Introduction to Parallel Thinking and Programming

Dr. Axel Kohlmeyer

Research Professor, Department of Chemistry
Associate Director, ICMS
College of Science and Technology
Temple University, Philadelphia
axel.kohlmeyer@temple.edu

The Abdus Salam International Centre for Theoretical Physics

PC Hardware is Very Parallel

Instruction level parallelism

- Pipelined / Superscalar design, Hyperthreading:
  -> multiple functional units operate concurrently

Register level parallelism

- Vector Unit (MMX/SSE/AVX):
  -> process multiple data elements at same time

Program level parallelism

- Multi-core CPU designs:
  -> single socket contains multiple full CPU units
  -> server hardware often also multi-socket

Heterogeneous parallelism

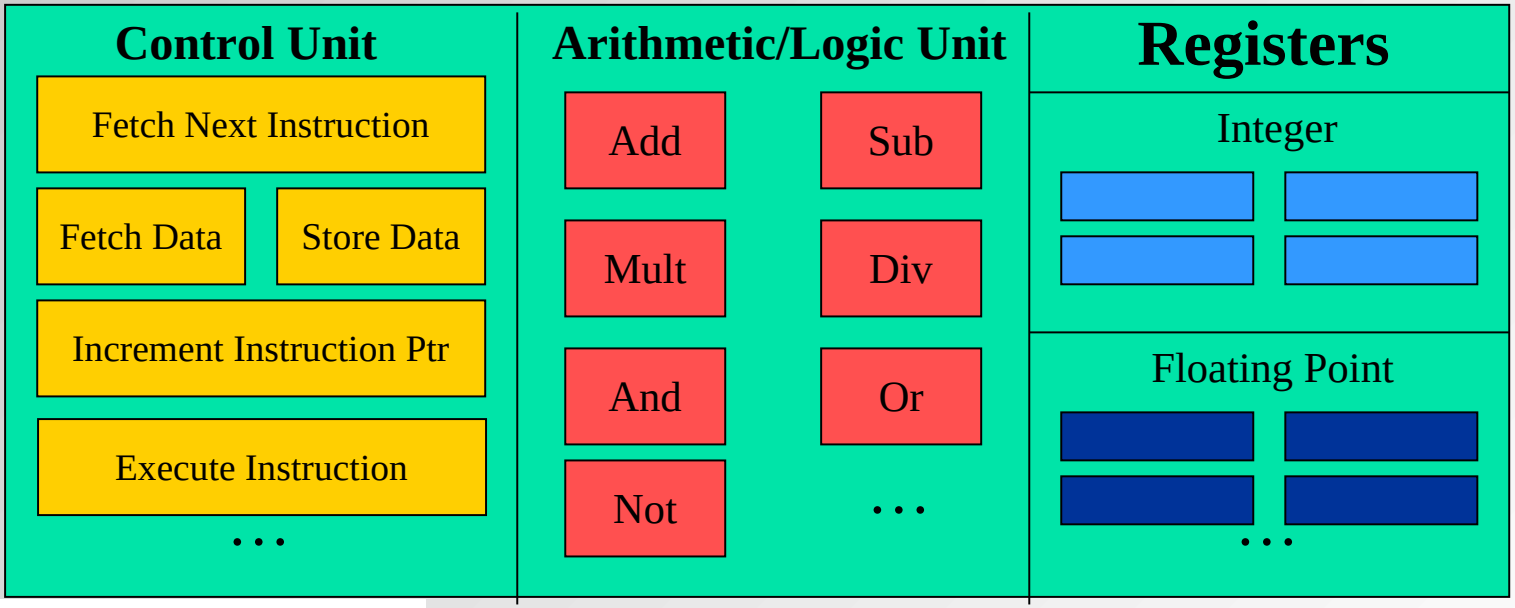
- Accelerator (GPU / Xeon Phi / FPGA):
  -> suitable parts of calculation can be offloaded

A Simple Calculator

1) Enter number on keyboard => register 1
2) Turn handle down -> add up -> subtract => count turns in register 2
3) Register 3 => accumulator
4) Multiply => add multiple times with shifts

A Simple CPU

- The basic CPU design is not much different from the mechanical calculator.
- Data still needs to be fetched into registers for the CPU to be able to operate on it.



