CSC Summer School in High-Performance Computing 2022

June 26 - July 5, 2022

CSC - IT Center for Science, Finland

CSC

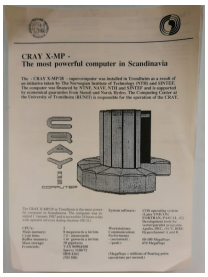
Training for Brilliant Minds

Introduction to supercomputing

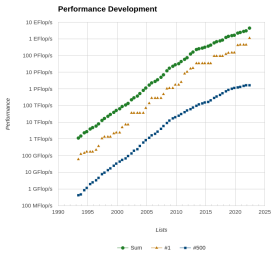
What is high-performance computing?

- Utilising computing power that is much larger than available in a typical desktop computer
- Performance of HPC systems (i.e. supercomputers) 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 > 10^{18} flop/s (1 Eflop / s)

What is high-performance computing?


CRAY X-MP :
The most powerful computer in Scandinavia

Top 500 list



What are supercomputers used for?

General use cases

- Simulations of very different scales
  - From subatomic particles to cosmic scales
- Problems with very large datasets
- Complex computational problems
- Problems that are hard to experiment on
  - Simulations with decade-long timescales
- Very time consuming or even impossible to solve on a standard computer

Application areas

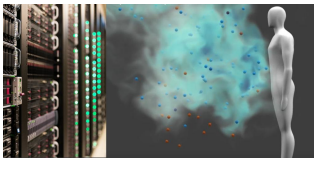
- Fundamental sciences such as particle physics and cosmology
- Climate, weather and Earth sciences
- Life sciences and medicine
- Chemistry and material science
- Energy, e.g oil and gas exploration and fusion research
- Engineering, e.g. infrastructure and manufacturing
- etc.

Climate change

- Simulating ice sheets, air pollutants, sea-level rise etc.
- Building short and long-term simulations
- Analyzing with different parameters for future predictions and possible solutions
- Modeling space weather

Covid-19 fast track with Puhti

- Modeling particles in airflows
- A large part of the calculations used for solving turbulent flow
- A third of Puhti was reserved for running the simulations
- The results have had an impact on e.g. ventilation instructions and the use of masks

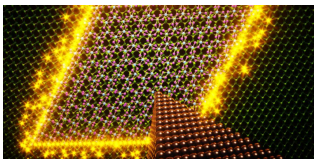


Gravitational waves

- Computational modeling of sources of gravitational waves
- Identifying a phase transition of the Higgs boson “turning on” (10 picoseconds after Big Bang)
- Large simulations with over ten thousand CPU cores
- Experimental data from ESA’s LISA satellite (Launch date 2037)

Topological superconductors

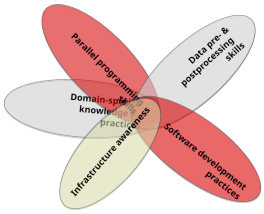
- Topological superconductors are possible building blocks for qubits
- Based on an elusive quantum state of electrons in thin layers
- Electronic properties simulated with the density-functional theory
  - These confirm that experimentally measured signals are due to this special quantum state



Deep language model of Finnish

- Web-scale Finnish language data together with very deep neural networks utilizing GPUs
- New model for Finnish
  - Comparable in coverage and quality to the best language models available today for any language

Utilizing HPC in scientific research

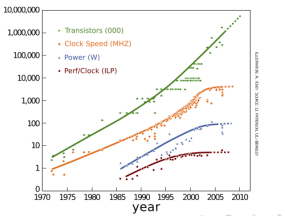


- **Goal for this school: everyone is able to write and modify HPC applications!**

What are supercomputers made of?

CPU frequency development

- Power consumption of CPU: f^3

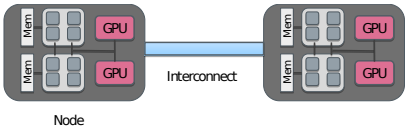


Parallel processing

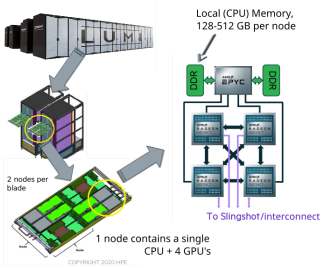
- Modern (super)computers rely on parallel processing
- **Multiple** CPU cores & accelerators (GPUs)
  - #1 system has ~9 000 000 cores and ~40 000 GPUs
- Vectorization
  - A single instruction can process multiple data (SIMD)
- Pipelining
  - Core executes different parts of instructions in parallel

Anatomy of a supercomputer

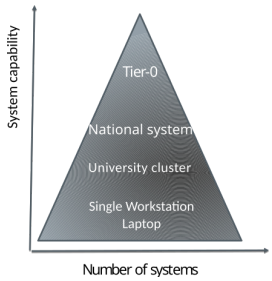
- Supercomputers consist of nodes connected by a high-speed network
  - Latency ~1 µs, bandwidth ~20 GB / s
- A node can contain several multicore CPU’s
- Additionally, a node can contain one or more accelerators
- Memory within the node is directly usable by all CPU cores



Supercomputer autopsy – Lumi



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

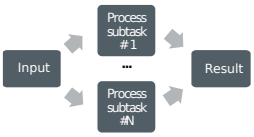
Cloud computing and HPC

- Suitability of cloud computing for HPC depends heavily on application
  - Single node performance is often sufficient
- Virtualization adds overhead especially for the networking
  - Some providers offer high-speed interconnects at a higher price
- Moving data out from the cloud can be time-consuming
- 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

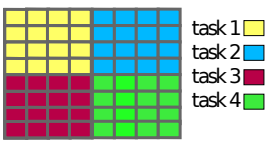


Types of parallel problems

- 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

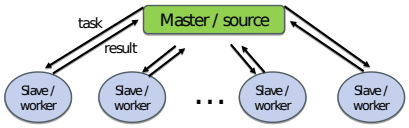
Exposing parallelism

- 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

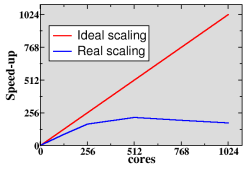
- Task farm (master / worker)



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

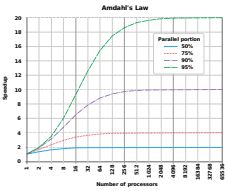
Parallel scaling

- 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
  - Variation in workload over different cores
- 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

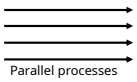
Programming languages

- 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
  - SYCL, Kokkos, PETSc, Trilinos
  - TensorFlow, PyTorch for deep learning
  - Spark for MapReduce

Parallel programming models

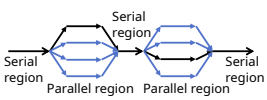
- Parallel execution is based on threads or processes (or both) which run at the same time on different CPU cores
- Processes
  - Interaction is based on exchanging messages between processes
  - MPI (Message passing interface)
- Threads
  - Interaction is based on shared memory, i.e. each thread can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

Parallel programming models



MPI: Processes

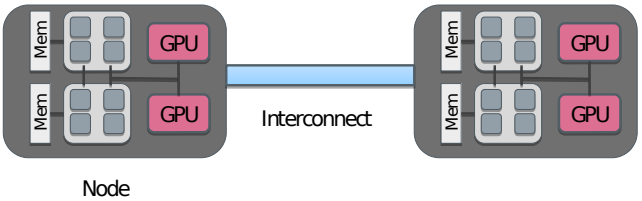
- Independent execution units
- MPI launches N processes at application startup
- Works over multiple nodes



OpenMP: Threads

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

Parallel programming models



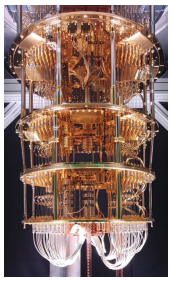
Future of High-performance computing

Post-Exascale challenges

- Performance of supercomputers has increased exponentially for a long time
- However, there are still challenges in continuing onwards from exascale supercomputers ($> 1 \times 10^{18}$ flop/s)
  - Power consumption: current #1 energy efficient system requires ~20 MW for exascale performances
  - Cost & Maintaining: Global chip shortage
  - 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
- Use cases still largely experimental and hypothetical
- Hybrid approaches



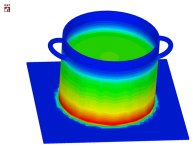
Case study: heat equation

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

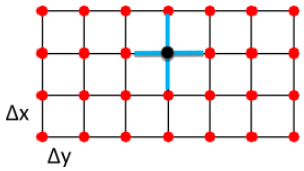


Numerical solution

- Discretize: Finite difference Laplacian in two dimensions

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

Temperature field $u(i, j)$



Time evolution

- Explicit time evolution with time step Δt

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

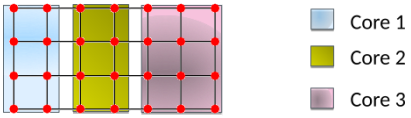
- Note: algorithm is stable only when

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

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

Solving heat equation in parallel

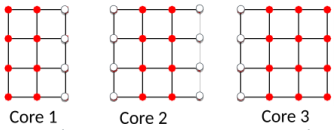
- Temperature at each grid point can be updated independently
- Domain decomposition



- Straightforward in shared memory computer

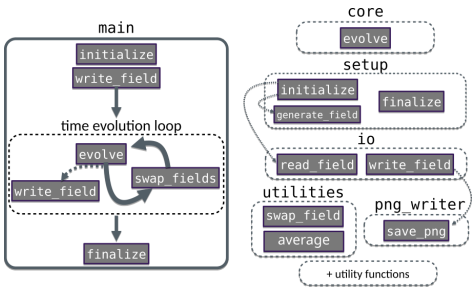
Solving heat equation in parallel

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

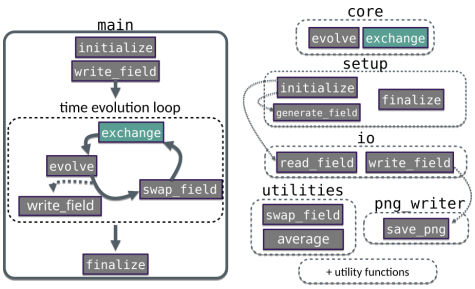


- Before each update cycle, CPU cores communicate boundary data: halo exchange

Serial code structure



Parallel code structure



Working with Unix and version control

Unix skills needed for the summer school

- To be able to follow the summer school, please make sure that you:
  - know how to start and use a ***command line interface*** (that is, console, shell, terminal)
  - know what is meant by ***directories*** and ***files*** and understand ***unix file paths***
  - know what ***environment variables*** are and how to use them
  - know how to ***change directories*** (cd), ***list the contents of a directory*** (ls), ***move and copy files*** (cp, mv), and ***remove files*** (rm)
  - know how to ***use a text editor*** (nano, vim, emacs or similar)

Unix philosophy: Do one thing and do it well

- All input and output of different tools should be single flat files, so that they can be combined through the use of pipes
- Examples:
  - count how many filenames contain the string `model` in the current directory and all it's subdirectories:

```
find . | grep model | wc -l
```

  - concatenate multiple files while replacing all occurrences of the string `/some/path` with `/another/path`:

```
cat *.sh | sed -e "s#/some/path#/another/path#g" > output.sh
```

Skills needed to work with supercomputers

- Logging into, and copying files from/to a supercomputer (ssh, scp)
- Setting up an environment to compile and run programs
  - Module system (before compiling)
  - Batch queue system (for running)
- Compiling codes
  - Basic use of version control (git)
  - Make
  - Compilers (gcc)

SSH: log in and copy files to/from remote hosts

- The `ssh` command is used to log into a remote host
  - You will be asked for your password to establish a connection

```
ssh user@host
```

  - Additionally, ssh can also be used to run a command on the host:
    `ssh user@host command`
- The `scp` command is used to copy files and directories between different hosts on a network

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

Hands-on: ssh and scp

- Create a text file on your computer (e.g. `name.txt`) with your first name as the content
- Copy the file to Puhti using `scp`:

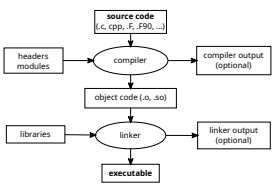
```
scp local.txt trainingxxx@puhti.csc.fi:~/
```

- Log in to Puhti using `ssh`, check that the file is there, and modify it (for example add your last name)
- Copy the modified file back to your local computer and check that the file has been changed

Make

Compiling and linking

- A compiler turns a source code file into an object file that contains machine code that can be executed by the processor
- A linker combines several compiled object files into a single executable file
- Together, compiling and linking can be called building

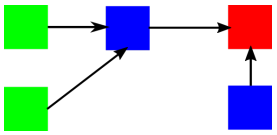


Compiling and linking: possible problems

- Programs should usually be separated into several files
  ⇒ complicated dependency structures
- Building large programs takes time
  - could we just rebuild the parts that changed?
- Having different options when building
  - debug versions, enabling/disabling features, etc.

Make

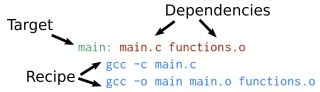
- Make allows you to define how to build individual parts of a program and how they depend on each other. This enables:
  - Building parts of a program and only rebuilding necessary parts
  - Building different version and configurations of a program



Make rules

- Make **rules** define how some part of your program is built
  - **Target**: the output file (or aim) of your rule
  - **Dependency**: which other targets your target depends on
  - **Recipe**: how you produce your target
- Rules are defined in a file which is by default called `Makefile`

A make rule



Simple Makefile example

- Dependencies can be files or other targets
- Recipes consist of one or more shell commands
  - Recipe lines start with a **tabulator**
- If the dependencies are newer than the target, make runs the recipe
- Run make with: `make target`
  - Without an argument, make runs the first rule in the file

```
main: main.c functions.o
    gcc -c main.c
    gcc -o main main.o functions.o

functions.o: functions.c
    gcc -c functions.c

.PHONY: clean
clean:
    rm -f main.o functions.o main
```

Variables and patterns in rules

- It is possible to define variables, for example compiler and link commands and flags
- Targets, dependencies and recipes can contain special wild cards

```
CC=cc
CCFLAGS=-O3

# Files of the form filename.o depend on
# filename.c
%.o: %.c
        $(CC) $(CCFLAGS) -c $< -o $@
```

Modules and batch job system

Module environment

- Supercomputers have a large number of users with different needs for development environments and applications
- *Environment modules* provide a convenient way to dynamically change the user's environment
- In this way, different compiler suites and application versions can be used more easily
  - They basically just change where things are "pointing". So when you run `gcc` the loaded module decides whether you are using version 4.9 or 5.3 or 6.0 and so on
  - Most programs require loading a module to be accessible

Common module commands

`module load mod`
: Load module **mod** in shell environment

`module unload mod`
: Remove module **mod** from environment

`module list`
: List loaded modules

`module avail`
: List all available modules

`module spider mod`
: Search for module **mod**

`module show mod`
: Get information about module **mod**

`module switch mod1 mod2`
: Switch loaded **mod1** to **mod2**

Batch queue system

- On a cluster, instead of running a program instantly, you submit your program/simulation (aka job) to a queue and the system will then execute it once the resources are available
  - The queue enables effective and fair resource usage
  - CSC uses SLURM as the queue system
- When running a job on a supercomputer you need to:
  - Describe how you want to run the job and what resources you need
  - Add a command that launches your program
  - Submit your job to a queue
- This is done with a batch job script

Example SLURM batch job script

```
#!/bin/bash
#SBATCH --job-name=example
#SBATCH --account=yourproject
#SBATCH —-partition=test
#SBATCH —-time=00:10:00
#SBATCH —-ntasks=80
#SBATCH --mem-per-cpu=4000

srun myprog
```

- More examples:
  - https://docs.csc.fi/computing/running/example-job-scripts-puhti/
  - https://docs.csc.fi/computing/running/example-job-scripts-mahti/

Running batch jobs under SLURM

- 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`:

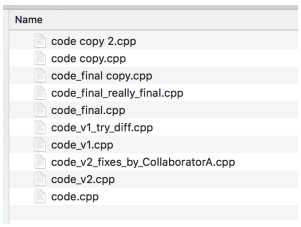
```
scancel jobid
```

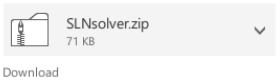
(here the jobid is the numeric ID of the job)

- Show job resource usage (for completed jobs) with `sacct`:

```
sacct jobid
```

Hands up - is this familiar?





Why is this a problem for you ?

- Keeping track of all of your changes requires a lot of work
- Sharing code with colleagues and collaborating is slow and prone to errors

You spend less time on things that actually matter!

Writing your code/scripts, running simulations and getting results

Good science is reproducible!

What is the exact version/revision of your code used to produce some results?

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 compare different versions of the code
- 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

Version control with git

Starting a git repository: git init

- We need to tell git to start “monitoring” our project folder
- Git won’t do or save anything unless we tell it to
- creates a `.git` folder. Don’t touch this!
  - contains all the info and data used by git, manual editing can/will break it
- **Terminology:** A git *repository* = A project folder were git is operating

```
$ mkdir recipe
$ cd recipe
$ git init
```

Choosing what we want to save: git add

- Selecting what changes we want to save
  - Moving and deleting files is also a change
  - Could be a folder, a couple of files, a single file or just part of a file
- You don't have to add all your changes at once!
- Does not change our files or save anything yet

```
.... editing some files
$ git add ingredients.txt
```

Creating a snapshot: git commit

- A commit is a saved state of your project
- All changes that were added using `git add` are now part of the commit
- A commit needs to have an associated commit message
  - The message should be clear and concise: "Fixed bug #1234"
- Git allows us to compare, merge, and checkout commits
  - Checkout a commit -> Set our project state to a specific commit

```
$ git commit
```

Git file states: git status

A file in a git repository can be in one of the following states

- **Untracked**: Untracked files are ignored by git
- **Unmodified**: The file is tracked by git, but no changes have been made
- **Modified**: The file is tracked by git and has been modified
- **Staged**: The file has been added (`git add file1`) and is ready to be committed to your repository

`git status` shows the current state of your repository

Project history: git log

- Each commit has a:
  - Message
  - Author
  - Unique id, also called a commit hash
  - Submission date
- We use `git log` to inspect the history of our repository

Whats the difference: git diff

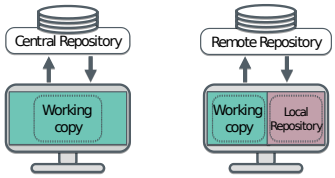
- Show the difference between two commits
- Or the difference between a commit and the current state of your project
- The commit hash is used to refer to a specific commit

Remote repositories: Where things get really useful

- You can have multiple copies of the same project in different places
  - A different folder, another computer, or a web-based repository
  - You have a copy, your coworker has a copy, and so on...
- This is what enables collaboration and sharing changes
- You can also just clone a git repository and after that never interact with anyone

Centralized vs. Distributed

- Git is a distributed version control system
  - I.e. every repository is self contained and *equally valid*



Remote repositories: Basic commands

- `git clone`, to copy a repository
- `git pull`, to retrieve changes from a remote repository
- `git push`, to send our changes to a remote repository

Remote repositories: Web services

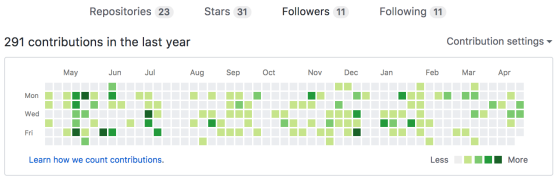
- Git is usually paired up with a web-based repository
  - GitHub
  - Bitbucket
  - GitLab
- **These are not the same as git**, they are services built on top of git
- We will be using GitHub, exercises and materials at https://github.com/csc-training/summerschool

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"



Creating repositories on GitHub

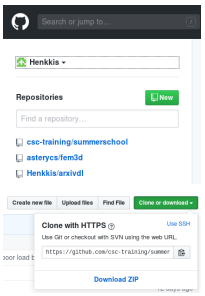
1. Create a new repository in GitHub
2. After this you can:
   - push a local repository to GitHub

```
$ git remote add origin https://github.com/User/repo.git
$ git push -u origin master
```

   - clone the repository

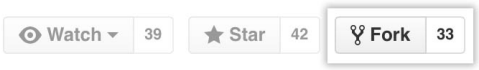
```
git clone https://github.com/User/repo.git
```

3. The local repository is now functioning and synced with GitHub



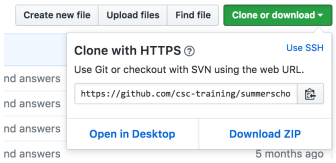
Forking repositories

- Copying a repository on GitHub to your own GitHub
  - So a GitHub feature, *not a part of git*
- This is called "forking"
- 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

1. New file changes are **pull**ed from the remote repository: `git pull`
   - Files are now up-to-date (with respect to the remote repository)
2. Something is modified
   - E.g. editing, adding or removing a file
3. Modifications are **add**ed for commit: `git add file.c`
   - Think of this as putting your file changes into a package
4. Changes are **commit**ted to the local repository: `git commit`
5. Commits are **push**ed to the remote repository: `git push`

Git Workflow

- At every point, you can check the current status of the local repository
  - `git status`
  - What has changed? What is staged for a 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!

Branching

- By default everything is happening at “main” branch
- A branch is just a pointer to a commit
- You can freely create new branches
  - `git branch new_branch_name`
  - `git checkout new_branch_name`
- Extremely useful when you are working on multiple things/features
  - Allows you to keep a properly working version of a program, even though you are in the middle of major modifications.
  - Great for still unstable features

Branching

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 many cool repositories!
- Experiment with git and try out different features, its better to make mistakes now than later.
- Errors can be easily corrected

Version control git commands

- Download repository: `git clone …`
- 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]`

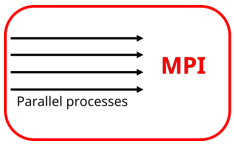
Message passing interface

Basic concepts in MPI

Message-passing interface

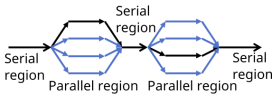
- MPI is an application programming interface (API) for distributed parallel computing
- MPI programs are portable and scalable
  - the same program can run on different types of computers, from laptops to supercomputers
- MPI is flexible and comprehensive
  - large (hundreds of procedures)
  - concise (only 10-20 procedures are typically needed)
- First version of standard (1.0) published in 1994, latest (4.0) in June 2021

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

Execution model in MPI

- Normally, 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`, `srun`, `aprun`, ...
  - `srun` on puhti.csc.fi and mahti.csc.fi
- MPI supports also dynamic spawning of processes and launching *different* programs communicating with each other
  - rarely used in HPC systems

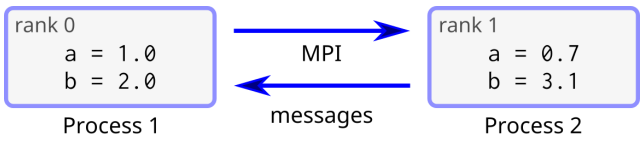
MPI ranks

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



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
  - Communicator manipulation, user defined datatypes, one-sided communication, ...

