function? */
float
foo(/*float f,*/int
bar) { int   i,c=0; float x;
x=0.1;for  (/*i=10
; i<2*bar*/  i=
0;i<bar;i++){x+=i*3.1/*1.2+
     */4+c;c=i+2;}return x;}
```

```
/* example function */
float foo(int bar) {
   int i, c=0;
   float x;

   x = 0.1;
   for (i=0; i<bar; i++) {
      x += i*3.14 + c;
      c = i + 2;
   }
   return x;
}
```

Simple Input / Output (I/O)

- stdio-library is almost always needed. To include it use:

  `#include <stdio.h>`

- For printing a text on the screen:

  `printf(<template> {, <variables>})`

  `<template>` arbitrary string with optional placeholders for data from variables

  `<variables>` (optional) a list of variables

Basic I/O

- Special characters that can be used in the template string:

  `\n` newline **`\t`** horizontal tab

  `\”` double quote **`\\`** literal backslash

- Placeholders are marked with **%** followed by formatting and type information
  - For now, we will just use the following formattings:

    `%d` integer value

    `%f` float value

    `%s` string

Basic I/O example

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("The answer is %d.\n", 42);
    printf("Pi equals to %.2f", 3.14159265);
    printf(" ...at least to the %dnd decimal.\n", 2);
    return 0;
}

//output:
// The answer is 42.
// Pi equals to 3.14 ...at least to the 2nd decimal.
```

The transition from Code to program

- Compiling:
  - Transforming the C source code to machine language
- Linker:
  - Combines object files generated by the compiler into a single executable program
- The result is an executable binary file

Let's try it out!

- Write the classic first program in C:

```
#include <stdio.h>

void main () {
    printf("Hello world!\n");
}
```

- Compile the code:

```
$ gcc -o hello hello.c
```

- Test your program:

```
$ ./hello
Hello world!
```

DATATYPES, VARIABLES, ASSIGNMENT AND OPERATORS

Basic data types

- Basic datatypes in C are:

  `char` character

  `int` integer number

  `long` long integer number

  `float` floating point number

  `double` double precision float

- C has also *pointer* types
- There is also a special type **`void`**

Variables

- Data types of the variables are static
- Declare variables before using them
- Valid variable names in C:
  [a-z] [A-Z] [0-9] _
- Scope of a variable is always local unless otherwise declared

```
// variable declaration
int i;
float f, g;
double total=1.9;

// valid names
int i3, myINT, I_o;
float o3Gf_ry9;
// invalid names
int 33, 9a, i-o;
float o3.Gf;

// data type matters
char c;
float f;

f = 1.234;
c = f;     // ERROR!
           // wrong data type
```

Variable assignment

- Assign a value to a variable:
  `variable = value;`
- Both should have the same data type
  - implicit conversion between compatible types
  - explicit conversion (typecast) syntax: **`(type) var`**

```
// examples of assignment
count = 10;
i = j = 0;
k = i*j + 1;

// assign at declaration
int i = 4;

// typecast from int to float
int i;
float f;

i = 5 * 21;
f = (float) i;

// watch out for operator order
(float) i/5 != (float) (i/5)
```

Arithmetics

- Operators:

+	addition
–	subtraction
*****	multiplication
/	division
%	modulus

- Operator precedence
  1) **% * /**
  2) **+ -**
- Parentheses group operations () and change evaluation order

Compound assignment

- C has a short-hand notation for combined arithmetic operation and assignment

a += b	equals	a = a + b
a -= b		a = a - b
a *= b		a = a * b
a /= b		a = a / b

- For special cases **a+=1** and **a-=1** there are special operators **++** and **--**

```
// example of compound assignment
int count = 10;
count += 5;  // now count is 15
count /= 3;  // and now count is 5

// Adding one to count
count++;

// This is also valid!
++count;
```

Arithmetics and assignment examples

```
// addition, substraction
i = 5 + 2;
i = 5 – 2;
i += 1;      // i = i + 1
f -= 3.4;
i++;         // i = i + 1
i--;

// multiplication, division
i = 5 * 2;
i = 5 / 2;      // integer division
f *= 3.0 / 4.2;

// modulus
m = 25 % 3;
```

```
// grouping with ()s
b = a * (1.3 + (25%3));

// watch out!
f = 1 + q / (1 - q);
f = (1 + q) / (1 - q);

// also functions use ()
f = r * sin(theta);
f = (r + p)*sin(theta);
```

Pointers

- A variable is actually a reference to a memory location
- The contents of the memory block determine the value of a variable
- Memory addresses can be stored (and manipulated) in pointers
- Pointers are one of the key features (and one of the most difficult) of the C language!

Pointers

- Pointers have a type (**`int, char`** etc.)
- Stores a memory address containing data of corresponding type

Pointers

- The address of a variable is obtained with **&**
- The value pointed at by a pointer (the contents of the address) can be accessed by de-referencing the pointer with *****

```
int *p, a;

a = 5;
// get address of a
p = &a;
// assign to a the value pointed at by *p plus 2
a = (*p) + 2; // a=7
```

Summary

- Short history and different standards
- Basic syntax
- Variables and their type, assignment of values
- Arithmetic operations

CONTROL STRUCTURES AND LOOPS

Control structures // if – else

```
if (i < 100) {
    data[i] = x[i] + c
} else {
    data[i] = x[i] - c
}
```

- `if (conditional) {TRUE} else {FALSE}`
- test operators:

==	equal to	>=	greater or equal
!=	not equal to		
&&	AND		
||	OR		
<	less than		
>	greater than		
<=	less or equal		

- negation: !
- non-zero value == TRUE

Control structures // if – else

```
// simple if-else
if (x > 1.2) {
   y = 4*x*r;    // TRUE
} else {
   y = 0.0;      // FALSE
}

// else is optional
if (x || y) {
   z += x + y;
}
```

```
// complex if-elseif-else
if ((x > 1.2) && (i != 0) )
{
   y = x / i;    // 1st TRUE
} else if (x < 1.0 ) {
   y = -x;       // 1st FALSE
                 // 2nd TRUE
} else {
   y = 0.0;      // 1st, 2nd FALSE
}
```

Control structures // switch

- `switch (...) {` — conditional expression
  `case (...): ...` — test value
  `break;` — end of branch
  `default: ... }` — `default` branch executed if nothing else matches
- condition:
  - single variable OR complex expression
- branch(=case) with matching value chosen
- `break` stops branch

Control structures // switch

```
// simple switch based on the
// value of integer i
switch (i) {
   case 1:
      printf("one\n");
      break;
   case 2:
      printf("two\n");
      break;
   default:
      printf("many\n");
      break;
}
```

good style to break even the last branch!

Control structures // switch

A

```
switch (i) {
   case 1:
      printf("one\n");
      break;
   case 2:
      printf("two\n");
      break;
   default:
      printf("many\n");
}
```

B

```
switch (i) {
   default:
      printf("many\n");
   case 1:
      printf("one\n");
      break;
   case 2:
      printf("two\n");
      break;
}
```

C

```
switch (i) {
   case 1:
      printf("one\n");
   case 2:
      printf("two\n");
   default:
      printf("many\n");
}
```

In each case, what would be printed on the screen if `i` is 1, 2, or 3?

Loops // while

```
// loop i = 0..99 with an increment of one
i = 0
while (i<100) {
    data[i] = x[i] + c
    i++;
}
```

- `while (...) {...}`
  condition — code block
- Code block executes repeatedly as long as condition is TRUE
- Condition executed *before* iteration
- `do {...} while (...)`
  - Block executed at least once

Loops // for

```
// loop i = 0..99 with an increment of one
for (i=0; i<100; i++) {
    data[i] = x[i] + c
}
```

- **for (<init>; <condition>; <after>) {...}**

<init>	execute once before entering the loop
<condition>	stop loop when condition is FALSE
<after>	execute after each iteration

Loops // for

```
// loop using a for-statement
//   i is incremented after each iteration
for (i=0; i<bar; i++) {
   x += i*3.14 + c;
   c = i + 2;
}

// the same loop but with a while-statement
i=0;
while (i < bar) {
   x += i*3.14 + c;
   c = i + 2;
   i++;
}
```

Jump statements

- **break**
  - end a loop (for, while, do) or a switch statement
- **continue**
  - continue with the next iteration of a loop

```
// jump out of a loop
for (i=0; i<10; i++) {
   printf("in loop");
   if (i == x)
      break;
}

// jump to the next iteration
for (i=0; i<10; i++) {
   if (i < x)
      continue;
   printf("in loop");
}
```

Summary

- Conditional branching: if-else, switch
- Looping as long as condition is true: while, do – while
- Fixed number of loop passes: for
- Jump statements: continue, break

DATA STRUCTURES AND FUNCTIONS

Arrays

- Static arrays declared as **<type> name[N]** where **N** is the size of the array

int i[10]	array of 10 integers
float f[3]	array of 3 floats
char c[60]	string of 60 letters

- elements indexed starting from zero

i[3]	4th element of array **i**

- multi-dimensional arrays possible

int m[10][20]	a 10x20 matrix

Arrays

```
// integer vector
int triplet[3];

triplet[0] = 1;
triplet[1] = 2 * triplet[0];
triplet[2] =
   triplet[0] + triplet[1];

// string
char word[] = "hello!";
printf(word); // hello!
```

```
// 2-dim matrix
int i,j;
float matrix[5][5];

for (i=0; i<5; i++)
   for (j=0; j<5; j++)
      matrix[i][j] = i*j;

// print the 2nd row
for (i=0; i<5; i++)
   printf("%f ", matrix[1][i]);
```

Pointer arithmetic and arrays

- Pointers can be added to or subtracted from
  - This shifts the pointer to a new memory address before or after the current one
  - Each step is **sizeof(pointer_datatype)** bytes, i.e. the memory size of the pointer's data type

```
float vector[10], *ptr;

ptr = &vector[0];   // *ptr -> vector[0]
ptr = vector;       // name of array works as pointer to the beginning
ptr++;              // *ptr -> vector[1]
ptr += 3;           // *ptr -> vector[4]
ptr--;              // *ptr -> vector[3]
```

- Watch out for out-of-bounds memory errors!

Structures

- Multiple variables of arbitrary type can be grouped together in a **struct**
- Member **y** of struct **x** accessed with **x.y**
  (or **x->y** if x is a pointer)

```
struct s {
    int i;
    float f;
} one;

typedef struct {
    int count;
    float price;
} apple;
apple a;

a.count = 7;
a.price = 1.24;
cost = a.count * a.price;

// with pointers
apple *ptr;
ptr = &a;
c = ptr->count;
ptr->price = 0.89;
```

Functions

- Functions are the only subroutine type in C
  - But functions do not have to return anything
- Function definitions are of form

```
type func-name(param-list)
{
   /* body of function */
}
```

- Here **type** is the return type of the function
  - **void** means that function does not return anything

main function

- Every C program has the main function with definition

```
int main(int argc, char *argv[])
{
   /* body of function */
}
```

- Command line arguments are passed to the program using **argc** and **argv**
- main should always return integer
  - zero means success, non-zero values are errors

Function example

- This function returns the sum of two integer arguments:

```
#include <stdio.h>
int add(int a, int b) {
   return a + b;
}

int main(void) {
   int val;
   val = add(3, 6);
   printf("Sum is %i\n", val);
   return 0;
}
```

Variable scoping

- Variable scoping in C is local unless defined otherwise
  - Variables declared in the function definition are only visible inside the function body
  - Variables of calling scope are not visible inside function body

```
void funcB(float a) {
   int counter;
   ...
}
int funcA(void) {
   float value_1, value2;
   funcB(value_1);
   ...
}
```

Not accessible from **funcA**

Not accessible from **funcB**

Arguments

- All arguments are passed as *values*
  - Copy of the value is passed to the function
  - Functions can not change the value of a variable in the *calling scope*
- Pointers can be used to pass a reference to a variable as an argument!

```
void funcB(int a) {
   a += 10;
}

int funcA(void) {
   int var = 0;
   funcB(var);
   // var is still 0!
   ...
}
```

Functions with pointers

- Pointers allow "returning" of multiple values

```
// passing by reference
void add_pi(float *a, int *b) {
   float pi=3.14;
   *a += pi;
   (*b)++;
}
float a=1.0;
int b=1;

add_pi(&a, &b); // a=4.14  b=2
```

DYNAMIC ARRAYS

Dynamic memory management

- In most cases the exact size of all data structures is not known at compile time because they depend on the input data
- Dynamic memory management is accomplished by using pointers and manually managing memory (de)allocation
- Relevant functions are **malloc()**, **realloc()**, **free()**

malloc

- malloc function is defined in header file `stdlib.h`:
  `void *malloc(size_t size);`
- **malloc** returns a pointer of type **void** to a memory location with allocated space of size **size** bytes
  - If allocation fails, **malloc** returns **NULL!**
- The memory area returned by the malloc is uninitialized!

Dynamic arrays

- **malloc** can be used to allocate space for arrays
  - **malloc** returns a pointer to the beginning of the array
  - Elements can be accessed with normal array syntax
- The size of memory allocation can be determined using **sizeof** operator which returns the size of the argument in bytes
- Example for a dynamic array of five integers:

```
int *ptr = (int *) malloc(5 * sizeof(int));
```

free

- **free** deallocates previous allocated object
  `void free(void *ptr);`
- After freeing you should not try to access any part of the allocation
- Calling free with a pointer that does not point to a allocated memory can crash the code
  - Calling free twice is a common mistake!

Dynamic arrays

```
int n_elems = 32;
float *prices;

// allocate memory for the required amount of floats
prices = malloc(n_elems*sizeof(float));
for (i=0; i<n_elems; i++) {
    prices[i] = i*1.23;
}
// add space for one more float
prices = realloc(prices, sizeof(float)*(n_elems+1));
prices[n_elems] = 0.91;
// de-allocate the memory block
free(prices);
```

Pointers to pointers

- It is possible to have pointer references to pointers
  - This is very useful when functions have to manipulate values of arguments of pointer type
  - Multidimensional arrays are also naturally mapped into pointers of pointers

Pointer to a pointer example

```
#include<stdio.h>

int main(void) {
    int a = 5;
    int *int_ptr;
    int **int_ptr_ptr;

    int_ptr_ptr = &int_ptr;
    int_ptr = &a;

    printf("a=%i\n", **int_ptr_ptr);
    return 0;
}

// Result: a=5
```

Dynamic multi-dimensional arrays

- Doable, but becomes complicated
- No real multi-dimensional arrays in C, so really just arrays of arrays
  - Two dimensional array maps to a variable that is a pointer to a pointer
- Memory management by hand
  - There are at least two different ways to do the allocation
  - Easy to make mistakes, beware here lieth dragons!
- Often best to just use 1D array and map N dimensional indices

Allocating row-by-row

```
int i;
int rows = 4, cols = 8;
float **matrix;

/* allocate memory */
matrix = malloc(rows * sizeof(float *));
for (i = 0; i < rows; i++)
    matrix[i] = malloc(cols * sizeof(float));

// start using the 2D array
matrix[0][2] = 3.14;
```

Do not do this!

Dynamic multi-dimensional arrays

Dynamic 2D array in *contiguous* memory:

- First, allocate memory for pointers to the first element of each row
- Second, allocate memory for all elements
- Third, point each row at the first element of that row

Dynamic multi-dimensional arrays

```
/* allocate memory */
matrix = malloc(rows * sizeof(float *));
matrix[0] = malloc(rows * cols * sizeof(float));

/* point the beginning of each row at the correct address */
for (i = 0; i < rows; i++)
    matrix[i] = matrix[0] + i * cols;

// start using the 2D array
matrix[0][2] = 3.14;
```

Memory layout

- Allocating space for the whole array using a single malloc call is the recommended way
  - Number of expensive malloc calls is minimized
  - Array is represented as one contiguous block in memory
  - It can be copied without looping over rows
  - Most IO and numerical libraries assume contiguous storage

Dynamic multi-dimensional arrays

```
/* When using contiguous memory */
free(matrix[0]);
free(matrix);
```

- After using a dynamic multi-dimensional array, remember to free each array inside the main array

Using indexing to represent 2D array

- Allocate memory for all elements as normal 1D array
- Compute indices so that the 1D memory area map to the 2D array we want to represent
- Benefits
  - The compiler understands better the code, and hence performance may be better
  - Allocation and deallocation easier
  - Straightforward to generalize to higher dimensions
- Drawbacks
  - Code may look less elegant

Using indexing to represent 2D array

```
/* matrix now normal pointer */
float *matrix;

/* allocate memory */
matrix = malloc(rows * cols * sizeof(float));

/* start using the array, element [i][j] in 2D array now maps
   to [i * cols + j]*/
matrix[0 * cols + 2] = 3.14;

/* free memory */
free(matrix);
```

PREPROCESSING AND LIBRARIES

Preprocessing directives

- C preprocessor is a part of the compiler that does initial text substitution, manipulation, inclusion and other activities before the actual translation is done
  - We have already used **#include**, which includes a file
- C relies heavily on preprocessor to accomplish
  - Portability of code
  - Source code control
  - Debugging

Directives

- Preprocessor directives start with **#**, which has to be first token on a line
- Directives are limited to one line
  - Line can be continued using ****
- Directives are not statements, do not end the line with **;**

```
// These are ok
#define one
  #define two
// Not the first token - WRONG!
int i; #define one
```

#define

- **#define** can be used to define "objects" or macros
- It has the form of
  #define identifier replacement
- After the definition all instances of ***identifier*** are replaced with ***replacement***

```
// Example
#define ONE 1
printf("Value is %i\n", ONE);
// Result: Value is 1
```

Directives

- **#ifdef, #ifndef**
  - checks whether a macro is defined (/ not defined)
  - if check is true, then includes the enclosed code

```
#include <stdio.h>
#include "myfile.h"

#define DEBUG 1
#define PI 3.14159
#define MAX(a,b) (a)>(b)?(a):(b)

#ifdef DEBUG
/* debug mode is enabled */
#endif
```

Definitions on compiler command line

- It is also possible to set preprocessor definitions on the compiler command line
- Most compilers accept option **-D** for this purpose:

```
gcc -DONE=1 -DUSE_FEATURE
```

is equivalent with

```
#define ONE 1
#define USE_FEATURE
```

The transition from Code to program

- Compiling:
  - Transforming the C source code to machine language
- Linker:
  - Combines object files generated by the compiler into a single executable program
- The result is an executable binary file

Working with several files and libraries

- Advantages:
  - Different programmers can work on different files
  - Function definitions can be re-used in other programs
  - Changing a file, only that file needs to be re-compiled to rebuild the program
- UNIX 'make' can be very useful tool for this!
  - Most IDEs also provide a tools for building this kind of compilation

Compilation: working with several files

- We use header files to define functions that we can use later
- Making .h files for your functions allows you to 'include' them in your code

```
// we implement the function 'add'
int add(int first, int second)
{
    return first + second;
}
```

File: add.c

```
// we define the function 'add'
#ifndef ADD_H
#define ADD_H
int add(int first, int second);
#endif
```

File: add.h

This is called **function prototype.** It introduces the return value, name and arguments of the function to the compiler

Compilation: working with several files

- Another .c file can then use the function add() by including the header file:

```
// we can use #include "" or #include <>
#include "add.h"

int example(int x)
{
   return add(x, add(x,x));
}
```

Compilation: working with several files

- So, this is how headers work:
  - main() would be in one file, the others will contain functions
  - Headers usually only contain definitions of data types, function prototypes and C preprocessor commands
  - We include the header into the C files
  - We compile the different files and the compiler calls the header file

The transition from Code to program

- In complex projects:
  - Compile each source code file (.c) into an object file (.o)

```
$ gcc -c main.c
$ gcc -c add.c
```

  - Link object files into a binary and execute the binary

```
$ gcc -o foo main.o add.o
$ ./foo
Modify "add.c"
$ gcc -c add.c
$ gcc -o foo main.o add.o
$ ./foo
```

Libraries

- Libraries contain functions, data types and constants that can be used by multiple programs
- Two components:
  - header: function etc. definitions
  - object code in a library file (shared .so or static .a)
- When creating the executable binary file, an additional flag (`-l`*`library_name`*) is given to the linker to include the library

Libraries

- Math operations

  `#include <math.h>`

  + link with `-lm` flag!
- It includes the most common mathematical functions, e.g. sin(), cos(), log() etc.
- Also includes constants such as M_PI for π
- For power operation function pow() is used (^ is bit-wise XOR in C !)

```
#include <math.h>

int main () {
  float r, theta;
  double area, y;

  // radius and an angle
  r = 1.2;
  theta = 0.456;

  // calculate something
  area = M_PI * pow(r,2);
  y = sin(theta)
    + cos(theta/2.0);
  y += exp(-3.1 * area);

  // echo results
  printf("area is %f\n", area);
  printf("y=%.18e\n", y);
}
```

Linking with libraries

- Add libraries in linking phase

```
$ gcc -c main.c
$ gcc -o foo main.o -lm
$ ./foo
```

Compiler optimization

- Compiler flags provides possibilities of also for code optimization. Few instances of the optimization flags are `-O`, `-O0`, `-O1`, `-O2`, `-O3`, `-Os`.

```
$ gcc -O3 -o toroid -lm toroid.c
$ ./toroid
```

- For further information check the manual of `gcc`:

```
$ man gcc
```

FILE I/O

File I/O

- `fopen(<filename>, <mode>) → FILE*`

name of a file

r, w, a	read, write, append
r+, w+, a+	read+write
rb, wb, ...	binary mode

pointer to file

- `fclose(FILE*)` → closes file
- `fflush(FILE*)` → flush I/O buffer to disk

File I/O

- `stdout, stderr`
  - special output streams for STDOUT and STDERR (stdio.h)
- `fprintf(<file>, <template> {, <variables>})`
  - `<file>` pointer to an open file
  - `<template>` arbitrary string with optional placeholders for data from variables
  - `<variables>` (optional) a list of variables

File I/O

```
#include <stdio.h>

fprintf(stdout, "The answer is %d.\n", 42);
//output (STDOUT):
// The answer is 42.

fprintf(stderr,
   "Pi equals to %.2f (at least to the %dnd decimal).\n",
   3.14159265, 2);
//output (STDERR):
// Pi equals to 3.14 (at least to the 2nd decimal).
```

File I/O

```
#include <stdio.h>

FILE *fp;
int pid=2146;

fp = fopen("mylog", "w");
fprintf(fp, "logging my progress for process %d...", pid);
fprintf(fp, "ok\n");
fclose(fp);

//output (in file 'mylog'):
// logging my progress for process 2146...ok
```

File I/O

- `fgets(char *buffer, int size, FILE *stream)`
  - Reads a single line from file pointer stream
  - Assigns string to buffer
  - The `size` parameter defines the max. size of characters to read.