CPU Pipeline

- One CPU “operation” has multiple steps/stages: fetch instr, decode instr, execute instr, memory lookup, write back => multiple functional units
- Using a pipeline allows for a faster CPU clock => like assembly line
- Dependencies and branches may force CPU to stall pipeline
- Complex operations usually not pipelined

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

How Would This Statement Be Executed?

```
z = a * b + c * d;
```

Actual steps:

```
z1 = a * b;
```

Data load can start while multiplying

```
z2 = c * d;
```

Start data load for next command

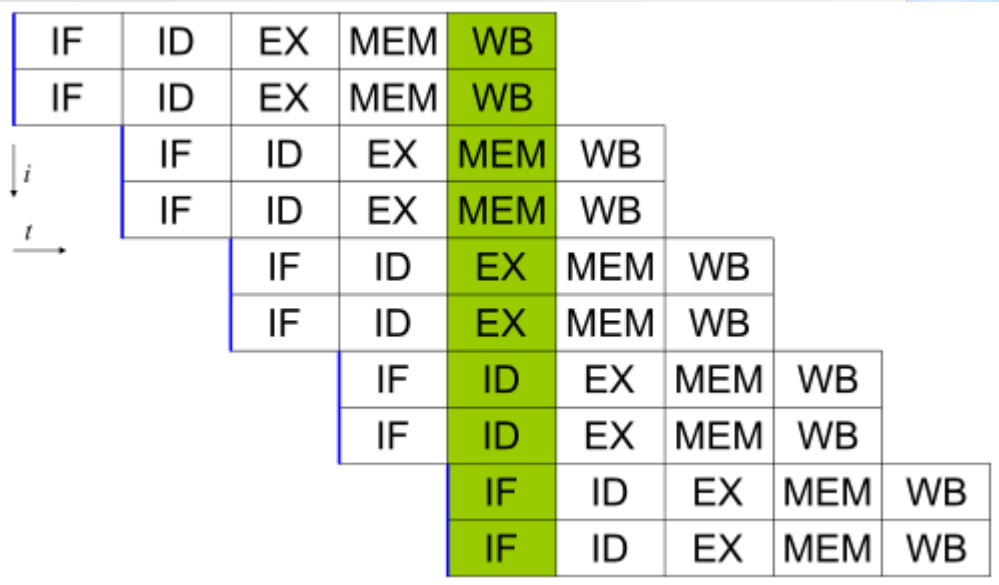
```
z= z1 + z2;
```

1. Load **a** into register **R0**
2. Load **b** into **R1**
3. Multiply **R2** = **R0** * **R1**
4. Load **c** into **R3**
5. Load **d** into **R4**
6. Multiply **R5** = **R3** * **R4**
7. Add **R6** = **R2** + **R5**
8. Store **R6** into **z**

Pipeline savings:
1 step out of 8, plus 3 more if next operation independent

Superscalar CPU design

- Superscalar CPU => instruction level parallelism
- Redundant functional units in single CPU core
  => multiple instructions executed at same time
  => typically combined with pipelined CPU design
- This is **not** SIMD!
- How to program for this:
  - write simple code
  - no data dependencies
  - avoid branches
  - compiler optimization



Superscalar & Pipelined CPU Execution

```
z = a * b + c * d;
```

Actual steps:

```
z1 = a * b;
```

```
z2 = c * d;
```

Start data load for next command

```
z= z1 + z2;
```

1. Load **a** into register **R0**
   and load **b** into **R1**
2. Multiply **R2** = **R0** * **R1**
   and load **c** into **R3**
   and load **d** into **R4**
3. Multiply **R5** = **R3** * **R4**
4. Add **R6** = **R2** + **R5**
5. Store **R6** into **z**

Superscalar pipeline savings:
3 out of 8 steps, plus 3 if next operation independent

Hyper-threading (HT) or Simultaneous Multi-threading (SMT)

- 2 or 4 sets of registers share functional units
  => integrates well into superscalar CPU design
- Operating system “sees” two processors
  => use multi-thread or MPI-parallelization
  => concurrent tasks are independent
- Performance gain very application dependent
  - overhead for scheduling twice as many cores
  - independent data access => cache trashing
  - HT/SMT tasks contend for the network with

  little performance gain (20%), often turned off

Vectorized Loop

```
for (i = 0; i < length; i++) {
  z[i] = a[i] * b[i] + c[i] * d[i];
}
```

Vector registers on a CPU can hold multiple numbers and load, store or process them in parallel (**SIMD**):

```
for (i = 0; i < length; i +=2) {
 z[i] = a[i]  *b[i]   + c[i]  *d[i];
 z[i+1]=a[i+1]*b[i+1] + c[i+1]*d[i+1];
}
```

Executed together

This is **in addition** to superscalar pipelining and with using special vector instructions (SSE,AVX,etc.)

How To Write Vector Code

- Modern compilers can automatically vectorize:
  => only works for very simple cases
  => many constraints for fast data access
- Use intrinsic vector instructions of compiler:
  => easier than writing assembly code directly
  => can be quite complicated, not-very portable
- Use optimized libraries
  => libraries for linear algebra and similar often include vector optimized variants
- Larger gain for float vs. double & AVX vs. SSE

A High-Performance Problem

Program-Level Parallelism

Functional parallelism: different people perform **different** tasks at the same time. Tasks typically have **different duration**.