MPI communicator

- Communicator is an object connecting a group of processes, i.e. the communication framework
- Most MPI functions require communicator as an argument
- Initially, there is always a communicator **MPI_COMM_WORLD** which contains all the processes
- Users can define custom communicators

Programming MPI

- The MPI standard defines interfaces to C and Fortran programming languages
  - No C++ bindings in the standard, C++ programs use the C interface
  - There are unofficial bindings to eg. Python, Rust, R
- C call convention (*case sensitive*)

  `rc = MPI_Xxxx(parameter,...)`
  - some arguments have to be passed as pointers
- Fortran call convention (*case insensitive*)

  `call mpi_xxxx(parameter,..., rc)`
  - return code in the last argument

Writing an MPI program

- C: include the MPI header file

```c
#include <mpi.h>
```

- Fortran: use MPI module

```fortran
use mpi_f08
```

(older Fortran codes might have `use mpi` or `include 'mpif.h'`)

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

Compiling an MPI program

- MPI is a library (+ runtime system)
- In principle, MPI programs can be build with standard compilers (*i.e.* `gcc` / `g++` / `gfortran`) with the appropriate `-I` / `-L` / `-l` options
- Most MPI implementations provide convenience wrappers, typically `mpicc` / `mpicxx` / `mpif90`, for easier building
  - no need for MPI related options

```
mpicc -o my_mpi_prog my_mpi_code.c
mpicxx -o my_mpi_prog my_mpi_code.cpp
mpif90 -o my_mpi_prog my_mpi_code.F90
```

Presenting syntax

- MPI calls are presented as pseudocode
  - actual C and Fortran interfaces are given in reference section
  - Fortran error code argument not included

`MPI_Function(arg1, arg2)`

arg1
input arguments in red

arg2
output arguments in blue. Note that in C the output arguments are always pointers

First five MPI commands: Initialization and finalization

`MPI_Init`
(in C `argc` and `argv` pointer arguments are needed)

MPI_Finalize :

First five MPI commands: 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: Synchronization

- Wait until everybody within the communicator reaches the call

`MPI_Barrier(comm)`

comm
communicator

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

Web resources

- List of MPI functions with detailed descriptions
  - http://mpi.deino.net/mpi_functions/
  - https://www.rookiehpc.com/mpi/docs/
- Good online MPI tutorials
  - https://hpc-tutorials.llnl.gov/mpi/
  - http://mpitutorial.com/tutorials/
  - https://www.youtube.com/watch?v=BPSgXQ9aUXY
- MPI coding game in C
  https://www.codingame.com/playgrounds/47058/have-fun-with-mpi-in-c/lets-start-to-have-fun-with-mpi

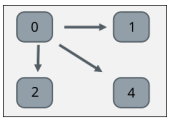
Web resources

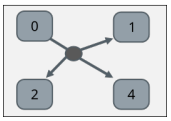
- MPI 4.0 standard http://www.mpi-forum.org/docs/
- MPI implementations
  - OpenMPI http://www.open-mpi.org/
  - MPICH https://www.mpich.org/
  - Intel MPI
    https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/r
    library.html

Point-to-point communication

Communication

- Data is local to the MPI processes
  - They need to *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

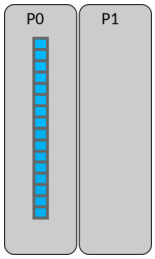




MPI point-to-point operations

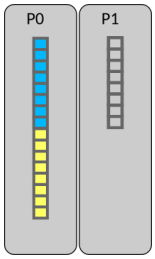
- One process *sends* a message to another process that *receives* it with `MPI_Send` and `MPI_Recv` routines
- Sends and receives in a program should match – one receive per send
- Each message (envelope) contains
  - The actual *data* that is to be sent
  - The *datatype* of each element of data
  - The *number of elements* the data consists of
  - An identification number for the message (*tag*)
  - The ranks of the *source* and *destination* process

Case study: parallel sum

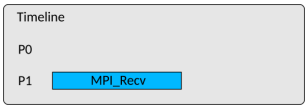


- Array initially on process #0 (P0)
- Parallel algorithm
  - **Scatter** Half of the array is sent to process 1
  - **Compute** P0 & P1 sum independently their segments
  - **Reduction** Partial sum on P1 sent to P0 P0 sums the partial sums

Case study: parallel sum

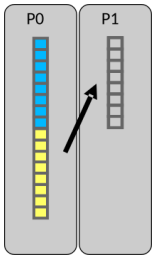


Step 1.1: Receive call in scatter

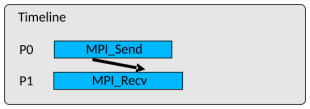


P1 issues MPI_Recv to receive half of the array from P0

Case study: parallel sum

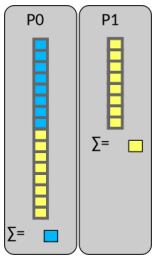


Step 1.2: Send call in scatter

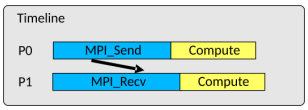


P0 issues an MPI_Send to send the lower part of the array to P1

Case study: parallel sum

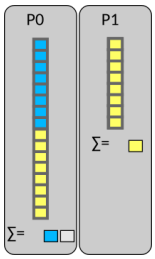


Step 2: Compute the sum in parallel

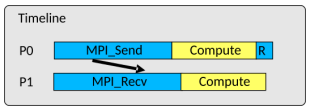


Both P0 & P1 compute their partial sums and store them locally

Case study: parallel sum

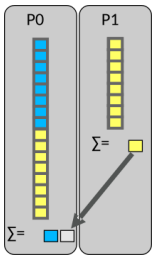


Step 3.1: Receive call in reduction

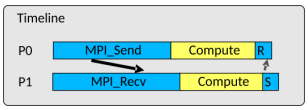


P0 issues an MPI_Recv operation for receiving P1's partial sum

Case study: parallel sum

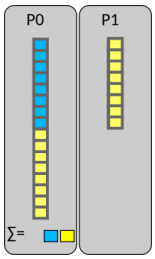


Step 3.2: Send call in reduction

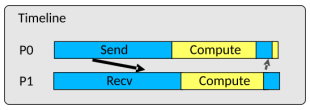


P1 sends the partial sum to P0

Case study: parallel sum



Step 3.3: compute the final answer



P0 computes the total sum

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

"Buffers" in MPI

- The "buffer" arguments are memory addresses
- MPI assumes contiguous chunk of memory
  - `count` elements are send starting from the address
  - received elements are stored starting from the address
- In Fortran, arguments are passed by reference, *i.e.* variables can be passed as such
  - Note: be careful if passing non-contiguous array segmens such as `a(1, 1:N)`
- In C/C++ "buffer" is pointer
  - `data()` method of C++ `<array>` and `<vector>` containers can be used

MPI datatypes

- On low level, MPI sends and receives stream of bytes
- MPI datatypes specify how the bytes should be interpreted
  - Allows data conversios in heterogenous environments (*e.g.* little endian to big endian)
- MPI has a number of predefined basic datatypes corresponding to C or Fortran datatypes
  - C examples: `MPI_INT` for `int` and `MPI_DOUBLE` for `double`
  - Fortran examples: `MPI_INTEGER` for `integer`, `MPI_DOUBLE_PRECISION` for `real64`
- One can also define custom datatypes for communicating more complex data

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

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 the status parameter is of type `mpi_status`
  - Old interface: 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` (old version `status(MPI_TAG)`)
- Rank of the sender C: `status.MPI_SOURCE` Fortran: `status%mpi_source` (old version `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` and `MPI_Recv`
- Status parameter of `MPI_Recv` contains information about the message after the receive is completed

"Special" parameters

MPI programming practices

- For the sake of illustration, we have so far hard-coded the `source` and `destination` arguments, and placed the MPI calls within `if` constructs
- This produces typically code which is difficult to read and to generalize to arbitrary number of processes
- Store source and destination in variables and place MPI calls outside **if**s when possible.

```
if (myid == 0) then
   call mpi_send(message, msgsize, MPI_INTEGER, 1, &
        1, MPI_COMM_WORLD, rc)
   call mpi_recv(recvBuf, arraysize, MPI_INTEGER, 1, &
        1, MPI_COMM_WORLD, status, rc)
else if (myid == 1) then
   call mpi_send(message, msgsize, MPI_INTEGER, 0, &
        1, MPI_COMM_WORLD, rc)
   call mpi_recv(recvBuf, arraysize, MPI_INTEGER, 0, &
        1, MPI_COMM_WORLD, status, rc)
```

```
! Modulo operation can be used for wrapping around
dst = mod(myid + 1, ntasks)
src = mod(myid - 1 + ntasks, ntasks)

call mpi_send(message, msgsize, MPI_INTEGER, dst, &
              1, MPI_COMM_WORLD, rc)
call mpi_recv(recvBuf, arraysize, MPI_INTEGER, src, &
              1, MPI_COMM_WORLD, status, rc)
```

MPI programming practices

- As rank 0 is always present even in the serial case, it is normally chosen as the special task in scatter and gather type operations

```
if (0 == myid) {
  for (int i=1; i < ntasks; i++) {
     MPI_Send(&data, 1, MPI_INT, i, 22, MPI_COMM_WORLD);
  }
} else {
     MPI_Recv(&data, 1, MPI_INT, 0, 22, MPI_COMM_WORLD, &status);
}
```

Coping with boundaries

- In some communication patterns there are boundary processes that do not send or receive while all the other processes do
- A special constant `MPI_PROC_NULL` can be used for turning `MPI_Send` / `MPI_Recv` into a dummy call
  - No matching `receive` / `send` is needed

```
if (myid == 0) then
    src = MPI_PROC_NULL
end if
if (myid == ntasks - 1) then
    dst = MPI_PROC_NULL
end if

call mpi_send(message, msgsize, MPI_INTEGER, dst, ...
call mpi_recv(message, msgsize, MPI_INTEGER, src, ...
```

Arbitrary receives

- In some communication patterns one might want to receive from arbitrary sender or a message with arbitrary tag
- `MPI_ANY_SOURCE` and `MPI_ANY_TAG`
  - The actual sender and tag can be queried from **status** if needed
- There needs to be still `receive` for each `send`
- `MPI_ANY_SOURCE` may introduce performance overhead
- Use only when there is clear benefit *e.g.* in load balancing

```
if (0 == myid) {
  for (int i=1; i < ntasks; i++) {
     MPI_Recv(&data, 1, MPI_INT, MPI_ANY_SOURCE, 22,
              MPI_COMM_WORLD, &status);
     process(data)
  }
} else {
     MPI_Send(&data, 1, MPI_INT, 0, 22, MPI_COMM_WORLD);
}
```

Ignoring status

- When `source`, `tag`, and number of received elements are known, there is no need to examine `status`
- A special constant `MPI_STATUS_IGNORE` can be used for the `status` parameter
- Saves memory in the user program and allows optimizations in the MPI library

Special parameter values in sending

`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 in receiving

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

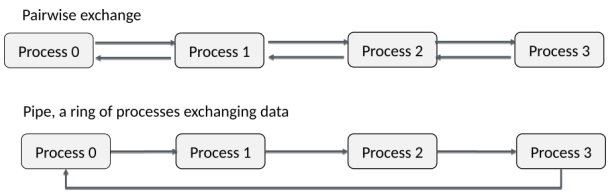
Parameter	Special value	Implication
`source`	`MPI_PROC_NULL`	No sender=no operation takes place
	`MPI_ANY_SOURCE`	Receive from any sender
`tag`	`MPI_ANY_TAG`	Receive messages with any tag
`status`	`MPI_STATUS_IGNORE`	Do not store any status data

Common communication patterns

Main - worker

- Each process is only sending or receiving at the time

Pairwise neighbour communication



- Incorrect ordering of sends/receives may give a rise to a deadlock or unnecessary idle time
- Can be generalized to multiple dimensions

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 and improves performance
- Parameters as in `MPI_Send` and `MPI_Recv`
- Destination rank and source rank can be same or different
- `MPI_PROC_NULL` can be used for coping with the boundaries

Summary

- Generally, it is advisable to make MPI programs to work with arbitrary number of processes
- When possible, MPI calls should be placed outside `if` constructs
- Employing special parameter values may simplify the implementations of certain communication patterns
- Individual `MPI_Send` and `MPI_Recv` are suitable for irregular communication
- When there is always both sending and receiving, `MPI_Sendrecv` can prevent deadlocks and serialization of communication

Parallel debugging

Debugging

- Bugs are evident in any non-trivial program
- Crashes (Segmentation fault) or incorrect results
- Parallel programs can have also deadlocks and race conditions

Finding bugs

- Building code with Sanitizer
  - Detects out-of-bounds memory access and OpenMP race conditions
  - `-fsanitize=address` (GCC / Clang)
  - `-fsanitize=thread` (GCC / Clang)
- Using MPI correctness checkers
  - MUST
  - Intel Trace Analyzer

Finding bugs

- Print statements in the code
  - Typically cumbersome, especially with compiled languages
  - Might result in lots of clutter in parallel programs
  - Order of printouts from different processes is arbitrary
- "Standard" debuggers
  - `gdb`: common command line debugger
  - Debuggers within IDEs, e.g. VS Code
  - No proper support for parallel debugging
- Parallel debuggers
  - **Allinea DDT**, Totalview, gdb4hpc (commercial products)

Common features in debuggers

- Setting breakpoints and watchpoints
- Executing code line by line
- Stepping into / out from functions
- Investigating values of variables
- Investigating program stack
- Parallel debuggers allow all of the above on per process/thread basis

Web resources

- Defensive programming and debugging online course https://www.futurelearn.com/courses/defensive-programming-and-debugging
- MUST https://www.i12.rwth-aachen.de/go/id/nrbe
- Using `gdb` for parallel debugging https://www.open-mpi.org/faq/?category=debugging
- Memory debugging with Valgrind https://valgrind.org/docs/manual/mc-manual.html#mc-manual.mpiwrap

Demo: using Allinea DDT

Using Allinea DDT

- Code needs to be compiled with debugging option `-g`
- Compiler optimizations might complicate debugging (dead code elimination, loop transformations, *etc.*), recommended to compile without optimizations with `-O0`
  - Sometimes bugs show up only with optimizations
- In CSC environment DDT is available via `module load ddt`
- Debugger needs to be started in an interactive session

```
module load ddt
export SLURM_OVERLAP=1
salloc --nodes=1 --ntasks-per-node=2 --account=project_xxx -p small
ddt srun ./buggy
```

- VNC remote desktop is recommended for smoother GUI performance

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)
```

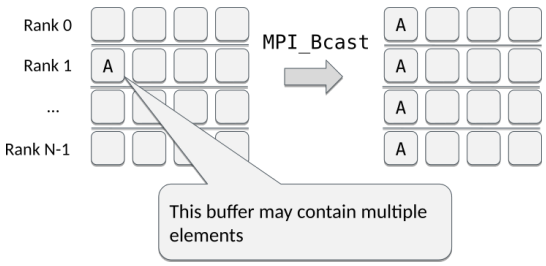
Communicating a vector **a** consisting of 1M float elements from the task 0 to all other tasks

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

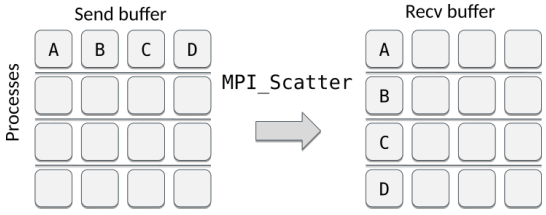
datatype
data type of buffer

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

- Task root sends an equal share of data 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

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

```
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(:)
```

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

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

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

```
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

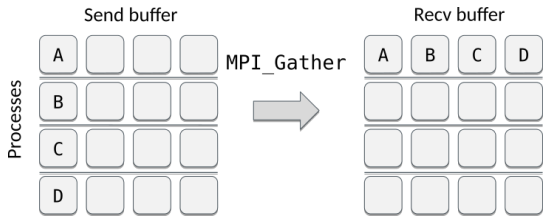
```
if (my_id==0) then
  do i = 1, 10
    a(i) = i
  end do
end if

scounts(0:3) = [ 1, 2, 3, 4 ]
displs(0:3) = [ 0, 1, 3, 6 ]

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

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

A) 1 2 3 **B)** 7 8 9 10 **C)** 1 2 3 4 5 6 7 8 9 10

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 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

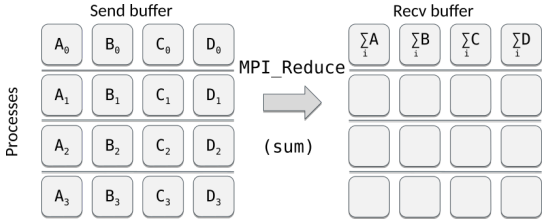
Vector version of MPI_Gather

```
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



Available reduction operations

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

Operation	Meaning
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

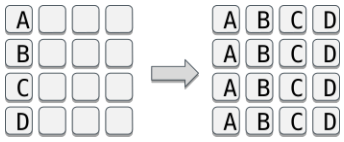
Allreduce example: parallel dot product

```
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_WORL
                   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



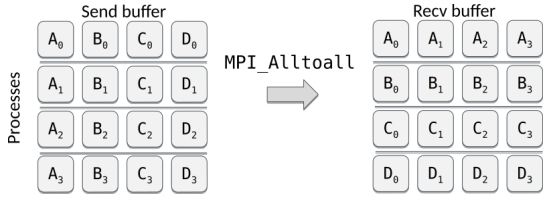
All gather

```
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

- 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
sendtype
data type of send buffer elements
recvbuf
receive buffer
recvcount
number of elements received
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`

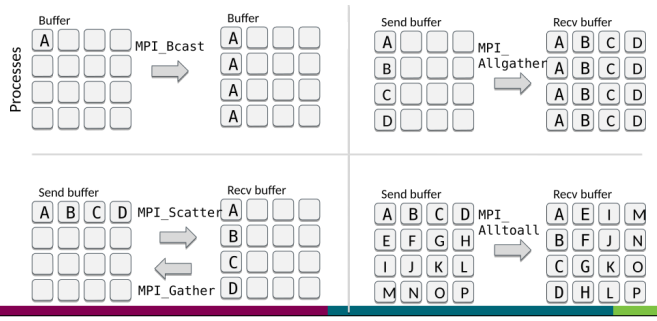
Common mistakes with collectives

- 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



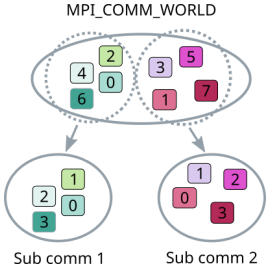
User-defined communicators

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
  - Collective communication within a group of processes
  - Parallel I/O

Communicators

- Communicators are dynamic
- A task can belong simultaneously to several communicators
  - Unique rank in each communicator



Creating new communicator

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

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

`comm`
communicator

`color`
processes with the same "color" belong to the same new communicator

`key`
control of rank assignment

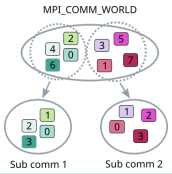
`newcomm`
new communicator handle

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,
    "%d in Comm %d.\n", myid, mysubid, co
```

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



Using an own communicator

```
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
```

Before broadcast:

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

After broadcast:

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

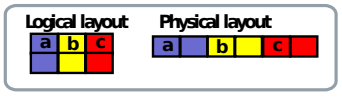
User defined datatypes (part 1)

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` (32 bits) 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 in communication
  - `MPI_Type_commit(newtype)`
- A type should be freed after it is no longer needed
  - `MPI_Type_free(newtype)`

Datatype constructors

Datatype	Usage
`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_CONTIGUOUS

`MPI_Type_contiguous(count, oldtype, newtype)`

`count`
number of oldtypes

`oldtype`
old type

`newtype`
new datatype

- Usage mainly for programming convenience
  - derived types in all communication calls

```
! Using derived type
call mpi_send(buf, 1, conttype, ...)
call mpi_send(buf, 1, non_conttype, ...)
```

```
! Equivalent call with count and basic ty
call mpi_send(buf, count, MPI_REAL, ...)
call mpi_send(buf, 1, non_conttype, ...)
```

MPI_TYPE_VECTOR

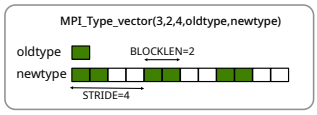
- Creates a new type from equally spaced identical blocks

`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
type(mpi_datatype) :: 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)`

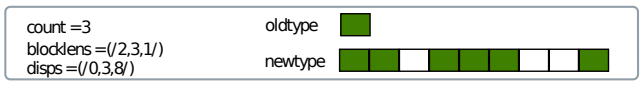
`count`
number of blocks

`blocklens`
lengths of the blocks (array)

`displs`
displacements (array) in extent of oldtypes

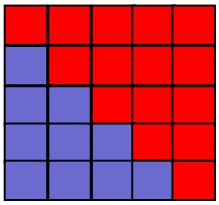
`oldtype`
original type

`newtype`
new type



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 tr 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_WO
MPI_Type_free(&upper);
```



Subarray

- Subarray datatype describes a N-dimensional subarray within a N-dimensional array
- Array can have either C (row major) or Fortran (column major) ordering in memory

MPI_TYPE_CREATE_SUBARRAY

```
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`

oldtype
oldtype

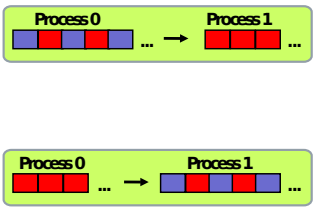
newtype
resulting type

Example: subarray

```
int a_size[2]    = {5,5};
int sub_size[2]  = {2,3};
int sub_start[2] = {1,2};
MPI_Datatype sub_type;
double array[5][5];

for(i = 0; i < a_size[0]; i++)
  for(j = 0; j < a_size[1]; j++)
    array[i][j] = rank;

MPI_Type_create_subarray(2, a_size, sub_size,
       sub_start, MPI_ORDER_C, MPI_DOUBLE, &sub_type);

MPI_Type_commit(&sub_type);

if (rank==0)
  MPI_Recv(array[0], 1, sub_type, 1, 123,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
if (rank==1)
  MPI_Send(array[0], 1, sub_type, 0, 123,
  MPI_COMM_WORLD);

MPI_Type_free(&sub_type);
```

```
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
```

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 not necessarily performance – manual packing can be 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

User defined datatypes (part 2)

Datatype constructor examples

`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

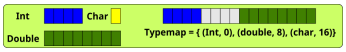
`MPI_Type_create_hvector`
like vector, but uses bytes for spacings

`MPI_Type_create_hindexed`
like index, but uses bytes for spacings

`MPI_Type_create_struct`
fully general datatype

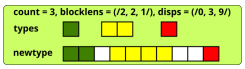
Understanding datatypes: typemap

- 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; it creates a new type from heterogeneous blocks
  - e.g. Fortran types and C structures
  - input is used to generate a correct typemap



Datatype constructors: MPI_TYPE_CREATE_STRUCT

`MPI_Type_create_struct(count, blocklens, displs, types, newtype)`

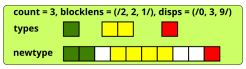
count
number of blocks

blocklens
lengths of blocks (array)

displs
displacements of blocks in bytes (array)

types
types of blocks (array)

newtype
new datatype



Example: sending a C struct

```c
/* Structure for particles */
struct ParticleStruct {
    int charge;         /* particle charge */
    double coord[3];    /* particle coordinates */
    double velocity[3]; /* particle velocity vector */
};
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);
```

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)`

pointer
pointer to the variable of interest

address
address of the variable, type is
- `MPI_Aint` (C)
- `integer(mpi_address_kind)` (Fortran)

Determining displacements

```c
/* Structure for particles */
struct ParticleStruct {
    int charge;         /* particle charge */
    double coords[3];   /* particle coordinates */
    double velocity[3]; /* particle velocity vector */
};

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;
```

Gaps between datatypes

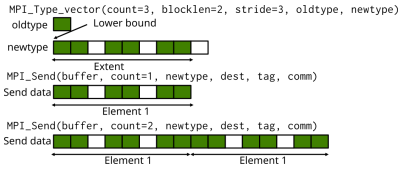
- Sending of an array of the `ParticleStruct` structures may have a portability issue: it assumes that array elements are packed in memory
  - Implicit assumption: the **extent** of the datatype was the same as the size of the C struct
  - This is not necessarily the case
- If there are gaps in memory between the successive structures, sending does not work correctly

Type extent

Sending multiple elements: Extent

- When communicating multiple elements, MPI uses the concept of extent
  - next element is read or write *extent* bytes apart from the previous one in the buffer
- Extent is determined from the displacements and sizes of the basic types in the typemap
  - The lower bound (LB) = min(displacement)
  - Extent = max(displacement + size) - LB + padding
- Communicating multiple user-defined types at once may not behave as expected if there are gaps in the beginning or end of the derived type

Multiple MPI_TYPE_VECTORs



Getting 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)

Setting extent and lower bound

`MPI_Type_create_resized(type, lb, extent, newtype)`

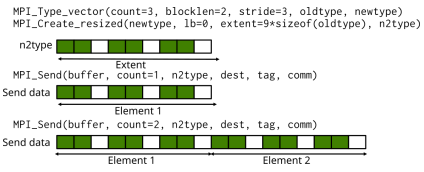
`type`
Old datatype

`lb`
New lower bound (in bytes)

`extent`
New extent (in bytes)

`newtype`
New datatype, commit before use

Multiple MPI_TYPE_VECTORs



Example: sending an array of structs portably

```c
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

```c
struct ParticleStruct particle[1000];
int psize;

psize = sizeof(particle[0]);
MPI_Send(particle, 1000*psize, MPI_BYTE, ...);
```

- 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

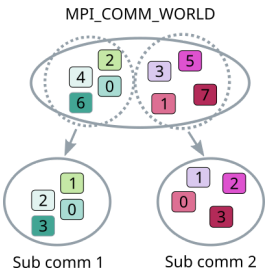
Summary

- 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
- This time we focused on the most general type specification: `MPI_Type_create_struct`
- Introduced the concepts of extent and typemap

Process topologies

Communicators

- Communicators are dynamic
- A task can belong simultaneously to several communicators
  - Unique rank in each communicator



Process topologies

- MPI topology mechanism adds additional information about the communication pattern to a communicator
- MPI topology can provide convenient naming scheme of processes
- MPI topology may assist the library and the runtime system in optimizations
  - In most implementations main advantage is, however, better programmability
- Topologies are defined by creating special user defined communicators

Virtual topologies

- MPI topologies are virtual, *i.e.* they do not have have necessarily relation to the physical structure of the supercomputer
- The assignment of processes to physical CPU cores happens typically outside MPI (and before MPI is initialized)
- The physical structure can in principle be taken account when creating topologies, however, MPI implementations may not implement that in practice

Virtual topologies

- A communication pattern can be represented by a graph: nodes present processes and edges connect processes that communicate with each other
- We discuss here only Cartesian topology which represents a regular multidimensional grid.