File I/O

- `fscanf(FILE *stream, char *format, ...)`
  - scan input according to format
    e.g. `"name: %s  age: %d"`
  - store values of placeholders (%s etc.) to pointers

```
// read max 64 characters from FILE *fp
fgets(buffer, 64, fp);

   // read a matching string from FILE *fp, pick a string and
   // an integer and assign them to variables name and age
fscanf(fp, "name: %s  age: %d", &name, &age);
```

Summary

- Basic syntax
- Overall structure: #include, functions, main()
- Data types: int, float, ..., struct
- Control structures: for, while, if/else, switch/case
- Memory management: pointers, malloc(), ...
- I/O: files, stdin, stdout
- Compiling & linking

Web resources

- C Programming
  http://en.wikibooks.org/wiki/C_Programming
- The C Library Reference Guide
  http://www.acm.uiuc.edu/webmonkeys/book/c_guide/
- C syntax
  http://en.wikipedia.org/wiki/C_syntax
- Ten Reasons to Teach and Learn Computer Programming in C
  http://iel.ucdavis.edu/publication/WhyC.html
- International Obfuscated C Code Contest
  http://en.wikipedia.org/wiki/International_Obfuscated_C_Code_Contest

Say what?

```
    static signed char a[] = {0x6D,
  110, 118, 97, 108, 105, 0x64, 1-1,
 0x6d, 111, 118, 101, 1<<1<<1<<1<<1,
 114, 105, 0x6e, 105, 32, 'o'/7, 100,
 32, 102, 114, 111, 0x6d, 32, 115, 116,
 97, 100-001, 107, 32,37, 2*'2', '@'>>1,
 116, '%' + '%', '%','w'-'w',115, 0x74,
 97, 3*'!', 107, '&' - '&', 37, 10*'\n',
 10, 0}, * b = a + (1<<1<<1<<1), * w, x,
 *q, c, r; int main(int d, char *e [])
 {
 return q = (signed char *)(e+1+1), (r =
 e[0] && e[1] ? 0 : 0 * puts (a) + 1) ||
 (r = e[1<<1] && d != 1 <<1 && 0 * puts(
 a) + 1) || e[1- -1] ||  (r = atoi(e[1])
 < -0200 || atoi (e [1]) > 0x7f || ( x =
 atoi( e[1] ) ) == 0 ? 0 * puts(a) + 1 :
 0) || e [1- -1] || (x- -x > 1-1 ? (q[0]
  = x, q[1] = q[0] + 1, q[2] = 2) : (
    memset ( w = ( signed  char * )

    malloc(-x), '1', -x), puts (w),
  q[0] = x, q[1] = '0', q[2] = q[1] =
 0)), r || (q[1] ? (c = 6 - q[1] - q[2],
 (q[0] += 1) ? q[0]-- , d = q[2], q[2] =
 c, main(2, e), c = q[2], q[2] = d, q[0]
 ++ - 0, printf(b, q[0], q[1], q[2]), (q
 [0] += 1) ? q[0]-- , d = q[1],q[1] = c ,
 main(2, e), c = q[1], q[1] = d, q[0] ++
 : 0) :  - 1 - q[0] - 1 == 0 ? (w[- x - 1
 - (q[1] & 1 ^ 1)] = q[1], puts (w), w [
 - x - 1 - (q[1] & 1)] = q[1], puts(w) )
 : - 1 - q[0] == 0 ?  (w[- x - 1] = q[ 1
  ], puts(w)) : (q[0] += 1- ( q[1] & 1
    ^ 1), main(2, e), q[0] -= 1 + ( q
        [1] & 1 ^ 1), q[1] & 1 ? (q
            [0]+=1, q[1]^=1, main
            (2, e), q[1]^=1, q[0]-=1
            =1) : 0, w[q[0] - x] = q
            [1], puts(w), q[1] & 1 ?
          0 : (q[0]+=1+1, q[1]^=1,
        main (2, e), q[1]^=1, q
        [0]-=1+1), q[0] += 1 +
        (q[1] & 1),main(2,e)
  , q[0] -= 1 + (q[1]
 & 1) ) ), r; }
```

© 1993 E. Jay Berkenbilt

```
                                                  char
                                                _314592654[1141
        )__ _3141[3141];_314159[31415]_ _3141[31415];main(){register char *
      _3_141,*_3_1415,*_3_1415; register int _314,_31415,*_31;
    _3_14159,_3_1415;_3142592654=_31415=2,_314592654[0]_314592654
  -1]=2[__3142]=5,__3_1415=6;dw(_3_14159o_314e0,__3_31415=(for(__31415
  =0;_31415<(3_14-4)*__31415;_31415++)_31415[_3142]=_31415[_31415]= -
 1,_3142[*_314159+_3_14159];_314,_3_141=_314592654=_3_1415;_3_1415=
 __3_1415    =__3141;for                    (_31415 = 3141-
            __3_1415                        _31415;_31415-
            ,_3_141 +=,                     _3_1415++){_314
            ++_314<<2 ;                     _314<<=1;_314+=
            *_3_1415;_31                    *=_314159+_314;
            if((*_3141)                     )* _31 = _314 /
            __31415,_314                    [_3142]=_314 %
            __31415 ;*(                     _3_1415+_3_141
           )+= *_3_1415                     _= *_31;while(*
            _3_1415 >>                      31415/3142 ) *=
            _3_1415+= -                     10,(*--_3_1415
           )++;_314+_314                    [_3142]; if ( !
            _3_14159 && *                   _3_1415)_3_14159
            *1,__3_1415 =                   3141-_31415;;if(
            _314+(__31415                   >>1);=__31415 )
            while { *= *                    _3_141=3142/314
           )*_3_141--=0                     ;;while(_3_14159
            ) ; { char *                    _3_14e *3_14159";
            write(3,3),                     (--*__3_14,__3_14
            ),(_3_14159                     ++,*=_3_14159)!=
           3.1415926; }                     for ( __31415 = 1;
           31415;3141-                      1;_31415++)write(
         314159 314-(                       3_14),_3141592654[
        _31415     ) +                      "0123456789",*314*
        [ 3]+3__314;                        puts(("_3141592654+0
      _3142592654))                         ;_314+ *"3.141592";}
```

© 1989 Roemer B. Lievaart

C S C

Introduction to C++

INTRODUCTION TO C++

What is C++?

- Originally started as an extension of C with classes and improvements on type system
- AT&T labs, Bjarne Stroustrup
- Should be considered as completely different language than C
  - C95 is not a strict subset, but very close
  - One can pick some features and have "improved" C

Evolution of C++

- C++ has evolved heavily during last 20 years
  - Modern C++ looks quite different compared to the first standard, C++98
  - Changes in C are very modest in comparison
- C++11 standard brought many fundamental changes and big improvements

Compiling

- Separate language means that C compiler can not compile C++ programs
- Usually the compiler is called either c++ or CC
  - Gnu Compiler Collection C++ compiler is g++
- C++ source code files have usually on of the following suffixes:
  - `.hpp`, `.h`, `.hxx` for headers and
  - `.cpp`, `.cxx`, `.CC` for source

Control structures

- Similarity with C is most prominent when comparing the control structures and syntax
  - For example, `if`, `while`, `for`, `switch-case` are same
    - There is also a new version of for-loops in C++11
  - Normal function declarations and implementations are the same
  - Pointers are still there if you have to use them
    - One useful new features is the concept of **reference**

References

- Remember C: function arguments are always passed by value → modifying argument values in the calling scope is only possible with pointers
  - Can be cumbersome, especially with structures
- C++ adds the concept of references
  - Programmer can specify in the function declaration that an argument is passed by reference, not by value
    - That is, modifying the variable modifies the value of the calling scope too

Example of reference

```
#include <cstdio>
void func(int &a) {
  a++;
}

int main(void) {
  int val = 10;
  printf("Value is %i\n", val);
  func(val);
  printf("Value is %i\n", val);
  return 0;
}
// Result:
// Value is 10
// Value is 11
```

OBJECTS

Object-oriented programming

- Object oriented programming is one of the major programming paradigms
  - Main idea: grouping of data and the program code that manipulates the data (methods) into same unit (object)
- C++ supports class-based object oriented programming
  - Objects are instances of classes
  - Class definitions have descriptions of data fields and methods

Objects and C++

- Natural language construct for OO extensions is the **struct**
  - Collection of variables, that is, data of different types
  - In C++ is just a special type of an object declaration
- Bundle the code that manipulates the variables into the same definition

Class declaration example

Type declaration

```
// In c++ struct is a class with
// public members
struct my_struct {
  double a;
  int b;
};

class my_class {
public:
  double a;
  int b;
};
```

Variable declaration

```
// In C:
struct my_struct my_variable;

// c++
my_struct my_variable;
my_class my_object;
```

Members

- Variables and functions that are defined inside a class are called *members* of that class
  - Member functions are often called methods in OO programming context
- There are couple of special member functions, *constructor* and *destructor*
  - Constructor is called when object is created
  - Destructor is called when object is freed
  - If user does not specify these, default constructor and destructor will be used

Member functions

```
#include <cstdio>

class my_class {
public:
  double d;
  int a;
  void print_a() const { printf("a is %i\n", a); };
};

int main(void) {
  my_class c;
  c.a = 10;
  c.print_a();
  return 0;
}
```

Member visibility

- Visibility of members can be controlled using labels in class declaration
  - public: visible to everywhere where instance is visible
  - private: visible only to class methods

```
class my_class {
private:
  double d;
  int a;
public:
  void print_a() const {
    printf("a is %i\n", a);
  };
};
```

Scope resolution operator

- Adding large function bodies within the class declarations is impractical
  - Remember C: structure definitions are often added into header files so that the declaration can be included where ever needed
- Member function declarations and implementations can be separated from the class declaration body using *scope resolution operator* ::

Example

```
class my_class {
public:
  double d;
  int a;
  void print_a() const;
};

void my_class::print_a() co
  printf("a is %i\n", a
}
```

Note the scoping: member variable a from the class declaration is visible to the member function body

Operator overloading

- Member functions can also be defined for operators
  - When used correctly, can improve code readability
  - Can also make code very difficult to read if not used in a proper way
- Most common operators can be overloaded:
  - +, -, *, /
  - [ ], << (streams)

Operator overloading example

```
#include <cstdio>
class my_class {
public:
  int a;
  const int operator[](const int &b) const {
     return a+b;
  };
};

int main(void) {
  my_class val;
  val.a = 5;
  printf("Value of val[10]: %i\n", val[10]);
  return 0;
}
```

IOSTREAMS

Streams

- C++ introduced new IO system that is based on streams
- Streams provide a portable and unified IO access for different platforms and IO devices
- Two operators: << and >>

Using streams

- Steps needed when performing input and output with streams:
  1. Construct a stream object
  2. Connect the stream object to actual IO device (keyboard, file, etc.)
  3. Perform IO operations on the stream using functions that are defined in the streams interface
  4. Disconnect the stream and IO device
  5. Free the stream object

Example: writing file using streams

```
#include <fstream>   // File streams
#include <iostream>  // Standard io streams
int main(void) {
    // Writing text to the standard output
    std::cout << "Writing out a demo.txt file\n");
    // Output file stream object
    std::ofstream ofile;
    // Open file for writing, overwrite existing files
    ofile.open("demo.txt", std::ios_base::out | std::ios_base::trunc);
    // Write one line
    ofile << "This is a demo text\n";
    // Close file
    ofile.close()
    return 0;
}
```

Summary

- C++ is a modern multi-paradigm programming language
- Started as an improved C
  - Should be considered as a separate programming language though it can still be used as improved C
- Supports object-oriented programming

C++ Standard Template Library

C++ STANDARD TEMPLATE LIBRARY

Generic code

- Many times you would like to (and should!) write more generic code
- Strongly typed languages like C do not allow this

```
int my_add(int a, int b) {
    return a+b;
}

double my_add(double a, double b) {
    return a+b;
}
```

Templates

- Templates are C++'s solution to this
- Also known as parameterized types
- Templates can be extremely powerful constructs

```
int my_add_int(int a, int b) {
    return a+b;
}

double my_add(double a, double b) {
    return a+b;
}
```

```
template<typename T>
T my_add(T a, T b) {
    return a+b;
}
```

Templates

- Defined with `template<typename T>` as prefix
- For usage template functions need to be specialized with `my_func<TYPE>`
- Class templates are also supported

```
template<typename T>
T my_add(T a, T b) {
    return a+b;
}
// …

int a=1;
int b=2;

double c=1.0;
double d=2.0;

int res1    = my_add<int>(a,b);
double res2 = my_add<double>(c,d);
```

Templates under the hood

source code | template definition | compiler output

```
my_add<int>
```

```
my_add<double>
```

```
template<typename T>
T my_add(T a, T b) {
    return a+b;
}
```

```
int my_add(int a, int b) {
    return a+b;
}
```

```
double my_add(double a,
              double b) {
    return a+b;
}
```

Standard Template Library (STL)

- C++ library of ready-made code
  - containers, algorithms, iterators,…
- Components are are heavily parameterized
  - Generic programming with templates
- You gain more generic code, faster to write, easier to understand, (often) more efficient,…

std::vector

- Dynamic & generic container
  - think C-style dynamic arrays
- `std::vector<T> vec;`
- works on any type!
- no need to think of memory
  - done by compiler

```
#include <vector>

//…

std::vector<int> vecA;
std::vector<float> vecB;
std::vector<myOwnClass> vecC;

//…

int   A0 = vecA[0];
float B5 = vecB[5];
myOwnClass& C0 = vecC[0];
```

std::vector initialization

- Vectors are dynamic in size
  - program will expand them when needed
- Initialization is flexible
  - empty vector
  - zero-initialization with predefined size
  - pre-initialization with a list

```
//empty vector
std::vector<int> vecA;

//zero-initialized (5 elements)
std::vector<int> vecB(5);

// pre-initialized
std::vector<int> vecC = {0,1,2,3};
```

std::vector usage

- Vector (like most containers) have a standard interface
  - `v.size()` - current size
  - `v.empty()` - is it empty?
  - `v.resize()` - change size (inserts/deletes elements!)
  - `v.begin()` - iterator to first element (more later on…)
  - `v.end()` - iterator to last element
  - `v.push_back(e)` - append new element `e` to the end
  - `v.insert(i,e)` - insert new element `e` to location `i`

std::vector basic usage

```
#include <iostream>
#include <vector>

int main() {

    //initialize empty vector
    std::vector<int> vec;

    //append elements into vec
    vec.push_back(101);
    vec.push_back(102);
    vec.push_back(103);

    std::cout << "size of vec is: << vec.size() << '\n';

    for(int i=0; i<vec.size(); i++) {
        std::cout << vec[i] << '\n';
    }
}
```

std::array

- Static, i.e., fixed size
  - think C-style fixed arrays
- generic container
- `std::array<T,D> arr;`

```
#include <array>

//…

std::array<int,100> arrA;
std::array<float,5> arrB = {1.0, 2.0, 3.0, 4.0, 5.0}
std::array<myOwnClass,5> arrC;
```

Iterators

- A general object to traverse containers
- Resembles C-style pointers a bit
  - they point to an element in container
  - but also have information of the location
- type of the iterator is always fixed to a type of container
  - `std::vector<T>::iterator`
- also `const_iterator, reverse_iterator,…`

Iterators

- Value is obtained as `T x = *it;`
- location can be shifted by changing the iterator itself
  - `it++` (move forward one element)
  - `it--` (move backwards one element)
  - `it+=5` (move forward 5 elements)

Iterators in practise

- In practise they are quite simple & familiar from C

```
#include <iostream>
#include <vector>
#include <iterator>
…
std::vector<int> vec = {0,1,2,3,4};

for(std::vector<int>::iterator it=vec.begin();
    it != vec.end();
    it++) {
        std::cout << *it[i] << '\n';
}
```

- Modern C++ makes the usage even more succinct

```
for(auto it=vec.begin(); it != vec.end(); it++) {
        std::cout << *it[i] << '\n';
}
```

Algorithms library

- A library of algorithms for often used functions
  - often paired together with containers
- `min/max` (for comparison of two values)
- `min_elem/max_elem` (for arrays)
- `count`
- `find`
- `sort`

std::find

- Returns a first element in the range [first, last) that is equal to some given value
- Relies on iterators to give the lookup range
- if element is not found, returns iterator that points to end

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

…
std::vector<int> vec = {101, 102, 103, 104, 105};

int elem = 102;
std::vector<int>::iterator it = std::find(vec.begin(), vec.end(), elem1)
```

std::find

- Yes, the syntax can be looooong…
  - Modern C++ syntax to the rescue!

```
std::vector<int> vec = {101, 102, 103, 104, 105};
int elem = 102;

std::vector<int>::iterator it = std::find(vec.begin(), vec.end(), elem);
```

```
//…
auto it = std::find(vec.begin(), vec.end(), elem);
```

std::find

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

…
std::vector<int> vec = {101, 102, 103, 104, 105};

int A = 102;
auto itA = std::find(vec.begin(), vec.end(), A)

int B = 100; // element is not in array
auto itB = std::find(vec.begin(), vec.end(), B)

// found
if( itA != vec.end() ) { std::cout << "vec contains " << A << '\n'; }
else {std::cout << "vec does not contain << A << '\n'; }

// not found
if( itB != vec.end() ) { std::cout << "vec contains " << B << '\n'; }
else {std::cout << "vec does not contain << B << '\n'; }
```

std::sort

- Sort elements of container in the range [first, last)
- Possibility to use custom comparison functions/functors too

```
#include <iostream>
#include <vector>
#include <algorithm>
…
std::vector<int> vec = {3, 0, 2, 4, 1};
std::sort(vec.begin(), vec.end());

for(auto elem : vec) std::cout << elem << '\n'; // prints 0,1,2,3,4
```

Futher topics

- We are just scratching the surface!
- More complex containers
  - std::list, std::set, std::map, std::unordered_map,…
- std::string for string manipulation
- customizable iterators
- functor usage

Summary

- STL is essential part of modern C++
- Containers can deal with complex data types
  - std::vector
- Iterators provide an unified interface for container manipulation
- Algorithm library further extends and generalizes how you treat your abstract data
- Easier, faster, better code!

Sami Ilvonen
Joonas Nättilä
Seija Sirkiä

PARALLEL PROGRAMMING WITH OPENMP

OpenMP – overview

Friday

9:00-9:45	Introduction to OpenMP
9:45-10:15	Exercises
10:15-10:30	Coffee break
10:30-11:00	Work sharing constructs
11:00-12:00	Exercises
12:00-13:00	Lunch
13:00-13:45	Thread synchronisation
13:45-14:30	Exercises
14:30-14:45	Coffee break
14:45-15:15	Task parallelisation
15:15-16:00	Exercises

BASIC CONCEPTS: PROCESSES AND THREADS

Processes and threads

Process

- Independent execution units
- Have their own state information and *own memory* address space

Thread

- A single process may contain multiple threads
- Have their own state information, but *share* the *same memory* address space

Processes and threads

MPI

Process

- Long-lived: spawned when parallel program starts, killed when program is finished
- Explicit communication between processes

OpenMP

Thread

- Short-lived: created when entering a parallel region, destroyed (joined) when region ends
- Communication through shared memory

What is OpenMP?

- A collection of *compiler directives* and *library routines* for multi-threaded, shared-memory parallelization
- Fortran 77/9X/03 and C/C++ are supported
- Latest version of the standard is 4.5 (November 2015)
  - support for attached devices
  - support for thread affinity

Why would you want to learn OpenMP?

- OpenMP parallelized program can be run on your many-core workstation or on a node of a cluster
- Enables one to parallelize one part of the program at a time
  - Get some speedup with a limited investment in time
  - Efficient and well scaling code still requires effort
- Serial and OpenMP versions can easily coexist
- Hybrid MPI+OpenMP programming

Three components of OpenMP

- Compiler directives, i.e. *language extensions*, for shared memory parallelization

	directive	*construct*	*clauses*
C/C++	`#pragma omp`	`parallel`	`shared(data)`
Fortran	`!$omp`	`parallel`	`shared(data)`

- Runtime library routines (Intel: libiomp5, GNU: libgomp)
  - Conditional compilation to build serial version
- Environment variables
  - Specify the number of threads, thread affinity etc.