Data parallelism: different people perform the **same** task, but on different, equivalent, and independent objects. Tasks have the **same duration**.

Load Imbalance and Synchronization

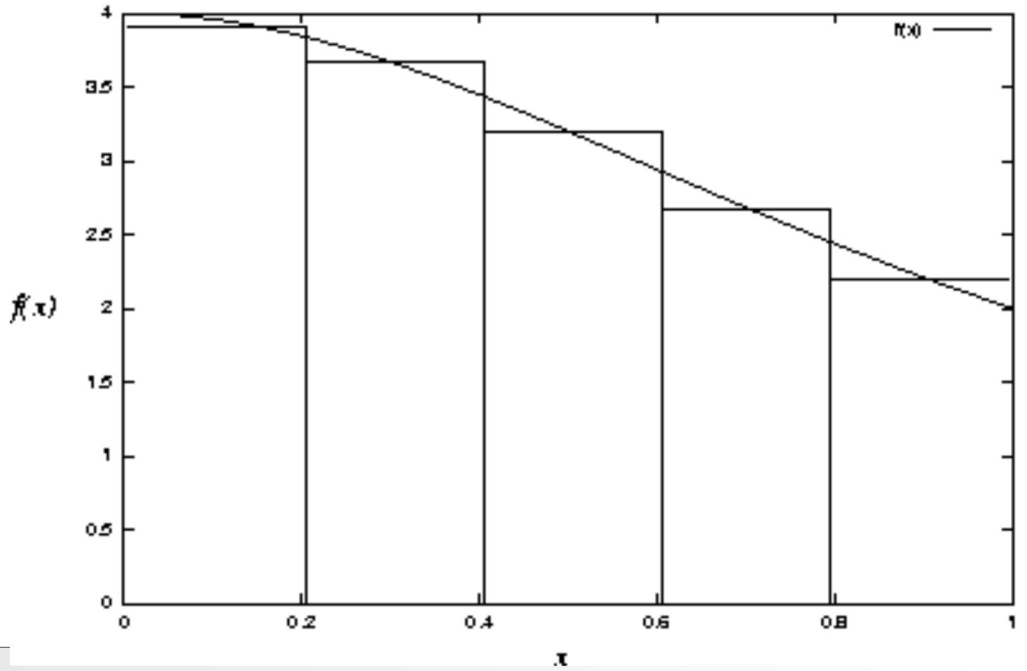
- Load Imbalance:
  Different tasks take a different amount of time. Workers that have done one “fast” task could perform a second
- Synchronization:
  The “lollipop man” makes certain, the driver does not try to continue before all parallel tasks are done

Computing Pi in Parallel (1)

- Algorithm:

$$\int_0^1 \frac{4}{1+x^2}\,dx = 4\left(\arctan(1)-\arctan(0)\right) = \pi$$

- Integrate, i.e determine the area under function numerically using slices of *h * f(x)* at midpoints. Smaller slices lead to more accurate result. Computing groups of slices in parallel + sum.



Computing Pi in Parallel (2)

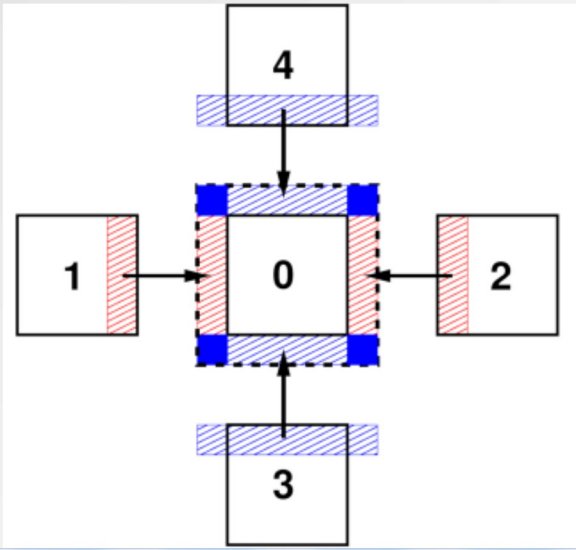
- This is a data parallel problem:
  - Each parallel process computes the area for different slices of the numerical integration
  - Data does not need to be communicated since each parallel task can infer from the loop index which slices it needs to compute and sum up
  - The final result is a sum over the per-parallel task sums. This step is called a Reduction
  - Reductions can be computed for sums, products, min, max, but not differences and divisions (why?)
- Accuracy limited by # of slices and floating-point math

Computing Pi in Parallel (3)