Two dimensional Cartesian grid

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

- Row major numbering
- Topology places no restrictions on communication
  - any process can communicate with any other process
- Any dimension can be finite or periodic

Communicator in Cartesian grid: MPI_Cart_create

```
MPI_Cart_create(oldcomm, ndims, dims, periods, reorder, newcomm)
```

oldcomm
communicator

ndims
number of dimensions

dims
integer array (size ndims) that defines the number of processes in each dimension

periods
array that defines the periodicity of each dimension

reorder
is MPI allowed to renumber the ranks

newcomm
new Cartesian communicator

Determining division: MPI_Dims_create

- Decompose a given number of processes into balanced distribution

```
MPI_Dims_create(ntasks, ndims, dims)
```

ntasks
number of tasks in a grid

ndims
number of dimensions

dims
integer array (size ndims). A value of 0 means that MPI fills in suitable value

Translating rank to coordinates

- Checking the Cartesian communication topology coordinates for a specific rank

`MPI_Cart_coords(comm, rank, maxdim, coords)`

comm
Cartesian communicator

rank
rank to convert

maxdim
length of the coords vector

coords
coordinates in Cartesian topology that corresponds to rank

Translating coordinates to rank

- Checking the rank of the process at specific Cartesian communication topology coordinates

`MPI_Cart_rank(comm, coords, rank)`

comm
Cartesian communicator

coords
array of coordinates

rank
a rank corresponding to coords

Creating a Cartesian communication topology

```
dims = 0
period=(/ .true., .false. /)

call mpi_dims_create(ntasks, 2, dims, rc)
call mpi_cart_create(mpi_comm_world, 2, dims, period, .true., comm2d, rc)
call mpi_comm_rank(comm2d, my_id, rc)
call mpi_cart_coords(comm2d, my_id, 2, coords, rc)
```

How to communicate in a Cartesian topology

`MPI_Cart_shift(comm, direction, displ, source, dest)`

comm
Cartesian communicator

direction
shift direction (0 or 1 in 2D)

displ
shift displacement (1 for next cell etc, < 0 for source from “down”/“right” directions)

source
rank of source process

dest
rank of destination process

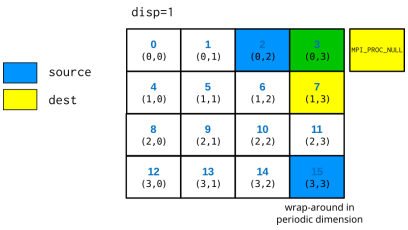
How to communicate in a Cartesian topology

- Note! *Both* `source` and `dest` are *output* parameters. The coordinates of the calling task is implicit input.
- `source` and `dest` are defined as for a shift like operation: receive from source, send to destination

$$\text{displ} = 1 \implies \begin{cases} \text{source} = \text{mycoord} - 1 \\ \text{dest} = \text{mycoord} + 1 \end{cases}$$

- With a non-periodic grid, source or dest can land outside of the grid
  - `MPI_PROC_NULL` is then returned

How to communicate in a Cartesian topology



Halo exchange

```
call mpi_cart_shift(comm2d, 0, 1, nbr_up, nbr_down, rc)
call mpi_cart_shift(comm2d, 1, 1, nbr_left, nbr_right, rc)
...

! left boundaries: send to left, receive from right
call mpi_sendrecv(buf(1,1), 1, coltype, nbr_left, tag_left, &
                  buf(1,n+1), 1, coltype, nbr_right, tag_left, &
                  comm2d, mpi_status_ignore, rc)

! right boundaries: send to right, receive from left
...
! top boundaries: send to above, receive from below
call mpi_sendrecv(buf(1,1), 1, rowtype, nbr_up, tag_up, &
                  buf(n+1,1), 1, rowtype, nbr_down, tag_up, &
                  comm2d, mpi_status_ignore, rc)

! bottom boundaries: send to below, receive from above
...
```

Summary

- Process topologies provide a convenient referencing scheme for grid-like decompositions
- Usage pattern
  - Define a process grid with `MPI_Cart_create`
  - Use the obtained new communicator as the `comm` argument in communication routines
  - For getting the ranks of the neighboring processes, use `MPI_Cart_shift` or wrangle with `MPI_Cart_coords` and `MPI_Cart_rank`
- MPI provides also more general graph topologies

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
- Collective operations are also available as non-blocking versions

Non-blocking send

Parameters similar to `MPI_Send` but has an additional request parameter.

`MPI_Isend(buffer, count, datatype, dest, tag, comm, request)`

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 (`type(mpi_request)` in Fortran, `MPI_Request` in C)

Non-blocking receive

Parameters similar to `MPI_Recv` but has no status parameter.

`MPI_Irecv(buffer, count, datatype, source, tag, comm, request)`

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)
  - Remember: successfully finished send does not mean successful receive!
- 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)`

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

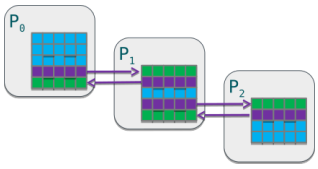
status
status for the completed operation

A call to `MPI_Test` is non-blocking. It allows one to schedule alternative activities while periodically checking for completion.

`MPI_Probe` is a similar kind of operation (see later slides).

Typical usage pattern

```
MPI_Irecv(ghost_data)
MPI_Isend(border_data)
compute(ghost_independent_data)
MPI_Waitall compute(border_data)
```



Additional completion operations

Routine	Meaning
MPI_Waitany	Waits until any one operation has completed
MPI_Waitsome	Waits until at least one operation has completed
MPI_Test	Tests if 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

Returns when one or more operations is/are 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

Allows incoming messages to be checked, without actually receiving them.

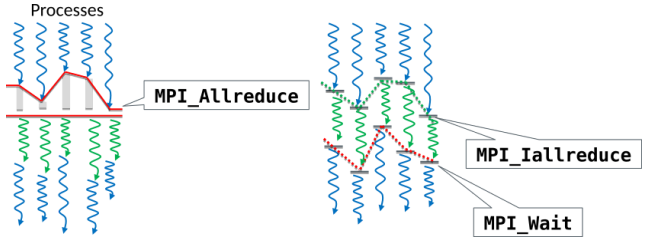
Non-blocking collectives

- Non-blocking collectives ("`I`-collectives") enable the overlapping of communication and computation together with the benefits of collective communication.
- Same syntax as for blocking collectives, besides
  - "I" at the front of the name (`MPI_Alltoall` -> `MPI_Ialltoall`)
  - Request parameter at the end of the list of arguments
  - 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 is not allowed

Non-blocking collectives



(Computation) work 1 (Computation) work 2, not involving data in the `Allreduce` operation

Example: Non-blocking broadcasting

```
MPI_Ibcast(buffer, count, datatype, root, comm, request)
```

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"
- Three separate phases:
  1. Initiation of communication
  2. Starting of communication
  3. Completing communication
- Recently published MPI 4.0 includes also persistent collectives
  - Not supported by all implementations yet

Persistent communication

- Inititiate communication by creating requests
  - `MPI_Send_init` and `MPI_Recv_init`
  - Same arguments as in `MPI_Isend` and `MPI_Irecv`
- Start communication
  - `MPI_Start` / `MPI_Startall`
  - Request or array of requests as argument
- Complete communication
  - `MPI_Wait` / `MPI_Waitall`
  - Same as in standard non-blocking communication

Persistent point-to-point communication

```
MPI_Request recv_req, send_req;
...
// Initialize send/request objects
MPI_Recv_init(buf1, cnt, MPI_DOUBLE, src, tag, MPI_COMM_WORLD, &recv_req);
MPI_Send_init(buf2, cnt, MPI_DOUBLE, dst, tag, MPI_COMM_WORLD, &send_req);
for (int i=1; i<BIGNUM; i++){
// Start communication described by recv_obj and send_obj
    MPI_Start(&recv_req);
    MPI_Start(&send_req);
    // Do work, e.g. update the interior domains
    ...
    // Wait for send and receive to complete
    MPI_Wait(&send_req, MPI_STATUS_IGNORE);
    MPI_Wait(&recv_req, MPI_STATUS_IGNORE);
}
//Clean up the requests
MPI_Request_free (&recv_req); MPI_Request_free (&send_req);
```

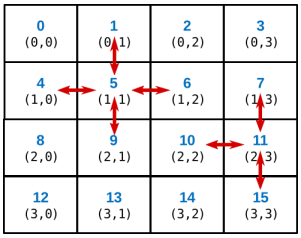
Neighborhood collectives

Neighborhood collectives

- Neighborhood collectives build on top of process topologies
- Provide optimization possibilities for MPI library for communication patterns involving neighbors
  - Nearest neighbors in cartesian topology
    - Processes connected by a edge in a general graph
- Similar to ordinary collectives, all tasks within a communicator need to call the routine
- Possible to have multidimensional halo-exchange with a single MPI call

Neighborhood collectives in cartesian grid

- Only nearest neighbors, *i.e.* those corresponding to `MPI_Cart_shift` with displacement=1.
- Boundaries in finite dimensions treated as like with `MPI_PROC_NULL`



Neighborhood collectives

- Two main neighborhood operations
  - `MPI_Neighbor_allgather` : send same data to all neighbors, receive different data from neighbors
    - `MPI_Neighbor_alltoall` : send and receive different data between all the neighbors
- Also variants where different number or type of elements is communicated
- Non-blocking versions with similar semantics than non-blocking collectives
  - Request parameter at the end of the list of arguments

Summary

- Non-blocking communication is often useful 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
- Collective operations can also be done in non-blocking mode

Summary

- In persistent communication the communication pattern remains constant
- All the parameters for the communication are set up in the initialization phase
  - Communication is started and finalized in separate steps
- Neighborhood collectives enable communication between neighbors in process topology with a single MPI call
- Persistent and neighborhood communication provide optimization opportunities for MPI library

Further MPI topics

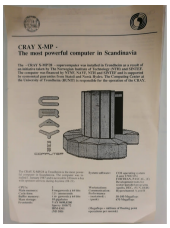
- One-sided communication
- MPI shared memory programming
- MPI error handling
- Intercommunicators
- Point-to-point communication modes
- Dynamic process creation

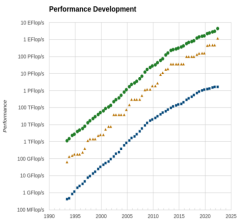
Parallel input/output

What is high-performance computing?

- Utilizing computer power that is much larger than available in typical desktop computer - would take too long, explore many different parameters, not enough memory, not enough disk space
- 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 ~1 x 10^{18} flop/s (1 Eflop / s)

Exponential growth of performance





- frequency growth stopped around 2005
- modern (super)computers rely on parallel processing

HPC and I/O data

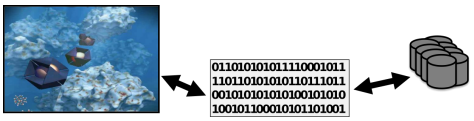
- Reading initial conditions or datasets for processing
- Storing numerical output from simulations for later analysis
- Checkpointing to files for restarting in case of system failure
- 'out-of-core' techniques when the data is larger than one can fit in system memory

Heat Equation example

- Parallel heat equation
- Check the scaling for different saving intervals

Parallel I/O

- Mapping problem: how to convert internal structures and domains to files which are streams of bytes
- Transport problem: how to get the data efficiently from hundreds to thousands of nodes on the supercomputer to physical disks



- ***Users need to understand the I/O infrastructure!***

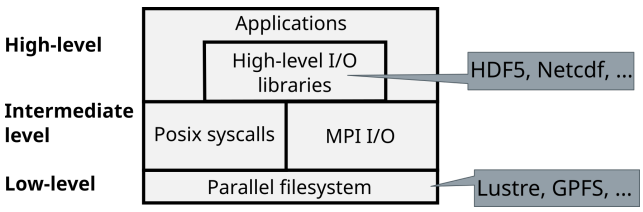
Parallel I/O

- 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

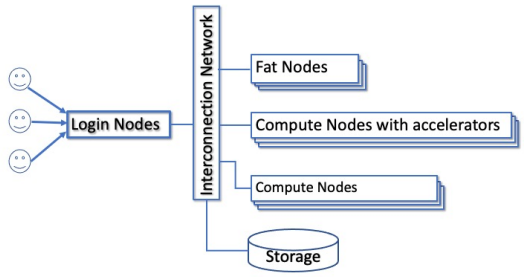


I/O library choice

- POSIX and MPI-I/O libraries
  - 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
  - Gives tools for writing data + metadata, e.g. HDF5
  - Or even provide a application or domain specific schema for what data + metadata to describe a particular kind of data

Parallel File systems

Anatomy of supercomputer



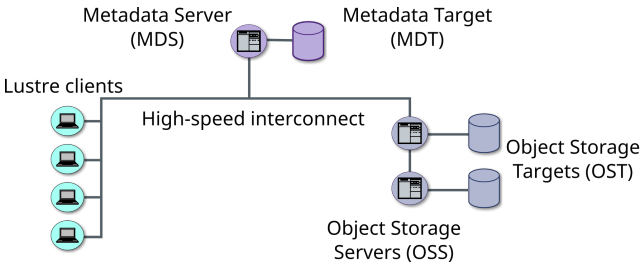
File systems

- All large parallel computer systems provide a parallel file system area
  - Files can be accessed from all tasks
  - Large systems often have dedicated I/O nodes
- Some systems also provide a local disk area for temporary storage
  - Only visible to tasks on the same node
  - Results have to be copied after simulation

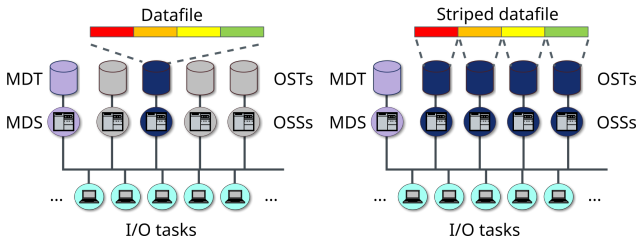
Lustre

- Lustre is a popular parallel file system that is used in many large systems
  - Also at CSC (Puhti, Mahti, Lumi)
- Separated storage of data and metadata
  - Single metadata server
  - Clustered data storage
  - Supports e.g. striping of large datafiles for higher performance

Lustre architecture



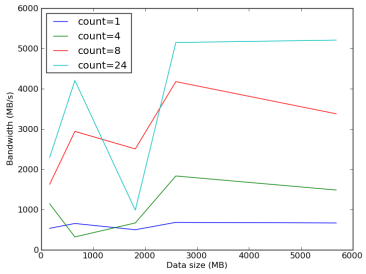
Lustre file striping



Lustre file striping

- Striping pattern of a file/directory can be queried or set with the `lfs` command
- `lfs getstripe` <*dir*|*file*>
- `lfs setstripe` -c *count dir*
  - Set the default stripe count for directory *dir* to *count*
  - All the new files within the directory will have the specified striping
  - Also stripe size can be specified, see *man lfs* for details
- Proper striping can enhance I/O performance a lot

Performance with Lustre striping



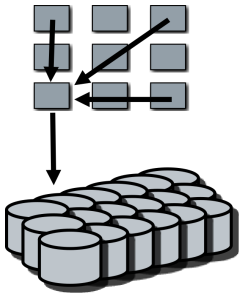
Performance considerations

- Files are often assigned in a round robin fashion over multiple OSTs
- You can overwhelm an OST, OSS or MDS
- Per node limits on bandwidth
- Testing can be super noisy, performance depends on the load
- Very large cache effects
- More stripes does not automatically improve performance

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)

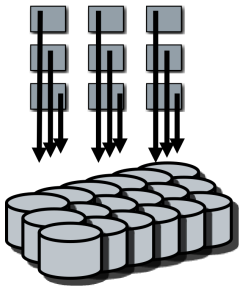


Example: spokesperson strategy

```
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
```

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 the filesystem (for example Lustre metadata)



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 stdout/err!
  - Interim results, timings,...

Summary

- Parallel file system is needed for efficient parallel I/O
  - Striping of files
- Primitive parallel I/O can be achieved using just normal Posix calls (+ MPI communication)
  - Spokesman strategy
  - Every man for himself
  - Subset of writers/readers

MPI-IO

- Defines parallel 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-IO

- 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

Basic concepts in MPI-IO

- File *view*
  - part of a parallel file which is visible to process
  - enables efficient noncontiguous access to file
- *Collective* and *independent* I/O
  - Collective = MPI coordinates the reads and writes of processes
  - Independent = no coordination by MPI, every man for himself

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`, ...
- Mode parameters can be combined with + in Fortran and | in C/C++

info
Hints to implementation for optimal performance (No hints: `MPI_INFO_NULL`)

fhandle
parallel file handle

- File is closed using `MPI_File_close(fhandle)`

File writing at explicit location

`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 from where to write the data

count
number of elements in the buffer

datatype
datatype of elements to write

status
similar to status in `MPI_Recv`, stores number of elements actually written

Example: parallel write

```
program output
use mpi_f08
implicit none
type(mpi_file) :: file
integer :: err, i, myid, 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
```

- First process writes integers 1-100 to the beginning of the file, etc.

Example: parallel write

```
...

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 at explicit location

`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 in the buffer

datatype
datatype of elements to read

status
similar to status in `MPI_Recv`, stores number of elements actually read

Setting file pointer

- In previous examples the location for writing or reading was provided explicitly in the `write` / `read` calls.
- It is also possible to set the location of file pointer separately with

`MPI_File_seek(fhandle, disp, whence)`

disp
displacement in bytes (with the default file view)

whence
`MPI_SEEK_SET`: the pointer is set to `disp`
`MPI_SEEK_CUR`: the pointer is set to the current pointer position plus `disp`
`MPI_SEEK_END`: the pointer is set to the end of file plus `disp` (can be negative)

File writing at file pointer

`MPI_File_write(fhandle, buffer, count, datatype, status)`

buffer
buffer in memory which holds the data

count
number of elements to write

datatype
datatype of elements to write

status
status object

- Updates position of individual file pointer after writing
  - Not thread safe

File reading at file pointer

`MPI_File_read(fhandle, buffer, count, datatype, status)`

buffer
buffer in memory where to store 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`

- Updates position of individual file pointer after reading
  - Not thread safe

Collective operations

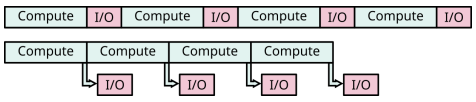
- I/O can be performed *collectively* by all processes in a communicator
  - `MPI_File_read_all`
  - `MPI_File_write_all`
  - `MPI_File_read_at_all`
  - `MPI_File_write_at_all`
- Same parameters as in independent I/O functions (`MPI_File_read` etc.)

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. Specifies the 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

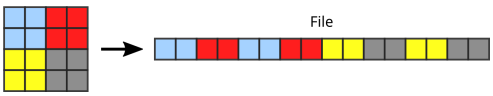
info
Hints for implementation that can improve performance (`MPI_INFO_NULL`: no hints)

File view

- The values for `datarep` and the extents of `etype` must be identical on all processes in the communicator.
  - values for `disp`, `filetype`, and `info` may vary.
- The datatypes passed in must be committed.

File view for non-contiguous data

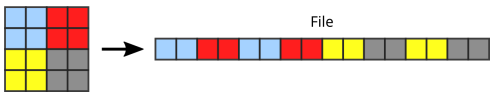
Decomposition for a 2D array



- 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




```
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-IO

- 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
  - Collective write can be over hundred times faster than the individual for large arrays!
- Remember to use the correct striping for each file size
  - Striping can also be set in the `MPI_Info` parameter using the `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

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
  - Poison for many parallel filesystems
- MPI-IO 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
  - Visualisation much easier for complicated cases
- Typically, I/O libraries support also metadata
  - Self-describing files, pays off in the long run
- Parallel I/O is typically build on top of MPI-IO
- Generic
  - **HDF5**, **NetCDF**, ADIOS, SIONlib
- Framework specific/builtin

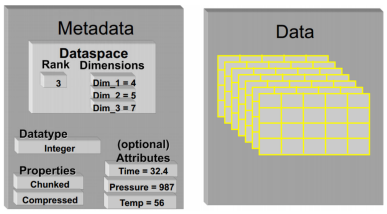
I/O libraries

- Consider the full data cycle
  - Raw IO performance is important but not the sole thing to focus on
- Standards and common formats in your field
  - Weather, PIC, AMR
- Lots of tuning still required, read the documentation!
  - Understanding the underlying system
- Future scalability
  - Dream big, but don't overengineer.
- File format and data format are different things.

HDF5

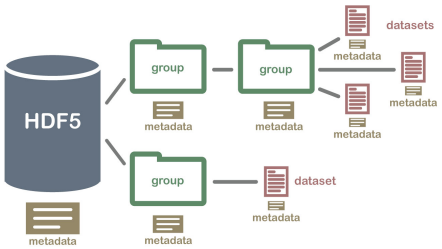
- A data model, library, and file format for storing and managing multidimensional data
- Can store complex data objects and meta-data
- File format and files are *portable*
- Possibility for parallel I/O on top of MPI-IO
- Library provides Fortran and C/C++ API
  - Third party interfaces for Python, R, Java
  - Many tools can work with HDF5 files (Paraview, Matlab, ...)
- The HDF5 data model and library are complex

HDF5 example dataset



Source: http://press3.mcs.anl.gov/computingschool/files/2014/01/QKHDF5-intro-v2.pdf

HDF5 hierarchy



Exercise

Discuss within the group

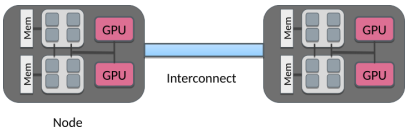
- Are there any very common data formats in your field
- What does your process for the data look like?
  - Keeping track of what is what
  - Is the format hurting or helping your data-analysis
  - Will it be shared and with what kind of target audience?
- Estimate the size and bandwidth requirements of your input and output for current/future use cases

Hybrid programming

Introduction

Anatomy of supercomputer

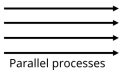
- Supercomputers consist of nodes connected with high-speed network
  - Latency ~1 µs, bandwidth ~200 Gb / s
- A node can contain several multicore CPUS
- Additionally, nodes can contain one or more accelerators
- Memory within the node is directly usable by all CPU cores

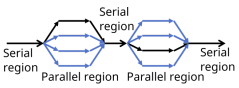


Parallel programming models

- 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

Parallel programming models





MPI: Processes

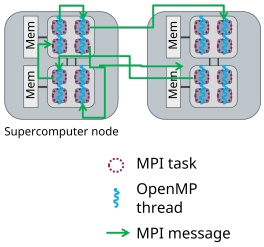
- Independent execution units
- Have their **own** memory space
- MPI launches N processes at application startup
- Works over multiple nodes

OpenMP: Threads

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

Hybrid programming: Launch threads (OpenMP) *within* processes (MPI)

- Shared memory programming inside a node, message passing between nodes
- Matches well modern supercomputer hardware
- Optimum MPI task per node ratio depends on the application and should always be experimented.



Example: Hybrid hello

```
#include <mpi.h>
#include <omp.h>

int main(int argc, char *argv[]) {
    int my_id, omp_rank;
    int provided, required=MPI_THREAD_FUNNELED;

    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();
}
```

```
$ mpicc -fopenmp hybrid-hello.c -o hybrid-hello
$ srun  --ntasks=2 --cpus-per-task=4
  ./hybrid-hello

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
```

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 to different tasks
  - e.g. a thread dedicated to communication or parallel I/O
- Dynamic parallelization patterns often easier to implement with OpenMP

Disadvantages of hybridization

- Increased overhead from thread creation/destruction
- More complicated programming
  - Code readability and maintainability issues
- Thread support in MPI and other libraries needs to be considered

Alternatives to OpenMP within a node

- pthreads (POSIX threads)
- Multithreading support in C++ 11
- Performance portability frameworks (SYCL, Kokkos, Raja)
- Intel Threading Building Blocks

OpenMP

What is OpenMP?

- A collection of *compiler directives* and *library routines* , together with a *runtime system*, for **multi-threaded**, **shared-memory parallelization**
- Fortran 77/9X/0X and C/C++ are supported
- Latest version of the standard is 5.2 (November 2021)
  - Full support for accelerators (GPUs)
  - Support latest versions of C, C++ and Fortran
  - Support for a fully descriptive loop construct
  - and more
- Compiler support for 5.0 is still incomplete
- This course discusses mostly features present in < 4.5

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
  - Expresses shared memory parallelization
  - Preceded by sentinel, can compile serial version
- Runtime library routines
  - Small number of library functions
  - Can be discarded in serial version via conditional compiling
- Environment variables
  - Specify the number of threads, thread affinity etc.