OpenMP directives

- Sentinels precede each OpenMP directive
  - C/C++: `#pragma omp`
  - Fortran: `!$omp`
- Conditional compilation with `_OPENMP` macro:

```
#ifdef _OPENMP
  OpenMP specific code
#else
  Code without OpenMP
#endif
```

Compiling an OpenMP program

- Compilers that support OpenMP usually require an option that enables the feature
  - GNU: **`-fopenmp`**
  - Intel: **`-qopenmp`**
  - Cray: **`-h omp`**
    - OpenMP enabled by default, **`-h noomp`** disables
  - PGI: **`-mp[=nonuma,align,allcores,bind]`**
- Without these options a serial version is compiled!

Example: Helloworld with OpenMP

```fortran
program hello
  use omp_lib
  integer :: omp_rank
!$omp parallel private(omp_rank)
  omp_rank = omp_get_thread_num()
  print *, 'Hello world! by &
    thread ', omp_rank
!$omp end parallel
end program hello
```

```
> ftn -h omp omp_hello.f90 -o omp
> aprun -e OMP_NUM_THREADS=4 -n 1 -d 4 ./omp
 Hello world! by thread          0
 Hello world! by thread          2
 Hello world! by thread          3
 Hello world! by thread          1
```

```c
#include <stdio.h>
#include <omp.h>
int main(int argc, char argv[]){
  int omp_rank;
#pragma omp parallel private(omp_rank)
 {
  omp_rank = omp_get_thread_num();
  printf("Hello world! by
          thread %d", omp_rank);
 }
}
```

```
> cc -h omp omp_hello.c -o omp
> aprun -e OMP_NUM_THREADS=4 -n 1 -d 4 ./omp
 Hello world! by thread          2
 Hello world! by thread          3
 Hello world! by thread          0
 Hello world! by thread          1
```

PARALLEL REGIONS AND DATA SHARING

Parallel construct

- Defines a parallel region
  C/C++:
  `#pragma omp parallel [clauses]`
  Fortran:
  `!$omp parallel [clauses]`
  - Prior to it only one thread, master
  - Creates a team of threads: master+slave threads
  - At end of the block is a barrier and all shared data is synchronized

SPMD: Single Program Multiple Data

How do the threads interact?

- Because of the shared address space threads can interact using *shared variables*
- Threads often need some *private work space* together with shared variables
  - for example the index variable of a loop
- Visibility of different variables is defined using *data-sharing clauses* in the parallel region definition

`omp parallel`: data-sharing clauses

`private(list)`
- Private variables are stored in the private stack of each thread
- Undefined initial value
- Undefined value after parallel region

`firstprivate(list)`
- Same as private variable, but with an initial value that is the same as the original objects defined outside the parallel region

`omp parallel`: data-sharing clauses

`shared(list)`
- All threads can write to, and read from a shared variable
- Variables are shared by default

Race condition = a thread accesses a variable while another writes into it

`default(private/shared/none)`
- Sets default for variables to be shared, private or not defined
- In C/C++ `default(private)` is not allowed
- `default(none)` can be useful for debugging as each variable has to be defined manually

Default behaviour

- Most variables are *shared* by default
  - Global variables are shared among threads
    - C: `static` variables, file scope variables
    - Fortran: `save` and `module` variables, `common` blocks
    - `threadprivate(list)` can be used to make a private copy
- Private by default:
  - Stack variables of functions called from parallel region
  - Automatic variables within a block

Data sharing example

main.c

```
int A[5];

int main(void) {
    int B[2];
#pragma omp parallel
{
    float c;
    do_things(B);
    ...
}
    return 0;
}
```

kernel.c

```
extern int A[5];

void do_things(int *var) {
    double wrk[10];
    static int status;
    ...
}
```

private

shared

WORK SHARING CONSTRUCTS

Work sharing

- Parallel region creates an "Single Program Multiple Data" instance where each thread executes the same code
- How can one split the work between the threads in a parallel region?
  - Loop construct
  - Single/Master construct
  - Sections construct
  - Task construct
  - Workshare construct (Fortran only)

Loop construct

- Directive instructing compiler to share the work of a loop
  C/C++ : `#pragma omp for [clauses]`
  Fortran : `!$omp do [clauses]`
  - The construct must be followed by a loop construct
    - in C/C++ limited only to for-loops
  - It must reside inside a parallel region
    - Combined construct with `omp parallel`:
      `#pragma omp parallel for / !$omp parallel do`
- The loop index is private by default
- Work sharing can be adjusted with the `schedule` clause

Loop construct

```
!$omp parallel shared(x,y,z) private(i)
!$omp do
  do i = 1, n
    z(i) = x(i) + y(i)
  end do
!$omp end do
!$omp end parallel
```

```
#pragma omp parallel shared(x,y,z) private(i)
{
#pragma omp for
  for (i = 0; i < n ; i++)
    z[i] = x[i] + y[i]
}
```

in C/C++ only for-loops with a "canonical" form, such as here, can be used. One can also use iterator based loops in C++ for random access containers (e.g. std::vector)

Loop scheduling clause

`schedule(static[,chunk])`

- Blocks of iterations of size chunk for each thread

`schedule(dynamic[,chunk])`

- Chunk iterations off a queue until everything is done

`schedule(guided[,chunk])`

- Threads grab blocks of iterations, size of blocks starts from larger size and shrinks down to chunk

`schedule(runtime)`

- Schedule and chunk size are taken from environment variable OMP_SCHEDULE

Loop scheduling clause

```
int kernel(int w, int h, complex dc, int x, int y) {
  float fx = (float)x / w; float fy = (float)y / h;
  complex c = cmin + fx * creal(dc) + fy * cimag(dc) * I;
  int iteration = 0; complex z = c;
  while(iteration < MAX_ITER_COUNT && abs2(z) < 2 * 2) {
      z = z * z + c;
      iteration++;
  }
  return iteration;
}
#pragma omp parallel for schedule(guided)
  for (int i = 0; i < Nx; i++) {
    for (int j = y0; j < Ny; j++) {
      iter_counts[j * w + i] = kernel(w, h, cmax - cmin, i, j);
    }
  }
}
```

Lastprivate data-sharing clause

`lastprivate(list)`

- Private variable where the original variable is updated with the value from the "last" parallel *iteration step* or *section*
- Variable can be both firstprivate and lastprivate

```
int n = 10;
int a, b, i;
#pragma omp parallel for private(i,a) lastprivate(b)
for (i = 0; i < n; i++) {
   a = i;
   b = i;
}
printf("private int is %d, lastprivate int is %d\n", a, b);
```

```
>./test
>private int is 32765, lastprivate int is 9
```

Workshare directive (Fortran only)

- The `workshare` directive divides the execution of the enclosed structured block into separate units of work, each of which is executed only once
  - Array assignments, `forall` and `where` statements etc.

```
real :: a(n,n), b(n,n), c(n,n) d(n,n)
...
!$omp parallel shared(a,b,c,d)
!$omp workshare
      c = a * b
      d = a + b
!$omp end workshare
!$omp end parallel
```

Note that performance may be bad in some compilers, in particular with Intel

Race condition

- Race conditions take place when multiple threads read and write a variable simultaneously, for example

```
asum = 0.0d0
!$omp parallel do shared(x,y,n,asum) private(i)
 do i = 1, n
   asum = asum + x(i)*y(i)
 end do
!$omp end parallel do
```

- Random results depending on the order the threads access `asum`
- We need some mechanism to control the access

Race condition: example

Reductions

- Summing elements of array is an example of reduction operation

Computed by thread 0

Computed by thread 1

Combined result

$$S = \sum_{j=1}^{N} A_j = \sum_{j=1}^{\left\lceil \frac{N}{2} \right\rceil} A_j + \sum_{j=\left\lceil \frac{N}{2} \right\rceil+1}^{N} A_j = B_1 + B_2 = \sum_{j=1}^{2} B_j$$

- OpenMP provides support for common reductions within parallel regions and loops with the *reduction* clause

Reduction clause

`reduction(operator:list)`

- Performs reduction on the (scalar) variables in list
- Private reduction variable is created for each thread's partial result
- Private reduction variable is initialized to operator's initial value
- After parallel region the reduction operation is applied to private variables and result is aggregated to the shared variable

Reduction operators

Operator	Initial value
+	0
-	0
*	1

C/C++ only

Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

Fortran only

Operator	Initial value
.AND.	.true.
.OR.	.false.
.NEGV.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.
MIN	max pos.
MAX	min neg.

Race condition avoided with reduction

```
!$OMP PARALLEL DO SHARED(x,y,n) PRIVATE(i) REDUCTION(+:asum)
  do i = 1, n
    asum = asum + x(i)*y(i)
  end do
!$OMP END PARALLEL DO
```

```
#pragma omp parallel for shared(x,y,n) private(i) reduction(+:asum)
for(i = 0; i < n; i++) {
  asum = asum + x[i] * y[i];
}
```

Summary

- Work sharing construct distribute work between threads
- Parallel do/for loops
  - Remember the restrictions on loop construct (C/C++)
  - Scheduling can be adjusted
  - Race condition when accessing shared variables
    - Reduction clause

THREAD SYNCHRONIZATION

Execution controls

- Sometimes a part of parallel region should be executed only by the master thread or by a single thread at time
  - IO, initializations, updating global values, etc.
  - Remember the synchronization!
- OpenMP provides clauses for controlling the execution of code blocks

Failing example

```
#pragma omp parallel shared(global_counter), private(tnum, delay, rem)
{
  tnum = omp_get_thread_num();
  delay.tv_sec = 0;
  delay.tv_nsec = 10000 * tnum;
  do {
    printf("This is iteration %i\n", global_counter);
    global_counter++;
    nanosleep(&delay, &rem);
  } while(global_counter < 10);
}
```

Race condition!

Execution control constructs

`barrier`

- When a thread reaches a barrier it only continues after all the threads in the same thread team have reached it
  - Each barrier must be encountered by all threads in a team, or none at all
  - The sequence of work-sharing regions and barrier regions encountered must be same for all threads in team
- Implicit barrier at the end of: `do`, `parallel`, `sections`, `single`, `workshare`

Execution control constructs

`master`

- Specifies a region that should be executed only by the master thread
- Note that there is no implicit barrier at end

`single`

- Specifies that a regions should be executed only by a single (arbitrary) thread
- Other threads wait (implicit barrier)

Execution control constructs

`critical[(name)]`

- A section that is executed by only one thread at a time
- Optional name specifies global identifier for critical section
- Unnamed critical sections are treated as the **same** section

`flush[(name)]`

- Synchronizes the memory of all threads
- Only needed in some special cases
- Implicit flush at
  - All explicit and implicit barriers
  - Entry to / exit from critical section and lock routines

Execution control constructs

`atomic`

- Strictly limited construct to update a single value, can not be applied to code blocks
- Only guarantees atomic update, does not protect function calls
- Can be faster on hardware platforms that support atomic updates

Example: reduction using critical section

```
!$OMP PARALLEL SHARED(x,y,n,asum) PRIVATE(i, psum)
  psum = 0.0d
  !$OMP DO
  do i = 1, n
    psum = psum + x(i)*y(i)
  end do
  !$OMP END DO
  !$OMP CRITICAL(dosum)
  asum = asum + psum
  !$OMP END CRITICAL(dosum)
!$OMP END PARALLEL
```

Example: initialization and output

```
int total = 0;
#pragma omp parallel shared(total) private(sum,new)
{
#pragma omp single
  initialise();

  int new, sum = 0;
  do {
    new = compute_something();
    sum += new;
  } while (new);
#pragma omp critical(addup)
  total += sum;
#pragma omp master
  printf("Grand total is: %5.2f\n", total);
}
```

OPENMP RUNTIME LIBRARY AND ENVIRONMENT VARIABLES

OpenMP and execution environment

- OpenMP provides several means to interact with the execution environment. These operations include e.g.
  - Setting the number of threads for parallel regions
  - Requesting the number of CPUs
  - Changing the default scheduling for work-sharing clauses
- Improves portability of OpenMP programs between different architectures (number of CPUs, etc.)

Environment variables

- OpenMP standard defines a set of environment variables that all implementations have to support
- The environment variables are set before the program execution and they are read during program start-up
  - Changing them during the execution has no effect
- We have already used **OMP_NUM_THREADS**

Runtime functions

- Runtime functions can be used either to read the settings or to set (override) the values
- Function definitions are in
  - C/C++ header file **`omp.h`**
  - **`omp_lib`** Fortran module (**`omp_lib.h`** header in some implementations)
- Two useful routines for distributing work load:
  - **`omp_get_num_threads()`**
  - **`omp_get_thread_num()`**

List of environment variables (OpenMP 4.5)

OMP_SCHEDULE
OMP_NUM_THREADS
OMP_DYNAMIC
OMP_PROC_BIND
OMP_PLACES
OMP_NESTED
OMP_STACKSIZE
OMP_WAIT_POLICY
OMP_MAX_TASK_PRIORITY
OMP_MAX_ACTIVE_LEVELS
OMP_THREAD_LIMIT
OMP_CANCELLATION
OMP_DISPLAY_ENV
OMP_DEFAULT_DEVICE

Some useful environment variables

Variable	Action
OMP_NUM_THREADS	Number of threads to use
OMP_SCHEDULE	Default scheduling
OMP_PROC_BIND	Bind threads to CPUs
OMP_PLACES	Specify the bindings between threads and CPUs
OMP_DISPLAY_ENV	Print the current OpenMP environment info on stderr
OMP_STACKSIZE	Default size of thread stack
OMP_THREAD_LIMIT	Maximum number of threads to use

Parallelizing a loop with library functions

Don't use this in production!

```
#pragma omp parallel private(i,nthrds,thrd)
{
  nthrds = omp_get_num_threads();
  thrd = omp_get_thread_num();
  for(i = thrd; i < n; i += nthrds) {
    vec_c[i] = vec_a[i] + vec_b[i];
  }
}
```

Fixed example

```
#pragma omp parallel shared(global_counter), private(tnum, delay, rem)
{
  tnum = omp_get_thread_num();
  delay.tv_sec = 0;
  delay.tv_nsec = 10000 * tnum;
  do {
#pragma omp barrier
#pragma omp single {
      printf("This is iteration %i\n", global_counter);
      global_counter++;
    }
    nanosleep(&delay, &rem);
  } while(global_counter < 10);
}
```

Summary

- OpenMP has many synchronization pragmas
  - Critical sections
  - Atomic
  - Single and Master
  - And some that we did not present (yet)
- OpenMP runtime behavior can be controlled using environment variables

TASK PARALLELISATION

Limitations of work sharing so far

- Number of iterations in a loop must be constant
  - No while loops or early exits in *for/do* loops
- All the threads have to participate in workshare
- OpenMP provides also dynamic tasking in addition to static sections
- Irregular and dynamical parallel patterns via tasking
  - While loops
  - Recursion

What is a task in OpenMP?

- A task has
  - Code to execute
  - Data environment
  - Internal control variables
- Tasks are added to a *task queue*, and executed then by single thread
  - Can be same or different thread that created the task
  - OpenMP runtime takes care of distributing tasks to threads

OpenMP task construct

- Create a new task and add it to task queue
  - Memorize data and code to be executed
  - Task constructs can arbitrarily nested
- Syntax (C/C++)
  #pragma omp task *[clause[[,] clause],…]*
  structured-block
- Syntax (Fortran)
  !$omp task*[clause[[,] clause],…]*
  block
  !$omp end task

OpenMP task construct

- All threads that encounter the construct create a task
- Typical usage pattern is thus that single thread creates the tasks

```
#pragma omp parallel
#pragma omp single
{
  int i=0;
  while (i < 12 ) {
    #pragma omp task
    printf("Task %d by thread %d \n", i, omp_get_thread_num());

    i++;
  }
}
```

OpenMP task construct

How many tasks does the following code create when executed with 4 threads?

```
#pragma omp parallel
{
  int i=0;
  while (i < 6 ) {
    #pragma omp task
    do_some_heavy_work();

    i++;
  }
}
```

a) 6 b) 4 c) 24

Task execution model

- Tasks are executed by an arbitrary thread
  - Can be same or different thread that created the task
  - By default, tasks are executed in arbitrary order
  - Each task is executed only once
- Synchronization points
  - Implicit or explicit barriers
  - `#pragma omp taskwait` / `!$omp taskwait`
    - Encountering task suspends until child tasks complete

Data environment of a task

- Tasks are created at one time, and executed at another
  - What data does the task see when executing?
- Variables that are *shared* in the enclosing construct contain the data at the time of execution
- Variables that are *private* in the enclosing construct are made *firstprivate* and contain the data at the time of creation
- Data scoping clauses (*shared*, *private*, *firstprivate*, *default*) can change the default behaviour

Data environment of a task

What is the value of i that is printed out?

```
#pragma omp parallel
{
  int i=0;
  #pragma omp master
  {
    while (i < 6 ) {
      #pragma omp task
      if (omp_get_thread_num() != 0) i=100;

      i++;
    }
  }
  #pragma omp barrier
  if (omp_get_thread_num() == 0) printf("i is %d\n", i);
}
```

a) i=0 b) i=6 c) i=100

Data environment of a task

What is the value of i that is printed out?

```
#pragma omp parallel
{
  int i=0;
  #pragma omp master
  {
    while (i < 6 ) {
      #pragma omp task shared(i)
      if (omp_get_thread_num() == 1) i=100;

      i++;
    }
  }
  #pragma omp barrier
  if (omp_get_thread_num() == 0) printf("i is %d\n", i);
}
```

a) i=0 b) i=6 c) i≥100

Recursive algorithms with tasks

- A task can itself generate new tasks
  - Useful when parallelising recursive algorithms
- Recursive algorithm for Fibonacci numbers:
  $F_n=F_{n-1}+F_{n-2}$ $F_0=0$, $F_1=1$

```
#pragma omp parallel
{
  fibonacci(10);
}
```

```
int fibonacci(int n) {
  int fn, fnm;
  if (n<2)
    return n;
#pragma omp task shared(fn)
  fn = fibonacci(n-1);

#pragma omp task shared(fnm)
  fnm = fibonacci(n-2);

/* Wait for child tasks to finish */
#pragma omp taskwait
  return fn+fnm;
}
```

OpenMP programming best practices

- Maximize parallel regions
  - Reduce fork-join overhead, e.g. combine multiple parallel loops into one large parallel region
  - Potential for better cache re-usage
- Parallelize outermost loops if possible
  - Move PARALLEL DO construct outside of inner loops
- Reduce access to shared data
  - Possibly make small arrays private
- Use more tasks than threads
  - Too large number of tasks leads to performance loss

OpenMP summary

- OpenMP is an API for thread-based parallelization
  - Compiler directives, runtime API, environment variables
  - Relatively easy to get started but specially efficient and/or real-world parallelization non-trivial
- Features touched in this intro
  - Parallel regions, data-sharing attributes
  - Work-sharing and scheduling directives
  - Task based parallelization

OpenMP summary

Things that we did not cover

- Other work-sharing clauses
  - **`workshare`**, **`sections`**, **`simd`**
  - **`teams`**, **`distribute`**
- More advanced ways to reduce synchronization overhead with **`nowait`** and **`flush`**
- **`threadprivate`**, **`copyin`**, **`cancel`**
- A user can define dependencies between tasks with the **depend** clause
- Support for attached devices with **`target`**

Web resources

- OpenMP homepage
  http://openmp.org/
- Good online tutorial:
  https://computing.llnl.gov/tutorials/openMP/
- More online tutorials:
  http://openmp.org/wp/resources/#Tutorials

Jussi Enkovaara
Pekka Manninen
Seija Sirkiä

PARALLEL PROGRAMMING WITH MESSAGE-PASSING INTERFACE (MPI)

MPI – overview

Saturday

13.00-14.00	Introduction to MPI
14.00-14.15	Coffee break
14.15-15.20	Point-to-point communication
15.20-17.00	Exercises

Sunday

9.00-9.45	Collective communication
9.45-10.30	Exercises
10.30-10.45	Coffee break
10.45-11.00	User-defined communicators
11.00-12.00	Exercises
12.00-13.00	Lunch break
15.15-15.35	Non-blocking communication, part I
15.35-16.00	Exercises
16.00-16.20	Non-blocking communication, part II
16.20-17.00	Exercises

MPI – overview

Monday

9.00-9.25	User-defined datatypes, part I
9.25-10.00	Exercises
10.00-10.45	User-defined datatypes, part II
10.45-11.00	Coffee break
11.00-12.00	Exercises
12.00-13.00	Lunch break

Web resources

- List of MPI functions with detailed descriptions
  http://mpi.deino.net/mpi_functions/index.htm
- Good online MPI tutorials
  https://computing.llnl.gov/tutorials/mpi
  http://mpitutorial.com/tutorials/
- MPI 3.1 standard
  http://www.mpi-forum.org/docs/
- MPI implementations
  MPICH2 http://www.mcs.anl.gov/research/projects/mpich2/
  OpenMPI http://www.open-mpi.org/

CSC

Basic concepts in MPI

Message-passing interface

- MPI is an application programming interface (API) for *distributed* parallel computing
- MPI programs are portable and scalable
  - the same program can run on different types of computers, from laptops to supercomputers
- MPI is flexible and comprehensive
  - large (hundreds of procedures)
  - concise (only 10-20 procedures are typically needed)

Processes and threads

MPI

Process

- Independent execution units
- Have their own state information and *own memory* address space

OpenMP

Thread

- A single process may contain multiple threads
- Have their own state information, but *share* the *same memory* address space