- Ideal problem for efficient parallelization
- Data parallel, but no communication needed to distribute or replicate the data to all processes. Instead the data can be quickly generated as needed by the parallel processes
- Good load balance because the amount of work to compute the area of each slice is same
- Only communication is reduction for a single item of data, so communication overhead is low
- Cache efficient, very little data needed

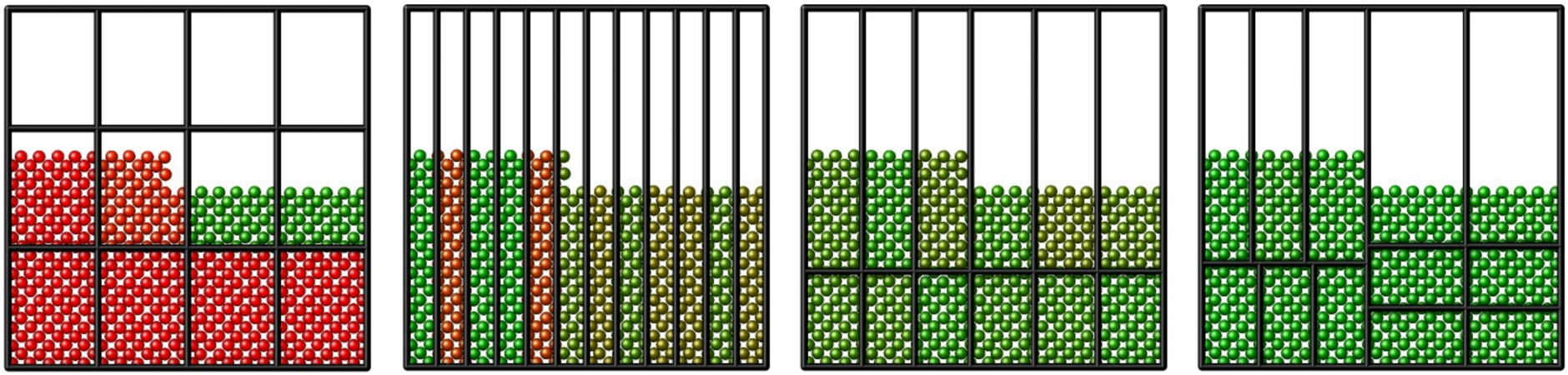
Distributed versus Replicated Data (1)

- Replicated data is easier to implement:
  - Broadcast data and then distribute loop over data across parallel processes → minimal code changes
  - Bad strong and weak scaling → more communication
- Distributed data can scale to more processes…
  - Data set gets smaller with more processors → better cache efficiency
  - Update “halo” of data from neighboring domains every step, can be done in parallel



Distributed versus Replicated Data (2)

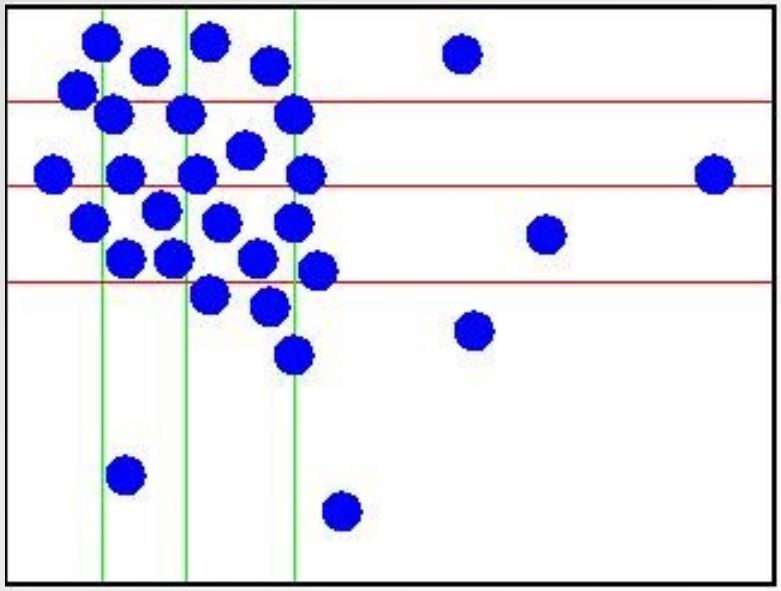
- Distributed data based on domain decomposition may result in load imbalance for sparse systems



- Changing the domain decomposition can reduce how much load imbalance impacts overall performance: 2d-decomposition, 1d-decompositions, 2d-with shifted domain boundaries, recursive bi-sectioning.