OpenMP directives

- OpenMP directives consist of a *sentinel*, followed by the directive name and optional clauses
- C/C++:

```
#pragma omp directive [clauses]
```

- Fortran:

```
!$omp directive [clauses]
```

- Directives are ignored when code is compiled without OpenMP support

Compiling an OpenMP program

- Compilers that support OpenMP usually require an option (flag) for enabling it
  - Most compilers (GNU, Intel, Cray) nowadays support `-fopenmp`
  - Intel legacy foption: `-qopenmp`
  - NVIDIA: `-mp`

Parallel construct

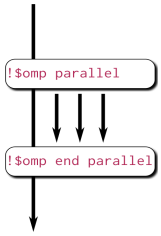
- Defines a *parallel region*
  - C/C++:

```
#pragma omp parallel [clauses]
  structured block
```

  - Fortran:

```
!$omp parallel [clauses]
   structured block
!$omp end parallel
```

- Prior to it only one thread (main)
- Creates a team of threads
- Barrier at the end of the block
- SPMD: Single Program Multiple Data



Example: “Hello world” with OpenMP

```
program omp_hello

   write(*,*) "Hello world! -main"
!$omp parallel
   write(*,*) ".. worker reporting for duty."
!$omp end parallel
   write(*,*) "Over and out! -main"

end program omp_hello
```

```
$ gfortran -fopenmp omp_hello.F90 -o omp
$ OMP_NUM_THREADS=3 ./omp
 Hello world! -main
 .. worker reporting for duty.
 .. worker reporting for duty.
 .. worker reporting for duty.
 Over and out! -main
```

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  printf("Hello world! -main\n");
#pragma omp parallel
  {
    printf(".. worker reporting for duty.\n");
  }
  printf("Over and out! -main\n");
}
```

```
$ gcc -fopenmp omp_hello.c -o omp
$ OMP_NUM_THREADS=3 ./omp
Hello world! -main
.. worker reporting for duty.
.. worker reporting for duty.
.. worker reporting for duty.
Over and out! -main
```

How to distribute work?

- Each thread executes the same code within the parallel region
- OpenMP provides several constructs for controlling work distribution
  - `for`/`do` construct
  - `workshare` construct (Fortran only)
  - `single`/`master`/`masked` construct
  - `task` construct
  - `distribute` construct (for GPUs)
  - `loop` construct
  - `sections` construct
- Thread ID can be queried and used for distributing work manually (similar to MPI rank)

for/do construct

- Directive instructing compiler to share the work of a loop
  - Each thread executes only part of the loop

```
#pragma omp for [clauses]
...
```

```
!$omp do [clauses]
...
!$omp end do
```

- in C/C++ limited only to “canonical” for-loops. Iterator base loops are also possible in C++
- Construct must reside inside a parallel region
  - Combined construct with omp parallel:
    `#pragma omp parallel for` / `!$omp parallel do`

for/do construct

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

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

Workshare directive (Fortran only)

- In Fortran many array operations can be done conveniently with array syntax, *i.e.* without explicit loops
  - Array assignments, forall and where statements *etc.*
- The `workshare` directive divides the execution of the enclosed structured block into separate units of work, each of which is executed only once

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

Summary

- OpenMP can be used with compiler directives
  - Ignored when code is build without OpenMP
- Threads are launched within **parallel** regions
- Simple loops can be parallelized easily with a `for`/`do` construct

OpenMP runtime library and environment variables

OpenMP runtime library and environment variables

- 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`

Some useful environment variables

Variable	Action
OMP_NUM_THREADS	Number of threads to use
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

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 finding out thread ID and number of threads:
  - `omp_get_thread_num()`
  - `omp_get_num_threads()`

OpenMP conditional compilation

- Conditional compilation with `_OPENMP` macro:

```
#ifdef _OPENMP
    OpenMP specific code with, e.g., library calls
#else
    Code without OpenMP
#endif
```

Example: Hello world with OpenMP

```
program hello
  use omp_lib
  integer :: omp_rank
!$omp parallel
#ifdef _OPENMP
  omp_rank = omp_get_thread_num()
#else
  omp_rank = 0
#endif
  print *, 'Hello world! by &
        thread ', omp_rank
!$omp end parallel
end program hello
```

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char argv[]) {
  int omp_rank;
#pragma omp parallel
  {
#ifdef _OPENMP
    omp_rank = omp_get_thread_num();
#else
    omp_rank = 0;
#endif
    printf("Hello world! by thread %d\n",
           omp_rank);
  }
}
```

Parallel regions and data sharing

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
- If threads write to the same shared variable a **race condition** appears
  - Undefined end result
- Visibility of different variables is defined using *data-sharing clauses* in the parallel region definition

Race condition in Hello world

```
#include <stdio.h>
#include <omp.h>
#include <unistd.h>

int main(int argc, char argv[]) {
  int omp_rank;
#pragma omp parallel
  {
    omp_rank = omp_get_thread_num();
    sleep(1);
    printf("Hello world! by thread %d\n", omp_rank);
  }
}
```

- All the threads write out the same thread number
- The result varies between successive runs

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
- **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:
  - Local variables of functions called from parallel region
  - Variables declared within a block (C/C++)
  - Outermost loop variables
- Good programming practice: declare all variables either shared or private

Hello world without a race condition

```c
#include <stdio.h>
#include <omp.h>
#include <unistd.h>

int main(int argc, char argv[]) {
  int omp_rank;
#pragma omp parallel private(omp_rank)
  {
    omp_rank = omp_get_thread_num();
    sleep(1);
    printf("Hello world! by thread %d\n", omp_rank);
  }
}
```

Data sharing example

main.c

```c
int A[5];  // shared

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

kernel.c

```c
extern int A[5];  // shared

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

Summary

- OpenMP runtime behavior can be controlled using environment variables
- OpenMP provides also library routines
- Visibility of variables in parallel region can be specified with data sharing clauses
  - **private** : each thread works with their own variable
  - **shared** : all threads can write to and read from a shared variable
- Race conditions possible when writing to shared variables
- Avoiding race conditions is key to correctly functioning OpenMP programs

OpenMP reductions

Race condition in reduction

- 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

Reductions

- Summing elements of array is an example of reduction operation

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

- OpenMP provides support for common reductions within parallel regions and loops

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 in C/C++

Operator	Initial value
+	0
-	0
*	1
&&	1
||	0

Bitwise Operator	Initial value
&	~0
|	0
^	0

Reduction operators in Fortran

Operator	Initial value
+	0
-	0
*	1
max	least
min	largest
.and.	.true.
.or.	.false.
.eqv.	.true.
.neqv.	.false.

Bitwise Operator	Initial value
.iand.	all bits on
.ior.	0
.ieor.	0

Race condition avoided with reduction clause

```
!$omp parallel do shared(x,y,n) private(i) reduction(+:asum)
  do i = 1, n
     asum = asum + x(i)*y(i)
  end do
!$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];
}
```

OpenMP execution controls

Execution controls

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

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 a team
- Implicit barrier at the end of: `for/do`, `parallel`, `single`, `workshare` unless a `nowait` clause is specified

Execution control constructs

`master`

- Specifies a region that should be executed only by the master thread
- Other threads do not wait, *i.e.* no implicit barrier at the end
- Deprecated in OpenMP 5.1 and replaced with `masked`

`single`

- Specifies that a region should be executed only by a single (arbitrary) thread
- Other threads wait (implicit barrier) unless a `nowait` clause is specified

Execution control constructs

`critical`

- A section that is executed by only one thread at a time
- No implicit barrier at the end

`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

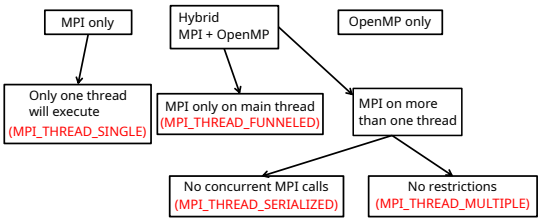
Summary

- Several parallel reduction operators available via `reduction` clause
- OpenMP has many synchronization constructs
  - `barrier`
  - `single` and `master/masked`
  - `critical` and `atomic`

Web resources

- OpenMP homepage: http://openmp.org/
- Online tutorials: https://www.openmp.org/resources/tutorials-articles/

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)

Multiple thread communication

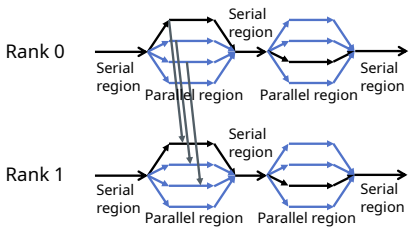
- Hybrid programming is relatively straightforward in cases where communication is done by only a single thread at a time
- With the "multiple" mode all threads can make MPI calls independently

```
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

MPI thread support levels

- Modern MPI libraries support all threading levels
  - OpenMPI: Build time configuration, check with
    ```
    ompi_info | grep 'Thread support'
    ```
  - Intel MPI: When compiling with `-qopenmp` a thread safe version of the MPI library is automatically used
  - Cray MPI: Set `MPICH_MAX_THREAD_SAFETY` environment variable to `single`, `funneled`, `serialized`, or `multiple` to select the threading level
- 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

Summary

- Multiple threads may make MPI calls simultaneously
- Thread specific tags and/or communicators
- For collectives it is often better to use a single thread for communication

Thread and process affinity

Thread and process affinity

- Normally, operating system can run threads and processes in any logical core
- Operating system may even move the running process/thread from one core to another
  - Can be beneficial for load balancing
  - For HPC workloads often detrimental as private caches get invalidated and NUMA locality is lost
- User can control where tasks are run via affinity masks
  - Task can be *pinned* to a specific logical core or set of logical cores

Controlling affinity

- Affinity for a *process* can be set with a `numactl` command
  - Limit the process to logical cores 0,3,7:
    ```
    numactl --physcpubind=0,3,7 ./my_exe
    ```
  - Threads "inherit" the affinity of their parent process
- Affinity of a thread can be set with OpenMP environment variables
  - `OMP_PLACES=[threads,cores,sockets]`
  - `OMP_PROC_BIND=[true, close, spread, master]`
- OpenMP runtime prints the affinity with `OMP_DISPLAY_AFFINITY=true`

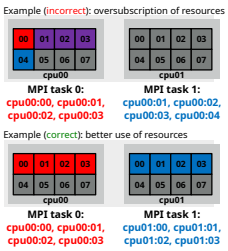
Controlling affinity

```
export OMP_AFFINITY_FORMAT="Thread %0.3n affinity %A"
export OMP_DISPLAY_AFFINITY=true
./test
Thread 000 affinity 0-7
Thread 001 affinity 0-7
Thread 002 affinity 0-7
Thread 003 affinity 0-7
```

```
OMP_PLACES=cores ./test
Thread 000 affinity 0,4
Thread 001 affinity 1,5
Thread 002 affinity 2,6
Thread 003 affinity 3,7
```

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!



Slurm configuration at CSC

- Within a node, `--tasks-per-node` MPI tasks are spread `--cpus-per-task` apart
- Threads within a MPI tasks have the affinity mask for the corresponging `--cpus-per-task` cores

```
export OMP_AFFINITY_FORMAT="Process %P thread %0.3n affinity %A"
export OMP_DISPLAY_AFFINITY=true
srun ... --tasks-per-node=2 --cpus-per-task=4 ./test
Process 250545 thread 000 affinity 0-3
...
Process 250546 thread 000 affinity 4-7
...
```

- Slurm configurations in other HPC centers can be very different
  - Always experiment before production calculations!

Summary

- Performance of HPC applications is often improved when processes and threads are pinned to CPU cores
- MPI and batch system configurations may affect the affinity
  - very system dependent, try to always investigate

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 tasks enable parallelisation of irregular and dynamical patterns
  - 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 a single thread
  - Can be same or different thread that created the task
  - OpenMP runtime takes care of distributing tasks to threads
- Execution may be deferred or started immediately after tasks is created
- Tasks are created by
  - Standard worksharing constructs (implicit tasks)
  - `task` construct (explicit tasks)

OpenMP task construct

- Create a new task and add it to task queue
  - Memorise data and code to be executed
  - Task constructs can be arbitrarily nested
- Syntax (C/C++)

```
#pragma omp task [clause[[,] clause],...]
{
    ...
}
```

- Syntax (Fortran)

```
!$omp task[clause[[,] clause],...]
...
!$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?

a) 6 b) 4 c) 24

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

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 an arbitrary order
  - Each task is executed only once
- Synchronisation 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 `private` in the enclosing construct are made `firstprivate` and contain the data at the time of creation
- Variables that are `shared` in the enclosing construct contain the data at the time of execution
- Data scoping clauses (`shared`, `private`, `firstprivate`, `default`) can change the default behaviour

Recursive algorithms with tasks

- A task can itself generate new tasks
  - Useful when parallelising recursive algorithms
- Recursive algorithm for Fibonacci numbers:

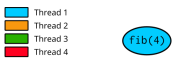
$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}$

```
#pragma omp parallel
{
    #pragma omp single
    fib(10);
}
```

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

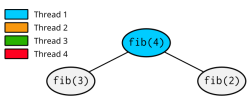
Tasking illustrated

- Thread 1 enters `fib(4)`



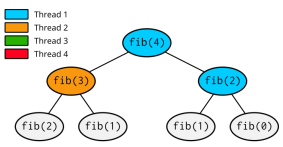
Tasking illustrated

- Thread 1 enters `fib(4)`
- Thread 1 creates tasks for `fib(3)` and `fib(2)`



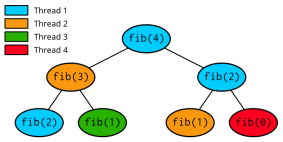
Tasking illustrated

- Thread 1 enters `fib(4)`
- Thread 1 creates tasks for `fib(3)` and `fib(2)`
- Threads 1 and 2 execute tasks from the queue and create four new tasks



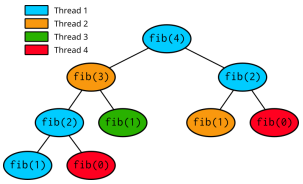
Tasking illustrated

- Thread 1 enters `fib(4)`
- Thread 1 creates tasks for `fib(3)` and `fib(2)`
- Threads 1 and 2 execute tasks from the queue and create four new tasks
- Threads 1-4 execute tasks



Tasking illustrated

- Thread 1 enters `fib(4)`
- Thread 1 creates tasks for `fib(3)` and `fib(2)`
- Threads 1 and 2 execute tasks from the queue and create four new tasks
- Threads 1-4 execute tasks
- ...



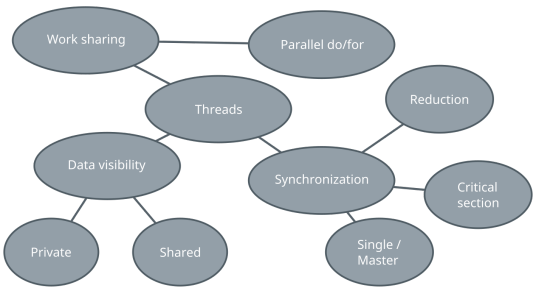
OpenMP programming best practices

- Maximise parallel regions
  - Reduce fork-join overhead, e.g. combine multiple parallel loops into one large parallel region
  - Potential for better cache re-usage
- Parallelise 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 parallelisation
  - Compiler directives, runtime API, environment variables
  - Relatively easy to get started but specially efficient and/or real-world parallelisation non-trivial
- Features touched in this intro
  - Parallel regions and work sharing constructs
  - Data-sharing clauses
  - Combining MPI and OpenMP
  - Task based parallelisation

OpenMP summary



Things that we did not cover

- sections construct
- scheduling clauses of `for`/`do` constructs
- task dependencies
- taskgroup and taskloop constructs
- simd construct
- ...

Web resources

- OpenMP homepage: http://openmp.org/
- Online tutorials: http://openmp.org/wp/resources/#Tutorials

Application performance

A day in life at CSC

CSC customer

I'm performing simulations with my Fortran code. It seems with MKL library in the new system than with IMSL library

No

CSC specialist

Have you profiled your code?

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          ! Removing these unnecessary loop iterations reduced the
 do j=1,m         ! wall-time of one simulation run from 17 hours to 3 seconds…
    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
```

Minding performance

Why worry about 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)
  - Several multicore CPUs and GPUs within one node
  - Superscalar out-of-order instruction execution
  - Multilevel coherent caches
  - SIMD vector units
  - SMT capabilities for multithreading
  - Separate memories for GPUs and CPUs
  - Complex communication topologies
- To get most out of the hardware, performance engineering is needed

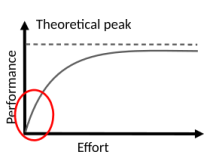
Factors affecting performance in HPC

- Single node performance
  - single core performance
  - single GPU performance
  - threading (and MPI within a node)
- Communication between nodes
- Input/output to disk

How to improve single node performance?

- Choose good algorithm
  - e.g. $O(N \log N)$ vs. $O(N^2)$
  - remember prefactor!
- Use high performance libraries
  - linear algebra (BLAS/LAPCK), FFTs, …
- Experiment with compilers and options
- Experiment with threading options
  - Thread pinning, loop scheduling, …
- Optimize the program code

```
./fibonacci 20
With loop, Fibonacci number i=20 is 6765
Time elapsed 79 ums
With recursion, Fibonacci number i=20 is 676
Time elapsed 343773 ums
```



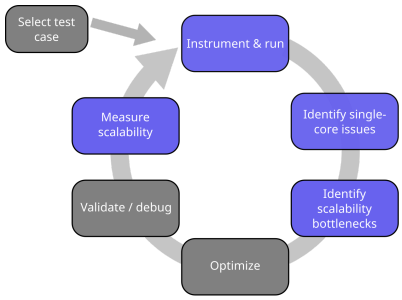
How to improve parallel scalability?

- Choose good algorithm
  - Note: the best serial algorithm is not always the best parallel algorithm
  - Computating more can be beneficial if less communication is needed
- Use high performance libraries
  - linear algebra (ScaLAPACK, ELPA), FFTs, …
- Mind the load balance
  - Load balancing algorithms might be complex and add overheads
- Utilize advanced MPI features
- Experiment with parameters of the MPI library
  - Short/long message threshold, collective algorithms, …

Performance engineering

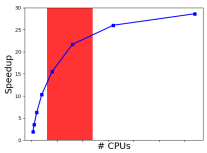
- First step should always be measuring the performance and finding performance critical parts
  - Typically small part of the code (~10 %) consumes most (~90%) of the execution time
- Optimize only the parts of code that are relevant for the total execution time!
  - “Premature code optimization is the root of all evil”
- Important to understand interactions
  - Algorithm – code – compiler – libraries – hardware
- Performance is not portable

Code optimization cycle



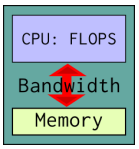
Selecting the test case

- 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



How to assess application’s performance?

- Two fundamental limits for CPU
- Peak floating point performance
  - clock frequency
  - number of instructions per clock cycle
  - number of FLOPS per instruction
  - number of cores
  - no real application achieves peak in sustained operation
- Main memory bandwidth
  - How fast data can be fed to the CPU



How to assess application’s performance?

- Example: maximum performance of **axpy**

```
x[i] = a x[i] + y[j]
```

  - Two FLOPS (multiply and add) per `i`
  - Three memory references per `i`
  - With double precision numbers arithmetic intensity $I = \frac{\text{FLOPS}}{\text{memorytraffic}} = \frac{2}{3*8} = 0.08$ FLOPS/byte
  - In Puhti, memory bandwidth is ~200 GB/s, so maximum performance is ~16 GFLOPS/s
  - Theoretical peak performance of Puhti node is ~2600 GFLOPS/s

How to assess application’s performance?

- Example: matrix-matrix multiplication

```
C[i,j] = C[i,j] + A[i,k] * B[k,j]
```

  - $2N^3$ FLOPS
  - $3N^2$ memory references
  - With double precision numbers arithmetic intensity $I = \frac{2N}{3}$ FLOPS/byte
  - With large enough N limited by peak performance

How to assess application’s parallel performance?

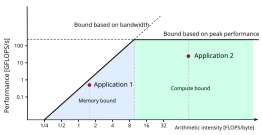
- First step should be always to optimize single core performance
  - May affect computation / communication balance
- Maximize single node performance
  - Dynamic scaling of clock frequency, shared caches etc. make scalability within node complex concept
  - Example: in Mahti independent computations (no parallel overheads) run 10 % faster with only one core per node than with full node
  - Memory bound applications may benefit from undersubscribing the node
- Lower limit for acceptable scalability between nodes
  - Speedup of 1.5 when doubling number of nodes

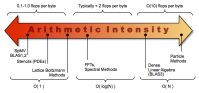
Roofline model

- Simple visual concept for maximum achievable performance
  - can be derived in terms of arithmetic intensity I, peak performance π and peak memory bandwidth β

$$P = min \begin{cases} \pi \\ \beta \times I \end{cases}$$

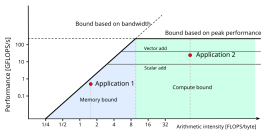
- Machine balance = arithmetic intensity needed for peak performance
  - Typical values 5-15 FLOPS/byte





Roofline model

- Model does not tell if code can be optimized or not
  - Application 1 may not be *fundamentally* memory bound, but only implemented badly (not using caches efficiently)
  - Application 2 may not have *fundamentally* prospects for higher performance (performs only additions and not fused multiply adds)
- However, can be useful for guiding the optimization work



Roofline model

- How to obtain the machine parameters?
  - CPU specs
  - own microbenchmarks
  - special tools (Intel tools, Empirical Roofline Tool)
- How to obtain application's GFLOPS/s and arithmetic intensity?
  - Pen and paper and timing measurements
  - Performance analysis tools and hardware counters
  - *True* number of memory references can be difficult to obtain

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
- Roofline model can work as a guide in optimization

How to start?

- What limits the performance?
  - Serial / OpenMP / GPU (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

Web resources

- Roofline performance model and Empiral Roofline Tool
  - https://crd.lbl.gov/departments/computer-science/par/research/roofline/

Single node performance optimization

Doesn't the compiler do everything?

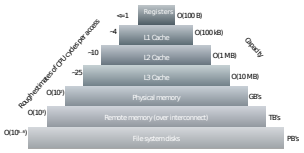
- You can make a big difference to code performance with how you express things
- Helping the compiler spot optimisation opportunities
- Using the insight of your application
  - language semantics might limit compiler
- Removing obscure (and obsolescent) "optimizations" in older code
  - Simple code 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

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

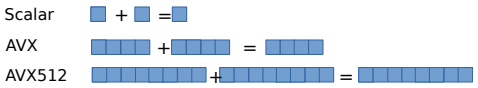
Cache memory

- In order to alleviate the memory bandwidth bottleneck, CPUs have multiple levels of cache memory
  - when data is accessed, it will be first fetched into cache
  - when data is reused, subsequent access is much faster
- L1 is closest to the CPU core and is fastest but has smallest capacity
- Each successive level has higher capacity but slower access



Vectorization

- Modern CPUs have SIMD (Single Instruction, Multiple Data) units and instructions
  - Operate on multiple elements of data with single instructions
- AVX2 256 bits = 4 double precision numbers
- AVX512 512 bits = 8 double precision numbers
  - single AVX512 fused multiply add instruction can perform 16 op's / cycle



Compiler flag examples

Feature	Gnu and Clang	Intel	Cray
Balanced optimization	`-O3`	`-O2`	`-O3`
Agressive optimization	`-Ofast -funroll-loops`	`-Ofast`	`-O3 -hfp3`
Architecture specific tuning	`-march=<target>`	`-x<target>`	`-h cpu=<target>`
Fast math	`-ffast-math`	`-fp-model fast=2`	`-h fp3`

What the compiler is doing?

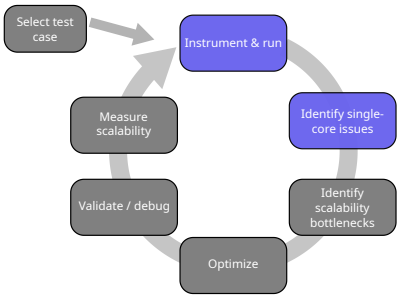
- Compilers have vast amount of heuristics for optimizing common programming patters
- Most compilers can provide a report about optimizations performed, with various amount of detail
  - See compiler manuals for all options
- Look into assembly code with `-S -fverbose-asm`

Compiler	Opt. report
GNU	`-fopt-info`
Clang	`-Rpass=.*`
Intel	`-qopt-report`
Cray	`-hlist