Execution model in MPI

- Parallel program is launched as set of *independent, identical processes*
  - execute the *same program code* and instructions
  - processes can reside in different nodes (or even in different computers)
- The way to launch parallel program depends on the computing system
  - mpiexec, mpirun, aprun, srun, …
  - aprun on sisu.csc.fi, srun on taito.csc.fi

MPI ranks

- MPI runtime assigns each process *a unique rank*
  - identification of the processes
  - ranks start from 0 and extent to N-1
- Processes can perform different tasks and handle different data basing on their rank

```
...
if ( rank == 0 ) {
   ...
   }
if ( rank == 1) {
   ...
   }
...
```

Data model

- All variables and data structures are local to the process
- Processes can exchange data by sending and receiving messages

Data model

The MPI library

- Information about the communication framework
  - number of processes
  - rank of the process
- Communication between processes
  - sending and receiving messages between two processes
  - sending and receiving messages between several processes
- Synchronization between processes
- Advanced features

Programming MPI

- The MPI standard defines interfaces to C and Fortran programming languages
  - There are unofficial bindings to Python, Perl and Java
- C call convention

  `rc = MPI_Xxxx(parameter,...)`

  - some arguments have to be passed as pointers

  Case sensitive, "MPI" and the first letter after the underscore capitalized

- Fortran call convention

  `call mpi_xxxx(parameter,..., rc)`

  - return code in the last argument

  Case insensitive

Writing an MPI program

- Include the MPI header files

  C **`#include <mpi.h>`**

  Fortran **`use mpi`**
- Start by calling the routine **`MPI_Init`**
- Write the program
- Call **`MPI_Finalize`** before exiting from the main program

Presenting syntax

First five MPI commands

- Set up the MPI environment

  `MPI_Init()`

  In C, argc and argv need to be passed to MPI_Init (as pointers)

- Information about the communicator

  `MPI_Comm_size(comm, size)`

  comm communicator
  size number of processes in the communicator

  `MPI_Comm_rank(comm, rank)`

  comm communicator
  rank rank of this process

First five MPI commands

- Synchronize processes

  `MPI_Barrier(comm)`
- Finalize MPI environment

  `MPI_Finalize()`

MPI communicator

- Communicator is an object connecting a group of processes, i.e. the communication framework
- Most MPI functions require communicator as an argument, i.e. take place in a given context
- Initially, there is always a communicator `MPI_COMM_WORLD` which contains all the processes
  - and `MPI_COMM_SELF` that includes only the calling rank
- Users can define own communicators

Interim summary

- In parallel programming with MPI, the key concept is a set of independent processes
- Data is always local to the process
- Processes can exchange data by sending and receiving messages
- The MPI library contains functions for communication and synchronization between processes

Point-to-point communication

MPI communication

- MPI processes are independent, they communicate to coordinate work
- Point-to-point communication
  - Messages are sent between two processes
- Collective communication
  - Involving a number of processes at the same time

MPI point-to-point operations

- One process *sends* a message to another process that *receives* it
- Sends and receives in a program should match – one receive per each send

Case study: parallel sum

- Array originally on process #0 (P0)
- Parallel algorithm
  - Scatter
    - Half of the array is sent to process 1
  - Compute
    - P0 & P1 sum independently their segments
  - Reduction
    - Partial sum on P1 sent to P0
    - P0 sums the partial sums

Case study: parallel sum

Step 1.1: Receive operation in scatter

P1 issues MPI_Recv to receive half of the array from P0

Case study: parallel sum

Step 1.2: Send operation in scatter

P0 issues an MPI_Send to send the lower part of the array to P1

Case study: parallel sum

Step 2: Compute the sum in parallel

Both P0 & P1 compute their partial sums and store them locally

Case study: parallel sum

Step 3.1: Receive operation in reduction

P0 issues an MPI_Recv operation for receiving P1's partial sum

Case study: parallel sum

Step 3.2: send operation in reduction

P1 sends the partial sum to P0

Case study: parallel sum

Step 3.3: compute the final answer

P0 computes the total sum

MPI point-to-point operations

- Each MPI message contains
  - The actual *data*
  - Information about the *datatype* of each element of data
  - The *number of elements* the data consists of
  - An identification number for the message (*tag*)
  - The ranks of the *source* and *destination* process

Send operation

```
MPI_Send(buffer, count, datatype, dest, tag, comm)
```

buffer	The data to be sent
count	Number of elements in buffer
datatype	Type of elements in buffer (see later slides)
dest	The rank of the receiver
tag	An integer identifying the message
comm	Communicator
error	Error value; in C/C++ it's the return value of the function, and in Fortran an additional output parameter

Receive operation

```
MPI_Recv(buffer, count, datatype, source, tag, comm,
         status)
```

buffer	Buffer for storing received data
count	Number of elements in buffer, not the number of element that are actually received
datatype	Type of each element in buffer
source	Sender of the message
tag	Number identifying the message
comm	Communicator
status	Information on the received message
error	As for send operation

MPI datatypes

- MPI has a number of predefined datatypes to represent data
- Each C or Fortran datatype has a corresponding MPI datatype
  - C examples: MPI_INT for int and MPI_DOUBLE for double
  - Fortran example: MPI_INTEGER for integer
- One can also define custom datatypes

MORE FEATURES IN POINT-TO-POINT COMMUNICATION

Blocking routines & deadlocks

- MPI_Send and MPI_Recv are *blocking* routines
  - MPI_Send exits once the send buffer can be safely read and written to
  - MPI_Recv exits once it has received the message in the receive buffer
- Completion depends on other processes => risk for *deadlocks*
  - For example, all processes are in MPI_Recv
  - If deadlocked, the program is stuck forever

Typical point-to-point communication patterns

Pairwise exchange

Pipe, a ring of processes exchanging data

- Incorrect ordering of sends/receives may give a rise to a deadlock (or unnecessary idle time)

Combined send & receive

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest,
  sendtag, recvbuf, recvcount, recvtype, source,
  recvtag, comm, status)
```

- Sends one message and receives another one, with a single command
  - Reduces risk for deadlocks
- Parameters as in `MPI_Send` and `MPI_Recv`
- Destination rank and source rank can be same or different

Special parameter values

```
MPI_Send(buffer, count, datatype, dest, tag, comm)
```

Parameter	Special value	Implication
dest	MPI_PROC_NULL	Null destination, no operation takes place

Special parameter values

```
MPI_Recv(buffer, count, datatype, source, tag, comm,
  status)
```

Parameter	Special value	Implication
source	MPI_PROC_NULL	No sender=no operation takes place
	MPI_ANY_SOURCE	Receive from any sender
tag	MPI_ANY_TAG	Receive messages with any tag
status	MPI_STATUS_IGNORE	Do not store any status data

Status parameter

- The status parameter in `MPI_Recv` contains information about the received data after the call has completed, e.g.
  - Number of received elements
  - Tag of the received message
  - Rank of the sender
- In C the status parameter is a struct
- In Fortran it is an integer array of size `MPI_STATUS_SIZE`

Status parameter

- Received elements

  Use the function

  `MPI_Get_count(status, datatype, count)`
- Tag of the received message

  C: `status.MPI_TAG`

  Fortran: `status(MPI_TAG)`
- Rank of the sender

  C: `status.MPI_SOURCE`

  Fortran: `status(MPI_SOURCE)`

Summary

- Point-to-point communication = messages are sent between two MPI processes
- Point-to-point operations enable any parallel communication pattern (in principle)
  - `MPI_Send` & `MPI_Recv`
  - `MPI_Sendrecv`
- Employing special argument values may simplify the implementations of certain communication patterns
- Status parameter of `MPI_Recv` contains information about the message after the receive is completed

Collective communication

Introduction

- Collective communication transmits data among all processes in a process group (communicator)
- Collective communication includes
  - data movement
  - collective computation
  - synchronization

Introduction

- Collective communication typically outperforms point-to-point communication
- Code becomes more compact and easier to read:

```
if (my_id == 0) then
  do i = 1, ntasks-1
    call mpi_send(a, 1048576, &
        MPI_REAL, i, tag, &
        MPI_COMM_WORLD, rc)
  end do
else
  call mpi_recv(a, 1048576, &
      MPI_REAL, 0, tag, &
      MPI_COMM_WORLD, status, rc)
end if
```

```
call mpi_bcast(a, 1048576, &
        MPI_REAL, 0, &
        MPI_COMM_WORLD, rc)
```

Communicating a vector a consisting of 1M float elements from the task 0 to all other tasks

Introduction

- These routines *must be called by all the processes* in the communicator
- Amount of sent and received data must match
- No tag arguments
  - Order of execution must coincide across processes

Broadcasting

- Replicate data from one process to all others

Broadcasting

- With `MPI_Bcast`, the task *root* sends a *buffer* of data to all other tasks

```
MPI_Bcast(buffer, count, datatype, root, comm)
```

buffer	data to be distributed
count	number of entries in buffer
datatype	data type of buffer
root	rank of broadcast root
comm	communicator

Scattering

- Send equal amount of data from one process to others

- Segments A, B, … may contain multiple elements

Scattering

- `MPI_Scatter` = Task *root* sends an equal share of data (*sendbuf*) to all other processes

```
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf,
            recvcount, recvtype, root, comm)
```

sendbuf	send buffer (data to be scattered)
sendcount	number of elements sent to each process
sendtype	data type of send buffer elements
recvbuf	receive buffer
recvcount	number of elements to receive at each process
recvtype	data type of receive buffer elements
root	rank of sending process
comm	communicator

Examples

```
if (my_id==0) then
  do i = 1, 16
    a(i) = i
  end do
end if

call mpi_bcast(a,16,MPI_INTEGER,0, &
               MPI_COMM_WORLD,rc)
if (my_id==3) print *, a(:)
```

```
if (my_id==0) then
  do i = 1, 16
    a(i) = i
  end do
end if
call mpi_scatter(a,4,MPI_INTEGER, &
                 aloc,4,MPI_INTEGER, &
                 0,MPI_COMM_WORLD,rc)
if (my_id==3) print *, aloc(:)
```

Assume 4 MPI tasks. What would the (full) program print?

```
A.  1 2 3 4
B.  13 14 15 16
C.  1 2 3 4
    5 6 7 8
    9 10 11 12
    13 14 15 16
```

```
A.  1 2 3 4
B.  13 14 15 16
C.  1 2 3 4
    5 6 7 8
    9 10 11 12
    13 14 15 16
```

Vector version of MPI_Scatter

- Like `MPI_Scatter`, but messages can have different sizes and displacements

```
MPI_Scatterv(sendbuf, sendcounts, displs, sendtype,
             recvbuf, recvcount, recvtype, root, comm)
```

sendbuf	send buffer
sendcounts	array (of length ntasks) specifying the number of elements to send to each processor
displs	array (of length ntasks). Entry i specifies the displacement (relative to sendbuf)
sendtype	data type of send buffer elements
recvbuf	receive buffer
recvcount	number of elements to receive
recvtype	data type of receive buffer elements
root	rank of sending process
comm	communicator

Scatterv example

```
if (my_id==0) then
  do i = 1, 10
    a(i) = i
  end do
end if
counts = [ 1, 2, 3, 4 ]
displs = [ 0, 1, 3, 6 ]

call mpi_scatterv(a, counts, &
    displs, MPI_INTEGER, &
    aloc, counts(my_id), &
    MPI_INTEGER, 0, &
    MPI_COMM_WORLD, rc)
```

```
A.  1 2 3
B.  7 8 9 10
C.  1 2 3 4 5 6 7 8 9 10
```

Assume 4 MPI tasks. What are the values in aloc in the last task (#3)?

Gathering data

- Collect data from all the process to one process

- Segments A, B, … may contain multiple elements

Gathering data

- `MPI_Gather`: Collect an equal share of data (in *sendbuf*) from all processes to *root*

```
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf,
           recvcount, recvtype, root, comm)
```

sendbuf	send buffer (data to be gathered)
sendcount	number of elements pulled from each process
sendtype	data type of send buffer elements
recvbuf	receive buffer
recvcount	number of elements in any **single** receive
recvtype	data type of receive buffer elements
root	rank of receiving process
comm	communicator

Vector version of MPI_Gather

- Like `MPI_Gather`, but messages can have different sizes and displacements

```
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf,
            recvcounts, displs, recvtype, root, comm)
```

sendbuf	send buffer
sendcount	the number of elements to send
sendtype	data type of send buffer elements
recvbuf	receive buffer
recvcounts	array (of length ntasks). Entry i specifies how many to receive from that process
displs	array (of length ntasks). Entry i specifies the displacement (relative to recvbuf)
recvtype	data type of receive buffer elements
root	rank of receiving process
comm	communicator

Reduce operation

- Applies an operation over set of processes and places result in single process

Reduce operation

- Available reduction operations (argument op)

Operation	Meaning
MPI_MAX	Max value
MPI_MIN	Min value
MPI_SUM	Sum
MPI_PROD	Product
MPI_MAXLOC	Max value + location
MPI_MINLOC	Min value + location
MPI_LAND	Logical AND
MPI_BAND	Bytewise AND
MPI_LOR	Logical OR
MPI_BOR	Bytewise OR
MPI_LXOR	Logical XOR
MPI_BXOR	Bytewise XOR

Reduce operation

- Applies a reduction operation *op* to *sendbuf* over the set of tasks and places the result in *recvbuf* on *root*

```
MPI_Reduce(sendbuf, recvbuf, count, datatype, op,
           root, comm)
```

sendbuf	send buffer
recvbuf	receive buffer
count	number of elements in send buffer
datatype	data type of elements in send buffer
op	operation
root	rank of root process
comm	communicator

Global reduction

- `MPI_Allreduce` combines values from all processes and distributes the result back to all processes
  - Compare: `MPI_Reduce` + `MPI_Bcast`

```
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm)
```

sendbuf	starting address of send buffer
recvbuf	starting address of receive buffer
count	number of elements in send buffer
datatype	data type of elements in send buffer
op	operation
comm	communicator

Allreduce example: parallel dot product

```
real :: a(1024), aloc(128)
...
if (my_id==0) then
    call random_number(a)
end if
call mpi_scatter(a, 128, MPI_INTEGER, &
                 aloc, 128, MPI_INTEGER, &
                 0, MPI_COMM_WORLD, rc)
rloc = dot_product(aloc,aloc)
call mpi_allreduce(rloc, r, 1, MPI_REAL,&
                   MPI_SUM, MPI_COMM_WORLD,
                   rc)
```

```
> aprun -n 8 ./mpi_pdot
 id= 6 local= 39.68326  global= 338.8004
 id= 7 local= 39.34439  global= 338.8004
 id= 1 local= 42.86630  global= 338.8004
 id= 3 local= 44.16300  global= 338.8004
 id= 5 local= 39.76367  global= 338.8004
 id= 0 local= 42.85532  global= 338.8004
 id= 2 local= 40.67361  global= 338.8004
 id= 4 local= 49.45086  global= 338.8004
```

All gather

- `MPI_Allgather` gathers data from each task and distributes the resulting data to each task
  - Compare: `MPI_Gather` + `MPI_Bcast`

```
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf,
              recvcount, recvtype, comm)
```

sendbuf	send buffer
sendcount	number of elements in send buffer
sendtype	data type of send buffer elements
recvbuf	receive buffer
recvcount	number of elements received from any process
recvtype	data type of receive buffer

All to all

- Send a distinct message from each task to every task

- "Transpose" like operation

All to all

- `MPI_Alltoall` sends a distinct message from each task to every task
  - Compare: "All scatter"

```
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf,
             recvcount, recvtype, comm)
```

sendbuf	send buffer
sendcount	number of elements to send to each process
sendtype	data type of send buffer elements
recvbuf	receive buffer
recvcount	number of elements received from any process
recvtype	data type of receive buffer elements
comm	communicator

All-to-all example

```
if (my_id==0) then
  do i = 1, 16
    a(i) = i
  end do
end if
call mpi_bcast(a, 16, MPI_INTEGER, 0, &
    MPI_COMM_WORLD, rc)

call mpi_alltoall(a, 4, MPI_INTEGER, &
                  aloc, 4, MPI_INTEGER, &
                  MPI_COMM_WORLD, rc)
```

Assume 4 MPI tasks. What will be the values of aloc in the process #0?

```
A. 1, 2, 3, 4
B. 1, ..., 16
C. 1, 2, 3, 4, 1, 2, 3, 4,
   1, 2, 3, 4, 1, 2, 3, 4
```

Common mistakes with collectives

- Using a collective operation within one branch of an if-test of the rank

```
if (my_id == 0) call mpi_bcast(...
```

  - All processes, both the root (the sender or the gatherer) and the rest (receivers or senders), must call the collective routine!
- Assuming that all processes making a collective call would complete at the same time
- Using the input buffer as the output buffer

```
call mpi_allreduce(a, a, n, mpi_real,...
```

  - One should employ `MPI_IN_PLACE` for this purpose

Summary

- Collective communications involve all the processes within a communicator
  - All processes must call them
- Collective operations make code more transparent and compact
- Collective routines allow optimizations by MPI library

Summary

USER-DEFINED COMMUNICATORS

Communicators

- The communicator determines the "communication universe"
  - The source and destination of a message is identified by process rank within the communicator
- So far: MPI_COMM_WORLD
- Processes can be divided into subcommunicators
  - Task level parallelism with process groups performing separate tasks
  - Parallel I/O

Communicators

- Communicators are dynamic
- A task can belong simultaneously to several communicators
  - In each of them it has a unique ID, however
  - Communication is normally within the communicator

Creating a new communicator

Creating new communicator

- `MPI_Comm_split` creates new communicators based on 'colors' and 'keys'

```
MPI_Comm_split(comm, color, key, newcomm)
```

comm	communicator handle
color	control of subset assignment, processes with the same color belong to the same new communicator
key	control of rank assignment
newcomm	new communicator handle

If color = MPI_UNDEFINED, a process does not belong to any of the new communicators

Creating new communicator

```
if (myid%2 == 0) {
  color = 1;
} else {
  color = 2;
}
MPI_Comm_split(MPI_COMM_WORLD, color, myid, &subcomm);
MPI_Comm_rank(subcomm, &mysubid);
printf ("I am rank %d in MPI_COMM_WORLD, but %d in
        Comm %d.\n", myid, mysubid, color);
```

```
I am rank 2 in MPI_COMM_WORLD, but 1 in Comm 1.
I am rank 7 in MPI_COMM_WORLD, but 3 in Comm 2.
I am rank 0 in MPI_COMM_WORLD, but 0 in Comm 1.
I am rank 4 in MPI_COMM_WORLD, but 2 in Comm 1.
I am rank 6 in MPI_COMM_WORLD, but 3 in Comm 1.
I am rank 3 in MPI_COMM_WORLD, but 1 in Comm 2.
I am rank 5 in MPI_COMM_WORLD, but 2 in Comm 2.
I am rank 1 in MPI_COMM_WORLD, but 0 in Comm 2.
```

Using an own communicator

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

```
if (myid%2 == 0) {
  color = 1;
} else {
  color = 2;
}
MPI_Comm_split(MPI_COMM_WORLD, color, myid, &subcomm);
MPI_Comm_rank(subcomm, &mysubid);
MPI_Bcast(sendbuf, 8, MPI_INT, 0, subcomm);
```

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15

Communicator manipulation

`MPI_Comm_size`	Returns number of processes in communicator's group
`MPI_Comm_rank`	Returns rank of calling process in communicator's group
`MPI_Comm_compare`	Compares two communicators
`MPI_Comm_dup`	Duplicates a communicator
`MPI_Comm_free`	Marks a communicator for deallocation

Summary

- Defining new communicators usually required in real-world programs
  - Task parallelism, using libraries, I/O,...
- We introduced one way of creating new communicators via `MPI_Comm_split`
  - Tasks assigned with a color, which can be MPI_UNDEFINED if the task is excluded in all resulting communicators
  - Other ways (via MPI groups) exist

Non-blocking communication

Non-blocking communication

- Non-blocking communication operations return immediately and perform sending/receiving in the background
  - MPI_Isend & MPI_Irecv
- Enables some computing concurrently with communication
- Avoids many common dead-lock situations
- Also collective operations available as non-blocking versions

Non-blocking send

```
MPI_Isend(buffer, count, datatype, dest, tag,
          comm, request)
```

Parameters similar to MPI_Send but has an additional request parameter

buffer	send buffer that must not be written to until one has checked that the operation is over
request	a handle that is used when checking if the operation has finished (integer in Fortran, MPI_Request in C)

Non-blocking receive

```
MPI_Irecv(buffer, count, datatype, source, tag,
          comm, request)
```

Parameters similar to MPI_Recv but has no status parameter

buffer	receive buffer guaranteed to contain the data only after one has checked that the operation is over
request	a handle that is used when checking if the operation has finished

Non-blocking communication

- Important: Send/receive operations have to be finalized
  - MPI_Wait, MPI_Waitall,…
    - Waits for the communication started with MPI_Isend or MPI_Irecv to finish (blocking)
  - MPI_Test,…
    - Tests if the communication has finished (non-blocking)
- You can mix non-blocking and blocking routines
  - e.g., receive a message sent by MPI_Isend with MPI_Recv

Wait for non-blocking operation

```
MPI_Wait(request, status)
```

Parameters

request	handle of the non-blocking communication
status	status of the completed communication, see MPI_Recv

A call to MPI_WAIT returns when the operation identified by request is complete

Wait for non-blocking operations