Limitations of Parallel Computing

- Fraction of serial code limits parallel speedup
- Degree to which tasks/data can be subdivided is limit to concurrency and parallel execution
- Load imbalance:
  - parallel tasks have a different amount of work
  - CPUs are partially idle
  - redistributing work helps but has limitations:

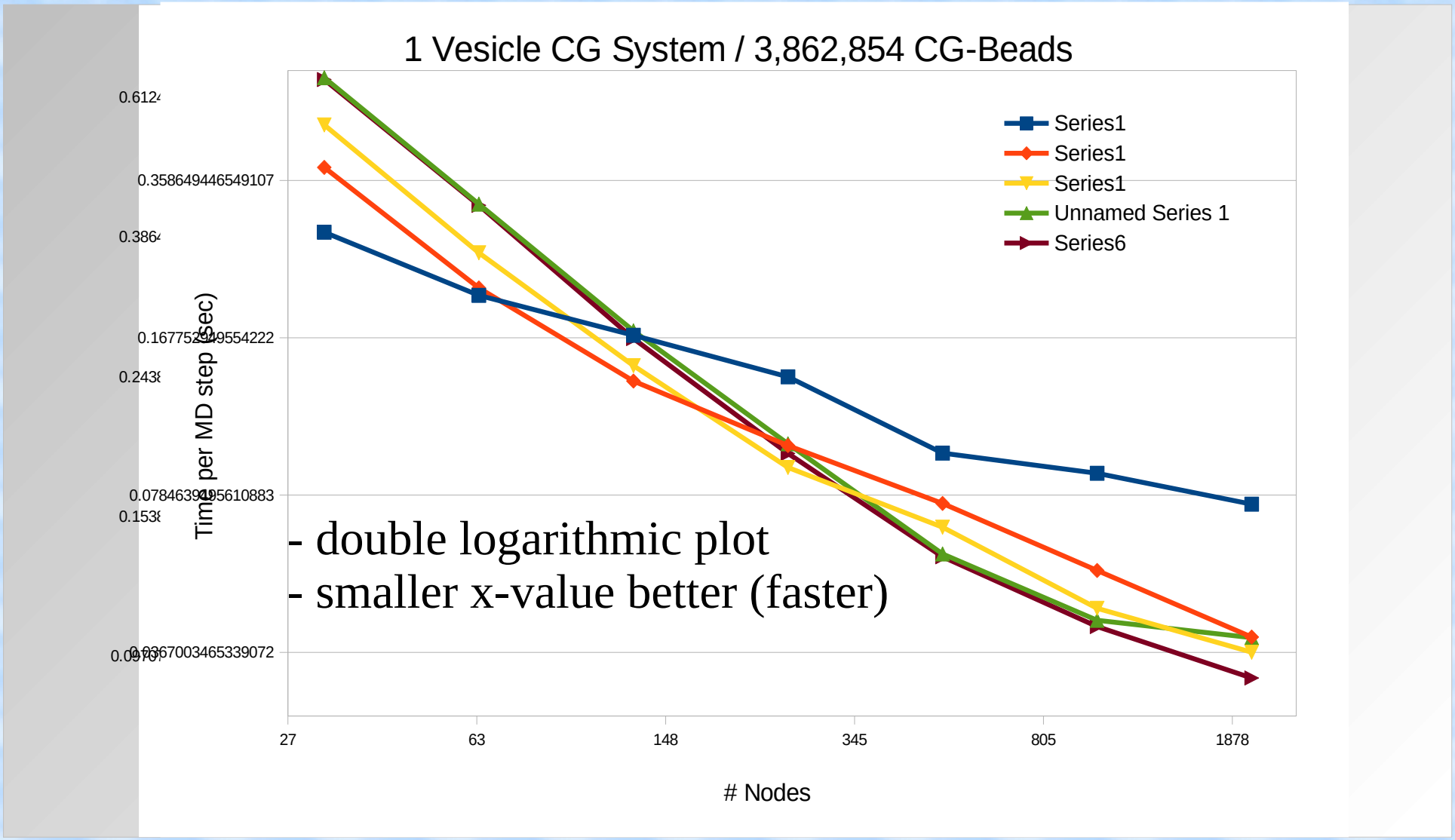


- Communication and synchronization overhead:

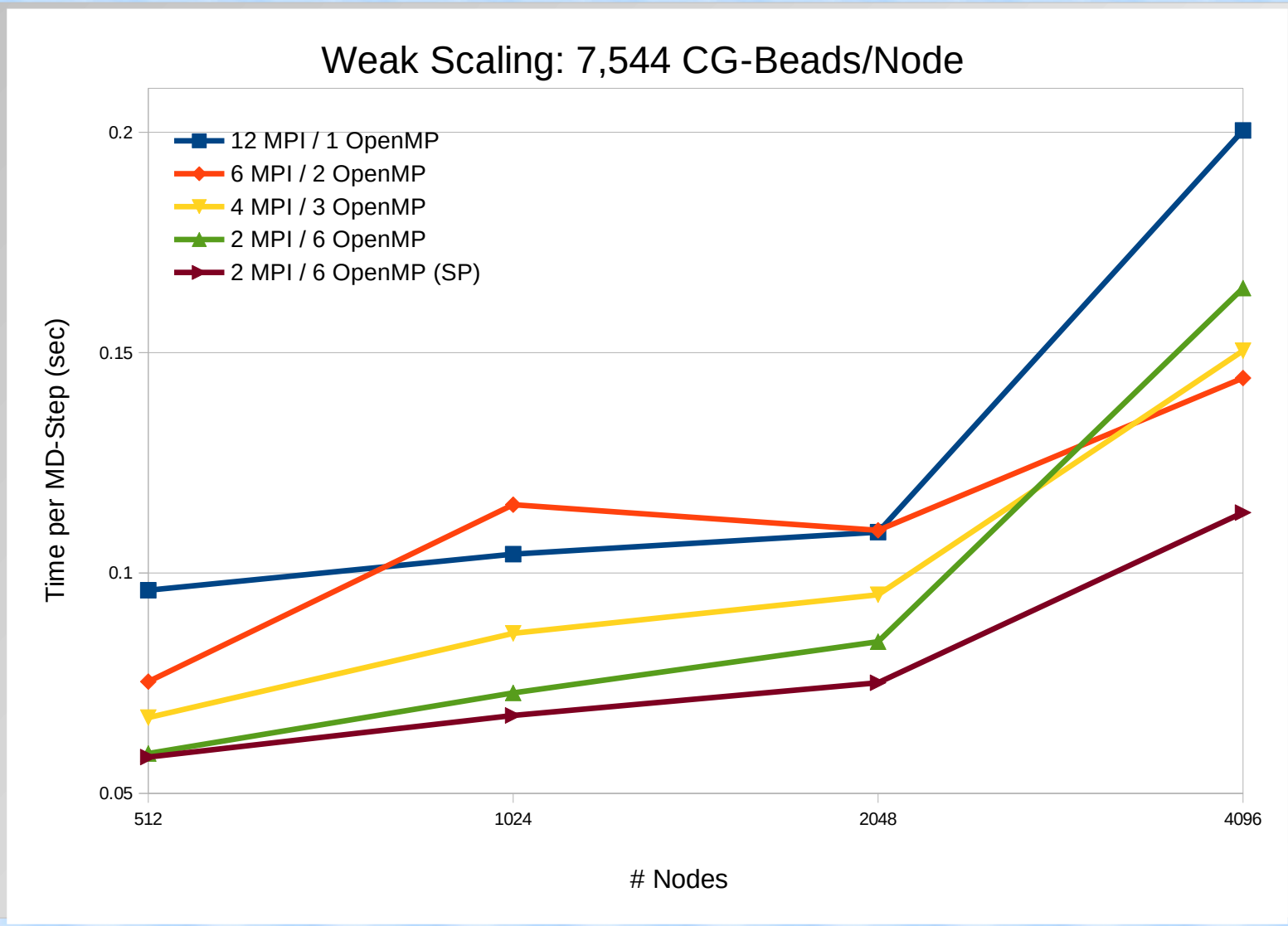
Performance of SC Applications

- Strong scaling: fixed data/problem set; measure speedup with more processors
- Weak scaling: data/problem set increases with more processors; measure if speed is same
- Linpack benchmark: weak scaling test, more efficient with more memory => 50-90% peak
- Climate modeling (WRF): strong scaling test, work distribution limited, load balancing, serial overhead => < 5% peak (similar for MD)