```
...
  vfmadd213pd %ymm0, %ymm2, %ymm10
  vfmadd213pd %ymm0, %ymm2, %ymm9
  vfmadd213pd %ymm0, %ymm2, %ymm8
...
```

Single node performance analysis

Performance analysis cycle



Measuring performance

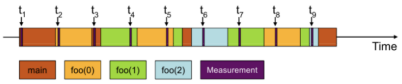
- Don't speculate about performance – measure it!
- Performance analysis tools help to
  - Find hot-spots
  - Identify the cause of less-than-ideal performance
- Tools covered here
  - Intel VTune
- Other tools
  - Perf, CrayPAT, Tau, Scalasca, gprof, PAPI, ...
  - NVIDIA Nsight, AMD ROCm Profiler, ...
  - 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?

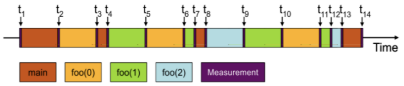
Sampling vs. Tracing

- 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



Sampling vs. Tracing

- Tracing records events, e.g., every function call
  - Usually requires modification to the executable
    - These modifications are called instrumentation
  - More accurate, but may affect program behavior
  - Often 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

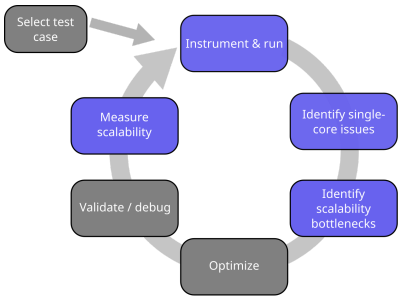
VTune

- 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 cycle



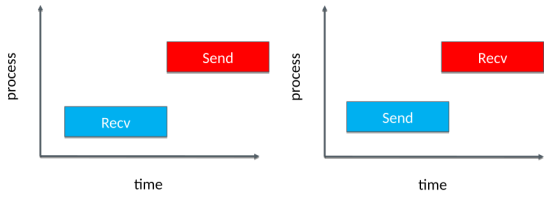
Introduction

- 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

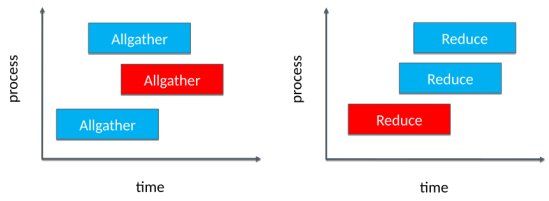
- Bad computation to communication ratio
  - In 2D heat equation with one dimensional parallelization
    $\frac{T_{comp}}{T_{comm}} \sim \frac{N}{p}$
- Load imbalance
- Synchronization overhead
- Non-optimal communication patterns

Common inefficient communication patterns



Late sender | Late receiver

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

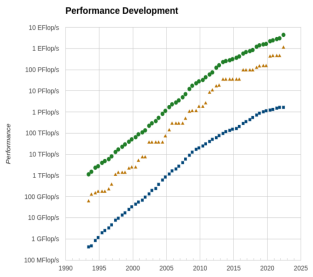
More detailed information

- 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/

GPU Programming: OpenMP offload

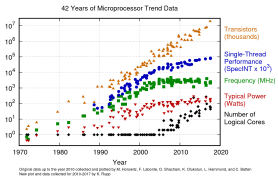
High-performance computing

- High performance computing is fueled by ever increasing performance
- Increasing performance allows breakthroughs in many major challenges that humankind faces today
- Not only hardware performance, algorithmic improvements have also added orders of magnitude of real performance



HPC through the ages

- Achieving performance has been based on various strategies throughout the years
  - Frequency, vectorization, multi-node, multi-core ...
  - Now performance is mostly limited by power consumption
- Accelerators provide compute resources based on a very high level of parallelism to reach high performance at low relative power consumption

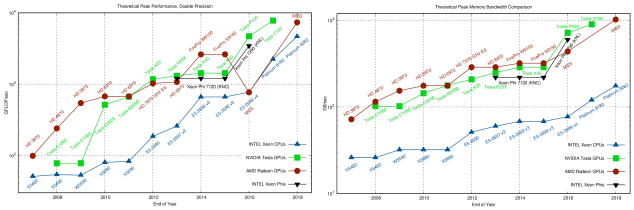


Accelerators

- Specialized parallel hardware for floating point operations
  - Co-processors for traditional CPUs
  - Based on highly parallel architectures
  - Graphics processing units (GPU) have been the most common accelerators during the last few years
- Promises
  - Very high performance per node
- Usually major rewrites of programs required

Why use them?

CPU vs Accelerator



https://github.com/karlrupp/cpu-gpu-mic-comparison

Different design philosophies

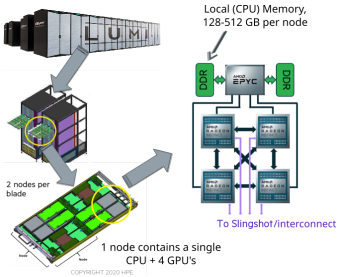
CPU

- General purpose
- Good for serial processing
- Great for task parallelism
- Low latency per thread
- Large area dedicated cache and control

GPU

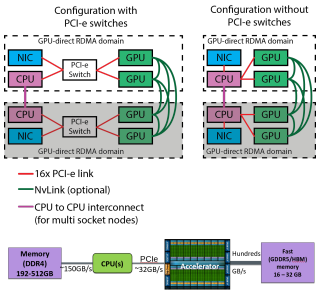
- Highly specialized for parallelism
- Good for parallel processing
- Great for data parallelism
- High-throughput
- Hundreds of floating-point execution units

Lumi - Pre-exascale system in Finland



Accelerator model today

- GPU is connected to CPUs via PCIe
- Local memory in GPU
  - Smaller than main memory (32 GB in Puhti)
  - Very high bandwidth (up to 3200 GB/s in LUMI)
  - Latency high compared to compute performance
- Data must be copied from CPU to GPU over the PCIe bus

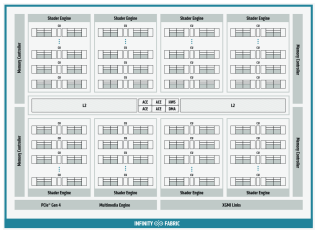


Heterogeneous Programming Model

- GPUs are co-processors to the CPU
- CPU controls the work flow:
  - *offloads* computations to GPU by launching *kernels*
  - allocates and deallocates the memory on GPUs
  - handles the data transfers between CPU and GPUs
- CPU and GPU can work concurrently
  - kernel launches are normally asynchronous

GPU architecture

- Designed for running tens of thousands of threads simultaneously on thousands of cores
- Very small penalty for switching threads
- Running large amounts of threads hides memory access penalties
- Very expensive to synchronize all threads



AMD Instinct MI100 architecture (source: AMD)

Advance features & Performance considerations

- Memory accesses:
  - data resides in the GPU memory; maximum performance is achieved when reading/writing is done in continuous blocks
  - very fast on-chip memory can be used as a user programmable cache
- *Unified Virtual Addressing* provides unified view for all memory
- Asynchronous calls can be used to overlap transfers and computations.

Challenges in using Accelerators

Applicability: Is your algorithm suitable for GPU?

Programmability: Is the programming effort acceptable?

Portability: Rapidly evolving ecosystem and incompatibilities between vendors.

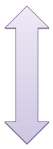
Availability: Can you access a (large scale) system with GPUs?

Scalability: Can you scale the GPU software efficiently to several nodes?

Using GPUs

1. Use existing GPU applications
2. Use accelerated libraries
3. Directive based methods
   - **OpenMP**, OpenACC
4. Use native GPU language
   - CUDA, HIP, SYCL, Kokkos,...

Easier, but more limited



More difficult, but more opportunities

Directive-based accelerator languages

- Annotating code to pinpoint accelerator-offloadable regions
- OpenACC
  - created in 2011, latest version is 3.1 (November 2020)
  - Mostly Nvidia
- OpenMP
  - Earlier only threading for CPUs
  - initial support for accelerators in 4.0 (2013), significant improvements & extensions in 4.5 (2015), 5.0 (2018), 5.1 (2020 and 5.2 (2021)
- Focus on optimizing productivity
- Reasonable performance with quite limited effort, but not guaranteed

Native GPU code: HIP / CUDA

- CUDA
  - has been the *de facto* standard for native GPU code for years
  - extensive set of optimised libraries available
  - custom syntax (extension of C++) supported only by CUDA compilers
  - support only for NVIDIA devices
- HIP
  - AMD effort to offer a common programming interface that works on both CUDA and ROCm devices
  - standard C++ syntax, uses nvcc/hcc compiler in the background
  - almost a one-on-one clone of CUDA from the user perspective
  - ecosystem is new and developing fast

GPUs @ CSC

- **Puhti-AI**: 80 nodes, total peak performance of 2.7 Petaflops
  - Four Nvidia V100 GPUs, two 20 cores Intel Xeon processors, 3.2 TB fast local storage, network connectivity of 200Gbps aggregate bandwidth
- **Mahti-AI**: 24 nodes, total peak performance of 2. Petaflops
  - Four Nvidia A100 GPUs, two 64 cores AMD Epyc processors, 3.8 TB fast local storage, network connectivity of 200Gbps aggregate bandwidth
- **LUMI-G**: 2560 nodes, total peak performance of 500 Petaflops
  - Four AMD MI250X GPUs, one 64 cores AMD Epyc processors, 2x3 TB fast local storage, network connectivity of 800Gbps aggregate bandwidth

Summary

- GPUs provide significant speed ups for certain applications
- GPUs are co-processors to CPUs
  - CPU offloads computations and manages memory
- High amount of parallelism required for efficient utilization of GPUs
- Programming GPUs
  - Directive based methods
  - CUDA, HIP

What is OpenMP offloading ?

- Set of OpenMP constructs for heterogenous systems
  - **GPU**s, FPGAs, ...
- Code regions are offloaded from a host CPU to be computed on an accelerator
  - High level GPU programming
- In principle same code can be run on various systems
  - CPUs only
  - NVIDIA GPUs, AMD GPUs, Intel GPUs, ...
- Standard defines both C/C++ and Fortran bindings

OpenMP vs. OpenACC

- OpenACC is very similar compiler directive based approach for GPU programming
  - Open standard, however, NVIDIA major driver
- Why OpenMP and not OpenACC?
  - OpenMP is going to have a more extensive platform and compiler support
  - Currently, OpenACC support in AMD GPUs is limited
  - Currently, OpenACC can provide better performance in NVIDIA GPUs

OpenACC support for AMD GPUs

- OpenACC support for AMD GPUs in GNU compilers under development
- OpenACC support in general for Clang/Flang is under development
- Cray compilers
  - Fortran compiler supports OpenACC v2.7, support for latest OpenACC coming in 2022
  - C/C++ compiler does not support OpenACC
- In LUMI, only Fortran will be supported with OpenACC
- For now, OpenACC is not a recommended approach for new codes targeting AMD GPUs
  - If a Fortran code already uses OpenACC, it may be possible to use it

OpenMP vs. CUDA/HIP

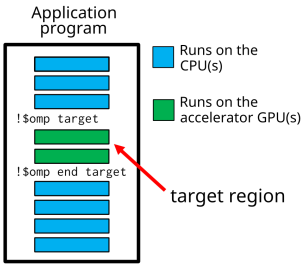
- Why OpenMP and not CUDA/HIP?
  - Easier to work with
  - Porting of existing software requires less work
  - Same code can be compiled to CPU and GPU versions easily
- Why CUDA/HIP and not OpenMP?
  - Can access all features of the GPU hardware
  - More optimization possibilities

OpenMP execution model

- Host-directed execution with an attached accelerator
  - Large part of the program is usually executed by the host
  - Computationally intensive parts are *offloaded* to the accelerator that executes *parallel regions*
- Accelerator can have a separate memory
  - OpenMP exposes the separate memories through *data environment* that defines the memory management and needed copy operations

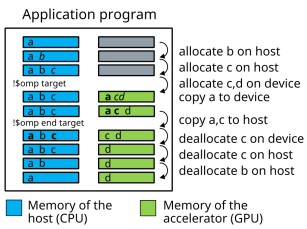
OpenMP execution model

- Program runs on the host CPU
- Host offloads compute-intensive regions (*kernels*) and related data to the accelerator GPU
- Compute kernels are executed by the GPU



OpenMP data model in offloading

- If host memory is separate from accelerator device memory
  - host manages memory of the device
  - host copies data to/from the device
- When memories are not separate, no copies are needed (difference is transparent to the user)



OpenMP directive syntax

- OpenMP uses compiler directives for defining compute regions (and data transfers) that are to be performed on a GPU
- OpenMP directives consist of a *sentinel*, followed by the directive name and optional clauses

	sentinel	directive	clauses
C/C++	`#pragma omp`	`target`	`map(data)`
Fortran	`!$omp`	`target`	`map(data)`

OpenMP directive syntax

- In C/C++, directive applies to the following structured block

```c
#pragma omp parallel
{
  // calculate in parallel
    printf("Hello world!\n");
}
```

- In Fortran, and `end` directive specifies the end of the construct

```fortran
!$omp parallel
  ! calculate in parallel
    write(*,*) "Hello world!"
!$omp end parallel
```

Compiling an OpenMP program for GPU offloading

- In addition to normal OpenMP options (*i.e.* `-fopenmp`), one needs to typically specify offload target (NVIDIA GPU, AMD GPU, ...)

Compiler	Options for offload
NVIDIA	`-mp=gpu` (`-gpu=ccNN`)
Cray	`-fopenmp-targets=xx -Xopenmp-target=xx`
Clang	`-fopenmp-targets=xx`
GCC	`-foffload=yy`

- Without these options a regular CPU version is compiled!

Compiling an OpenMP program for GPU offloading

- Conditional compilation with `_OPENMP` macro:

```c
#ifdef _OPENMP
device specific code
#else
host code
#endif
```

- Example: Compiling with NVIDIA HPC in Mahti

```
nvc -o my_exe test.c -mp=gpu -gpu=cc80
```

OpenMP internal control variables

- OpenMP has internal control variables
  - `OMP_DEFAULT_DEVICE` controls which accelerator is used.
- During runtime, values can be modified or queried with `omp_<set|get>_default_device`
- Values are always re-read before a kernel is launched and can be different for different kernels

Runtime API functions

- Low-level runtime API functions can be used to
  - Query the number of devices in the system
  - Select the device to use
  - Allocate/deallocate memory on the device(s)
  - Transfer data to/from the device(s)
- Function definitions are in
  - C/C++ header file `omp.h`
  - `omp_lib` Fortran module

Useful API functions

- `omp_is_initial_device()` : returns True when called in host, False otherwise
- `omp_get_num_devices()` : number of devices available
- `omp_get_device_num()` : number of device where the function is called
- `omp_get_default_device` : default device
- `omp_set_default_device` : set the default device

OpenMP offload constructs

Target construct

- OpenMP `target` constructs specifies a region to be executed on GPU
- Initially, runs with a single thread
- By default, execution in the host continues only after target region is finished.
- May trigger implicit data movements between the host and the device

```c
#pragma omp target
{
  // code executed in device
}
```

```fortran
!$omp target
  ! code executed in device
!$omp end target
```

Teams construct

- Target construct does not create any parallelism, but additional constructs are needed
- `teams` creates a league of teams
  - number of teams is implementation dependent
- Initially, single thread per team runs the following structured block
- No synchronization between teams is possible
- Probable mapping: team corresponds to a “thread block” / “workgroup” and runs within streaming multiprocessor / compute unit

Creating threads within a team

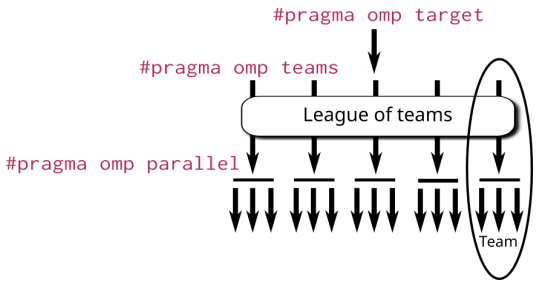
- The league of teams cannot typically leverage all the parallelism available in the accelerator
- A `parallel` construct within a `teams` region creates threads within each team
  - number of threads per team is implementation dependent
- With N teams and M threads per team there will be N x M threads in total
- Threads within a team can synchronize
- Number of teams and threads can be queried with the `omp_get_num_teams()` and `omp_get_num_threads()` API functions

Creating teams and threads

```
#pragma omp target
#pragma omp teams
#pragma omp parallel
{
  // code executed in device
}
```

```
!$omp target
!$omp teams
!$omp parallel
  ! code executed in device
!$omp end parallel
!$omp end teams
!$omp end target
```

League and teams of threads



Worksharing in the accelerator

- `teams` and `parallel` constructs create threads, however, all the threads are still executing the same code
- `distribute` constructs distributes loop iterations over the teams
- `for` / `do` construct can be used within parallel region

Worksharing in the accelerator

```
#pragma omp target
#pragma omp teams
#pragma omp distribute
for (int i = 0; i < N; i++)
  #pragma omp parallel
  #pragma omp for
  for (int j = 0; j < M; j++) {
    ...
  }
```

```
!$omp target
!$omp teams
!$omp distribute
do i = 1, N
  !$omp parallel
  !$omp do
  do j = 1, N
    ...
  end do
  !$omp end do
  !$omp end parallel
end do
!$omp end distribute
!$omp end teams
!$omp end target
```

Controlling number of teams and threads

- By default, the number of teams and the number of threads is up to the implementation to decide
- `num_teams` clause for `teams` construct and `num_threads` clause for `parallel` construct can be used to specify number of teams and threads
  - May improve performance in some cases
  - Performance is most likely not portable

```
#pragma omp target
#pragma omp teams num_teams(32)
#pragma omp parallel num_threads(128)
{
  // code executed in device
}
```

Composite directives

- In many cases composite directives are more convenient
  - possible to parallelize also single loop over both teams and threads

```
#pragma omp target teams
#pragma omp distribute parallel for
for (int i = 0; i < N; i++) {
  p[i] = v1[i] * v2[i]
}
```

```
!$omp target teams
!$omp distribute parallel do
do i = 1, N
  p(i) = v1(i) * v2(i)
end do
!$omp end distribute parallel do
!$omp end target teams
```

Loop construct

- In OpenMP 5.0 a new `loop` worksharing construct was introduced
- Less prescriptive, leaves more freedom to the implementation to do the work division
  - Tells the compiler/runtime only that the loop iterations are independent and can be executed in parallel

```
#pragma omp target
#pragma omp loop
for (int i = 0; i < N; i++) {
  p[i] = v1[i] * v2[i]
}
```

```
!$omp target
!$omp loop
do i = 1, N
  p(i) = v1(i) * v2(i)
end do
!$omp end loop
!$omp end target
```

Compiler diagnostics

Compiler diagnostics

- Compiler diagnostics is usually the first thing to check when starting the OpenMP work
  - It can tell you what operations were actually performed
  - Data copies that were made
  - If and how the loops were parallelized
- The diagnostics is very compiler dependent
  - Compiler flags
  - Level and formatting of information

NVIDIA compiler

- Diagnostics is controlled by compiler flag `-Minfo[=option]`
- Useful options:
  - `mp` – operations related to the OpenMP
  - `all` – print all compiler output
  - `intensity` – print loop computational intensity info

Example: `-Minfo`

```
nvc++ -O3 -mp=gpu -gpu=cc80 -c -Minfo=mp,intensity core.cpp
evolve:
     63, #omp target teams distribute parallel for
         63, Generating Tesla and Multicore code
             Generating "nvkernel_evolve_F1L63_1" GPU kernel
         68, Loop parallelized across teams and threads, schedule(static)
     69, Intensity = 19.00
```

Summary

- OpenMP enables directive-based programming of accelerators with C/C++ and Fortran
- Host-device model
  - host offloads computations to the accelerator
- Host and device may have separate memories
  - Host controls copying into/from the device
- Key concepts:
  - league of teams
  - threads within a team
  - worksharing between teams and threads within a team

Useful resources

- NVIDIA HPC SDK Documentation https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/
- Cray Compilers Documentation <https://dcray/hpc-compilers-user-guide/>

OpenMP data environment

- GPU device has a separate memory space from the host CPU
  - unified memory is accessible from both sides
  - OpenMP supports both unified and separate memory
- OpenMP includes constructs and clauses to ***allocate variables on the device*** and to define ***data transfers to/from the device***
- Data needs to be mapped to the device to be accessible inside the offloaded target region
  - host is not allowed to touch the mapped data during the target region
  - variables are implicitly mapped to a target region unless explicitly defined in a data clause
    - scalars as *firstprivate*, static arrays copied to/from device

Example: implicit mapping

```
int N=1000;
double a=3.14;
double x[N], y[N];
// some code to initialise x and y

#pragma omp target
#pragma omp parallel for
for (int i=0; i < N; i++) {
    y[i] += a * x[i];
}
```

- Implicit copy of **a**, **x** and **y** to the target device when the target region is opened and back when it is closed

Explicit mapping

`#pragma omp target map(type:list)`

- Explicit mapping can be defined with the `map` clause of the `target` construct
  - *list* is a comma-separated list of variables
  - *type* is one of:

`to`
: copy data to device on entry

`from`
: copy data from device on exit

`tofrom`
: copy data to device on entry and back on exit

`alloc`
: allocate on the device (uninitialised)

Example: explicit mapping

```
int N=1000;
double a=3.14;
double x[N], y[N];
// some code to initialise x and y

#pragma omp target map(to:x) map(tofrom:y)
#pragma omp parallel for
for (int i=0; i < N; i++) {
    y[i] += a * x[i];
}
```

- Both **x** and **y** are copied the device, but only **y** is copied back to the host
- Implicit copy of **a** to the device

Dynamically allocated arrays

- With dynamically allocated arrays, one needs to specify the number of elements to be copied

```
int N=1000;
double *data = (double *) malloc(N * sizeof(double));

#pragma omp target map(tofrom:data[0:N])
// do something ..
```

Motivation for optimizing data movement

- When dealing with an accelerator GPU device attached to a PCIe bus, **optimizing data movement** is often **essential** to achieving good performance
- The four key steps in porting to high performance accelerated code
  1. Identify parallelism
  2. Express parallelism
  3. Express data movement
  4. Optimise loop performance
  5. Go to 1!

Data region

- Define data mapping for a structured block that may contain multiple target regions
  - C/C++: `#pragma omp target data map(type:list)`
  - Fortran: `!omp target data map(type:list)`
  - only maps data, one still needs to define a target region to execute code on the device
- Data transfers take place
  - from **the host** to **the device** upon entry to the region
  - from **the device** to **the host** upon exit from the region

Example: data mapping over multiple target regions

```
int N=1000; double a=3.14, b=2.1;
double x[N], y[N], z[N];
// some code to initialise x, y, and z

#pragma omp target data map(to:x)
{
    #pragma omp target map(tofrom:y)
    #pragma omp parallel for
    for (int i=0; i < N; i++)
        y[i] += a * x[i];

    #pragma omp target map(tofrom:z)
    #pragma omp parallel for
    for (int i=0; i < N; i++)
        z[i] += b * x[i];
}
```

Update

- Update a variable within a data region with the `update` directive
  - C/C++: `#pragma omp target update type(list)`
  - Fortran: `!omp target update type(list)`
  - a single line executable directive
- Direction of data transfer is determined by the *type*, which can be either `to` (= copy to device) or `from` (= copy from device)

Why update?

- Useful for producing snapshots of the device variables on the host or for updating variables on the device
  - pass variables to host for visualization
  - communication with other devices on other computing nodes
- Often used in conjunction with
  - asynchronous execution of OpenMP constructs
  - unstructured data regions

Example: update within a data region

```
int N=1000; double a=3.14, b=2.1;
double x[N], y[N], z[N];
// some code to initialise x, y, and z

#pragma omp target data map(to:x)
{
    #pragma omp target map(tofrom:y)
    #pragma omp parallel for
    for (int i=0; i < N; i++)
        y[i] += a * x[i];

    // ... some host code that modifies x ...
    #pragma omp target update to(x)
    #pragma omp target map(tofrom:z)
    #pragma omp parallel for
    for (int i=0; i < N; i++)
        z[i] += b * x[i];
}
```

Reduction

`reduction(operation:list)`

- Applies the *operation* on the variables in *list* to reduce them to a single value
  - local private copies of the variables are created for each thread
  - initialisation depends on the *operation*
- Variables need to be shared in the enclosing parallel region
  - at the end, all local copies are reduced and combined with the original shared variable
- Directives that support reduction: `parallel`, `teams`, `scope`, `sections`, `for`, `simd`, `loop`, `taskloop`

Reduction operators in C/C++ and Fortran

Arithmetic Operator	Initial value
+	0
-	0
*	1
max	least
min	largest

Reduction operators in C/C++ only

Logical Operator	Initial value
&&	1
||	0

Bitwise Operator	Initial value
&	~0
|	0
^	0

Reduction operators in Fortran only

Logical Operator	Initial value
.and.	.true.
.or.	.false.
.eqv.	.true.
.neqv.	.false.

Bitwise Operator	Initial value
iand	all bits on
ior	0
ieor	0

Example: reduction

```
int N=1000;
double sum=0.0;
double x[N], y[N];
// some code to initialise x and y

#pragma omp target
#pragma omp parallel for reduction(+:sum)
for (int i=0; i < N; i++) {
    sum += y[i] * x[i];
}
```

Unstructured data regions

Unstructured data regions

- Unstructured data regions enable one to handle cases where allocation and freeing is done in a different scope
- Useful for e.g. C++ classes, Fortran modules
- `enter data` defines the start of an unstructured data region
  - C/C++: `#pragma omp enter data [clauses]`
  - Fortran: `!$omp enter data [clauses]`
- `exit data` defines the end of an unstructured data region
  - C/C++: `#pragma omp exit data [clauses]`
  - Fortran: `!$omp exit data [clauses]`

Unstructured data

```
class Vector {
    Vector(int n) : len(n) {
        v = new double[len];
        #pragma omp enter data alloc(v[0:len])
    }
    ~Vector() {
        #pragma omp exit data delete(v[0:len])
        delete[] v;
    }
    double v;
    int len;
};
```

Enter data clauses

`map(alloc:var-list)`
Allocate memory on the device
`map(to:var-list)`
Allocate memory on the device and copy data from the host to the device

Exit data clauses

`map(delete:var-list)`
Deallocate memory on the device
`map(from:var-list)`
Deallocate memory on the device and copy data from the device to the host

Declare target

- Makes a variable resident in accelerator memory
- Added at the declaration of a variable
- Data life-time on device is the implicit life-time of the variable
  - C/C++: `#pragma omp declare target [clauses]`
  - Fortran: `!$omp declare target [clauses]`

Porting and unified memory

- Porting a code with complicated data structures can be challenging because every field in type has to be copied explicitly
- Recent GPUs have *Unified Memory* and support for automatic data transfers with page faults

```
typedef struct points {
  double *x, *y;
  int n;
}

void process_points() {
  points p;

  #pragma omp target data map(alloc:p)
  {
    p.size = n;
    p.x = (double) malloc(...
    p.y = (double) malloc(...
    #pragma omp target update map(to:p)
    #pragma omp target update map(to:p.x[
    ...
```

Unified memory

- OpenMP 5.0 added a `requires` construct so that program can declare it assumes shared memory between host and device
- Compiler support in progress

```
typedef struct points {
  double *x, *y;
  int n;
}

void process_points() {
  points p;

  #pragma omp requires unified_shared_mem
  p.size = n;
  p.x = (double) malloc(...
  p.y = (double) malloc(...
  ...
```

Summary

- Implicit/explicit mapping
  - mapping types: `to`, `from`, `tofrom`, `alloc`, `delete`
- Structured data region
- Unstructured data region
  - `enter data` and `exit data`
- Data directives: update, reduction, declare target

Device functions

Function calls in compute regions

- Often it can be useful to call functions within loops to improve readability and modularisation
- By default OpenMP does not create accelerated regions for loops calling functions
- One has to instruct the compiler to compile a device version of the function

Directive: `declare target`

- Define a function to be compiled for an accelerator as well as the host
- In C/C++ one puts `declare target` and `end declare target` around function declarations
- In Fortran one uses `!$declare target (name)` form
- The functions will now be compiled both for host and device execution

Example: `declare target`

C/C++

```
#pragma omp declare target
void foo(float* v, int i, int n) {
    for ( int j=0; j<n; ++j) {
        v[i*n+j] = 1.0f/(i*j);
    }
}
#pragma omp end declare target

#pragma omp target teams loop
for (int i=0; i<n; ++i) {
    foo(v,i);  // executed on the device
}
```

Fortran

```
subroutine foo(v, i, n)
  !$omp declare target
  real :: v(:,:)
  integer :: i, n

  do j=1,n
     v(i,j,n) = 1.0/(i*j)
  enddo
end subroutine

!$omp target teams loop
do i=1,n
  call foo(v,i,n)
enddo
!$omp end target teams loop
```

Summary

- Declare target directive
  - Enables one to write device functions that can be called within parallel loops

Interoperability with libraries

Interoperability with libraries

- Often it may be useful to integrate the accelerated OpenMP code with other accelerated libraries
- MPI: MPI libraries are GPU-aware
- HIP/CUDA: It is possible to mix OpenMP and HIP/CUDA
  - Use OpenMP for memory management
  - Introduce OpenMP in existing GPU code
  - Use HIP/CUDA for tightest kernels, otherwise OpenMP
- Numerical GPU libraries: CUBLAS, CUFFT, MAGMA, CULA...
- Thrust, etc.

Device data interoperability

- OpenMP includes methods to access the device data pointers in the host side
- Device data pointers can be used to interoperate with libraries and other programming techniques available for accelerator devices
  - HIP/CUDA kernels and libraries
  - GPU-aware MPI libraries
- Some features are still under active development, many things may not yet work as they are supposed to!

Data constructs: `use_device_ptr` and `use_device_addr`

`omp target data use_device_ptr(var-list)`

- Define a device pointer to be available on the host
- Within the construct, all the pointer variables in `var-list` contain the device address

`omp target data use_device_addr(var-list)`

- Within the construct, all the variables in `var-list` have the address of the corresponding object in the device
- Can be used with non-pointer variables

use_device_ptr: example with cublas

```
cublasInit();
double *x, *y;
//Allocate x and y, and initialise x
#pragma omp target data map(to:x[:n]), map(from:y[:n]))
{
    #pragma omp target data use_device_ptr(x, y) {
        cublasDaxpy(n, a, x, 1, y, 1);
    }
}
```

Calling CUDA-kernel from OpenMP-program

- In this scenario we have a (main) program written in C/C++ (or Fortran) and this driver uses OpenMP directives
  - CUDA-kernels must be called with help of OpenMP `use_device_ptr`
- Interface function in CUDA-file must have `extern "C" void func(...)`
- The CUDA-codes are compiled with NVIDIA `nvcc` compiler e.g.:
  - `nvcc -c -O3 --restrict daxpy_cuda.cu`
- The OpenMP-codes are compiled with NVIDIA `nvc` or `nvc++` compiler e.g.:
  - `nvc -c -mp=gpu -O3 call_cuda_from_openmp.c`
- For linking, `-lcudart -L$CUDA_HOME/lib64` is needed

Calling CUDA/HIP-kernel from C OpenMP-program

```
// call_cuda_from_openmp.c

extern void daxpy(int n, double a,
                  const double *x, double *y);

int main(int argc, char *argv[])
{
    int n = (argc > 1) ? atoi(argv[1]) : (1 << 27);
    const double a = 2.0;
    double *x = malloc(n * sizeof(*x));
    double *y = malloc(n * sizeof(*y));

    #pragma omp target data map(alloc:x[0:n], y[0:n])
    {
        // Initialize x & y
        ...
        // Call CUDA-kernel
        #pragma omp target data use_device_ptr(x, y)
        daxpy(n, a, x, y);
        ...
    } // #pragma omp target data
```

```
// daxpy_cuda.cu

__global__
void daxpy_kernel(int n, double a,
                  const double *x, double *y)
{ // The actual CUDA-kernel
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    while (tid < n) {
        y[tid] += a * x[tid];
        tid += stride;
    }
}
extern "C" void daxpy(int n, double a,
                      const double *x, double *y)
{ // This can be called from C/C++ or Fortran
    dim3 blockdim = dim3(256, 1, 1);
    dim3 griddim = dim3(65536, 1, 1);
    daxpy_kernel<<<griddim, blockdim>>>(n, a, x, y);
}
```

Calling CUDA/HIP-kernel from Fortran OpenMP-program

```
// call_cuda/hip_from_openmp.f90
MODULE CUDA_INTERFACES
    INTERFACE
      subroutine f_daxpy(n, a, x, y) bind(C,name=daxpy)
      use iso_c_binding
      integer(c_int), value :: n
      double(c_double), value :: a
      real :: x(*), y(*)
    END INTERFACE
END MODULE CUDA_INTERFACES

...