```
MPI_Waitall(count, requests, status)
```

count	number of requests
requests	array of requests
status	array of statuses for the operations that are waited for

A call to MPI_Waitall returns when *all* operations identified by the array of requests are complete

Non-blocking test for non-blocking operations

```
MPI_Test(request, flag, status)
```

request	request
flag	True if the operation has completed
status	status for the completed operation

A call to MPI_Test is non-blocking. It allows one to schedule alternative activities while periodically checking for completion.

Typical usage pattern

```
MPI_Irecv(ghost_data)
MPI_Isend(border_data)
compute(ghost_independent_data)
MPI_Waitall
compute(border_data)
```

Additional completion operations

- Other routines for completing the communication or testing whether it has finished

Routine	Meaning
MPI_Waitany	Waits until any one operation has completed
MPI_Waitsome	Waits until at least one operation has completed
MPI_Test	Tests whether an operation has completed (non-blocking)
MPI_Testall	Tests whether a list of operations have completed
MPI_Testany	Like Waitany but non-blocking
MPI_Testsome	Like Waitsome but non-blocking
MPI_Probe	Check for incoming messages without receiving them

Wait for non-blocking operations

MPI_Waitany(count, requests, index, status)

count	number of requests
requests	array of requests
index	index of request that completed
status	status for the completed operations

A call to MPI_Waitany returns when one operation identified by the array of requests is complete

Wait for non-blocking operations

MPI_Waitsome(count, requests, done, index, status)

count	number of requests
requests	array of requests
done	number of completed requests
index	array of indexes of completed requests
status	array of statuses of completed requests

A call to MPI_Waitsome returns when one or more operations identified by the array of requests is complete

Message probing

MPI_Iprobe(source, tag, comm, flag, status)

source	rank of sender (or MPI_ANY_SOURCE)
tag	message of the tag (or MPI_ANY_TAG)
comm	communicator
flag	true if there is a message that matches the pattern and can be received
status	status object

MPI_Iprobe and MPI_Probe allow incoming messages to be checked, without actually receiving them

Non-blocking collectives

- Non-blocking collectives ("I-collectives") enable the overlapping of communication and computation together with the benefits of collective communication
- The same syntax as for blocking collectives, besides
  - "I" at the front of the name (MPI_Alltoall -> MPI_Ialltoall)
  - The request parameter at the end of the list of arguments
  - The completion needs to be waited

Non-blocking collectives

- Restrictions
  - Have to be called in same order by all ranks in a communicator
  - Mixing of blocking and non-blocking collectives not allowed

Non-blocking collectives

Example: Non-blocking broadcasting

```
MPI_Ibcast(buffer, count, datatype, root, comm, request)
```

Parameters

buffer	data to be distributed
count	number of entries in buffer
datatype	data type of buffer
root	rank of broadcast root
comm	communicator
request	a handle that is used when checking if the operation has finished

PERSISTENT COMMUNICATION

Persistent communication

- Often a communication with same argument list is repeatedly executed
- It may be possible to optimize such pattern by *persistent communication* requests
  - Can be thought as a "communication port"
- Usage
  - Create requests: MPI_Send_init & MPI_Recv_init
  - Start communication: MPI_Start / MPI_Startall
  - Complete communication: MPI_Wait / MPI_Waitall

Initiation and starting of communication separated in addition to communication and completion

Persistent communication

```
MPI_Request recv_obj, send_obj;
...
// Initialize send/request objects
MPI_Recv_init (buf1, cnt, MPI_DOUBLE, src, tag, MPI_COMM_WORLD, &recv_obj);
MPI_Send_init (buf2, cnt, MPI_DOUBLE, dst, tag, MPI_COMM_WORLD, &send_obj);
for (i=1; i<BIGNUM; i++)
{
    // Start communication described by recv_obj and send_obj
    MPI_Start (&recv_obj);
    do_work(buf1,buf2);
    MPI_Start (&send_obj);
    // Wait for send to complete
    MPI_Wait (&send_obj, MPI_STATUS_IGNORE);
    //Wait for receive to finish (no deadlock!)
    MPI_Wait(&recv_obj, MPI_STATUS_IGNORE);
}
//Clean up the requests
MPI_Request_free (&recv_obj); MPI_Request_free (&send_obj);
```

Summary

- Non-blocking communication is usually the smarter way to do point-to-point communication in MPI
- Non-blocking communication core features
  - Open receives with `MPI_Irecv`
  - Start sending with `MPI_Isend`
  - Possibly do something else while the communication takes place
  - Complete the communication with `MPI_Wait` or a variant
- MPI-3 contains also non-blocking collectives
- Persistent communication may enable optimization

User-defined datatypes

MPI datatypes

- MPI datatypes are used for communication purposes
  - Datatype tells MPI where to take the data when sending or where to put data when receiving
- Elementary datatypes (`MPI_INT`, `MPI_REAL`, ...)
  - Different types in Fortran and C, correspond to languages basic types
  - Enable communication using contiguous memory sequence of identical elements (e.g. vector or matrix)

Sending a matrix row (Fortran)

- Row of a matrix is not contiguous in memory in Fortran
- Several options for sending a row:
  - Use several send commands for each element of a row
  - Copy data to temporary buffer and send that with one send command
  - Create a matching datatype and send all data with one send command

User-defined datatypes

- Use elementary datatypes as building blocks
- Enable communication of
  - Non-contiguous data with a single MPI call, e.g. rows or columns of a matrix
  - Heterogeneous data (structs in C, types in Fortran)
  - Larger messages, count is int in C
- Provide higher level of programming
  - Code is more compact and maintainable
- Needed for getting the most out of MPI I/O

User-defined datatypes

- User-defined datatypes can be used both in point-to-point communication and collective communication
- The datatype instructs where to take the data when sending or where to put data when receiving
  - Non-contiguous data in sending process can be received as contiguous or vice versa

Using user-defined datatypes

- A new datatype is created from existing ones with a datatype constructor
  - Several routines for different special cases
- A new datatype must be committed before using it

```
MPI_Type_commit(newtype)
```

  newtype the new datatype to commit

- A type should be freed after it is no longer needed

```
MPI_Type_free(newtype)
```

  newtype the old datatype that is freed

Datatype constructors

`MPI_Type_contiguous`	contiguous datatypes
`MPI_Type_vector`	regularly spaced datatype
`MPI_Type_indexed`	variably spaced datatype
`MPI_Type_create_subarray`	subarray within a multi-dimensional array
`MPI_Type_create_hvector`	like vector, but uses bytes for spacings
`MPI_Type_create_hindexed`	like index, but uses bytes for spacings
`MPI_Type_create_struct`	fully general datatype

MPI_TYPE_VECTOR

- Creates a new type from equally spaced identical blocks

```
MPI_Type_vector(count, blocklen, stride, oldtype,
                newtype)
```

  count number of blocks
  blocklen number of elements in each block
  stride displacement between the blocks

Example: sending rows of matrix in Fortran

```
integer, parameter :: n=2, m=3
real, dimension(n,m) :: a
integer :: rowtype
! create a derived type
call mpi_type_vector(m, 1, n, mpi_real, rowtype, ierr)
call mpi_type_commit(rowtype, ierr)
! send a row
call mpi_send(a, 1, rowtype, dest, tag, comm, ierr)
! free the type after it is not needed
call mpi_type_free(rowtype, ierr)
```

MPI_TYPE_INDEXED

- Creates a new type from blocks comprising identical elements
  - The size and displacements of the blocks may vary

```
MPI_Type_indexed(count, blocklens, displs,
                 oldtype, newtype)
```

count	number of blocks
blocklens	lengths of the blocks (array)
displs	displacements (array) in extent of oldtypes

count = 3
blocklens = (/2,3,1/)
disps = (/0,3,8/)

Example: an upper triangular matrix

```
/* Upper triangular matrix */
double a[100][100];
int  disp[100], blocklen[100], int i;
MPI_Datatype upper;
/* compute start and size of rows */
for (i=0; i<100; i++)
  {
    disp[i] = 100*i+i;
    blocklen[i] = 100-i;
  }
/* create a datatype for upper triangular matrix */
MPI_Type_indexed(100,blocklen,disp,MPI_DOUBLE,&upper);
MPI_Type_commit(&upper);
/* ... send it ... */
MPI_Send(a,1,upper,dest, tag, MPI_COMM_WORLD);
MPI_Type_free(&upper);
```

From non-contiguous to contiguous data

```
if (myid == 0)
  MPI_Type_vector(n, 1, 2,
                  MPI_FLOAT, &newtype)
  ...
  MPI_Send(A, 1, newtype, 1, ...)
else
  MPI_Recv(B, n, MPI_FLOAT, 0, ...)
```

```
if (myid == 0)
  MPI_Send(A, n, MPI_FLOAT, 1, ...)
else
  MPI_Type_vector(n, 1, 2, MPI_FLOAT,
                  &newtype)
  ...
  MPI_Recv(B, 1, newtype, 0, ...)
```

Performance

- Main motivation for using datatypes is rarely performance – manual packing is often faster
- Performance depends on the datatype - more general datatypes are often slower
- Overhead is potentially reduced by:
  - Sending one long message instead of many small messages
  - Avoiding the need to pack data in temporary buffers
- Performance should be tested on target platforms

Summary

- Derived types enable communication of non-contiguous or heterogeneous data with single MPI calls
  - Improves maintainability of program
  - Allows optimizations by the system
  - Performance is implementation dependent
- Life cycle of derived type: create, commit, free
- MPI provides constructors for several specific types

TYPEMAPS & DESCRIBING HETEROGENEOUS DATA

Understanding datatypes: typemap

- A datatype is defined by a typemap
  - pairs of basic types and displacements (in bytes)
  - E.g. MPI_INT={(int,0)}

Datatype constructors: MPI_TYPE_CREATE_STRUCT

- The most general type constructor, creates a new type from heterogeneous blocks
  - E.g. Fortran types and C structures
  - Input is the typemap

Datatype constructors: MPI_TYPE_CREATE_STRUCT

`MPI_Type_create_struct(count, blocklens, displs, types, newtype)`

count	number of blocks
blocklens	lengths of blocks (array)
displs	displacements of blocks in bytes (array)
types	types of blocks (array)

Example: sending a C struct

```
/* Structure for particles */
struct ParticleStruct {
   int charge;         /* particle charge */
   double coord[3];    /* particle coords */
   double velocity[3]; /* particle velocity vector components */ };

struct ParticleStruct particle[1000];
MPI_Datatype Particletype;

MPI_Datatype type[3]={MPI_INT, MPI_DOUBLE, MPI_DOUBLE};
int blocklen[3]={1,3,3};
MPI_Aint disp[3]={0, sizeof(double), 4*sizeof(double)};
...

MPI_Type_create_struct(3, blocklen, disp, type, &Particletype);
MPI_Type_commit(&Particletype);

MPI_Send(particle, 1000, Particletype, dest, tag, MPI_COMM_WORLD);
MPI_Type_free(&Particletype);
```

Determining displacements

- The previous example defines and assumes a certain alignment for the data within the structure
- The displacements can (and should!) be determined by using the function
  `MPI_Get_address(pointer, address)`
  - The address of the variable is returned, which can then be used for determining relative displacements

Determining displacements

```
/* Structure for particles */
struct ParticleStruct {
   int charge;         /* particle charge */
   double coords[3];   /* particle coords */
   double velocity[3]; /* particle velocity vector components */ };

struct ParticleStruct particle[1000];
...
MPI_Aint disp[3];
MPI_Get_address(&particle[0].charge, &disp[0]);
MPI_Get_address(&particle[0].coords, &disp[1]);
MPI_Get_address(&particle[0].velocity, &disp[2]);
/* Make displacements relative */
disp[2] -= disp[0];
disp[1] -= disp[0];
disp[0] = 0;
...
```

Note the address type - MPI_Aint in C, `integer(mpi_address_kind)` in Fortran

Gaps between datatypes

- Sending of an array of the `ParticleStruct` structures may have a portability issue: it assumes that array elements are packed in memory
  - Implicit assumption: the extent of the datatype was the same as the size of the C struct
  - This is not necessarily the case
- If there are gaps in memory between the successive structures, sending does not work correctly

TYPE EXTENT

Sending multiple elements: Extent

- Sending multiple user-defined types at once may not behave as expected
- The *lower bound* describes where the datatype starts
  - LB: $\min(disp_j)$
- The *extent* describes the stride at which contiguous elements are read or written when sending multiple elements
  - Extent = $\max(disp_j + size_j) - LB + padding$

Multiple MPI_TYPE_VECTOR

```
MPI_Type_vector(count = 3, blocklen = 2, stride = 3, oldtype, newtype)
```

```
MPI_Send(buffer, count = 1, newtype, dest, tag, comm)
```

```
MPI_Send(buffer, count = 2, newtype, dest, tag, comm)
```

Setting extent and lower bound

- Read current extent and lower bound

  `MPI_Type_get_extent(type, lb, extent)`

type	Datatype
lb	Lower bound of type (in bytes)
extent	Extent of type (in bytes)

- Set new extent and lower bound

  `MPI_Type_create_resized(type, lb, extent, newtype)`

type	Old datatype
lb	New lower bound (in bytes)
extent	New extent (in bytes)
newtype	New datatype, commit before use

Multiple MPI_TYPE_VECTOR

```
MPI_Type_vector(count = 3, blocklen = 2, stride = 3, oldtype, newtype)
MPI_Create_resized(newtype, lb = 0, extent = 9*sizeof(oldtype), n2type)
```

```
MPI_Send(buffer, count = 1, n2type, dest, tag, comm)
```

```
MPI_Send(buffer, count = 2, newtype, dest, tag, comm)
```

Example: sending an array of structs portably

```
struct ParticleStruct particle[1000];
MPI_Datatype particletype, oldtype;
MPI_Aint lb, extent;
...
/* Check that the extent is correct
MPI_Type_get_extent(particletype, &lb, &extent);
if ( extent != sizeof(particle[0] ) {
   oldtype = particletype;
   MPI_Type_create_resized(oldtype, 0, sizeof(particle[0]),
                           &particletype);
   MPI_Type_commit(&particletype);
   MPI_Type_free(&oldtype);
}
...
```

Other ways of communicating non-uniform data

- Non-contiguous data by manual packing
  - Copy data into or out from temporary buffer
  - Use MPI_Pack and MPI_Unpack functions
  - Performance will likely be an issue
- Structures and types as continuous stream of bytes: Communicate everything using MPI_BYTE
  - Portability can be an issue - be careful

```
struct ParticleStruct particle[1000];
int psize;
psize = sizeof(particle[0]);
MPI_Send(particle, 1000*psize, MPI_BYTE, ...);
```

SUBARRAY TYPE

MPI_TYPE_CREATE_SUBARRAY

- Creates a type describing an *N*-dimensional subarray within an *N*-dimensional array

  `MPI_Type_create_subarray(ndims, sizes, subsizes, offsets, order, oldtype, newtype)`

ndims	number of array dimensions
sizes	number of array elements in each dimension (array)
subsizes	number of subarray elements in each dimension (array)
offsets	starting point of subarray in each dimension (array)
order	storage order of the array. Either MPI_ORDER_C or MPI_ORDER_FORTRAN

Example: subarray

```
Rank 0: original array
  0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0
Rank 0: array after receive
  0.0   0.0   0.0   0.0   0.0
  0.0   0.0   1.0   1.0   1.0
  0.0   0.0   1.0   1.0   1.0
  0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0
```

```
int array_size[2]     = {5,5};
int subarray_size[2]  = {2,3};
int subarray_start[2] = {1,2};
MPI_Datatype subtype;
double array[5][5];

for (i=0; i<array_size[0]; i++)
  for (j=0; j<array_size[1]; j++)
    array[i][j] = rank;
MPI_Type_create_subarray(2, array_size, subarray_size, subarray_start,
                         MPI_ORDER_C, MPI_DOUBLE, &subtype);
MPI_Type_commit(&subtype);
if (rank==0)
  MPI_Recv(array[0], 1, subtype, 1, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
if (rank==1)
  MPI_Send(array[0], 1, subtype, 0, 123, MPI_COMM_WORLD);
```

Example: halo exchange with user defined types

- A solver requires a stencil of two cells in each direction to compute the next timestep
- The distributed 2D grid
  - Two-element ghost layers, each process has allocated a 8 x 8 grid patch
  - Each process computes a 4 x 4 patch of total system
  - Need to communicate 2x4 and 4x2 patches

Example: halo exchange with user defined types

Create datatype for sending and receiving in y direction

```
int array_size[2] = {4 + 2 + 2, 4 + 2 + 2};
int x_size[2] = {4,2};
int offset[2] = {0,0};
MPI_Type_create_subarray(2, array_size, x_size,
     offset, MPI_ORDER_C, MPI_DOUBLE,
     &x_boundary);
```

Example: halo exchange with user defined types

Create datatype for sending and receiving in y direction

```
int array_size[2] = {4 + 2 + 2, 4 + 2 + 2};
int y_size[2] = {2,4};
int offset[2] = {0,0};
MPI_Type_create_subarray(2, array_size, y_size,
     offset, MPI_ORDER_C, MPI_DOUBLE,
     &y_boundary);
```

Example: halo exchange with user defined types

```
//shift to the left
MPI_Sendrecv(&array[2][2], 1, x_boundary, left, tag_left,
             &array[2][6], 1, x_boundary, right, tag_right,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);

...
```

Example: halo exchange with user defined types

```
//shift to the right
MPI_Sendrecv(&array[2][4], 1, x_boundary, right,
tag_right,
             &array[2][0], 1, x_boundary, left,
tag_left,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
...
//shift up
//shift down
```

Summary

- User-defined types enable communication of non-contiguous or heterogeneous data with single MPI communication operations
  - Improves code readability & portability
  - Allows optimizations by the MPI runtime
- In the second part we
  - Introduced the concepts of extent and typemap
  - Discussed describing structures/types
  - Covered another advanced type constructor, `MPI_Type_create_subarray`

Debugging

What is debugging?

- Searching for programming mistakes (bugs) that cause wrong program behavior
  - Note that also dependencies (libraries) can have bugs
- Different approaches
  - Compiler checks
  - Source code reviews
  - Source code modifications, adding extra output or logs
  - **Debuggers**

Compiler checks

- Most compilers have some additional checks that can also be used for debugging
  - Usually not enabled for production because of memory consumption or performance overhead
- Two analysis types:
  - Static analysis
    - Diagnostics during compilation
  - Runtime analysis:
    - Compiler adds extra code that checks e.g. array indices and memory access

Compiler check examples

- Using GCC address sanitizer to add run-time checks for invalid memory references
  - `gcc -g -fsanitize=address myprog.c`
- Add array bounds run-time checks to Fortran program
  - `gfortran -g -fbounds-check myprog.f90`
  - `ftn –g –h bounds myprog.f90`
  - `ifort –g –check bounds myprog.f90`
- Check for accessing uninitialized variables
  - `icc -check uninit myprog.c`

Compiling for debugging

- There is no one-to-one mapping from C or Fortran to machine language
  - For example, variables are mapped to registers and memory locations
- By default the information that is needed for interpreting the program state back to a corresponding source code is not available
  - Can be added to the program code during compilation
  - Combining debugging information with optimization is problematic

Compiling cont.

- Compiler option that adds the debug information depends on the compiler, but usually adding option **-g** is enough
- Optimizations and debugging do not mix
  - Optimizations can change the program logic so that mapping from source to program code is not even possible
  - Compilers may generate faulty code with aggressive optimization flags
  - Finding bugs that are caused by compiler optimizations are often difficult to find

Optimizations and debugging

- GCC has special optimization flag that can be used together with debugging, **`-Og`**
  - Otherwise same as optimization level O1, but does not enable optimizations that make debugging problematic
- Cray toolset has a fast-track debugging, that enables debugging of fully optimized code
  - Compile with **`-Gfast`**
  - Requires support from both compiler and debugger

DEBUGGERS

What are debuggers?

- Debugger is a program that can be used to inspect other programs
  - Debuggers can stop program execution
    - Step-by-step execution is also possible
  - They can show (and set) values of variables
  - Machine language code is also accessible with low-level information on registers e.g.

Stopping execution, breakpoint

- Breakpoints are specified source code locations where the debugger stops the program execution so that the state can be inspected
  - Usually the breakpoints are set so that program is stopped before the wrong behavior occurs
- Breakpoints can be conditional, that is, they can depend on values of variables etc.

Stack frame and backtrace

- When a function is called, some of the arguments are stored in registers and other are put into the *stack*
  - Also local variables of the function are reserved from the stack
- It is possible to move between *stack frames* and check the values of the calling scope
- *Backtrace* is the full listing of all stack frames
  - Important information on who actually called the current function

Core files

- OS can be instructed to store the program memory space to disk if the program crashes
  - Often not enabled by default
  - Parallel programs can easily generate terabytes of data
- If the program code is available with debugging symbols together with memory dump one can inspect the program state at the time of crash

GDB

- GNU project debugger that is very widely available
- Supports multiple programming languages
- Support for many operating systems
- Command line tool, but many GUIs are available

PARALLEL DEBUGGING

Parallel debugging

- Parallel debugging is much more difficult than debugging serial programs
  - Many bugs are due to synchronization issues and may appear randomly
  - Often bugs appear when running with large number of tasks
    - Inspecting the status of thousands of tasks is challenging
  - Debuggers have to be integrated to the batch job system and/or MPI process launcher and there are many different alternatives

What to do when MPI program crashes?

- Try to get as much statistics as possible
  - Backtrace is very useful, use compiler options or other tools to generate it
- If not all tasks crash, and the program memory usage is not too large, try to generate some core files and inspect them using e.g. gdb
- Use a parallel debugger!

Parallel debuggers

- Unfortunately there are no good open-source parallel debuggers available
  - Eclipse Parallel Tools Platform, not a stand-alone debugger, very complicated installation
  - PGDB, command line debugger on top of gdb, project is not active
- There are few commercial parallel debuggers
  - ARM (prev. Allinea) DDT
  - RogueWave Totalview

Available tools at CSC

- Sisu and Taito both have Totalview available, maximum size of the program is 1500 tasks
- Sisu has Cray's command line parallel debugger, lgdb

Parallel debugging demo

- Without a parallel debugger
  - Backtraces, serial debugger, core files
- RogueWave Totalview demo

Jussi Enkovaara
Sami Ilvonen

HYBRID MPI+OPENMP PROGRAMMING

Hybrid MPI+OpenMP programming overview

Monday

13:00-13:45	Basic concepts
13:00-13:45	Exercises
13:45-14:15	Further considerations
14:15-14:30	Break
14:30-15:45	Exercises

BASIC CONCEPTS

Processes and threads

Process

- Independent execution units
- Have their own state information and *own address spaces*

Thread

- A single process may contain multiple threads
- Have their own state information, but share the address space of the process

Processes and threads

Process

- Long-lived: spawned when parallel program started, killed when program is finished
- Explicit communication between processes

Thread

- Short-lived: created when entering a parallel region, destroyed (joined) when region ends
- Communication through shared memory

Hybrid programming

Process (MPI)

- Independent execution units
- MPI launches N processes at application startup

Thread (OpenMP)

- Threads share *memory space*
- Threads are created and destroyed (*parallel regions*)

Hybrid programming:
Launch threads (OpenMP) *within* processes (MPI)

Hybrid programming

- Shared memory programming inside a node, message passing between nodes
- Matches well modern supercomputer hardware
- Often one MPI task / socket, but one should experiment with the ratio

Hybrid hello