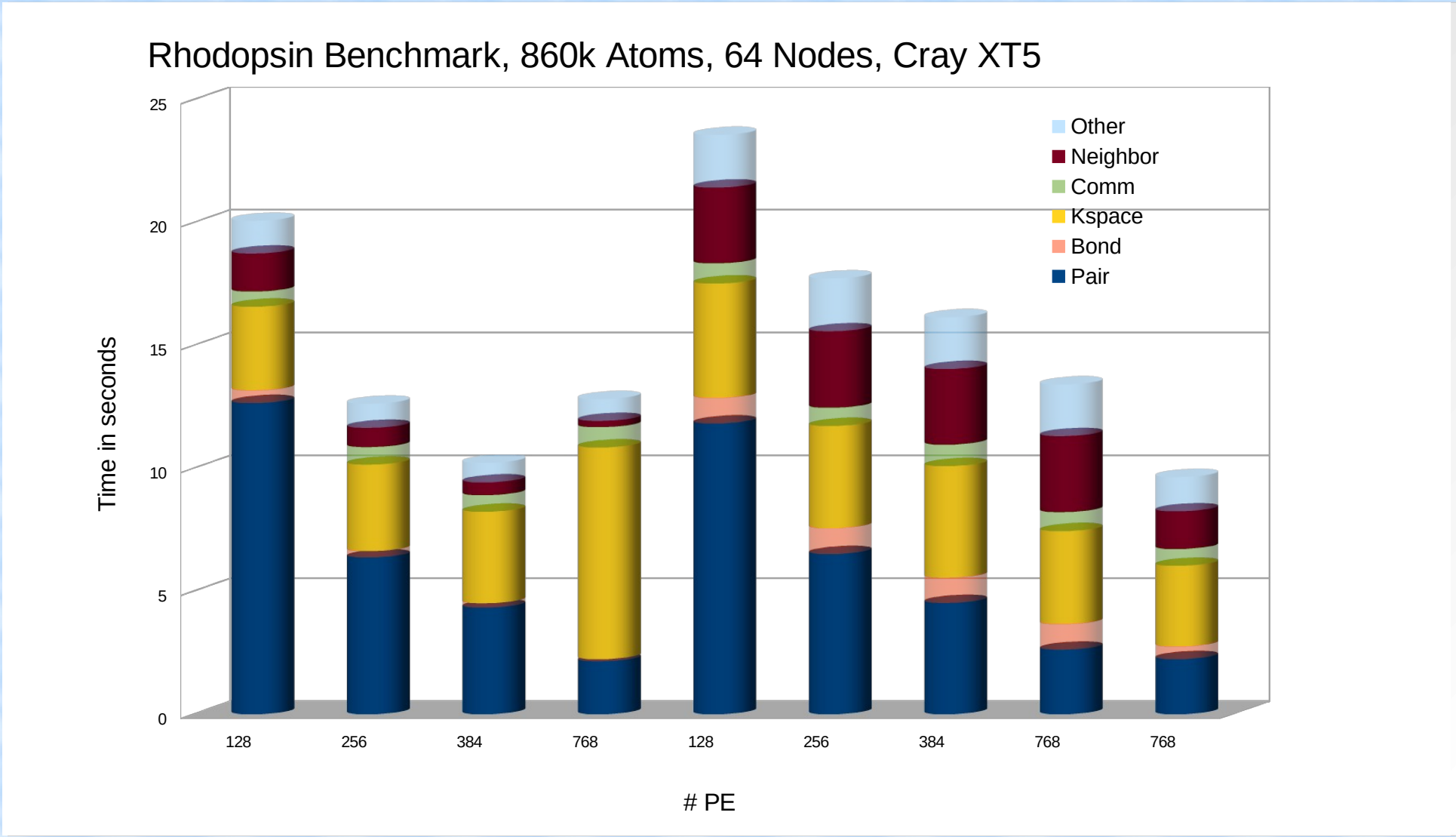
Strong Scaling Graph



Weak Scaling Graph

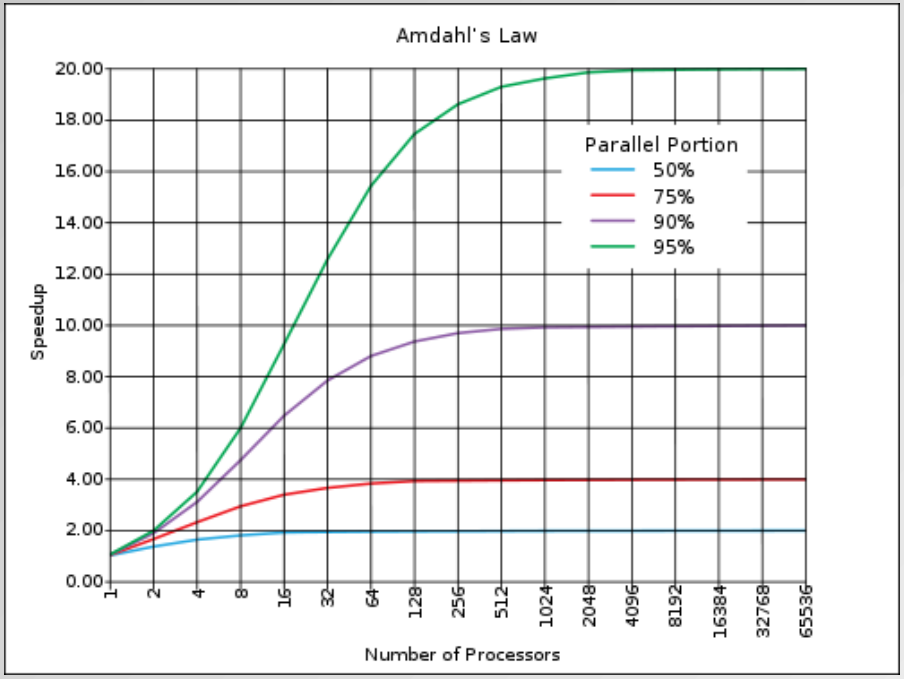


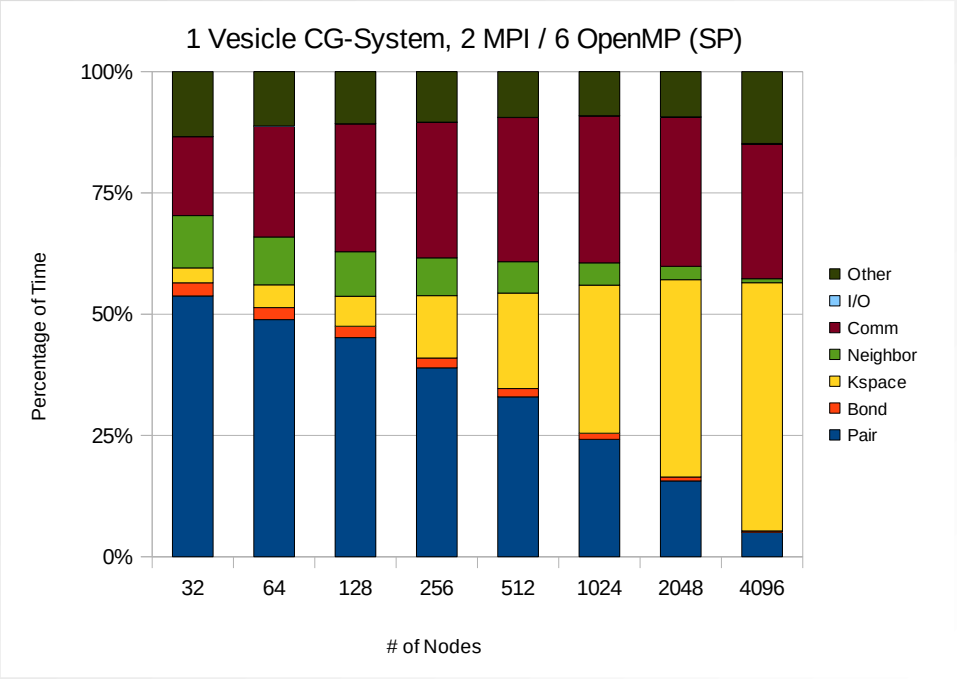
Performance within an Application



Amdahl's Law vs. Real Life

- The speedup of a parallel program is limited by the sequential fraction of the program.
- This assumes perfect scaling and no overhead





MPI Program Design

- Multiple and separate processes (can be local and remote) concurrently that are coordinated and exchange data through “messages”
  => a “share nothing” parallelization
- Best for coarse grained parallelization
- Distribute large data sets; replicate small data
- Minimize communication or overlap communication and computing for efficiency
  => Amdahl's law: speedup is limited by the fraction of serial code plus communication

What is MPI?

- A standard, i.e. there is a document describing how the API (= constants & subroutines) are named and should behave; multiple “levels”, MPI-1 (basic), MPI-2 (advanced), MPI-3 (new)
- A library (or API) to hide the details of low-level communication hardware and how to use it
- An implementation of the standard
  - Open source and commercial versions
  - Vendor specific versions for certain hardware
  - **Not binary compatible** between implementations

Goals of MPI

- Allow to write software (source code) that is portable to many different parallel hardware. i.e. agnostic to actual realization in hardware