// in the main programs
 use iso_c_binding
 ...
 integer(c_int) :: n
 double(c_double) :: a
 ...

# omp target data use_device_ptr(x, y)

  call f_daxpy(n,a,x,y)
```

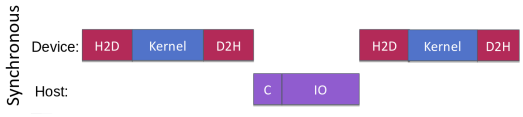
```
// call_cuda/hip_from_openmp.f90
MODULE CUDA_INTERFACES
    INTERFACE
      subroutine f_daxpy(n, a, x, y) bind(C,name=daxpy)
      use iso_c_binding
      integer(c_int), value :: n
      double(c_double), value :: a
      type(c_ptr), value :: x, y
    END INTERFACE
END MODULE CUDA_INTERFACES

...

// in the main programs
 use iso_c_binding
 ...
 integer(c_int) :: n
 double(c_double) :: a
 ...

# omp target data use_device_ptr(x, y)

  call f_daxpy(n,a,c_loc(x),c_loc(y))
```

Summary

- OpenMP programs can work in conjuction with GPU libraries or with own computational kernels written with lower level languages (e.g. CUDA/HIP).
- The pointer / reference to the data in device can be obtained with the `use_device_ptr` / `use_device_addr` clauses.

Motivation

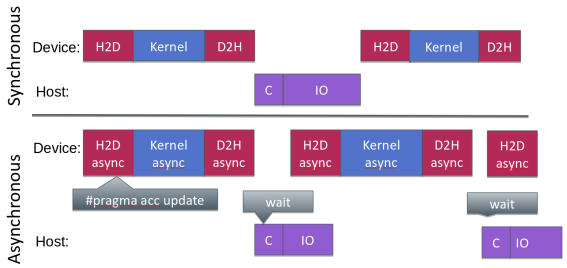
- By default, the host thread will wait until OpenMP `target` compute or data construct has completed its execution, *i.e.* there is an implicit barrier at the end of the `target`
- Potential parallelism in overlapping compute, data transfers, MPI, etc.



Asynchronous execution: nowait and taskwait

- `target` construct creates an implicit OpenMP task
- Similar to OpenMP explicit task constructs (`task`, `taskloop`), `target` has a `nowait` clause
  - removes the implicit barrier
- Completion can be ensured with the `taskwait` construct
- Work on host (concurrent to accelerator) can be parallelized with OpenMP tasks

OpenMP and asynchronous execution



OpenMP and asynchronous execution

```
#pragma omp target nowait
process_in_device();

process_in_host();
#pragma omp taskwait
```

```
!$omp target nowait
call process_in_device();
!$omp end target

process_in_host();
!$omp taskwait
```

Task dependencies

- OpenMP tasks support data flow model, where a task can have input and output dependencies

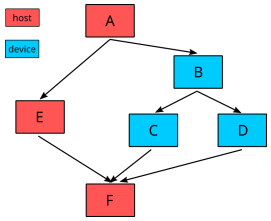
```
// The two tasks can be executed concurre
#pragma omp task
{do something}

#pragma omp task
{do something else}
```

```
// The two tasks can be executed concurre
#pragma omp task depend(out:a)
{do something which produces a}

#pragma omp task depend(in:a)
{do something which uses a as input}
```

- Also `target` tasks support dependencies

Task dependencies

```
#pragma omp task depend(out: A)
 {Code A}
#pragma omp target depend(in: A) depend(out: B) \
 nowait
 {Code B}
#pragma omp target depend(in: B) depend(out: C) \
 nowait
 {Code C}
#pragma omp target depend(in: B) depend(out: D) \
 nowait
 {Code D}
#pragma omp task depend(in: A) depend(in: A)
  {Code E}
#pragma omp task depend(in: C,D,E)
  {Code F}
```



Task dependencies

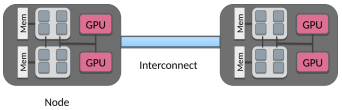
- Dependencies may be specifief also for a part of an array

```
// Preocessing array in blocks
for (int ib = 0; ib < n; ib += bf) {
  #pragma omp ... depend(out:A[ib*bf]) nowait
  {Processing step 1}
  #pragma omp ... depend(in:A[ib*bf]) nowait
  {Processing step 2}
}
```

Summary

- `target` construct creates an implicit OpenMP task
- OpenMP task functionalities (`nowait`, `taskwait`, `depend`) can be used for asynchronous execution on accelerators
- May enable better performance by overlapping different operations
  - Performance depends heavily on the underlying implementation

Multi-GPU programming with OpenMP



- Three levels of hardware parallelism in a supercomputer:
  1. GPU - different levels of threads
  2. Node - multiple GPUs and CPUs
  3. System - multiple nodes connected with interconnect
- Three parallelization methods:
  1. OpenMP offload
  2. OpenMP or MPI
  3. MPI between nodes

Multi-GPU communication cases

- Single node multi-GPU programming
  - All GPUs of a node are accessible from a single process and its OpenMP threads
  - Data copies either directly or through CPU memory
- Multi-node multi-GPU programming
  - Communication between nodes requires message passing (MPI)
- In this lecture we will discuss in detail only parallelization with MPI
  - It enables direct scalability from single to multi-node

Multiple GPUs

- OpenMP permits using multiple GPUs within one node by using the `omp_get_num_devices` and `omp_set_default_device` functions
- Asynchronous OpenMP calls, OpenMP threads or MPI processes must be used in order to actually run kernels in parallel
- Issue when using MPI:
  - If a node has more than one GPU, all processes in the node can access all GPUs of the node
  - MPI processes do not have any a priori information about the other ranks in the same node
  - Which GPU the MPI process should select?

Selecting a device with MPI

- Simplest model is to use **one** MPI task per GPU
- Launching job
  - Launch you application so that there are as many MPI tasks per node as there are GPUs
  - Make sure the affinity is correct - processes equally split between the two sockets (that nodes typically have)
  - Read the user guide of the system for details how to do this!
- In the code a portable and robust solution is to use MPI shared memory communicators to split the GPUs between processes
- Note that you can also use OpenMP to utilize all cores in the node for computations on CPU side

Selecting a device with MPI

```
MPI_Comm shared;
int local_rank, local_size, num_gpus;

MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0,
                    MPI_INFO_NULL, &shared);
MPI_Comm_size(shared, &local_size); // number of ranks in this node
MPI_Comm_rank(shared, &local_rank); // my local rank
num_gpus = omp_get_num_device(); // num of gpus in node
if (num_gpus == local_size) {
    omp_set_default_device(local_rank);
} // otherwise error
```

Sharing a device between MPI tasks

- In some systems (e.g. with NVIDIA Multi-Process Service) multiple MPI tasks (within a node) may share a device efficiently
- Can provide improved performance if single MPI task cannot fully utilize the device
- Oversubscribing can lead also to performance degregation, performance should always be tested
- Still, only MPI tasks within a node may share devices
- Load balance may be a problem if number of MPI tasks per device is not the same for all the devices

Assigning multiple MPI tasks per device

```
MPI_Comm shared;
int local_rank, local_size, num_gpus, my_device;

MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0,
                    MPI_INFO_NULL, &shared);
MPI_Comm_size(shared, &local_size); // number of ranks in this node
MPI_Comm_rank(shared, &local_rank); // my local rank
num_gpus = omp_get_num_device(); // num of gpus in node
my_device = local_rank % num_gpus; // round robin assignment with modulo
omp_set_default_device(my_device);
```

Data transfers

- Idea: use MPI to transfer data between GPUs, use OpenMP offloading for computations
- Additional complexity: GPU memory is separate from CPU memory
- GPU-aware MPI-library
  - Can use the device pointer in MPI calls - no need for additional buffers
  - No need for extra buffers and device-host-device copies
  - If enabled on system, data will be transferred via transparent RDMA
- Without GPU-aware MPI-library
  - Data must be transferred from the device memory to the host memory and vice versa before performing MPI-calls

Using device addresses with MPI

- For accessing device addresses of data on the host one can use `target data` with the `use_device_ptr` clause
- No additional data transfers needed between the host and the device, data automatically accessed from the device memory via **R**emote **D**irect **M**emory **A**ccess
- Requires *library* and *device* support to function!

MPI communication with GPU-aware MPI

- MPI send
  - Send the data from the buffer on the **device** with MPI
- MPI receive
  - Receive the data to a buffer on the **device** with MPI
- No additional buffers or data transfers needed to perform communication

MPI communication with GPU-aware MPI

```
/* MPI_Send with GPU-aware MPI */
#pragma omp target data use_device_ptr(data)
{
    MPI_Send(data, N, MPI_DOUBLE, to, MPI_ANY_TAG, MPI_COMM_WORLD);
}

/* MPI_Recv with GPU-aware MPI */
#pragma omp target data use_device_ptr(data)
{
    MPI_Recv(data, N, MPI_DOUBLE, from, MPI_ANY_TAG, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}
```

Summary

- Typical HPC cluster node has several GPUs in each node
  - Selecting the GPUs with correct affinity
- Data transfers using MPI
  - GPU-aware MPI avoids extra memory copies

GPU Programming: HIP

HIP

- Heterogeneous-computing Interface for Portability
  - AMD effort to offer a common programming interface that works on both CUDA and ROCm devices
- HIP is a C++ runtime API and kernel programming language
  - standard C++ syntax, uses nvcc/hcc compiler in the background
  - almost a one-on-one clone of CUDA from the user perspective
  - allows one to write portable GPU codes
- AMD offers also a wide set of optimised libraries and tools

HIP kernel language

- Qualifiers: `__device__`, `__global__`, `__shared__`, ...
- Built-in variables: `threadIdx.x`, `blockIdx.y`, ...
- Vector types: `int3`, `float2`, `dim3`, ...
- Math functions: `sqrt`, `powf`, `sinh`, ...
- Intrinsic functions: synchronisation, memory-fences etc.

HIP API

- Device init and management
- Memory management
- Execution control
- Synchronisation: device, stream, events
- Error handling, context handling, ...

HIP programming model

- GPU accelerator is often called a *device* and CPU a *host*
- Parallel code (kernel) is launched by the host and executed on a device by several threads
- Code is written from the point of view of a single thread
  - each thread has a unique ID

Example: Hello world

```
#include <hip/hip_runtime.h>
#include <stdio.h>

int main(void)
{
    int count, device;

    hipGetDeviceCount(&count);
    hipGetDevice(&device);

    printf("Hello! I'm GPU %d out of %d GPUs in total.\n", device, count);

    return 0;
}
```

AMD GPU terminology

- Compute Unit
  - one of the parallel vector processors in a GPU
- Kernel
  - function launched to the GPU that is executed by multiple parallel workers

AMD GPU terminology (continued)

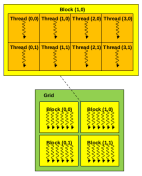
- Thread
  - individual lane in a wavefront
- Wavefront (cf. CUDA warp)
  - collection of threads that execute in lockstep and execute the same instructions
  - each wavefront has 64 threads
  - number of wavefronts per workgroup is chosen at kernel launch (up to 16)
- Workgroup (cf. CUDA thread block)
  - group of wavefronts (threads) that are on the GPU at the same time and are part of the same compute unit (CU)
  - can synchronise together and communicate through memory in the CU

GPU programming considerations

- GPU model requires many small tasks executing a kernel
  - e.g. can replace iterations of loop with a GPU kernel call
- Need to adapt CPU code to run on the GPU
  - rethink algorithm to fit better into the execution model
  - keep reusing data on the GPU to reach high occupancy of the hardware
  - if necessary, manage data transfers between CPU and GPU memories carefully (can easily become a bottleneck!)

Grid: thread hierarchy

- Kernels are executed on a 3D *grid* of threads
  - threads are partitioned into equal-sized *blocks*
- Code is executed by the threads, the grid is just a way to organise the work
- Dimension of the grid are set at kernel launch



- Built-in variables to be used within a kernel:
  - `threadIdx`, `blockIDx`, `blockDim`, `gridDim`

Kernels

- Kernel is a (device) function to be executed by the GPU
- Function should be of `void` type and needs to be declared with the `__global__` or `__device__` attribute
- All pointers passed to the kernel need to point to memory accessible from the device
- Unique thread and block IDs can be used to distribute work

Example: axpy

```
__global__ void axpy_(int n, double a, double *x, double *y)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if (tid < n) {
        y[tid] += a * x[tid];
    }
}
```

- Global ID `tid` calculated based on the thread and block IDs
  - only threads with `tid` smaller than `n` calculate
  - works only if number of threads ≥ `n`

Example: axpy (revisited)

```
__global__ void axpy_(int n, double a, double *x, double *y)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = gridDim.x * blockDim.x;

    for (; tid < n; tid += stride) {
        y[tid] += a * x[tid];
    }
}
```

- Handles any vector size, but grid dimensions should be still “optimised”

Launching kernels

- Kernels are launched with the function call `hipLaunchKernelGGL`
  - grid dimensions need to be defined (two vectors of type `dim3`)
  - execution is asynchronous

```
dim3 blocks(32);
dim3 threads(256);

hipLaunchKernelGGL(somekernel, blocks, threads, 0, 0, ...)
```

- Compared with the CUDA syntax:

```
somekernel<<<blocks, threads, 0, 0>>>(...)
```

Simple memory management

- In order to calculate something on the GPUs, we usually need to allocate device memory and pass a pointer to it when launching a kernel
- Similarly to `cudaMalloc` (or simple `malloc`), HIP provides a function to allocate device memory: `hipMalloc()`

```
double *x_
hipMalloc(&x_, sizeof(double) * n);
```

- To copy data to/from device, one can use `hipMemcpy()`:

```
hipMemcpy(x_, x, sizeof(double) * n, hipMemcpyHostToDevice);
hipMemcpy(x, x_, sizeof(double) * n, hipMemcpyDeviceToHost);
```

Error handling

- Most HIP API functions return error codes
- Good idea to **always** check for success (`hipSuccess`), e.g. with a macro such as:

```
#define HIP_SAFECALL(x) {       \
  hipError_t status = x;        \
  if (status != hipSuccess) {   \
    printf("HIP Error: %s\n", hipGetErrorString(status));  \
  } }
```

Example: fill (complete device code and launch)

```
#include <hip/hip_runtime.h>
#include <stdio.h>

__global__ void fill_(int n, double *x, double a)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = gridDim.x * blockDim.x;

    for (; tid < n; tid += stride) {
        x[tid] = a;
    }
}
```

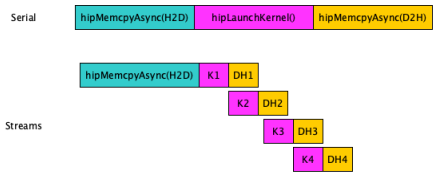
```
int main(void)
{
    const int n = 10000;
    double a = 3.4;
    double x[n];
    double *x_;

    // allocate device memory
    hipMalloc(&x_, sizeof(double) * n);

    // launch kernel
    dim3 blocks(32);
    dim3 threads(256);
    hipLaunchKernelGGL(fill_, blocks, threads, 0, 0, n, x_

    // copy data to the host and print
    hipMemcpy(x, x_, sizeof(double) * n, hipMemcpyDeviceTo
    printf("%f %f %f ... %f %f\n",
           x[0], x[1], x[2], x[3], x[n-2], x[n-1]);

    return 0;
}
```

Summary

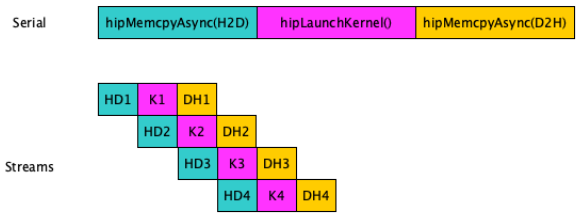
- HIP supports both AMD and NVIDIA GPUs
- HIP contains both API functions and declarations etc. needed to write GPU kernels
- Kernels are launched by multiple threads in a grid
  - in wavefronts of 64 threads
- Kernels need to be declared `void` and `__global__` and are launched with `hipLaunchKernelGGL()`

Streams

- What is a stream?
  - a sequence of operations that execute in order on the GPU
  - operations in different streams may run concurrently



Amount of concurrency

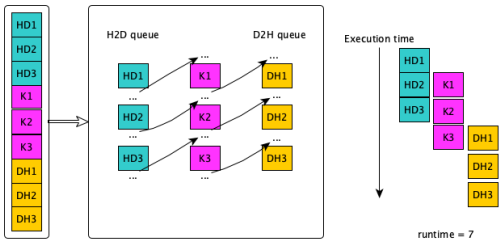


Default

- Only a single stream is used if not defined
- Operations are synchronized unless async versions are used

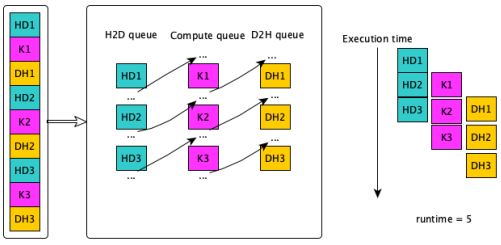
Example: issue of order (I)

- We have 3 streams and we do 3 operations (HD, K, DH)



Example: issue of order (II)

- Need to think the dependencies and how to improve the runtime



Synchronization and memory

`hipStreamSynchronize`
host waits for all commands in the specified stream to complete

`hipDeviceSynchronize`
host waits for all commands in all streams on the specified device to complete

`hipEventSynchronize`
host waits for the specified event to complete

`hipStreamWaitEvent`
stream waits for the specified event to complete

Stream/Events API

`hipStreamCreate`
creates an asynchronous stream

`hipStreamDestroy`
destroy an asynchronous stream

`hipStreamCreateWithFlags`
creates an asynchronous stream with specified flags

`hipEventCreate`
create an event

`hipEventDestroy`
destroy an event

`hipEventRecord`
record an event in a specified stream

`hipEventSynchronize`
wait for an event to complete

`hipEventElapsedTime`
return the elapsed time between two events

Implicit Synchronization

- hipHostMalloc
- hipMalloc
- hipMemcpy ...

Example: data transfer and compute

- Serial

```
hipEventRecord(startEvent,0);

hipMemcpy(d_a, a, bytes, hipMemcpyHostToDevice);
hipLaunchKernelGGL(kernel, n/blockSize, blockSize, 0, 0, d_a, 0);
hipMemcpy(a, d_a, bytes, hipMemcpyDeviceToHost);

hipEventRecord(stopEvent, 0);
hipEventSynchronize(stopEvent);

hipEventElapsedTime(&duration, startEvent, stopEvent);
printf("Duration of sequential transfer and execute (ms): %f\n", duration);
```

How to improve the performance?

- Use streams to overlap computation with communication

```
hipStream_t stream[nStreams];
for (int i = 0; i < nStreams; ++i)
    hipStreamCreate(&stream[i]);
```

- Use Asynchronous data transfer

```
hipMemcpyAsync(dst, src, bytes, hipMemcpyHostToDevice, stream);
```

- Execute kernels on different streams

```
hipLaunchKernelGGL(kernel, gridsize, blocksize, shared_mem_size, stream,
                   arg0, arg1, ...);
```

Synchronization (I)

- Synchronize everything

```
hipDeviceSynchronize()
```

- Synchronize a specific stream

```
hipStreamSynchronize(streamid)
```

  - Blocks host until all HIP calls are completed on this stream

Synchronization (II)

- Synchronize using Events
  - Create an event

```
hipEvent_t stopEvent
hipEventCreate(&stopEvent)
```

  - Record an event in a specific stream and wait until ready

```
hipEventRecord(stopEvent,0)
hipEventSynchronize(stopEvent)
```

  - Make a stream wait for a specific event

```
hipStreamWaitEvent(stream[i], stopEvent, unsigned int flags)
```

Synchronization in the kernel

`__syncthreads()`

synchronize threads within a block inside a kernel

```
__global__ void reverse(double *d_a) {
    __shared__ double s_a[256]; /* array of doubles, shared in this block */
    int tid = threadIdx.x;
    s_a[tid] = d_a[tid];     /* each thread fills one entry */
    __syncthreads();         /* all wavefronts must reach this point before
                                any wavefront is allowed to continue. */
    d_a[tid] = s_a[255-tid]; /* safe to write out array in reverse order */
}
```

Outline

- Memory model and hierarchy
- Memory management strategies
- Page-locked memory
- Coalesced memory access in device kernels

Memory model

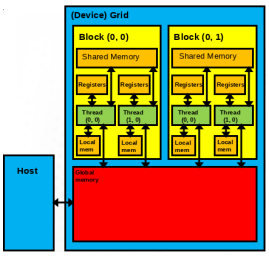
- Host and device have separate physical memories
- It is generally not possible to call malloc() to allocate memory and access the data from the GPU
- Memory management can be
  - Explicit (user manages the movement of the data and makes sure CPU and GPU pointers are not mixed)
  - Automatic, using Unified Memory (data movement is managed in the background by the Unified Memory driver)

Avoid moving data between CPU and GPU

- Data copies between host and device are relatively slow
- To achieve best performance, the host-device data traffic should be minimized regardless of the chosen memory management strategy
  - Initializing arrays on the GPU
  - Rather than just solving a linear equation on a GPU, also setting it up on the device
- Not copying data back and forth between CPU and GPU every step or iteration can have a large performance impact!

Device memory hierarchy

- Registers (per-thread-access)
- Local memory (per-thread-access)
- Shared memory (per-block-access)
- Global memory (global access)



Device memory hierarchy

- Registers (per-thread-access)
  - Used automatically
  - Size on the order of kilobytes
  - Very fast access
- Local memory (per-thread-access)
  - Used automatically if all registers are reserved
  - Local memory resides in global memory
  - Very slow access
- Shared memory (per-block-access)
  - Usage must be explicitly programmed
  - Size on the order of kilobytes
  - Fast access
- Global memory (per-device-access)
  - Managed by the host through HIP API
  - Size on the order of gigabytes
  - Very slow access

Device memory hierarchy (advanced)

- There are more details in the memory hierarchy, some of which are architecture-dependent, eg,
  - Texture memory
  - Constant memory
- Complicates implementation
- Should be considered only when a very high level of optimization is desirable

Important memory operations