```
int main(int argc, char *argv[]){
 int my_id, omp_rank;
 int required=MPI_THREAD_FUNNELED, provided;
 MPI_Init_thread(&argc,&argv,required,provided);
 MPI_Comm_rank(MPI_COMM_WORLD,&my_id);
#pragma omp parallel private(omp_rank)
 {
  omp_rank=omp_get_thread_num();
  printf("I'm thread %d in process %d\n",
         omp_rank, my_id);
 }
 MPI_Finalize();
}
```

```
> aprun -n 4 -d 4
./hybhello
I'm thread 0 in process 0
I'm thread 0 in process 1
I'm thread 2 in process 1
I'm thread 3 in process 1
I'm thread 1 in process 1
I'm thread 3 in process 0
I'm thread 1 in process 0
I'm thread 2 in process 0
I'm thread 0 in process 2
I'm thread 0 in process 3
I'm thread 1 in process 3
I'm thread 3 in process 3
I'm thread 2 in process 3
I'm thread 1 in process 2
I'm thread 3 in process 2
```

Thread support in MPI

Thread safe initialization

```
MPI_Init_thread(required, provided)
```

(argc, argv	Command line arguments in C)
required	Required thread safety level
provided	Supported thread safety level
error	Error value; in C/C++ it's the return value of the function, and in Fortran an additional output parameter

Pre-defined integer constants:

MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED < MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE

Hybrid programming styles: fine/coarse grained

- Fine-grained
  - Use `omp parallel do/for` on the most intensive loops
  - Possible to hybridize an existing MPI code with little effort and in parts
- Coarse-grained
  - Use OpenMP threads to replace MPI tasks
  - Whole (or most of) program within the same parallel region
  - More likely to scale over the whole node, enables all cores to communicate (if supported by MPI implementation)

Potential advantages of the hybrid approach

- Fewer MPI processes for a given amount of cores
  - Improved load balance
  - All-to-all communication bottlenecks alleviated
  - Decreased memory consumption if an implementation uses replicated data
- Additional parallelization levels may be available
- Possibility for dedicating threads for different tasks
  - e.g., dedicated communication thread or parallel I/O
  - Note that OpenMP worksharing constructs (e.g. OMP DO) will not be applicable when dedicating threads

Disadvantages of hybridization

- Increased overhead from thread creation/destruction
- More complicated programming
  - Code readability and maintainability issues
  - Synchronization
- Thread support in MPI and other libraries needs to be considered

Real-world hybrid applications

- Vlasiator – space weather simulations
  - Fluid in 6D spatial/velocity space: spatial grid with MPI, velocity grid with OpenMP
  - Lower memory footprint and better load balancing
- CP2K – electronic structure simulations
  - Especially so called hybrid functionals benefit from threading
  - Lower memory footprint allows more efficient computations

Real-world hybrid applications

- Vlasiator, 200 nodes (4800 cores) on Sisu

Threads per process	Hyperthreads per core	MPI ranks per node	Performance (10^8 cells/s)	Memory use per node (GB)
1	1	24	1.06	28.4
2	1	12	1.06	24.6
4	1	6	1.04	22.8
6	1	4	1.02	22.2
1	2	24	1.35	28.5
2	2	12	1.33	24.6
6	2	8	1.32	23.4
12	2	4	1.25	22.2

Summary

- Hybrid programming maps well to modern hardware
- In theory, hybrid programming offers several advantages
- In practice, all the advantages can be difficult to realize
- As number of cores inside a node increases, advantages of hybrid approach are likely to become more and more relevant
- MPI provides different levels of thread support

MORE ADVANCED CONCEPTS IN HYBRID PROGRAMMING

Multiple thread communication

- Hybrid programming is relatively straightforward in cases where communication is by only single thread at time
- With the so called multiple mode all threads can make MPI calls independently

```
int required=MPI_THREAD_MULTIPLE, provided;
MPI_Init_thread(&argc, &argv, required, &provided)
```

- When multiple threads communicate, the sending and receiving threads normally need to match
  - Thread-specific tags
  - Thread-specific communicators

Thread-specific tags

- In point-to-point communication the thread ID can be used to generate a tag that guides the messages to the correct thread

Thread-specific tags

- In point-to-point communication the thread ID can be used to generate a tag that guides the messages to the correct thread

```
!$omp parallel private(tid, tidtag, ierr)
tid = omp_get_thread_num()
tidtag = 2**10 + tid
! mpi communication to the corresponding thread on
! another process
call mpi_sendrecv(senddata, n, mpi_real, pid, tidtag, &
                  recvdata, n, mpi_real, pid, tidtag, &
                  mpi_comm_world, stat, ierr)

! ...
!$omp end parallel
```

Collective operations in the multiple mode

- MPI standard allows multiple threads to call collectives simultaneously
  - Programmer must ensure that the same communicator is not being concurrently used by two different collective communication calls at the same process
- In most cases, even with `MPI_THREAD_MULTIPLE` it is beneficial to perform the collective communication from a single thread (usually the master thread)
- Note that MPI collective communication calls do not guarantee synchronization of the thread order

Thread-specific communicators

```
! total number of openmp threads
nthr = omp_get_max_threads()
allocate(tcomm(nthr))
! split the communicator
do thrid=1,nthr
  col = thrid
  call mpi_comm_split(mpi_comm_world, &
       col, procid, tcomm(thrid), ierr)
end do
! collective threadwise communication
!$omp parallel private(tid, ierr)
tid = omp_get_thread_num()+1
! ...collective operations in tcomm(tid)
!$omp end parallel
```

- Collective calls do not have tag arguments
- Instead, we will generate thread-specific *communicators*

MPI thread support levels

- Modern MPI libraries support all threading levels
  - Cray MPI: Set **`MPICH_MAX_THREAD_SAFETY`** environment variable to `single`, `funneled`, `serialized`, `multiple` to select the threading level
  - Intel MPI: When compiling with `-qopenmp` a thread safe version of the MPI library is automatically used
- Note that using `MPI_THREAD_MULTIPLE` requires the MPI library to internally lock some data structures to avoid race conditions
  - May result in additional overhead in MPI calls

MPI+OpenMP thread affinity

- MPI library must be aware of the underlying OpenMP for correct allocation of resources
  - Oversubscription of CPU cores may cause significant performance penalty
- Additional complexity from batch job schedulers
- Heavily dependent on the platform used!

MPI+OpenMP thread affinity (Cray)

- Cray application launcher `aprun` fully supports interplay between MPI processes and OpenMP threads
- Some `aprun` options related to thread affinity
  - `-d <depth>`, reserve `depth` CPU cores for each MPI task
    - Recommended to be set equal to `OMP_NUM_THREADS`
  - `-cc <cpu|list|depth|numa_node|none>` sets CPU affinity for threads within each MPI task
    - Each thread can be pinned to a CPU core (`cpu`)
    - The threads may be set to float within a NUMA node (`numa_node`) or the –d reservation (`depth`). With `none` the affinity control is disabled.

MPI+OpenMP thread affinity (Intel)

- Intel MPI library allows affinity control of MPI domains via `I_MPI_PIN`, `I_MPI_PIN_DOMAIN` and `I_MPI_PIN_ORDER` environment variables
- Affinity control of the MPI library only applies to the placement of the MPI processes
  - `OMP_PLACES` (or `KMP_AFFINITY`) environment variables must be used to pin threads to cores within MPI domains

Summary

- Multiple threads may make MPI calls simultaneously
  - Thread specific tags and/or communicators
    - While waiting for the MPI-4 features
  - For collectives it is often better to use a single thread for communication
- Thread affinity matters
  - See manuals and user guides of the system if performance is not as expected

Jussi Enkovaara
Pekka Manninen

PARALLEL I/O

INTRODUCTION TO PARALLEL I/O

Parallel I/O

- How to convert internal data structures and domains to files that are essentially streams of bytes?
- How to get the data efficiently from thousands of nodes of a supercomputer to physical disks?
- Good I/O is non-trivial
  - Performance, scalability, reliability
  - Ease of use of output (number of files, format)
  - Portability
- One cannot achieve all of the above - one needs to prioritize

Parallel I/O

- Challenges
  - Number of tasks is rising rapidly
  - Size of the data is also rapidly increasing
  - Disparity of computing power vs. I/O performance is getting worse and worse
- The need for I/O tuning is algorithm & problem specific
- Without parallelization, I/O will become scalability bottleneck for practically every application!

I/O layers

I/O library choice

- First, POSIX and MPI I/O libraries will be discussed
  - Provides methods for writing raw data into files
  - Do not provide a schema or methods for writing metadata
  - The user has to develop a file format specification, or implement a given format
- Additionally there are also higher level libraries that
  - Give tools for writing data + metadata, e.g., HDF5
  - Provide an application or domain-specific schema for what data + metadata to describe some particular kind of information

PARALLEL I/O WITH POSIX

Parallel POSIX I/O

- Spokesman strategy
  - One process takes care of all I/O using normal (POSIX) routines
  - Requires a lot of communication
  - Writing/reading slow, single writer not able to fully utilize filesystem
  - Does not scale, single writer is a bottleneck
  - Can be good option when the amount of data is small (e.g. input files)

Example: spokesperson strategy

```
if (my_id == 0) then
  do i = 1, ntasks-1
    call mpi_recv(full_data(i*n), n, &
        MPI_REAL, i, tag, &
        MPI_COMM_WORLD, status, rc)
  end do
  open(funit, file=fname, access="stream")
  write(funit) full_data
  close(funit)
else
  call mpi_send(data, n, &
      MPI_REAL, 0, tag, &
      MPI_COMM_WORLD, rc)
end if
```

Special case: stdout and stderr

- Standard Output and Error streams are effectively serial I/O and will be a severe bottleneck for application scaling
- Disable debugging messages when running in production mode
  - "Hello, I'm task 32,000!"
- Ensure only the very minimum is written to the stdout/err!
  - Interim results, timings,…

Parallel POSIX I/O

- Every man for himself
  - Each process writes its local results to a separate file
  - Good bandwidth
  - Difficult to handle a huge number of files in later analysis
  - Can overwhelm filesystem (for example Lustre metadata)

MPI I/O

MPI-I/O

- MPI defines operations for reading and writing files
  - I/O to only one file and/or to many files
  - Contiguous and non-contiguous I/O
  - Individual and collective I/O
  - Asynchronous I/O
- Potentially good performance, easy to use (compared to implementing the same patterns on your own)
- Portable programming *interface*
  - By default, binary *files* are not portable

Basic concepts in MPI-I/O

- File *handle*
  - data structure which is used for accessing the file
- File *pointer*
  - *position* in the file where to read or write
  - can be individual for all processes or shared between the processes
  - accessed through file handle or provided as an explicit offset from the beginning of the file
  - Here we do not use shared file pointers – performance is poor

Opening & closing files

- All processes in a communicator open a file using
  `MPI_File_open(comm, filename, mode, info, fhandle)`

`comm`	communicator that performs parallel I/O
`mode`	MPI_MODE_RDONLY, MPI_MODE_WRONLY, MPI_MODE_CREATE, MPI_MODE_RDWR, …
`info`	Hints to implementation for optimal performance (No hints: MPI_INFO_NULL)
`fhandle`	parallel file handle

Can be combined with + in Fortran and | in C/C++

- File is closed using `MPI_File_close(fhandle)`

File writing

- Write file at explicitly defined offsets
  - Thread-safe
  - The file pointer is neither used or incremented

```
MPI_File_write_at(fhandle, disp, buffer, count,
                  datatype, status)
```

`disp`	displacement in bytes (with the default file view) from the beginning of file
`buffer`	buffer in memory to be written
`count`	number of elements to write
`datatype`	datatype of elements to write
`status`	similar to status in `MPI_Recv`, amount of data can be determined by `MPI_Get_count`

Example: parallel write

```
program output
 use mpi
 implicit none
 integer :: err, i, myid, file, intsize
 integer :: status(mpi_status_size)
 integer, parameter :: count=100
 integer, dimension(count) :: buf
 integer(kind=mpi_offset_kind) :: disp
 call mpi_init(err)
 call mpi_comm_rank(mpi_comm_world, myid,&
   err)
 do i = 1, count
  buf(i) = myid * count + i
 end do
...
```

Multiple processes write to a binary file 'test'. First process writes integers 1-100 to the beginning of the file, etc.

Example: parallel write

File offset determined explicitly

```
...
 call mpi_file_open(mpi_comm_world, 'test', &
      mpi_mode_create + mpi_mode_wronly, &
      mpi_info_null, file, err)
 intsize = sizeof(count)
 disp = myid * count * intsize
 call mpi_file_write_at(file, disp, buf, &
      count, mpi_integer, status, err)
 call mpi_file_close(file, err)
 call mpi_finalize(err)
end program output
```

File reading

- Read file at explicitly defined offsets
  - Thread-safe
  - The file pointer is neither referred or incremented

```
MPI_File_read_at(fhandle, disp, buffer, count,
                 datatype, status)
```

disp	displacement in bytes (with the default file view) from the beginning of file
buffer	buffer in memory where to read the data
count	number of elements to read
datatype	datatype of elements to read
status	similar to status in `MPI_Recv`, amount of data read can be determined by `MPI_Get_count`

Interim summary

- Primitive parallel I/O can be achieved using just normal Posix calls (+ MPI communication)
  - Spokesman strategy
  - Every man for himself
- MPI I/O: MPI library automatically handles all communication needed for parallel I/O access
  - File opening within a communicator
  - File writing / reading with explicit offsets

COLLECTIVE AND NON-BLOCKING MPI-IO

Collective operations

- I/O can be performed *collectively* by all processes in a communicator
  - `MPI_File_read_all`
  - `MPI_File_write_all`
  - `MPI_File_read_at_all`
  - `MPI_File_write_at_all`
- Same parameters as in independent I/O functions (MPI_File_read etc)

Collective operations

- All processes in communicator that opened file must call function
  - Collective operation
- If all processes write or read, you should always use the collective reads and writes.
  - Performance much better than for individual functions
  - Even if each processor reads a non-contiguous segment, in total the read may be contiguous
  - Lots of optimizations implemented in the MPI libraries

Non-blocking MPI-IO

- Non-blocking independent I/O is similar to non-blocking send/recv routines
  - `MPI_File_iread(_at)(_all)` / `MPI_File_iwrite(_at)(_all)`
- Wait for completion using MPI_Test, MPI_Wait, etc.
- Can be used to overlap I/O with computation:

NON-CONTIGUOUS DATA ACCESS WITH MPI-IO

File view

- By default, file is treated as consisting of bytes, and process can access (read or write) any byte in the file
- The *file view* defines which portion of a file is visible to a process
- A file view consists of three components
  - displacement: number of bytes to skip from the beginning of file
  - etype: type of data accessed, defines unit for offsets
  - filetype: portion of file visible to a process

File view

```
MPI_File_set_view(fhandle, disp, etype,
                  filetype, datarep, info)
```

disp	Offset from beginning of file. Always in bytes
etype	Basic MPI type or user defined type Basic unit of data access
filetype	Same type as etype or user defined type constructed of etype
datarep	Data representation (can be adjusted for portability) "native": store in same format as in memory
info	Hints for implementation that can improve performance `MPI_INFO_NULL`: No hints

The values for datarep and the extents of etype must be identical on all processes in the group; values for disp, filetype, and info may vary. The datatypes passed in must be committed.

File view for non-contiguous data

Decomposition for a 2D array

- Each process has to access small pieces of data scattered throughout a file
- Very expensive if implemented with separate reads/writes
- Use file type to implement the non-contiguous access

File view for non-contiguous data

Decomposition for a 2D array

Collective write can be over hundred times faster than the individual for large arrays!

```
...
integer, dimension(2,2) :: array
...
call mpi_type_create_subarray(2, sizes, subsizes, starts, mpi_integer, &
     mpi_order_c, filetype, err)
call mpi_type_commit(filetype)
disp = 0
call mpi_file_set_view(file, disp, mpi_integer, filetype, 'native', &
     mpi_info_null, err)
call mpi_file_write_all(file, buffer, count, mpi_integer, status, err)
```

Common mistakes with MPI I/O

✗ Not defining file offsets as `MPI_Offset` in C and integer (`kind=MPI_OFFSET_KIND`) in Fortran

✗ In Fortran, passing the offset or displacement directly as a constant (e.g., "0")
  - It has to be stored and passed as a variable of type `integer(MPI_OFFSET_KIND)`

Performance do's and don'ts

- Use collective I/O routines
- Remember to use the correct striping for each file size
  - Striping can also be set in the `MPI_Info` parameter using `MPI_Info_set` function:
    `MPI_Info_set(info, "striping_factor", "20")`
  - Optimal value system dependent, may be beneficial to benchmark
- Minimize metadata operations

Summary

- MPI-IO: MPI library automatically handles all communication needed for parallel I/O access
- File views enable non-contiguous access patterns
- Collective I/O can enable the actual disk access to remain contiguous

PARALLEL I/O LIBRARIES

I/O libraries

- How should HPC data be stored?
  - Large, complex, heterogeneous, esoteric, metadata ...
  - Parallel and random access
- Traditional relational databases poor fit
  - Cannot handle large objects
  - Many unnecessary features for HPC data
- MPI I/O is efficient but relatively low level

I/O libraries

- I/O libraries can produce files with standardized format
  - Portable files that can be manipulated with external software
- Typically, I/O libraries support also metadata
  - Self-describing files
- Parallel I/O is typically build on top of MPI I/O
- HDF5, NetCDF, ADIOS, SIONlib

HDF5

- A data model, library, and file format for storing and managing multidimensional data
- Can store complex data objects and meta-data
- File format and files are *portable*
- Possibility for parallel I/O on top of MPI-IO
- Library provides Fortran and C/C++ API
  - Third party interfaces for Python, R, Java
  - Many tools can work with HDF5 files (Paraview, Matlab, ...)
- The HDF5 data model and library are complex

HDF5 model

- Data can be organized hierarchically into groups in a tree and referred like in Unix filesystem
  - Data in root group: `/dataset1`
  - Data in the group *group_1*: `/group_1/dataset2`
- Root group is created automatically, other groups are created by the user
- Dataset stores multidimensional array of data elements
- Each object (group or dataset) can contain metadata
  - attributes = named values

Investigating HDF5 files

- HDF5 installation includes command line tools *h5ls* and *h5dump* for investigating contents of HDF5 files

```
$ h5ls example.hdf5
AtomicDensityMatrices    Dataset {1, 11946}
AtomicNumbers            Dataset {158}
…
CartesianPositions       Dataset {158, 3}
Dimensions               Group
…
```

```
$ h5dump -a Parameters/energyunit example.hdf5
…
DATA {
   (0): "Hartree«
…
```

NetCDF

- NetCDF is a set of data formats, programming interfaces, and software libraries that help read and write scientific data files
- Originally developed for the Earth Sciences community
- File formats are self-describing and platform independent
- Optimized for access to subsets of large datasets
- Version 4 implemented on top of HDF5
- Also separate "Parallel netCDF" library

NetCDF

- Data model is similar to HDF5
  - Data is stored in multidimensional datasets, which can be organized in a hierarchy
  - Metadata can be described in attributes
- Parallelization on top of MPI I/O

Other libraries

- ADIOS - Adaptable IO System
  - An I/O abstraction framework
  - Provides portable, fast, scalable, easy--to--use, metadata rich output
  - I/O strategy is separated from the user code
- SIONlib
  - Simple replacement for posix I/O with task local I/O
  - No standard file format or hierarchy
  - Potentially good performance

Summary

- I/O libraries are the recommended way for storing large data in production codes
- Portable files with standardized formats
  - No need to write application specific tools for investigating data
- Parallel I/O is normally built on top of MPI I/O
  - MPI I/O concepts are utilized also in libraries
- Many scientific domains have standard conventions build on top of I/O libraries

Jussi Enkovaara
Sami Ilvonen

APPLICATION PERFORMANCE

A day in life at CSC

CSC customer

I'm performing simulations with my Fortran code. It seems to perform much worse with MKL library in the new system than with IMSL library in the old system.

CSC specialist

Have you profiled your code?

No

A day in life at CSC

- Profiled the code: 99.9% of the execution time was being spent on these lines:

```
do i=1,n
  do j=1,m
    do k=1,fact(x)
      do o=1,nchoosek(x)
         where (ranktypes(:,:)==k)
            ranked(:,:,o)=rankednau(o,k)
         end where
      end do
    end do
  end do
end do
```

Removing these unnecessary loop iterations reduced the wall-time of one simulation run from 17 hours to 3 seconds…

MINDING PERFORMANCE

Improving application performance

- Obvious benefits
  - Better throughput => more science
  - Cheaper than new hardware
  - Save energy, compute quota etc.
- ..and some non-obvious ones
  - Potential cross-disciplinary research with computer science
  - Deeper understanding of application

Improving application performance

- Modern supercomputers are very complex (with evermore increasing complexity)
  - Multiple CPU cores within one socket
  - Superscalar out-of-order instruction execution
  - Multilevel coherent caches
  - SIMD vector units
  - SMT capabilities for multithreading
  - Complex communication topologies
- To get most out of the hardware, performance engineering is needed

Performance engineering

- Adapting the problem to the underlying hardware
- Key factors to application performance
  - Effective algorithms, doing things in a more clever way
    - e.g. $O(n \log(n))$ vs $O(n^2)$
  - Efficient memory access
  - High CPU cycle utilization
  - Parallel scalability
  - Efficient I/O

Performance engineering

- Important to understand interactions
  - Algorithm – code – compiler – libraries – hardware
- Performance is not portable
- Optimize *only* the parts of code that are relevant for the total execution time!
  - Typically small part of the code (~10 %) consumes most (~90 %) of the execution time

Compute bound vs. memory bound

- CPU bound: performance is limited by floating point operations CPU can execute
  - No waiting for data
  - Limit is theoretical peak performance
- Memory bound: performance is limited by how fast CPU can fetch data from memory
  - CPU performance has been increasing more rapidly than memory bandwidth
  - Limit is memory bandwidth
  - Optimized memory access is critical to performance

Memory hierarchy

SIMD vectorization

- SIMD instructions operate on multiple elements at one cycle
- AVX/AVX2: 256 bits:
  - 4 DP values
  - 8 SP values
  - AVX2 brings FMA
- AVX512: 512 bits
  - 8 DP values
  - 16 SP values

```
double * A, * B, * C;
int i, N;
/* C loops */
for (i=0; i<N; i++)
   C[i]=B[i]+A[i];
```

Code optimization cycle

Selecting the test case

- Test case should represent a real production run
- Measurements should be carried out on the target platform
  - "Toy" run on laptop is in most cases useless
- Measurements should be done on both sides of scalability limit
  - Rule of thumb: doubling number of CPUs increases performance less than factor of 1.5

ON COMPILERS AND PERFORMANCE

Optimal porting

- "Improving application performance without touching the source code"
- Potential to get significant performance improvements with little effort
- Should be revisited routinely
  - Hardware, OS, compiler and library upgrades
  - Can be automated

- Compilers
- Compiler flags
- Numerical libraries
- Intranode placement
- Internode placement
- Filesystem parameters

Choosing a compiler

- Many different options
  - GNU, PGI, Intel, Cray, etc.
- Compatibility
  - Different proprietary intrinsics
  - Different rounding rules
- Performance
  - There is no universally fastest compiler
  - Depends on the application or even input

Compiler optimization techniques

- Architecture-specific tuning
  - Tunes all applicable parameters to the defined microarchitecture
- Vectorization
  - Exploiting the vector units of the CPU (AVX etc.)
  - Improves performance in most cases
- Loop transformations
  - Fusing, splitting, interchanging, unrolling etc.
  - Effectiveness varies

Compiler flag examples

Feature	Cray	Intel	GNU
Reporting	-hlist=a	-qopt-report=3	-fopt-info-vec
Balanced Optimization	(default)	-O2	-O3
Aggressive Optimization	-O3,fp3	-Ofast	-Ofast –funroll-loops
Architecture specific tuning	-h cpu=<target>	-x<target>	-march=<target>
Fast math	-h fp3	-fp-model fast=2	-ffast-math

Doesn't the compiler do everything?

- You can make a big difference to code performance with how you express things
  - Helping the compiler spot optimisation opportunities
  - Using the insight of your application
  - Removing obscure (and obsolescent) "optimizations" in older code
    - Simple code is the best, until otherwise proven
- This is a dark art, mostly: optimize on case-by-case basis
  - First, check what the compiler is already doing

Take-home messages

- Mind the application performance: it is for the benefit of you, other users and the service provider
- Profile the code and identify the performance issues first, before optimizing anything
  - "Premature code optimization is the root of all evil"
- Quite often algorithmic or intrusive design changes are needed to improve parallel scalability
- Serial optimization is mostly about helping the compiler to optimize for the target CPU

How to start?

- What limits the performance?
  - Serial / OpenMP (single node performance)
  - MPI (internode performance)
  - I/O
- Intel Performance Snapshot can provide big picture for further analysis
  - Other possible tools: gprof, TAU, scalasca, CrayPAT

SINGLE NODE PERFORMANCE ANALYSIS

Performance analysis

Introduction

- Don't speculate about performance – measure it!
- Performance analysis tools help to
  - Find hot-spots
  - Identify the cause of less-than-ideal performance
- Tools covered here
  - Intel VTune Amplifier
- Other tools
  - Perf, gprof, CrayPAT, Tau, Scalasca, PAPI…
  - http://www.vi-hps.org/tools/tools.html

Profiling application

- Collecting all possible performance metrics with single run is not practical
  - Simply too much information
  - Profiling overhead can alter application behavior
- Start with an overview!
  - Call tree information, what routines are most expensive?

Sampling vs. tracing

- When application is profiled using sampling, the execution is stopped at predetermined intervals and the state of the application is examined
  - Lightweight, but may give skewed results
- Tracing records every event, e.g. function call
  - Usually requires modification to the executable
    - These modifications are called *instrumentation*
  - More accurate, but may affect program behavior
  - Generates lots of data

Hardware performance counters

- Hardware performance counters are special registers on CPU that count hardware events
- They enable more accurate statistics and low overhead
  - In some cases they can be used for tracing without any extra instrumentation
- Number of counters is much smaller than the number of events that can be recorded
- Different CPUs have different counters

Intel VTune

- VTune is a tool that can give detailed information on application resource utilization
  - Uses CPU hardware counters on Intel CPUs for more accurate statistics
- VTune has extensive GUI for result analysis and visualization

VTune

- Analysis in three steps
  1. **Collect:** Run binary and collect performance data – sampling based analysis
  2. **Finalize:** Prepare data for analysis – by default combined with collect
  3. **Report:** Analyze data with VTune

VTune

- In addition to the GUI, command-line tools can be used to collect the statistics
  - Works with batch jobs too
- Many different profiles (actions), for example
  - *hotspots* for general overview
  - *advanced-hotspots* for more detailed view with hardware counters
  - *hpc-performance* for HPC specific analysis
  - *memory-access* for detailed memory access analysis

VTUNE DEMO

MPI PERFORMANCE ANALYSIS

Performance analysis

Introduction

- Finding out the scalability bottlenecks of parallel application is non-trivial
- Bottlenecks can be very different with few CPUs than with thousands of CPUs
- Performance analysis needs to be carried in scalability limit with large enough test case
- Efficient tools are needed for the analysis of massively parallel applications

Potential scalability bottlenecks

- Bad computation to communication ratio
  - In 2d heat equation with one dimensional parallelization $\frac{T_{comp}}{T_{comm}} \sim \frac{N}{p}$
- Load imbalance
- Synchronization overhead
- Non-optimal communication patterns

Common inefficient communication patterns

Late sender

Late reducer

Common inefficient communication patterns

Wait at NxN

Early reduce / gather

MPI performance analysis

- Many tools can automatically identify common problematic communication patterns
- Flat profile
  - Time spent in MPI calls during the whole program execution
- Trace
  - Also the temporal profile of MPI calls
  - Potentially huge data

DEMO: INTEL TRACE ANALYZER

Summary

- Many tools can provide information about MPI performance problems
  - Manual investigation impossible in massively parallel scale
- Problems often caused by load imbalance or by badly designed communication pattern

Optimization overview

- Optimization is challenging topic
- Many technical low-level details
  - Often very architecture and problem specific
- Lots of trial and error
- Do the basics right
  - Use efficient algorithms
  - Use well-known libraries (BLAS, LAPACK, FFTW, …)
  - Test different compilers and compiler options
  - Avoid unnecessary I/O

More detailed information

- https://events.prace-ri.eu/event/718/material/slides/
- http://www.agner.org/optimize/
- https://pop-coe.eu/further-information/learning-material
- http://www.prace-ri.eu/best-practice-guides/

Seija Sirkiä

TOOLS FOR DATA ANALYSIS AND VISUALIZATION

Data science and visualization

What is data science?

- Not just statistics rebranded
- Not just Big Data hype
  - NB! Big data techniques are real, not hype
- Interdisciplinary field with a strong emphasis on communication and teamwork
- Emerged with the data deluge

By Calvin.Andrus - Ow n work, CC BY-SA 3.0, https://commons.w ikimedia.org/w/index.php?curid=20254353

Data science and HPC

- Many problem types in data science benefit nicely from parallel computing
  - matrix operations
  - K-means clustering
  - pairwise operations, graphs and adjacency matrices, sorting
  - MCMC
  - *chunk averaging*
  - and deep learning and GPUs

Apache Spark

- Apache Spark is a general-purpose cluster computing system
  - Spark SQL (for SQL and structured data) and MLlib (for machine learning) make it a high level framework for large scale data analysis
- Key concept is the RDD, resilient distributed dataset
  - fault tolerant
  - support *transformations* (like maps) and *actions* (like reduces)
- Question: Spark vs. MPI, which one wins?

The data in data science

- Data science commonly deals with "n observations, p variables" format of tabular ("rectangular") data, known as *data frames*
  - CSV (or any CSV-like) is a well suited file format for storing such data (when small)
- Other kinds of data and formats:
  - JSON, NetCDF, SQL
  - GIS
  - images, sound, video
  - trees, graphs

- An example of a data frame viewed in RStudio:

mpg

Filter

	manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
1	audi	a4	1.8	1999	4	auto(l5)	f	18	29	p	compact
2	audi	a4	1.8	1999	4	manual(m5)	f	21	29	p	compact
3	audi	a4	2.0	2008	4	manual(m6)	f	20	31	p	compact
4	audi	a4	2.0	2008	4	auto(av)	f	21	30	p	compact
5	audi	a4	2.8	1999	6	auto(l5)	f	16	26	p	compact
6	audi	a4	2.8	1999	6	manual(m5)	f	18	26	p	compact
7	audi	a4	3.1	2008	6	auto(av)	f	18	27	p	compact
8	audi	a4 quattro	1.8	1999	4	manual(m5)	4	18	26	p	compact
9	audi	a4 quattro	1.8	1999	4	auto(l5)	4	16	25	p	compact
10	audi	a4 quattro	2.0	2008	4	manual(m6)	4	20	28	p	compact

Showing 1 to 10 of 234 entries

Tidy data

- Tidy data (Codd's 3NF):
  1. Each variable must have its own column
  2. Each observation must have its own row

  (3. Each value must have its own cell)

subject	temp90	temp95
ctrl	0.91	0.22
trt	0.19	0.44

subject	temp	score
ctrl	90	0.91
trt	90	0.19
ctrl	95	0.22
trt	95	0.44

- Specifically, you do not want to have data values in variable names!

R vs. Python (or something else?)

- There is no clear winner between the two
- Relevant Python libraries: Pandas, NumPy, SciPy, matplotlib, scikit-learn
- Relevant R packages: all of tidyverse, particularly ggplot2
- Others to consider:
  - 3d visualization: VisIt, ParaView, MayaVi
  - Tableau and Trifacta
  - Matlab, SAS, Excel etc. are not dead! Julia is an option too

DATA VISUALISATION

Why visualize data?

- "The purpose of visualization is insight, not pictures." (Ben Shneiderman)
- "You can see a lot by just looking." (Yogi Berra)
- Humans are obviously visual creatures

Two famous examples (plus one)

- John Snow and the London cholera epidemic https://public.tableau.com/en-us/s/gallery/mapping-1854-cholera-outbreak
- Florence Nightingale in the Crimean war https://beta.sciencemuseum.org.uk/stories/2016/11/4/florence-nightingalethe-pioneer-statistician
- (Almost unrelated bonus: https://en.wikipedia.org/wiki/James_Lind)

Some distinctions in visualisation

- Exploration vs. communication
- *data* visualization vs. *scientific* visualization
- 3d visualization of surfaces
- maps
- animations
- infographics

How do you make good visualizations?

- Is it about knowing the correct graph type?
- If not right, is it about being able to name a lot of graph types?
- Is it knowing which visualization tool is the best and how to use it?

Albert Cairo: The truthful art

- Five qualities of great visualizations
  - Truthful
  - Functional
  - Beautiful
  - Insightful
  - Enlightening

You are telling a story

- The most important aspect of creating good, effective visualizations is knowing which story you are telling
  - and to **who**, **what** you want them to do with the story and **how** are you going to tell it
- "Compared to what, to whom, to when, to where...?"
  - part to whole; new to old; levels of a category to each other; single level to the rest; ...
- Context is important

Example: adding depth and breadth

From Cairo: The truthful art

- "Study finds more than a quarter of journalism grads wish they'd chosen another career."
  - compared to when? Years 1999 to 2012 present in the study, with not much change
  - compared to whom? No data on other majors
  - context? Wages and job opportunities in journalism?
- "Even if job prospects for journalists have worsened substantially and they may worsen even further in the future, the percentage of grads who wish they'd chosen another career hasn't changed at all in more than a decade."

Pre-attentive features and gestalt laws

- Some visual features are **pre-attentive**: they are processed without conscious effort, you *just see* it
  - such as size, length, shape, orientation, color hue and intensity, texture, grouping...
- Some visual features help to perceive a collection as a whole: these are known as **gestalt principles**
  - proximity, similarity, enclosure, closure, continuity, connection (and then some)

Visual encoding of data

- Cairo: "There is a time in every class and workshop when someone raises her hand and asks: **How do you know that you have chosen the right graphic form to represent your data?** --- I invariably reply, 'I have no idea, but I can give you some clues to make your own choices based on what we know about why and how visualization works."

Grammar of graphics

- Leland Wilkinson: The grammar of graphics (book)
- Hadley Wickham: The layered grammar of graphics (article)
- gpplot2 (an R package, part of *tidyverse*)
- the key idea in all of these:

  When you make a data visualisation, you are mapping variables in your data to graphical elements in your plot

Graphical features, Cleveland's hierarchy

- position, length, angle, slope, area, volume, curvature, shading, color saturation, color hue
  - (does this list sound familiar by now?)
  - ordered here roughly according to Cleveland: the first ones allow for more accurate and faster estimates
  - addition by Cairo: the latter ones may be better able to reveal general patterns

Beautiful: colors and decluttering

- Making your plots beautiful will lessen the cognitive load your audience will have to endure
- For colors, please consider using a tool rather than picking them by hand
  - colorbrewer2.org is great
- Decluttering:
  - do you really need grid lines? strong axes? axes at all? tickmarks?
  - 3D?? no, you don't (unless you do!)

Are pie charts always bad?

Jussi Enkovaara
Joonas Nättilä

HPC APPLICATION DESIGN

HPC application design

HPC application development

- How does one design (HPC) applications?
  - Why?
- What is (HPC) software design all about?
  - *<insert your field of science>* + computer sciences
  - *Programming*!
  - Collaborating
  - Parallellization & optimization
  - Planning & thinking ahead

Why?

- Do science and publish the results
  - hard-core scientific articles
  - method-oriented articles presenting novel algorithms, numerical methods etc. (+ code)
- Code as a product
  - prestige and fame
  - gateway into projects, collaborations, etc.
  - citations, co-authorships
  - work on the bleeding edge!

To who?

- Users (= customers)
  - you
  - your group
  - collaborators
  - community
- Different degrees of openness
  - open source vs. licensed
- The more people are using the code, the more citations (or money) you get!

By who?

- Most developers have little formal software engineering training
  - learn-by-doing
  - intensive courses & summer schools like this
- Different team sizes and thus roles
  - used and developed by *one scientist*
  - used and developed by a *single group*
  - larger codes often developed in collaboration by *multiple groups* and individuals
  - open source code supports *community*-driven development

What?

- What are you actually trying to do?
  - solve a particular problem?
  - write a more general code or a library?
- Expected lifetime: one time vs. eternity
  - beware: quick-and-dirty prototypes tend to live on for a long time….
- Not only code but also the input and output: *data*!
  - in what format is the data? is the format documented?
  - should any metadata be stored as well?
  - is it really worth anything after extracting the relevant metrics?

Main question: How?

- Are scientific programs fundamentally different from other pieces of software?
  - No, but there are some special considerations…
- Let's learn from computer sciences - they do this for a living!
- There are several software design processes that one can consider, e.g.:
  - "Just code"
  - Waterfall model
  - Agile method

"Just code"

- No process is used – typically the program is developed by one person simply starting to code

Why bother with subroutines when you can type fast?
—Vaughn Rokosz

Weeks of programming can save you hours of planning.
—unknown

"Just code"

- Benefits
  - little overhead, no formalitites
- Drawbacks
  - testing often done too late, costly fixes
  - the structure of the program grows organically
    - costly restructuring or unclear structure
  - selected models and algorithms are easily not the best ones available (or even outright bad)
  - collaboration difficult, often only one developer

Waterfall model

- Waterfall model is one of the original application development models
- All stages assumed to be complete, before moving on to the next one

Waterfall model

- Benefits:
  - good planning early on can help to avoid costly mistakes
- Drawbacks:
  - often impossible to plan perfectly, are all facts even known?
  - rigid; does not lend itself to fast reaction to new models, ideas
  - how can we publish during the process?

Agile software development

- An umbrella term for different software development methods
  - divide work into small tasks, define a short working period (1-2weeks) for completing the tasks, review, repeat
  - software requirements and solutions evolve through collaborative efforts of teams
- Focused on iterative and incremental development
  - quick prototyping
  - supports continuous publications
  - analysis, coding, design, and testing never end

Choosing a development model

- Development model should aid in reaching these goals:
  - Productivity
  - Performance (see previous lecture about Application performance)
  - Correctness (see next lecture about software engineering)
  - Novel results and impact

Goals of application design: Productivity

- Productivity
  - leverage existing libraries, models, frameworks
  - proper development tools
  - choosing correct programming language
  - development best practices – more on next lecture
  - aim for portability
  - aim for maintainability
    - clear code, comments, documentation (Doxygen!)

Productivity: Leverage existing software

- Libraries
  - numerical - BLAS, solvers
  - I/O
  - parallelization
- Models
  - physical models for processes
- Frameworks
  - application where you can plug in your model
  - couplers that can be used to bind different models
  - Petsc, Trilinos

Productivity: Leverage existing software

- Minimizes development time – somebody has already spent a lot of time to implement, validate and optimize a solution to your problem
  - *Avoid the "Not Invented Here" syndrome!*
- Some caveats:
  - is the library still supported/updated?
  - do you trust the source, is it widely used?
  - is there documentation?
  - does it support all features you need?

Productivity: Choosing a programming language

- Most common are C, C++ and Fortran
  - mostly a question of taste
  - C++ more full featured with object oriented features and many more data structures (maps, etc.)
  - Fortran has really good array syntax
- One should also consider Python
  - much faster coding cycle (and less error prone)
  - testing, debugging, and prototyping is much easier

Productivity: Choosing a programming language

- Different languages can be interfaced together
  - Best of both worlds
  - Low-level languages (C,C++, Fortran) for costly functions
  - High-level languages (Python, Julia, R,…) for main function
- Possible combinations
  - Python & C++ (pybind11) for object-oriented programming
  - Julia & Fortran/C (native binding)

Productivity: Working with an existing code

- Language selection is already done for you
- Instead of rewriting the whole code, try to understand what every main-level function does
- Sometimes it might be good to modernize an existing code
  - improved performance
  - deeper understanding of the code & methods
  - better extendability

Productivity: Working with an existing code

- Possible method for modernization:
  - Replicate the driver/main loop using a high-level language
  - Interface the existing functions and routines into it
  - start rewriting & redesigning old functions that you really need to focus on
- This way you will always have a working version of the code

Productivity: Community tools

- Community & collaboration tools are important
  - Git, svn…
- Open (and closed) repositories
  - Github, Gitlab, Bitbucket
    - issues
    - pull requests
    - wikispace
    - free webpage/manual for the project (gh-pages)
    - project boards for coordination

Productivity: Community tools

- Project chats
  - *Slack*, *Ryver* (free), *FlowDock*, …
  - IRC (free)
  - *Zapier* (free) can connect your repository and chat platform
    - Automatic updates of commits, issues, pull requests,…

Novel results and impact: Licensing model

- Open source
  - positive aspects
    - The Right Thing To Do™ – in the spirit of science
    - can open up collaborations, get users involved, make code into an impactful well-known code embraced by the community
    - easier to get help from PRACE and other such resources
    - Funding agencies (e.g., ERC) give extra points from this
  - on the negative side
    - some loss of control
    - do you lose your investment?

Novel results and impact: Licensing model

- Closed source
  - more control, but less chance of the code being widely used
  - can act as a gateway into projects

Summary

- Learn from computer scientists
  - adopt a clear code development model and their tools!
- Productivity
  - use existing libraries
  - use & adopt community and collaboration tools
  - think about programming language selection
- Novel results and impact
  - open source vs. closed source

GOOD SOFTWARE ENGINEERING PRACTICES

Three most important practices

- Rule 1: always use version control
- Rule 2: even if developing only by yourself, use version control
- Rule 3: if still in doubt, use version control

Outline

- Coding style
- Testing

Coding style

- An arbitrary set of rules on how the code should look like
  - aim is that multiple coders are together able to produce consistent program code
- A compromise that maybe only one coder (the main one) will be completely happy with!

www.gromacs.org/Developer_Zone/Programming_Guide

Coding style

- Coding style guides often contain rules for
  - naming of variables, functions and classes
  - indentation
  - white spaces in expressions
  - comments and documentation
- Good coding style helps developers:
  - easier to maintain the code
  - easier to understand the code
    - especially important to new developers!

Coding style: naming

- Use meaningful identifiers for names of variables and functions

```
a = b / c
```

vs.