- Provide flexibility for vendors to optimize the MPI functions for their hardware
- No limitation to a specific kind of hardware and low-level communication type. Running on heterogeneous hardware is possible.
- Fortran77 and C style API as standard interface Fortran90 deprecated, Fortran2008 module

Phases of an MPI Program

- 1) Startup
  - Parse arguments (mpirun may add some!)
  - Identify parallel environment and rank of process
  - Read and distribute all data
- 2) Execution
  - Proceed to subroutine with parallel work
    (can be same of different for all parallel tasks)
- 3) Cleanup
- CAUTION: this sequence may be run only once

MPI in C versus MPI in Fortran

- The programming interface (“bindings”) of MPI in C and Fortran77 are closely related (wrappers for many other languages exist)
- MPI in C (usually used in C++ as well):
  - Use '#include <mpi.h>' for constants and prototypes
  - Include only once at the beginning of a file
- MPI in Fortran77 (usually used in Fortran 90+):
  - Use 'include “mpif.h”' for constants
  - Include at the beginning of each module
  - All MPI functions are “subroutines” with the same name and same order and type of arguments as in C with return status added as the last argument

How much MPI do we need?

A fully functional MPI program can be written by using only 6 MPI functions:

- MPI_Init()
- MPI_Comm_size()
- MPI_Comm_rank()
- MPI_Bcast() *or* MPI_Send()
- MPI_Reduce() *or* MPI_Recv()
- MPI_Finalize()

Most parallel programs contain only a few more