```
// Allocate pinned device memory
hipError_t hipMalloc(void **devPtr, size_t size)

// Allocate Unified Memory; The data is moved automatically between host/device
hipError_t hipMallocManaged(void **devPtr, size_t size)

// Deallocate pinned device memory and Unified Memory
hipError_t hipFree(void *devPtr)

// Copy data (host-host, host-device, device-host, device-device)
hipError_t hipMemcpy(void *dst, const void *src, size_t count, enum hipMemcpyKind ki
```

Example of explicit memory management

```
int main() {
 int *A, *d_A;
 A = (int *) malloc(N*sizeof(int));
 hipMalloc((void**)&d_A, N*sizeof(int));
 ...
 /* Copy data to GPU and launch kernel */
 hipMemcpy(d_A, A, N*sizeof(int), hipMemcpyHostToDevice);
 hipLaunchKernelGGL(...);
 ...
 hipMemcpy(A, d_A, N*sizeof(int), hipMemcpyDeviceToHost);
 hipFree(d_A);
 printf("A[0]: %d\n", A[0]);
 free(A);
 return 0;
}
```

Example of Unified Memory

```
int main() {
 int *A;
 hipMallocManaged((void**)&A, N*sizeof(int));
 ...
 /* Launch GPU kernel */
 hipLaunchKernelGGL(...);
 hipStreamSynchronize(0);
 ...
 printf("A[0]: %d\n", A[0]);
 hipFree(A);
 return 0;
}
```

Unified Memory pros

- Allows incremental development
- Can increase developer productivity significantly
  - Especially large codebases with complex data structures
- Supported by the latest NVIDIA + AMD architectures
- Allows oversubscribing GPU memory on some architectures

Unified Memory cons

- Data transfers between host and device are initially slower, but can be optimized once the code works
  - Through prefetches
  - Through hints
- Must still obey concurrency & coherency rules, not foolproof
- The performance on the AMD cards is an open question

Unified Memory workflow for GPU offloading

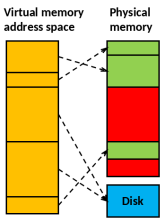
1. Allocate memory for the arrays accessed by the GPU with `hipMallocManaged()` instead of `malloc()`
   - It is a good idea to have a wrapper function and use conditional compilation for memory allocations
2. Offload compute kernels to GPUs
3. Check profiler backtrace for GPU->CPU Unified Memory page-faults (NVIDIA Visual Profiler, Nsight Systems, AMD profiler?)
   - This indicates where the data residing on the GPU is accessed by the CPU (very useful for large codebases, especially if the developer is new to the code)

Unified Memory workflow for GPU offloading

4. Move operations from CPU to GPU if possible, or use hints / prefetching (`hipMemAdvice()` / `hipMemPrefetchAsync()`)
   - It is not necessary to eliminate all page faults, but eliminating the most frequently occurring ones can provide significant performance improvements
5. Allocating GPU memory can have a much higher overhead than allocating standard host memory
   - If GPU memory is allocated and deallocated in a loop, consider using a GPU memory pool allocator for better performance

Virtual Memory addressing

- Modern operating systems utilize virtual memory
  - Memory is organized to memory pages
  - Memory pages can reside on swap area on the disk (or on the GPU with Unified Memory)



Page-locked (or pinned) memory

- Normal `malloc()` allows swapping and page faults
- User can page-lock an allocated memory block to a particular physical memory location
- Enables Direct Memory Access (DMA)
- Higher transfer speeds between host and device
- Copying can be interleaved with kernel execution
- Page-locking too much memory can degrade system performance due to paging problems

Allocating page-locked memory on host

- Allocated with `hipHostMalloc()` function instead of `malloc()`
- The allocation can be mapped to the device address space for device access (slow)
  - On some architectures, the host pointer to device-mapped allocation can be directly used in device code (ie. it works similarly to Unified Memory pointer, but the access from the device is slow)
- Deallocated using `hipHostFree()`

Asynchronous memcopies

- Normal `hipMemcpy()` calls are blocking (i.e. synchronizing)
  - The execution of host code is blocked until copying is finished
- To overlap copying and program execution, asynchronous functions are required
  - Such functions have Async suffix, eg. `hipMemcpyAsync()`
- User has to synchronize the program execution
- Requires page-locked memory

Global memory access in device code

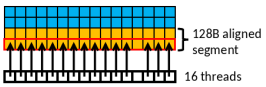
- Global memory access from the device is slow
- Threads are executed in warps, memory operations are grouped in a similar fashion
- Memory access is optimized for coalesced access where threads read from and write to successive memory locations
- Exact alignment rules and performance issues depend on the architecture

Coalesced memory access

- The global memory loads and stores consist of transactions of a certain size (eg. 32 bytes)
- If the threads within a warp access data within such a block of 32 bytes, only one global memory transaction is needed
- Now, 32 threads within a warp can each read a different 4-byte integer value with just 4 transactions
- When the stride between each 4-byte integer is increased, more transactions are required (up to 32 for the worst case)!

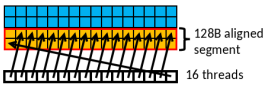
Coalesced memory access example

```
__global__ void memAccess(float *out, flo
{
 int tid = blockIdx.x*blockDim.x + thread
 if(tid != 12) out[tid + 16] = in[tid + 1
}
```



Coalesced access

```
__global__ void memAccess(float *out, flo
{
 int tid = blockIdx.x*blockDim.x + thread
 out[tid + 1] = in[tid + 1];
}
```



Unaligned sequential access (+1 tx)

Summary

- Host and device have separate physical memories
- Using Unified Memory can improve developer productivity and result in a cleaner implementation
- The number of data copies between CPU and GPU should be minimized
  - With Unified Memory, if data transfer cannot be avoided, using hints or prefetching to mitigate page faults is beneficial
- Coalesced memory access in the device code is recommended for better performance

Fortran

- First Scenario: Fortran + CUDA C/C++
  - Assuming there is no CUDA code in the Fortran files.
  - Hipify CUDA
  - Compile and link with hipcc
- Second Scenario: CUDA Fortran
  - There is no HIP equivalent
  - HIP functions are callable from C, using `extern C`
  - See hipfort

Hipfort

The approach to port Fortran codes on AMD GPUs is different, the hipify tool does not support it.

- We need to use hipfort, a Fortran interface library for GPU kernel
- Steps:
  1. We write the kernels in a new C++ file
  2. Wrap the kernel launch in a C function
  3. Use Fortran 2003 C binding to call the C function
  4. Things could change
- Use OpenMP offload to GPUs

Fortran SAXPY example

- CUDA Fortran, 29 lines of code
- Ported to HIP manually
  - two files of 52 lines, with more than 20 new lines
  - quite a lot of changes for such a small code!
- Should we try to use OpenMP offload instead?
- Code available at: https://github.com/csc-training/hip-programming/hipfort

Original CUDA Fortran

```
module mathOps
contains
  attributes(global) subroutine saxpy(x, y, a)
    implicit none
    real :: x(:), y(:)
    real, value :: a
    integer :: i, n
    n = size(x)
    i = blockDim%x * (blockIdx%x - 1) + threadIdx%x
    if (i <= n) y(i) = y(i) + a*x(i)
  end subroutine saxpy
end module mathOps
```

```
program testSaxpy
  use mathOps
  use cudafor
  implicit none
  integer, parameter :: N = 40000
  real :: x(N), y(N), a
  real, device :: x_d(N), y_d(N)
  type(dim3) :: grid, tBlock

  tBlock = dim3(256,1,1)
  grid = dim3(ceiling(real(N)/tBlock%x),1,1)

  x = 1.0; y = 2.0; a = 2.0
  x_d = x
  y_d = y
  call saxpy<<<grid, tBlock>>>(x_d, y_d, a)
  y = y_d
  write(*,*) 'Max error: ', maxval(abs(y-4.0))
end program testSaxpy
```

New Fortran with HIP calls

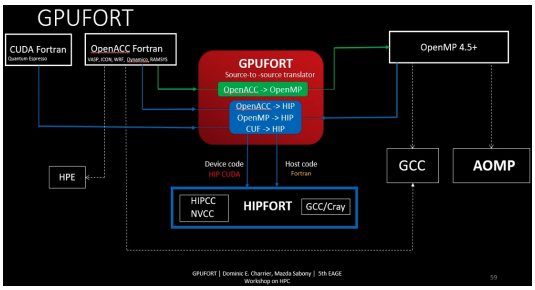
```
program testSaxpy
  use iso_c_binding
  use hipfort
  use hipfort_check

  implicit none
  interface
     subroutine launch(y,x,b,N) bind(c)
       use iso_c_binding
       implicit none
       type(c_ptr) :: y,x
       integer, value :: N
       real, value :: b
     end subroutine
  end interface

  type(c_ptr) :: dx = c_null_ptr
  type(c_ptr) :: dy = c_null_ptr
  integer, parameter :: N = 40000
  integer, parameter :: bytes_per_element = 4
  integer(c_size_t), parameter :: Nbytes = N*bytes_per
  real, allocatable,target,dimension(:) :: x, y

  real, parameter ::  a=2.0
  real :: x_d(N), y_d(N)
```

```
  call hipCheck(hipMalloc(dx,Nbytes))
  call hipCheck(hipMalloc(dy,Nbytes))

  allocate(x(N))
  allocate(y(N))
  x = 1.0
  y = 2.0

  call hipCheck(hipMemcpy(dx, c_loc(x), Nbytes, hipMemcpyHostT
  call hipCheck(hipMemcpy(dy, c_loc(y), Nbytes, hipMemcpyHostT

  call launch(dy, dx, a, N)
  call hipCheck(hipDeviceSynchronize())
  call hipCheck(hipMemcpy(c_loc(y), dy, Nbytes, hipMemcpyDevic

  write(*,*) 'Max error: ', maxval(abs(y-4.0))

  call hipCheck(hipFree(dx))
  call hipCheck(hipFree(dy))

  deallocate(x)
  deallocate(y)

end program testSaxpy
```

HIP kernel + C wrapper

```
#include <hip/hip_runtime.h>
#include <cstdio>

__global__ void saxpy(float *y, float *x, float a, int n)
{
  size_t i = blockDim.x * blockIdx.x  + threadIdx.x;
  if (i < n) y[i] = y[i] + a*x[i];
}


extern "C"
{
  void launch(float **dout, float **da, float db, int N)
  {
    dim3 tBlock(256,1,1);
    dim3 grid(ceil((float)N/tBlock.x),1,1);

    hipLaunchKernelGGL((saxpy), grid, tBlock, 0, 0, *dout, *da, db, N);
  }
}
```

GPUFort



GPUFort (II)

```
program saxpy

  implicit none
  integer, parameter :: N = 8192
  real :: y(N), x(N), a
  integer :: i
  a=2.0
  x(1)=5.0

  !$acc data copy(x(1:N),y(1:N))
  !$acc parallel loop
  do i = 1, N
   y(i) = a * x(i) + y(i)
  enddo
  !$acc end data

  print *, y(1)
end program
```

Ifdef original file

```
#ifdef __GPUFORT
  call gpufort_acc_enter_region()
  dev_x = gpufort_acc_copy(x(1:N))
  dev_y = gpufort_acc_copy(y(1:N))

  ! extracted to HIP C++ file
  call launch_axpy_12_b2e350_auto(0,c_null_ptr,dev_y,size(y,1),lbound(y,1),a,dev_x,size(x,1),lbound(x,1),n)
  call gpufort_acc_wait()
  call gpufort_acc_exit_region()
#else
!$acc data copy(x(1:N),y(1:N))

  !$acc parallel loop
  do i = 1, N
   y(i) = a * x(i) + y(i)
  enddo
  !$acc end data
#endif
```

GPUFort (III)

Extern C routine

```
extern "C" void launch_axpy_13_b2e350_auto(
  const int sharedmem,
  hipStream_t stream,
  float * __restrict__ y,
  const int y_n1,
  const int y_lb1,
  float a,
  float * __restrict__ x,
  const int x_n1,
  const int x_lb1,
  int n) {
 const int axpy_13_b2e350_blockX = 128;
 dim3 block(axpy_13_b2e350_blockX);
 const int axpy_13_b2e350_NX = (1 + ((n) - (1)));
 const int axpy_13_b2e350_gridX = divideAndRoundUp( axpy_13_b2e350_NX, axpy_13_b2e350_blockX );
 dim3 grid(axpy_13_b2e350_gridX);
// launch kernel
 hipLaunchKernelGGL((axpy_13_b2e350), grid, block, sharedmem, stream, y,y_n1,y_lb1,a,x,x_n1,x_lb1,n);
```

Kernel

```
__global__ void axpy_13_b2e350(
  float * __restrict__ y,
  const int y_n1,
  const int y_lb1,
  float a,
  float * __restrict__ x,
  const int x_n1,
  const int x_lb1,
  int n) {
 #undef _idx_y
 #define _idx_y(a) ((a-(y_lb1)))
 #undef _idx_x
 #define _idx_x(a) ((a-(x_lb1)))
 int i = 1 + (1)*(threadIdx.x + blockIdx.x * blockDim.x);
 if (loop_cond(i,n,1)) {
  y[_idx_y(i)]=(a*x[_idx_x(i)]+y[_idx_y(i)]);
 }
}
```

Outline

- GPU context
- Device management
- Programming models
- Peer access (GPU-GPU)
- MPI+HIP

Introduction

- Workstations or supercomputer nodes can be equipped with several GPUs
  - For the current supercomputers, the number of GPUs per node usually ranges between 2 to 6
  - Allows sharing (and saving) resources (disks, power units, e.g.)
  - More GPU resources per node, better per-node-performance

GPU Context

- Context is established when the first HIP function requiring an active context is called (eg. `hipMalloc()`)
- Several processes can create contexts for a single device
- Resources are allocated per context
- By default, one context per device per process (since CUDA 4.0)
  - Threads of the same process share the primary context (for each device)

Selecting device

- Driver associates a number for each HIP-capable GPU starting from 0
- The function `hipSetDevice()` is used for selecting the desired device

Device management

```
// Return the number of hip capable devices in `count`
hipError_t hipGetDeviceCount(int *count)

// Set device as the current device for the calling host thread
hipError_t hipSetDevice(int device)

// Return the current device for the calling host thread in `device`
hipError_t hipGetDevice(int *device)

// Reset and explicitly destroy all resources associated with the current device
hipError_t hipDeviceReset(void)
```

Querying device properties

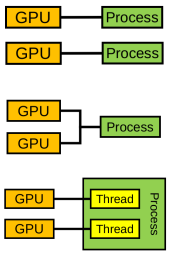
- One can query the properties of different devices in the system using `hipGetDeviceProperties()` function
  - No context needed
  - Provides e.g. name, amount of memory, warp size, support for unified virtual addressing, etc.
  - Useful for code portability

Return the properties of a HIP capable device in `prop`

```
hipError_t hipGetDeviceProperties(struct hipDeviceProp *prop, int device)
```

Multi-GPU programming models

- One GPU per process
  - Syncing is handled through message passing (eg. MPI)
- Many GPUs per process
  - Process manages all context switching and syncing explicitly
- One GPU per thread
  - Syncing is handled through thread synchronization requirements



Multi-GPU, one GPU per process

- Recommended for multi-process applications using a message passing library
- Message passing library takes care of all GPU-GPU communication
- Each process interacts with only one GPU which makes the implementation easier and less invasive (if MPI is used anyway)
  - Apart from each process selecting a different device, the implementation looks much like a single-GPU program
- **Multi-GPU implementation using MPI is discussed at the end!**

Multi-GPU, many GPUs per process

- Process switches the active GPU using `hipSetDevice()` function
- After setting the device, HIP-calls such as the following are effective only on the selected GPU:
  - Memory allocations and copies
  - Streams and events
  - Kernel calls
- Asynchronous calls are required to overlap work across all devices

Many GPUs per process, code example

```
for(unsigned int i = 0; i < deviceCount; i++)
{
  hipSetDevice(i);
  kernel<<<blocks[i],threads[i]>>>(arg1[i], arg2[i]);
}
```

Multi-GPU, one GPU per thread

- One GPU per CPU thread
  - I.e one OpenMP thread per GPU being used
- HIP API is threadsafe
  - Multiple threads can call the functions at the same time
- Each thread can create its own context on a different GPU
  - `hipSetDevice()` sets the device and creates a context per thread
  - Easy device management with no changing of device
- Communication between threads becomes a bit more tricky

One GPU per thread, code example

```
#pragma omp parallel for
for(unsigned int i = 0; i < deviceCount; i++)
{
  hipSetDevice(i);
  kernel<<<blocks[i],threads[i]>>>(arg1[i], arg2[i]);
}
```

Peer access

- Access peer GPU memory directly from another GPU
  - Pass a pointer to data on GPU 1 to a kernel running on GPU 0
  - Transfer data between GPUs without going through host memory
  - Lower latency, higher bandwidth

```
// Check peer accessibility
hipError_t hipDeviceCanAccessPeer(int* canAccessPeer, int device, int peerDevice)

// Enable peer access
hipError_t hipDeviceEnablePeerAccess(int peerDevice, unsigned int flags)

// Disable peer access
hipError_t hipDeviceDisablePeerAccess(int peerDevice)
```

Peer to peer communication

- Devices have separate memories
- With devices supporting unified virtual addressing, `hipMemCpy()` with `kind=hipMemcpyDefault`, otherwise, `hipMemcpyPeer()`:

```
// First option with unified virtual addressing
hipError_t hipMemcpy(void* dst, void* src, size_t count, hipMemcpyKind kind)

// Other option which does not require unified virtual addressing
hipError_t hipMemcpyPeer(void* dst, int  dstDev, void* src, int srcDev, size_t count
```

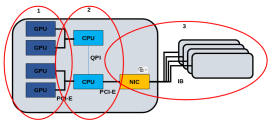
- If peer to peer access is not available, the functions result in a normal copy through host memory

Message Passing Interface (MPI)

- MPI is a widely adopted standardized message passing interface for distributed memory parallel computing
- The parallel program is launched as a set of independent, identical processes
  - Same program code and instructions
  - Each process can reside in different nodes or even different computers
- All variables and data structures are local to the process
- Processes can exchange data by sending and receiving messages

Three levels of parallelism

1. GPU - GPU threads on the multiprocessors
   - Parallelization strategy: HIP, SYCL, Kokkos, OpenMP
2. Node - Multiple GPUs and CPUs
   - Parallelization strategy: MPI, Threads, OpenMP
3. Supercomputer - Many nodes connected with interconnect
   - Parallelization strategy: MPI between nodes



MPI and HIP

- Trying to compile code with any HIP calls with other than the `hipcc` compiler can result in errors
- Either set MPI compiler to use `hipcc`, eg for OpenMPI:

```
OMPI_CXXFLAGS='' OMPI_CXX='hipcc'
```

- or separate HIP and MPI code in different compilation units compiled with `mpicxx` and `hipcc`
  - Link object files in a separate step using `mpicxx` or `hipcc`

MPI+HIP strategies

1. One MPI process per node
2. **One MPI process per GPU**
3. Many MPI processes per GPU, only one uses it
4. **Many MPI processes sharing a GPU**

- 2 is recommended (also allows using 4 with services such as CUDA MPS)
  - Typically results in most productive and least invasive implementation for an MPI program
  - No need to implement GPU-GPU transfers explicitly (MPI handles all this)
  - It is further possible to utilize remaining CPU cores with OpenMP (but this is not always worth the effort/increased complexity)

Selecting the correct GPU

- Typically all processes on the node can access all GPUs of that node
- The following implementation allows utilizing all GPUs using one or more processes per GPU
  - Use CUDA MPS when launching more processes than GPUs