```
speed = distance / time
```

- Use underscores or capital letters in long names to make the code more readable (e.g. `average_speed`, `DensityMatrix`)
- Consistency and clarity should be the leading factors!

Coding style: indentation

- Proper indentation and separation of code blocks can enhance readability dramatically

```
foo = 4.0
bar = 5.2
function add(a, b)
result = a + b
return result
end function
add(foo, bar)
```

vs.

```
foo = 4.0
bar = 5.2

function add(a, b)
   result = a + b
   return result
end function

add(foo, bar)
```

Coding style: comments

- Comments should be used to make it easier to understand the intended purpose of the code

```
# Reduce point group using operation matrices
for op_cc in self.op_scc:
    a_a = np.zeros(len(spos_ac), int)
    ok = True
    …
```

- Especially non-obvious things should be commented
- All comments are not useful

```
! Make addition
a = b + c
```

Coding style: comments

- Always document the function of a function
  - usually the input and output parameters should also be documented

```
def __init__(self, beta=0.1, nmaxold=3, weight=50.0):
        """Construct density-mixer object.

        Parameters
        ----------
        beta: float
            Mixing parameter between zero and one (one is most aggressive).
        nmaxold: int
            Maximum number of old densities.
        weight: float
            Weight parameter for special metric (for long wave-length changes).
        """
```

Coding style: comments

- Software development tools such as Doxygen can generate documentation automatically from comments

```
!> Initialize the Game of Life board
!! @param board_1 actual board
!! @param board_2 temporary board
!! @param height  height of the board
!! @param width   width of the board
subroutine init_board(board_1, board_2,
                      height, width)
 integer,dimension(:,:),pointer :: board_1
...
```

Test and verify

- Scientific software depend in most cases on numerical solutions to mathematical models
  - most important aspect is to get accurate results
- Practical aspects to consider:
  - accuracy of the numerical solution
  - software implementation
  - performance

Test and verify: numerical solution

- Many mathematical models have an analytical solution in special cases
  - solve special case numerically and compare to a known analytical result
- Symmetry considerations
- Conservation laws
  - energy, momentum, angular momentum, ...
- Comparison to previous numerical results
  - literature, other codes

Test and verify: unit testing

- Tests for smallest testable part of an application
  - Function or module
  - Class in object oriented programming
- Ideally, tests are independent of each other
- Frameworks for unit testing: cUnit, fUnit (xUnit family)
- Some languages have built-in support (Python)
- Client code which executes tests can be also hand-written

Test and verify: unit testing

- *Code coverage* : a metric how large part of the application is tested
  - Percentage of functions
  - Percentage of source code lines
- Test driven development
  - Tests are written before actually implementing the functionality

Test and verify: integration testing

- Combining individual software modules and testing as group
  - “trivial” changes in code can have surprising side effects
  - Example: testing a specific solver which utilizes several modules
  - At the highest level tests the whole application
- Recommended to be combined with unit testing

Test and verify: regression testing

- Uncover new errors in existing functionality after changes
- Bugs have a bad habit of lurking back!
  - When fixing a bug, it is a good practice to write a test that exposes it

Test and verify: challenges with HPC

- Behavior can depend on the number of threads and MPI tasks
  - Parallel components should be tested with multiple different parallelization schemes
- Large scale integration tests can be time consuming
- Changes in program code may also lead to degradation in performance and scalability
  - tests should track also the performance
- Performance is often system/architecture specific
  - preferably test on multiple architectures

Continuous integration

- Automatic testing
  - build test ensures that code is always in a "working state"
  - Unit and integration tests can be run automatically after build succeeds
- Nightly / weekly tests
  - Software is built and tested at specified times
- Test at "commit"
  - Test set is run when new code is pushed to main repository
- Nightly and "commit" tests can exclude time consuming tests

Continuous integration

- Test system should send a notification when test fails
  - Mailing list, IRC, Flowdock, …
  - Test status can be shown also in www-page
- Tools for continuous integration:
  - TravisCI
    - widely used, nice integration to Github
  - Jenkins
    - support for complex automation tasks
  - GitlabCI

Continuous integration

Summary

- Consistent coding style makes it easier to understand and maintain the code
- Testing should be essential part of the software development
  - Use automated build and test framework if possible!
- Version control, version control, version control
- Further reading
  http://coderefinery.org/lessons/

SHORT MPI REFERENCE

MPI FUNDAMENTALS

C interfaces for basic MPI operations

```
int MPI_Init(int *argc, char **argv)
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Barrier(MPI_Comm comm)
int MPI_Finalize()
```

Fortran interfaces for basic MPI operations

```
mpi_init(ierror)
mpi_comm_size(comm, size, ierror)
mpi_comm_rank(comm, rank, ierror)
mpi_barrier(comm, ierror)
mpi_finalize(ierror)
integer :: comm, size, rank, ierror
```

MPI datatypes

MPI type	C type
MPI_CHAR	signed char
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	

MPI datatypes

MPI type	Fortran type
MPI_CHARACTER	character
MPI_INTEGER	integer
MPI_REAL	real
MPI_REAL8	real*8 (nonstandard)
MPI_DOUBLE_PRECISION	double precision
MPI_COMPLEX	complex
MPI_DOUBLE_COMPLEX	double complex
MPI_LOGICAL	logical
MPI_BYTE	

POINT-TO-POINT COMMUNICATION

C interfaces for point-to-point operations

```
int MPI_Send(void *buffer, int count, MPI_Datatype datatype,int dest,
  int tag, MPI_Comm comm)
int MPI_Recv( void *buf, int count, MPI_Datatype datatype, int source,
  int tag, MPI_Comm comm, MPI_Status *status)
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
  int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype
  recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
```

Fortran interfaces for point-to-point operations

```
mpi_send(buffer, count, datatype, dest, tag, comm, ierror)
  type :: buf(*)
  integer :: count, datatype, dest, tag, comm, ierror
mpi_recv(buf, count, datatype, source, tag, comm, status, ierror)
  type :: buf(*)
  integer :: count, datatype, source, tag, comm, ierror
  integer :: status(MPI_STATUS_SIZE)
mpi_sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
  recvcount, recvtype, source, recvtag, comm, status, ierror)
  type :: sendbuf(*), recvbuf(*)
  integer :: sendcount, sendtype, dest, sendtag, recvcount, recvtype,
  source, recvtag, comm, ierror
  integer :: status(MPI_STATUS_SIZE)
```

COLLECTIVE COMMUNICATION

C interfaces for collective operations

```
int MPI_Bcast(void* buffer, int count, MPI_datatype datatype, int root,
  MPI_Comm comm)
int MPI_Scatter(void* sendbuf, int sendcount, MPI_datatype sendtype,
  void* recvbuf, int recvcount, MPI_datatype recvtype, int root,
  MPI_Comm comm)
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
  MPI_datatype sendtype, void* recvbuf, int recvcount, MPI_datatype
  recvtype, int root, MPI_Comm comm)
int MPI_Gather(void* sendbuf, int sendcount, MPI_datatype sendtype,
  void* recvbuf, int recvcount, MPI_datatype recvtype, int root,
  MPI_Comm comm)
int MPI_Gatherv(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void
  *recvbuf, int *recvcnts, int *displs, MPI_Datatype recvtype, int root,
  MPI_Comm comm)
```

C interfaces for collective operations

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype
  datatype, MPI_Op op, int root, MPI_Comm comm)
int MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype
  datatype, MPI_Op op, MPI_Comm comm)
int MPI_Allgather(void* sendbuf, int sendcount, MPI_datatype sendtype,
  void* recvbuf, int recvcount, MPI_datatype recvtype, MPI_Comm comm)
int MPI_Reduce_scatter(void* sendbuf, void* recvbuf,  int* recvcounts,
  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_datatype sendtype,
  void* recvbuf, int recvcount, MPI_datatype recvtype, MPI_Comm comm)
int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls,
  MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *rdispls,
  MPI_Datatype recvtype,MPI_Comm comm)
```

Fortran interfaces for collective operations

```
mpi_bcast(buffer, count, datatype, root, comm, ierror)
  type :: buffer(*)
  integer :: count, datatype, root, comm, ierror
mpi_scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
  root, comm, ierror)
  type :: sendbuf(*), recvbuf(*)
   integer :: sendcount, recvcount, sendtype, recvtype, root, comm,
          ierror
mpi_scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
  recvtype, root, comm, ierror)
  type :: sendbuf(*), recvbuf(*)
  integer :: sendcounts(*), displs(*), recvcount, sendtype,
          recvtype, root, comm, ierror
```

Fortran interfaces for collective operations

```
mpi_gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
  root, comm, ierror)
  type :: sendbuf(*), recvbuf(*)
  integer :: sendcount, recvcount, sendtype, recvtype, root, comm,
          ierror
mpi_gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
  recvtype, root, comm, ierror)
  type :: sendbuf(*), recvbuf(*)
  integer :: sendcount, recvcounts(*), displs(*), sendtype,
          recvtype, root, comm, ierror
mpi_reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
  type :: sendbuf(*), recvbuf(*)
  integer :: count, datatype, op, root, comm, ierror
```

Fortran interfaces for collective operations

```
mpi_allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)
  type :: sendbuf(*), recvbuf(*)
  integer :: count, datatype, op, comm, ierror
mpi_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
  recvtype, comm, ierror)
  type :: sendbuf(*), recvbuf(*)
  integer :: sendcount,  recvcount, sendtype, recvtype, comm, ierror
mpi_reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
  ierror)
  type :: sendbuf(*), recvbuf(*)
  integer :: recvcounts(*), datatype, op, comm, ierror
mpi_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
  recvtype, comm, ierror)
  type :: sendbuf(*), recvbuf(*)
  integer :: sendcount, recvcount, sendtype, recvtype, comm, ierror
```

Fortran interfaces for collective operations

```
mpi_alltoallv(sendbuf,sendcounts, sdispls, sendtype, recvbuf,
  recvcounts, rdispls, recvtype, comm, ierror)
  type :: sendbuf(*), recvbuf(*)
  integer :: sendcounts(*), recvcounts(*), sdispls(*), rdispls(*),
  sendtype, recvtype, comm, ierror
```

Reduce operation

- Available reduction operations (argument op)

Operation	Meaning
MPI_MAX	Max value
MPI_MIN	Min value
MPI_SUM	Sum
MPI_PROD	Product
MPI_MAXLOC	Max value + location
MPI_MINLOC	Min value + location
MPI_LAND	Logical AND
MPI_BAND	Bytewise AND
MPI_LOR	Logical OR
MPI_BOR	Bytewise OR
MPI_LXOR	Logical XOR
MPI_BXOR	Bytewise XOR

USER-DEFINED COMMUNICATORS

C interfaces for user-defined communicators

```
int MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm newcomm)
int MPI_Comm_compare (MPI_Comm comm1, MPI_Comm comm2, int result)
int MPI_Comm_dup ( MPI_Comm comm, MPI_Comm newcomm )
int MPI_Comm_free ( MPI_Comm comm )
```

Fortran interfaces for user-defined communicators

```
mpi_comm_split (comm, color, key, newcomm, ierror)
  integer :: comm, color, key, newcomm, ierror
mpi_comm_compare ( comm1, comm2, result, ierror)
  integer :: comm1, comm2, result, ierror
mpi_comm_dup ( comm, newcomm, ierror)
  integer :: comm, newcomm, ierror
mpi_comm_free ( comm, ierror)
  integer :: comm, ierror
```

NON-BLOCKING COMMUNICATION

C interfaces for non-blocking operations

```
int MPI_Isend( void *buf, int count, MPI_Datatype datatype, int dest,
  int tag, MPI_Comm comm, MPI_Request *request )
int MPI_Irecv( void *buf, int count, MPI_Datatype datatype, int source,
  int tag, MPI_Comm comm, MPI_Request *request )
int MPI_Wait(MPI_Request *request, MPI_Status *status)
int MPI_Waitall(int count, MPI_Request *array_of_requests, MPI_Status
  *array_of_statuses)
```

Fortran interfaces for non-blocking operations

```
mpi_isend(buf, count, datatype, dest, tag, comm, request,ierror)
  type :: buf(*)
  integer :: count, datatype, dest, tag, comm, request, ierror
mpi_irecv(buf, count, datatype, source, tag, comm, request,ierror)
  type :: buf(*)
  integer :: count, datatype, source, tag, comm, request, ierror
mpi_wait(request, status, ierror)
  integer :: request, status(mpi_status_size), ierror
mpi_waitall(count, array_of_requests, array_of_statuses, ierror)
  integer :: count, array_of_requests(:),
  array_of_statuses(mpi_status_size,:), ierror
```

USER-DEFINED DATATYPES

C interfaces for datatype routines

```
int MPI_Type_commit(MPI_Datatype *type)
int MPI_Type_free(MPI_Datatype *type)
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
  MPI_Datatype *newtype)
int MPI_Type_vector(int count, int block, int stride,
  MPI_Datatype oldtype, MPI_Datatype *newtype)
int MPI_Type_indexed(int count, int blocks[], int displs[], MPI_Datatype
  oldtype, MPI_Datatype *newtype)
int MPI_Type_create_subarray(int ndims, int array_of_sizes[], int
  array_of_subsizes[], int array_of_starts[], int order, MPI_Datatype
  oldtype, MPI_Datatype *newtype )
```

Fortran interfaces for datatype routines

```
mpi_type_commit(type, ierror)
  integer :: type, ierror
mpi_type_free(type, ierror)
  integer :: type, ierror
mpi_type_contiguous(count, oldtype, newtype, ierror)
  integer :: count, oldtype, newtype, ierror
mpi_type_vector(count, block, stride, oldtype, newtype, ierror)
  integer :: count, block, stride, oldtype, newtype, ierror
mpi_type_indexed(count, blocks, displs, oldtype, newtype, ierror)
  integer :: count, oldtype, newtype, ierror
  integer, dimension(count) :: blocks, displs
mpi_type_create_subarray(ndims, sizes, subsizes, starts, order,
  oldtype, newtype, ierror)
  integer :: ndims, order, oldtype, newtype, ierror
  integer, dimension(ndims) :: sizes, subsizes, starts
```

COMMUNICATION TOPOLOGIES

C interfaces for Cartesian process topologies

```
int MPI_Cart_create(MPI_Comm old_comm, int ndims, int *dims, int
  *periods, int reorder, MPI_Comm *comm_cart)
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdim, int *coords)
int MPI_Cart_rank(MPI_Comm comm, int *coords, int rank)
int MPI_Cart_shift( MPI_Comm comm, int direction, int displ, int *low,
  int *high )
```

Fortran interfaces for Cartesian process topologies

```
mpi_cart_create(old_comm, ndims, dims, periods, reorder, comm_cart,
  ierror)
  integer :: old_comm, ndims, dims(:), comm_cart, ierror
  logical :: reorder, periods(:)
mpi_cart_coords(comm, rank, maxdim, coords, ierror)
  integer :: comm, rank, maxdim, coords(:), ierror
mpi_cart_rank(comm, coords, rank, ierror)
  integer :: comm, coords(:), rank, ierror
mpi_cart_shift(comm, direction, displ, low, high, ierror)
  integer :: comm, direction, displ, low, high, ierror
```

ONE-SIDED COMMUNICATION

C interfaces for one-sided routines

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info
  info, MPI_Comm comm, MPI_Win *win)
int MPI_Win_fence(int assert, MPI_Win win)
int MPI_Put(const void *origin_addr, int origin_count, MPI_Datatype
  origin_datatype, int target_rank, MPI_Aint target_disp, int
  target_count, MPI_Datatype target_datatype, MPI_Win win)
int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype
  origin_datatype, int target_rank, MPI_Aint target_disp, int
  target_count, MPI_Datatype target_datatype, MPI_Win win)
int MPI_Accumulate(const void *origin_addr, int origin_count,
  MPI_Datatype origin_datatype, int target_rank, MPI_Aint  target_disp,
  int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win
  win)
```

Fortran interfaces for one-sided routines

```
mpi_win_create(base, size, disp_unit, info, comm, win, ierror)
  <type> :: base(*)
  integer(kind=mpi_address_kind) :: size
  integer :: disp_unit, info, comm, win, ierror
mpi_win_fence(assert, win, ierror)
  integer :: assert, win, ierror
mpi_put(origin_addr, origin_count, origin_datatype, target_rank,
  target_disp, target_count, target_datatype, win, ierror)
  <type> :: origin_addr(*)
  integer(kind=mpi_address_kind) :: target_disp
  integer :: origin_count, origin_datatype, target_rank, target_count,
  target_datatype, win, ierror
```

Fortran interfaces for one-sided routines

```
mpi_get(origin_addr, origin_count, origin_datatype, target_rank,
  target_disp, target_count, target_datatype, win, ierror)
  <type> :: origin_addr(*)
  integer(kind=mpi_address_kind) :: target_disp
  integer :: origin_count, origin_datatype, target_rank, target_count,
  target_datatype, win, ierror
mpi_accumulate(origin_addr, origin_count, origin_datatype, target_rank,
  target_disp, target_count, target_datatype, op, win, ierror)
  <type> :: origin_addr(*)
  integer(kind=mpi_address_kind) :: target_disp
  integer :: origin_count, origin_datatype, target_rank, target_count,
  target_datatype, op, win, ierror
```

MPI-IO

C interfaces for MPI-IO routines

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info
   info, MPI_File *fh)
int MPI_File_close(MPI_File *fh)
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
int MPI_File_read(MPI_File fh, void *buf, int count,
   MPI_Datatype datatype, MPI_Status *status)
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int
   count, MPI_Datatype datatype, MPI_Status *status)
int MPI_File_write(MPI_File fh, void *buf, int count, MPI_Datatype
   datatype, MPI_Status *status)
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int
   count, MPI_Datatype datatype, MPI_Status *status)
```

C interfaces for MPI-IO routines

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
   MPI_Datatype filetype, char *datarep, MPI_Info info)
int MPI_File_read_all(MPI_File fh, void *buf, int count,
   MPI_Datatype datatype, MPI_Status *status)
int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf, int
   count, MPI_Datatype datatype, MPI_Status *status)
int MPI_File_write_all(MPI_File fh, void *buf, int count, MPI_Datatype
   datatype, MPI_Status *status)
int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, void *buf, int
   count, MPI_Datatype datatype, MPI_Status *status)
```

Fortran interfaces for MPI-IO routines

```
mpi_file_open(comm, filename, amode, info, fh, ierror)
   integer ::  comm, amode, info, fh, ierror
   character* ::  filename
mpi_file_close(fh, ierror)
mpi_file_seek(fh, offset, whence, ierror)
   integer :: fh, whence, ierror
   integer(kind=MPI_OFFSET_KIND) :: offset
mpi_file_read(fh, buf, count, datatype, status, ierror)
mpi_file_write(fh, buf, count, datatype, status, ierror)
   integer :: fh, count, datatype, status(mpi_status_size)
   type :: buf(*)
mpi_file_read_at(fh, offset, buf, count, datatype, status, ierror)
mpi_file_write_at(fh, offset, buf, count, datatype, status, ierror)
   integer :: fh, offset, buf, count, datatype, status(mpi_status_size)
```

Fortran interfaces for MPI-IO routines

```
mpi_file_set_view(fh, disp, etype, filetype, datarep, info, ierror)
   integer :: fh, etype, filetype, info, ierror
   integer(kind=MPI_OFFSET_KIND) :: disp
   character* :: datarep
mpi_file_read_all(fh, buf, count, datatype, status, ierror)
mpi_file_write_all(fh, buf, count, datatype, status, ierror)
   integer :: fh, count, datatype, status(MPI_STATUS_SIZE), ierror
   type :: buf(*)
mpi_file_read_at_all(fh, offset, buf, count, datatype, status, ierror)
mpi_file_write_at_all(fh, offset, buf, count, datatype, status, ierror)
   integer :: fh, count, datatype, status(MPI_STATUS_SIZE), ierror
   integer(kind=MPI_OFFSET_KIND) :: offset
   type :: buf(*)
```

A

B

C

Notes