```
int deviceCount, nodeRank;
MPI_Comm commNode;
MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &commNod
MPI_Comm_rank(commNode, &nodeRank);
hipGetDeviceCount(&deviceCount);
hipSetDevice(nodeRank % deviceCount);
```

GPU-GPU communication through MPI

- CUDA/ROCm aware MPI libraries support direct GPU-GPU transfers
  - Can take a pointer to device buffer (avoids host/device data copies)
- Unfortunately, currently no GPU support for custom MPI datatypes (must use a datatype representing a contiguous block of memory)
  - Data packing/unpacking must be implemented application-side on GPU
- ROCm aware MPI libraries are under development and there may be problems
  - It is a good idea to have a fallback option to use pinned host staging buffers

Summary

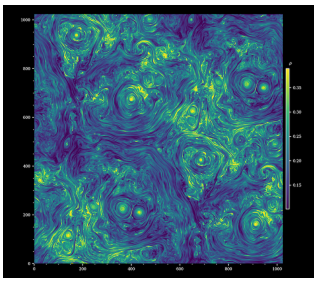
- There are many options to write a multi-GPU program
- Use `hipSetDevice()` to select the device, and the subsequent HIP calls operate on that device
- If you have an MPI program, it is often best to use one GPU per process, and let MPI handle data transfers between GPUs
- There is still little experience from ROCm aware MPIs, there may be issues
  - Note that a CUDA/ROCm aware MPI is only required when passing device pointers to the MPI, passing only host pointers does not require any CUDA/ROCm awareness

Application design

Design choices

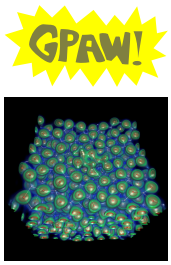
Runko

- Kinetic plasma simulation code
  - Particle-in-Cell with computational particles and electromagnetic fields on a grid
- Hybrid C++14/Python code
  - Domain super-decomposition with MPI
  - Massively parallel with runs on >10k cores



GPAW

- Density-functional theory -based electronic structure code
- Python + C + libraries (numpy, BLAS, LAPACK)
  - Various parallelization levels with MPI
  - Over 10k cores with some modes
  - ~20 developers all around the world

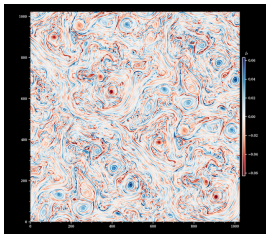


Why develop software?

- To do science
- To create a product
- Supercomputing platforms enable investigating bigger and more complex problems
- **Do science**
  - Scientific articles
  - Method-oriented articles presenting code and methods
- **Code as a product**
  - Prestige and fame
  - Gateway into projects, collaborations
  - Citations, co-authorships
  - Work on the bleeding edge

Case Runko: Going big...

- Kinetic plasma simulations are microscopical (<1cm)
  - Bigger simulation domains mean more realistic, larger systems
- Recently simulated turbulent plasma with 10^10 particles
  - New physics starts to appear at larger scale



Starting position

- New code or existing project / rewrite of old code?
- **Questions**: your software project?

Cases Runko & GPAW

Runko
- New code
- +1yr of development
- Allowed to start from scratch and use new technologies

GPAW
- Existing code with basic features mostly working (95 %) in 2005
- Choice of physical model and programming languages had been made
- Production ready in 2008-2009
  - Science done already earlier

Design model

- Development is not only about physics and numerics
  - Also about **how** you do it
- Instead of "Just code" it is advantageous to plan a little too!
- So-called Agile Development
  - Umbrella term for different software development methods
  - Divide work into small tasks, define short working period, review, repeat

Agile development model

- Focused on iterative and incremental development
  - Quick prototyping
  - Supports continuous publication
  - Analysis, coding, testing, etc. never end
- Development cycle
  - Plan
  - Design
  - Develop
  - Test
  - Release
  - Feedback

Parallelization strategies

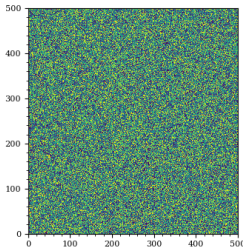
- Planning includes thinking what is the target platform
- Target machines: laptops, small clusters, supercomputers
  - OpenMP, MPI, MPI+OpenMP, GPUs
- From shared memory to distributed memory machines
  - Keep in mind that most machines are distributed memory systems = MPI
- Moving from <1000 cores to >10k cores
  - Parallellization strategies need to be considered
  - Non-blocking, avoiding global calls,...
- Accelerators
  - GPUs have their own tricks and quirks

Parallelization strategies

- Going **BIG** -> GPUs are mandatory
- But not all HPC needs to be exascale
  - Size is not a goal in itself

Case Runko: Parallellization

- Runko has uses a new novel parallellization strategy
  - Relies on dividing work among small subregions of the grid
  - Computational grid (i.e., what rank owns which tiles) is constantly changing to balance the load
- Moving beyond 1000 cores is non-trivial
  - Non-blocking communication
  - Removal of collectives
  - Re-design of IO



Programming languages

- Selection of languages
  - Most common are C, C++, Fortran
  - Mostly matter of taste
- C++ more object-oriented features and many more data structures (maps, lists, etc.); low-level memory management
- Fortran is really good for number crunching, good array syntax
- But also newcomers like Python/Julia
  - Faster coding cycle and less error prone
  - Testing, debugging, and prototyping much easier

Hybrid codes

- 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 functions
- Combinations/suggestions
  - Python & C++ (PyBind11) for object-oriented programming
  - Julia & Fortran (native) for functional programming

Case Runko: C++14/Python3 code

- Runko is an example of a hybrid code
- Low-level “kernels” are in C++
- High-level functionality is operated from Python scripts
- So far it has been an excellent choice
  - Fast code
  - Ease of use
  - Rapid prototyping

Modular code design: programming

- Good code is modular
  - Encapsulation
  - Self-contained functions
  - No global variables, input what you need
- Modular code takes more time to design but is **a lot** easier to extend and understand

Modular code design: tools

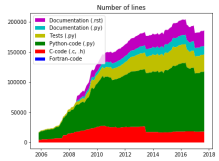
- Avoid **not invented here** syndrome
- Leverage existing software and libraries
  - Libraries
    - Numerical (BLAS, solvers,...)
    - I/O
    - Parallelization
  - Frameworks?
    - Plug your model into an existing framework?
    - PETSc, Trilinos, BoxLib++, AMReX, corgi,...
- Caveats:
  - Is the lib still supported/updated?
  - Do you trust the source, is it widely used
  - Is there documentation
  - Does it support all the features

Modular code design: development tools

- Software development is time consuming, many tools exist to help you in the process
- Build systems automate compiling
  - Makefiles, CMake, Ninja, ...
- Debuggers
- Compilers
  - Compilers are not the same, compiler bugs are real!
  - Test your code with different compilers (gnu, clang, intel, cray,...)
- **Questions**: Choices in your software and experiences about them?

Case GPAW: Modular design

- Object oriented features and Python modules heavily utilized
- Main numerical kernels well separated from high level algorithms
- New features can be developed independently



Data formats

- Data has to be “designed” too
- Data formats
  - Not just plain text files/binary files
  - Platform-independent formats (HDF5, NetCDF, ...)
  - Metadata together with the data?
- Log files. Especially useful with HPC applications
- Standard formats
  - Your field might have some data standards (e.g., PML for plasma codes)
- Remember also that large simulations produce lots of data
  - Storing “big data” is an issue

Case Runko: IO issues

- Runko uses rank-independent multiple-file IO strategy
  - Excellent performance as there is no synching
  - But, sometimes the burst performance is too good...
    - 10k cores writing ~TBs of data in seconds is nice for the user but file system might not like it

Summary

- Software design is all about planning (agile development)
- Productivity
  - Modular design
  - Use existing libraries
  - Use & adopt design, community, and collaboration tools
  - Programming language and design selection

Documenting and testing

Why to document your code?

- You will forget details
  - Code that you wrote 6 months ago is often indistinguishable from code that someone else has written
- Writing documentation may improve the design of your code
- Have other people to use (and cite!) your code
  - If the documentation is not good enough, people will not use it
- Allow other people to contribute to development
  - Practically impossible without documentation
- Even within a group, people move in and out

What to document?

- How to use the code
  - Installation instructions
  - Format of the input file and possible values of input parameters
  - Tutorials on specific cases
  - Examples that can be copy-pasted
  - FAQs
  - How to cite the code !

What to document?

- How to develop the code
  - Equations implemented in the code when appropriate
    - Mapping of physical symbols to variables in code
  - Coding style
  - Instructions for contributing
  - APIs (application programming interfaces)
  - Implementation details

How to document?

- Documentation should be part of the code
- Keep the documentation close to source
  - Same versioning as for the code
  - Guides in e.g. `docs` subdirectory
  - APIs in the source code, e.g. describe arguments of subroutine next to the definition
    - many tools can generate automatically API documention from comments
  - Non-obvious implementation choices in comments in source code
- Tools for documentation: RST and Markdown markup languages, wikis, Doxygen, github pages and readthedocs for hosting
- Good documentation does not save bad code!

How is your code documented?

Discuss within your table!

Documentation in GPAW & Runko

GPAW
- User developer guides, tutorials etc. written in RST
- www-site generated with Sphinx
- APIs in Python docstrings
- https://wiki.fysik.dtu.dk/gpaw

Runko
- Main structure described in a publication
- GitHub Readmes + www-site with sphinx
- API automatically documented with doxygen+breathe
- https://github.com/natj/runko
- https://runko.readthedocs.io

Testing

Simulations and analysis with untested software do not constitute science!

- Experimental scientist would never conduct an experiment with uncalibrated detectors
- Computational scientist should never conduct simulations with untested software

Why software needs to be tested?

- Ensure expected functionality
- Ensure expected functionality is preserved
  - Software is fragile, bugs lurk in easily
  - In large projects innocent looking changes can have surprising side effects
- Testing helps detecting errors early
- Testing is essential for reproducibility of results
- Tests make it easier to verify whether software is correctly installed

Defensive programming

- Would you trust a software ...
  - ... when its tests do not pass?
  - ... if the tests are never run?
  - ... if there are no tests at all?
- Assume mistakes will happen and introduce guards against them
- Test drive development
  - Tests are written before actually implementing the functionality

What should be tested in software ?

- Validity of the physical model
  - Comparison to experimental results
- Validity of numerical methods and implementation
  - Comparison to analytic solutions of special cases
  - Conservation laws and symmetries
- Correct functioning of individual subroutines and modules
- Performance
  - Changes in software may lead into degradation in performance
- Dependency variants
  - At least compiler and mpi implementation

Unit testing

- Tests for smallest testable part of an application
  - Function, module or class
- Ideally, tests are independent of each other
- Frameworks: cUnit, cppunit, pFUnit, Google Test, pytest, ...
- Client code which executes tests can be also hand-written
- Unit testing helps in managing complexity
  - Well structured and modular code is easy to test

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

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

Challenges with HPC

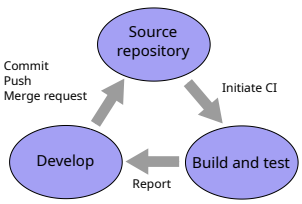
- Performance is often system/architecture specific
  - Preferably test on multiple architectures
- Complicated dependency chains makes testing even harder
  - Impossible to test exhaustively
- Systems are very noisy, especially on the filesystem and network level.

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
  - Jenkins
  - GitlabCI
  - GitHub Actions



How is your code tested?

Discuss within your table!

Testing in GPAW & Runko

GPAW
- Wide test set
  - Unit tests and integration tests
- Continuous integration with GitlabCI
- Developers should run tests manually before pushing
- Test set should run after installation

Runko
- Each module covered with unit tests (pytest)
- Physics tested with bigger integration tests
- Continuous integration with TravisCI

Take home messages

- Document your code
- Test your code, prefer automatic testing

Collaboration and release

Material based partly on Coderefinery lesson "Social coding and open software" by Radovan Bast, Richard Darst, Sabry Razick, Jyry Suvilehto, http://cicero.xyz/v3/remark/0.14.0/github.com/coderefinery/social-coding/master/talk.md

Collaboration in software development

- Scientific software is seldom developed only by one person
- Closed development within a limited group
  - Local research group
  - Group of collaborators
- Open development
  - Anyone can contribute (within the terms given by main developers)

What is needed for collaboration?

- Mostly same requirements for closed and open development
- Common version control is minimum requirement for collaboration
- Automated testing and reasonable documentation are also very beneficial
- Some communication tools
  - Mailing lists, instant messaging
  - Issue trackers
- Clear development guidelines
  - Code review process is likely beneficial

Benefits of sharing software

- Easier to find and reproduce (scientific reproducibility)
- More trustworthy: others can verify correctness and find and report bugs
- Enables others to build on top of your code (if the license allows it)
- Others can submit features/improvements and fix bugs
- Many tools and apps are free for open source (GitHub, Travis CI, Appveyor, Read the Docs)
- Good for your CV: you can show what you have built
- Starting to become a requirement in a lot of projects

Why do some researchers prefer not to share?

- Fear of being scooped
- Exposes possibly "ugly code"
- Others may find bugs
- Others may require support and ask too many questions
- Fear of losing control over the direction of the project
- "Bad" derivative projects may appear - fear that this will harm the reputation

To share or not to share?

- **Discuss** what do you think

Open vs. closed source vs. free software

- Free software does not mean that software is for free
- Open source license does not mean you need to share everything immediately
- Open source does not mean public domain: software in the public domain has no owner
- Open source does not mean non-commercial: plenty of companies produce and support it

Copyright vs. license

- Copyright: intellectual property granted to the creator of software
- License: Terms given by the copyright owner for using and redistributing the software
- Owner of the copyright depends on the local legislation, work contract etc.
  - As an example, CSC holds the copyright to the work created by CSC employees
- Common open source licenses: MIT, Apache, BSD, GPL, Creative Commons

Practical recommendations for licensing

- License your code very early in the project
  - Ideally develop publicly accessible open source code from day one
- Open source is in the spirit of openness of science
  - Take an OSI-approved license: makes it easier to evaluate
  - Do not use custom licenses for open source: compatibility not clear.
  - Open sourcing makes sure you are not locked out of your own code once you change affiliation.
- Add a `LICENSE` file to your repository (GitHub understands it):
  - Use GitHub web to add file named `LICENSE` and it helps you select!

Social coding

- Licensing is one thing... but will anyone ever contribute to code?
- Do you welcome people to your project?
- Do you give credit?
- Do you respond to issues and pull requests?
- Openness and transparency
- Document whether/how/where you want to be asked questions
  - Chat or mailing list
- If the project grows, agree on a decision process for controversial changes

Releasing software

- When software is ready for public releases?
  - Never (there are always bugs or missing features)
  - From day one (it will never be perfect in any case)
- Maybe you want to publish some science before releasing software?
- Having a citation for software is recommended
- Documentation and support processes are beneficial
- Generally, it is better to release often
  - Make clear release notes about changes

Discuss: Do you do software releases / versioning?

Case GPAW

- GPL from the beginning, copyright scattered over all developers
- Gitlab, mailing list, issue tracker, face-to-face workshops
- First method paper in 2005, major review article in 2010
  - Together over 1600 citations
- From single developer to tens of users and developers all over the world

MPI Reference

C interfaces

C interfaces for the “first six” MPI operations

```
int MPI_Init(int *argc, char **argv)

int MPI_Init_thread(int *argc, char **argv, int required, int *provided)

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()
```

C interfaces for the basic 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
                 void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, in
                 MPI_Comm comm, MPI_Status *status)

int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
```

MPI datatypes for C

MPI type	C type
MPI_CHAR	signed char
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	

C interfaces for collective operations

```
int MPI_Bcast(void* buffer, int count, MPI_datatype datatype, int root, MPI_Comm com

int MPI_Scatter(void* sendbuf, int sendcount, MPI_datatype sendtype,
                void* recvbuf, int recvcount, MPI_datatype recvtype, int root, MPI_C

int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs, MPI_datatype sendtype,
                 void* recvbuf, int recvcount, MPI_datatype recvtype, int root, MPI_

int MPI_Gather(void* sendbuf, int sendcount, MPI_datatype sendtype,
               void* recvbuf, int recvcount, MPI_datatype recvtype, int root, MPI_Co

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)
```

Available reduction operations

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

Operation	Meaning
MPI_LAND	Logical AND
MPI_BAND	Bytewise AND
MPI_LOR	Logical OR
MPI_BOR	Bytewise OR
MPI_LXOR	Logical XOR
MPI_BXOR	Bytewise XOR

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 )
```

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_stat
```

C interfaces for Cartesian process topologies

```
int MPI_Cart_create(MPI_Comm old_comm, int ndims, int *dims, int *periods, int reord
                    MPI_Comm *comm_cart)

int MPI_Dims_create(int ntasks, int ndims, int *dims);

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 )
```

C interfaces for persistent communication

```
int MPI_Send_init(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
              MPI_Comm comm, MPI_Request *request )

int MPI_Recv_init(void *buf, int count, MPI_Datatype datatype, int source, int tag,
              MPI_Comm comm, MPI_Request *request )

int MPI_Start(MPI_Request *request)

int MPI_Startall(int count, MPI_Request *array_of_requests);
```

C interfaces for neighborhood collectives

```
int MPI_Neighbor_allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                           void* recvbuf, int recvcount, MPI_datatype recvtype, MPI_Comm comm);

int MPI_Neighbor_allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                            void* recvbuf, int* recvcounts, int* displs, MPI_datatype recvtype,
                            MPI_Comm comm);

int MPI_Neighbor_alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                          void* recvbuf, int recvcount, MPI_datatype recvtype,
                          MPI_Comm comm);

int MPI_Neighbor_alltoallv(void* sendbuf, int* sendcounts, int* senddispls, MPI_Datatype sendtype,
                           void* recvbuf, int* recvcounts, int* recvdispls, MPI_datatype recvtype,
                           MPI_Comm comm);

int MPI_Neighbor_alltoallw(void* sendbuf, int* sendcounts, int* senddispls, MPI_Datatype* sendtypes,
                           void* recvbuf, int* recvcounts, int* recvdispls, MPI_datatype* recvtypes,
                           MPI_Comm comm);
```

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 )

int MPI_Type_create_struct(int count, const int array_of_blocklengths[],
                           const MPI_Aint array_of_displacements[],
                           const MPI_Datatype array_of_types[], MPI_Datatype *newtype)
```

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 t
            MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MP

int MPI_Accumulate(const void *origin_addr, int origin_count, MPI_Datatype origin_da
                   int target_rank, MPI_Aint target_disp, int target_count,
                   MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
```

C interfaces to MPI I/O routines

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File

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_Stat

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_Sta

int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
                      MPI_Datatype datatype, MPI_Status *status)
```

C interfaces to MPI I/O 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)
```

C interfaces for environmental inquiries

```
int MPI_Get_processor_name(char *name, int *resultlen)
```

Fortran interfaces

Fortran interfaces for the “first six” MPI operations

```
mpi_init(ierror)
  integer :: ierror

mpi_init_thread(required, provided, ierror)
  integer :: required, provided, ierror

mpi_comm_size(comm, size, ierror)
mpi_comm_rank(comm, rank, ierror)
  type(mpi_comm) :: comm
  integer :: size, rank, ierror

mpi_barrier(comm, ierror)
  type(mpi_comm) :: comm
  integer :: ierror

mpi_finalize(ierror)
  integer :: ierror
```

Fortran interfaces for the basic point-to-point operations

```
mpi_send(buffer, count, datatype, dest, tag, comm, ierror)
  <type> :: buf(*)
  integer :: count, dest, tag, ierror
  type(mpi_datatype) :: datatype
  type(mpi_comm) :: comm

mpi_recv(buf, count, datatype, source, tag, comm, status, ierror)
  <type> :: buf(*)
  integer :: count, source, tag, ierror
  type(mpi_datatype) :: datatype
  type(mpi_comm) :: comm
  type(mpi_status) :: status
```

Fortran interfaces for the basic point-to-point operations

```
mpi_sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, &
             recvtype, source, recvtag, comm, status, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: sendcount, recvcount, dest, source, sendtag, recvtag, ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm
  type(mpi_status) :: status

mpi_get_count(status, datatype, count, ierror)
  integer :: count, ierror
  type(mpi_datatype) :: datatype
  type(mpi_status) :: status
```

MPI datatypes for Fortran

MPI type	Fortran type
MPI_CHARACTER	character
MPI_INTEGER	integer
MPI_REAL	real32
MPI_DOUBLE_PRECISION	real64
MPI_COMPLEX	complex
MPI_DOUBLE_COMPLEX	double complex
MPI_LOGICAL	logical
MPI_BYTE	

Fortran interfaces for collective operations

```
mpi_bcast(buffer, count, datatype, root, comm, ierror)
  <type> :: buffer(*)
  integer :: count, root, ierror
  type(mpi_datatype) :: datatype
  type(mpi_comm) :: comm

mpi_scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm,
  <type> :: sendbuf(*), recvbuf(*)
  integer :: sendcount, recvcount,  root, ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm

mpi_scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, &
             root, comm, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: sendcounts(*), displs(*), recvcount, ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm
```

Fortran interfaces for collective operations

```
mpi_gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, i
  <type> :: sendbuf(*), recvbuf(*)
  integer :: sendcount, recvcount,  root, ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm

mpi_gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, &
            root, comm, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: sendcount, recvcounts(*), displs(*), ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm

mpi_reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: count, root, ierror
  type(mpi_datatype) :: datatype
```

Fortran interfaces for collective operations

```
mpi_allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: count, ierror
  type(mpi_datatype) :: datatype
  type(mpi_op) :: op
  type(mpi_comm) :: comm

mpi_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierr
  <type> :: sendbuf(*), recvbuf(*)
  integer :: sendcount, recvcount, ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm

mpi_reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: recvcounts(*), ierror
  type(mpi_datatype) :: datatype
```

Fortran interfaces for collective operations

```
mpi_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, ierro
  type:: sendbuf(*), recvbuf(*)
  integer :: sendcount, recvcount, ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm

mpi_alltoallv(sendbuf,sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, r
              comm, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: sendcounts(*), recvcounts(*), sdispls(*), rdispls(*), ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm
```

Available reduction operations

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

Operation	Meaning
MPI_LAND	Logical AND
MPI_BAND	Bytewise AND
MPI_LOR	Logical OR
MPI_BOR	Bytewise OR
MPI_LXOR	Logical XOR
MPI_BXOR	Bytewise XOR

Fortran interfaces for user-defined communicators

```
mpi_comm_split(comm, color, key, newcomm, ierror)
  integer :: color, key, ierror
  type(mpi_comm) :: comm, newcomm

mpi_comm_compare(comm1, comm2, result, ierror)
  integer :: result, ierror
  type(mpi_comm) :: comm1, comm2

mpi_comm_dup(comm, newcomm, ierror)
  integer :: ierror
  type(mpi_comm) :: comm, newcomm

mpi_comm_free(comm, ierror)
  integer :: ierror
  type(mpi_comm) :: comm
```

Fortran interfaces for non-blocking operations

```
mpi_isend(buf, count, datatype, dest, tag, comm, request,ierror)
  <type> :: buf(*)
  integer :: count, dest, tag, ierror
  type(mpi_datatype) :: datatype
  type(mpi_request) :: request
  type(mpi_comm) :: comm

mpi_irecv(buf, count, datatype, source, tag, comm, request,ierror)
  <type> :: buf(*)
  integer :: count, source, tag, ierror
  type(mpi_datatype) :: datatype
  type(mpi_request) :: request
  type(mpi_comm) :: comm

mpi_wait(request, status, ierror)
  integer :: ierror
  type(mpi_request) :: request
  type(mpi_status) :: status

mpi_waitall(count, array_of_requests, array_of_statuses, ierror)
  integer :: count, ierror
  type(mpi_request) :: array_of_requests(:)
  type(mpi_status) :: array_of_statuses(:)
```

Fortran interfaces for Cartesian process topologies

```
mpi_cart_create(old_comm, ndims, dims, periods, reorder, comm_cart, ierror)
  integer :: ndims, dims(:), ierror
  type(mpi_comm) :: old_comm, comm_cart
  logical :: reorder, periods(:)

mpi_dims_create(ntasks, ndims, dims, ierror)
  integer :: ntasks, ndims, dims(:), ierror

mpi_cart_coords(comm, rank, maxdim, coords, ierror)
  integer :: rank, maxdim, coords(:), ierror
  type(mpi_comm) :: comm

mpi_cart_rank(comm, coords, rank, ierror)
  integer :: coords(:), rank, ierror
  type(mpi_comm) :: comm

mpi_cart_shift(comm, direction, displ, low, high, ierror)
  integer :: direction, displ, low, high, ierror
  type(mpi_comm) :: comm
```

Fortran interfaces for persistent communication

```
mpi_send_init(buf, count, datatype, dest, tag, comm, request,ierror)
  <type> :: buf(*)
  integer :: count, dest, tag, ierror
  type(mpi_datatype) :: datatype
  type(mpi_request) :: request
  type(mpi_comm) :: comm

mpi_recv_init(buf, count, datatype, source, tag, comm, request,ierror)
  <type> :: buf(*)
  integer :: count, source, tag, ierror
  type(mpi_datatype) :: datatype
  type(mpi_request) :: request
  type(mpi_comm) :: comm

mpi_start(request, ierror)
  integer :: ierror
  type(mpi_request) :: request

mpi_startall(count, array_of_requests, ierror)
  integer :: count, ierror
  type(mpi_request) :: array_of_requests(:)
```

Fortran interfaces for neighborhood collectives

```
mpi_neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, c
  <type> :: sendbuf(*), recvbuf(*)
  integer :: sendcount, recvcount, ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm

mpi_neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, r
                        comm, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: sendcount, recvcounts(:), displs(:), ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm

mpi_neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, co
  <type> :: sendbuf(*), recvbuf(*)
  integer :: sendcount, recvcount, ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm
```

Fortran interfaces for neighborhood collectives

```
mpi_neighbor_alltoallv(sendbuf, sendcounts, sendtype, senddispls, &
                       recvbuf, recvcounts, recvdispls, recvtype, comm, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: sendcounts(:), recvcounts(:), senddispls(:), recvdispls(:), ierror
  type(mpi_datatype) :: sendtype, recvtype
  type(mpi_comm) :: comm

mpi_neighbor_alltoallw(sendbuf, sendcounts, sendtypes, senddispls, &
                       recvbuf, recvcounts, recvdispls, recvtypes, comm, ierror)
  <type> :: sendbuf(*), recvbuf(*)
  integer :: sendcounts(:), recvcounts(:), senddispls(:), recvdispls(:), ierror
  type(mpi_datatype) :: sendtypes(:), recvtypes(:)
  type(mpi_comm) :: comm
```

Fortran interfaces for datatype routines

```
mpi_type_commit(type, ierror)
  type(mpi_datatype) :: type
  integer :: ierror

mpi_type_free(type, ierror)
  type(mpi_datatype) :: type
  integer :: ierror

mpi_type_contiguous(count, oldtype, newtype, ierror)
  integer :: count, ierror
  type(mpi_datatype) :: oldtype, newtype

mpi_type_vector(count, block, stride, oldtype, newtype, ierror)
  integer :: count, block, stride, ierror
  type(mpi_datatype) :: oldtype, newtype
```

Fortran interfaces for datatype routines

```
mpi_type_indexed(count, blocks, displs, oldtype, newtype, ierror)
  integer :: count, ierror
  integer, dimension(count) :: blocks, displs
  type(mpi_datatype) :: oldtype, newtype

mpi_type_create_subarray(ndims, sizes, subsizes, starts, order, oldtype, newtype, ie
  integer :: ndims, order, ierror
  integer, dimension(ndims) :: sizes, subsizes, starts
  type(mpi_datatype) :: oldtype, newtype

mpi_type_create_struct(count, blocklengths, displacements, types, newtype, ierror)
  integer :: count, blocklengths(count), ierror
  type(mpi_datatype) :: types(count), newtype
  integer(kind=mpi_address_kind) :: displacements(count)
```

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, ierror
  type(mpi_info) :: info
  type(mpi_comm) :: comm
  type(mpi_win)  :: win

mpi_win_fence(assert, win, ierror)
  integer :: assert, ierror
  type(mpi_win)  :: win

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, target_rank, target_count,  ierror
  type(mpi_datatype) :: origin_datatype, target_datatype
  type(mpi_win)  :: win
```

Fortran interfaces for one-sided routines

```
mpi_get(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target
        target_datatype, win, ierror)
  <type> :: origin_addr(*)
  integer(kind=mpi_address_kind) :: target_disp
  integer :: origin_count, target_rank, target_count,  ierror
  type(mpi_datatype) :: origin_datatype, target_datatype
  type(mpi_win)  :: win

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, target_rank, target_count,  ierror
  type(mpi_datatype) :: origin_datatype, target_datatype
  type(mpi_op) :: op
  type(mpi_win)  :: win
```

Fortran interfaces for MPI I/O routines

```
mpi_file_open(comm, filename, amode, info, fh, ierror)
  integer :: amode, ierror
  character* :: filename
  type(mpi_info) :: info
  type(mpi_file) :: fh
  type(mpi_comm) :: comm

mpi_file_close(fh, ierror)
  integer :: ierror
  type(mpi_file) :: fh

mpi_file_seek(fh, offset, whence, ierror)
  integer(kind=MPI_OFFSET_KIND) :: offset
  integer :: whence, ierror
  type(mpi_file) :: fh
```

Fortran interfaces for MPI I/O routines

```
mpi_file_read(fh, buf, count, datatype, status, ierror)
mpi_file_write(fh, buf, count, datatype, status, ierror)
  <type> :: buf(*)
  integer :: count, ierror
  type(mpi_file) :: fh
  type(mpi_datatype) :: datatype
  type(mpi_status) :: status

mpi_file_read_at(fh, offset, buf, count, datatype, status, ierror)
mpi_file_write_at(fh, offset, buf, count, datatype, status, ierror)
  <type> :: buf(*)
  integer(kind=MPI_OFFSET_KIND) :: offset
  integer :: count, ierror
  type(mpi_file) :: fh
  type(mpi_datatype) :: datatype
  type(mpi_status) :: status
```

Fortran interfaces for MPI I/O routines

```
mpi_file_set_view(fh, disp, etype, filetype, datarep, info, ierror)
  integer :: ierror
  integer(kind=MPI_OFFSET_KIND) :: disp
  type(mpi_info) :: info
  character* :: datarep
  type(mpi_file) :: fh
  type(mpi_datatype) :: etype, datatype

mpi_file_read_all(fh, buf, count, datatype, status, ierror)
mpi_file_write_all(fh, buf, count, datatype, status, ierror)
  <type> :: buf(*)
  integer :: count, ierror
  type(mpi_file) :: fh
  type(mpi_datatype) :: datatype
  type(mpi_status) :: status
```

Fortran interfaces for MPI I/O routines

```
mpi_file_read_at_all(fh, offset, buf, count, datatype, status, ierror)
mpi_file_write_at_all(fh, offset, buf, count, datatype, status, ierror)
  <type> :: buf(*)
  integer(kind=MPI_OFFSET_KIND) :: offset
  integer :: count, ierror
  type(mpi_file) :: fh
  type(mpi_datatype) :: datatype
  type(mpi_status) :: status
```

Fortra interfaces for environmental inquiries

```
mpi_get_processor_name(name, resultlen, ierror)
  character(len=MPI_MAX_PROCESSOR_NAME) :: name
  integer :: resultlen, ierror
```

Notes


CSC