



Introduction to High Performance Computing

BGRS International Summer School

Prof. Dr. Thomas Ludwig
Universität Heidelberg, Germany

Email: t.ludwig@computer.org

Web: pvs.informatik.uni-heidelberg.de



Tutorial Outline

- Architecture of High Performance Computers
Slides 3-37
- Parallel Programming Principles
Slides 38-71
- Message Passing with MPI
Slides 72-103
- Advanced Issues with Message Passing
Slides 104-128



Architecture

Architecture of High Performance Computers

- Introduction
- Application Fields
- Architectural Concepts
- Technical Aspects
- The TOP500 List
- Selected Systems

Why High Performance Computing?

- Situation in science and engineering
 - replace complicated physical experiments by computer simulations
 - evaluate more fine-grained models
- User requirements
 - compute masses of individual tasks
 - compute complicated single tasks
- Available computational power
 - single workstation is not sufficient



What is it?

- High Performance Computing (HPC), Networking and Storage
 - deals with high and highest performance computers, with high speed networks, and powerful disk and tape storage systems
- Performance improvement
 - compared to personal computers and small workstations:
Factor 100...100.000



What do I Need?

- Small scale high performance computing
 - cheapest version: use what you have
workstations with disks and network
 - a bit more expensive: buy PCs
 - e.g. 16 personal computers with disks and gigabit ethernet
 - it's mainly a human resources problem
 - network of workstations is time consuming to maintain
 - software comes for free (Linux + libraries)



What do I Need? ...

- Large scale high performance computing
 - buy 10.000 PCs or a dedicated supercomputer
 - buy special hardware for networking and storage
 - build a special building
 - build an electric power station



How Much do I Have to Pay?

- Small scale (<64 nodes)
 - 1000€/node
- Medium scale (64-1024 nodes)
 - 2000€/node (multiprocessor, 64-bit)
 - 1500€/node for high speed network
 - 500€/node for high performance I/O
- Large scale (>1024 nodes)
 - money for building
 - money for power plant
 - current costs range between 20...400 million Euros



Application Fields

- Numerical calculations and simulations
 - particle physics
 - computational fluid dynamics
 - car crash simulations
 - weather forecast
 - financial calculations
 - ...
- Non-numerical computations
 - chess playing, theorem proving
 - commercial database applications



Application Fields...

- **All fields of Bioinformatics**

- computational genomics
- computational proteomics
- computational evolutionary biology
- ...

- In general

- everything that runs between 1 and 100.000 days
- everything that uses high volumes of data



Measures

Mega ($2^{20} \cong 10^6$) – Giga ($2^{30} \cong 10^9$) – Tera ($2^{40} \cong 10^{12}$)
Peta ($2^{50} \cong 10^{15}$) – Exa ($2^{60} \cong 10^{18}$)

■ Computational Performance (Flop/s)

Flop/s = floating point operations per second

- modern single processor: 5 GFlop/s
- Nr. 1 supercomputer: 1.026 TFlop/s (factor 200.000)

■ Network performance (bit/s)

- personal computer: 100/1000 Mbit/s
- supercomputer networks: 1-10 gigabit/s



Measures...

- Main memory (Byte)
 - personal computer: 2 GByte
 - best supercomputers: dozens of TByte
- Disk space (Byte)
 - single disk 2006: 300 GByte
 - best supercomputers: some PByte
- Tape storage (Byte)
 - personal computer: 400 GByte
 - best supercomputers: many PByte



Levels of Parallelism

- Parallel computer architectures
 - have units that work in parallel and in a well coordinated way to solve a problem
- Units
 - specialized components like e.g. processor pipelines
 - arithmetical/logical units in the processors
 - processors / co-processors
 - computers
 - parallel computers and/or cluster computers

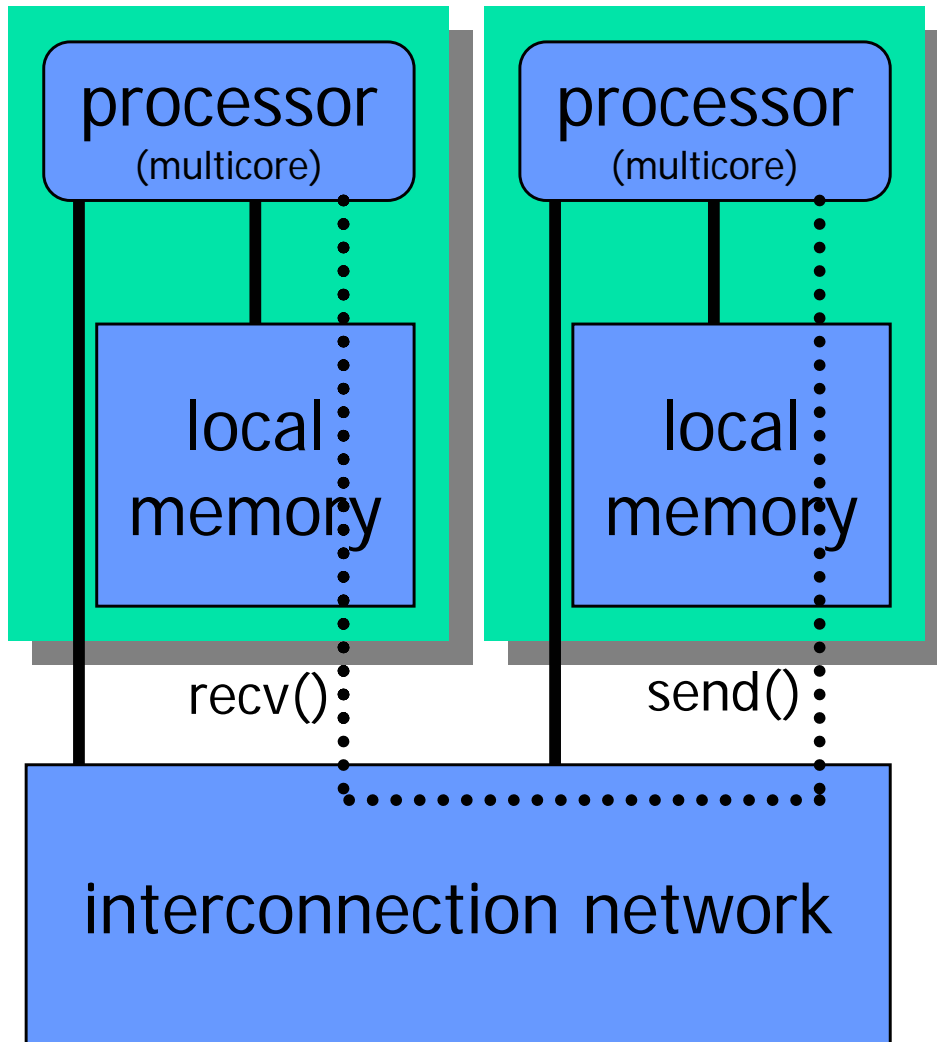


Architectural Concepts

- Basic classification concept:
How is the main memory organized?
 - distributed memory architecture
 - shared memory architecture

- Available systems
 - dedicated supercomputers
 - cluster systems

Distributed Memory Architecture



- autonomous computers connected via network
- processors have access to local memory only
- parallel program spawns processes over a set of processors
- called: multi computer system



Distributed Memory Architecture...

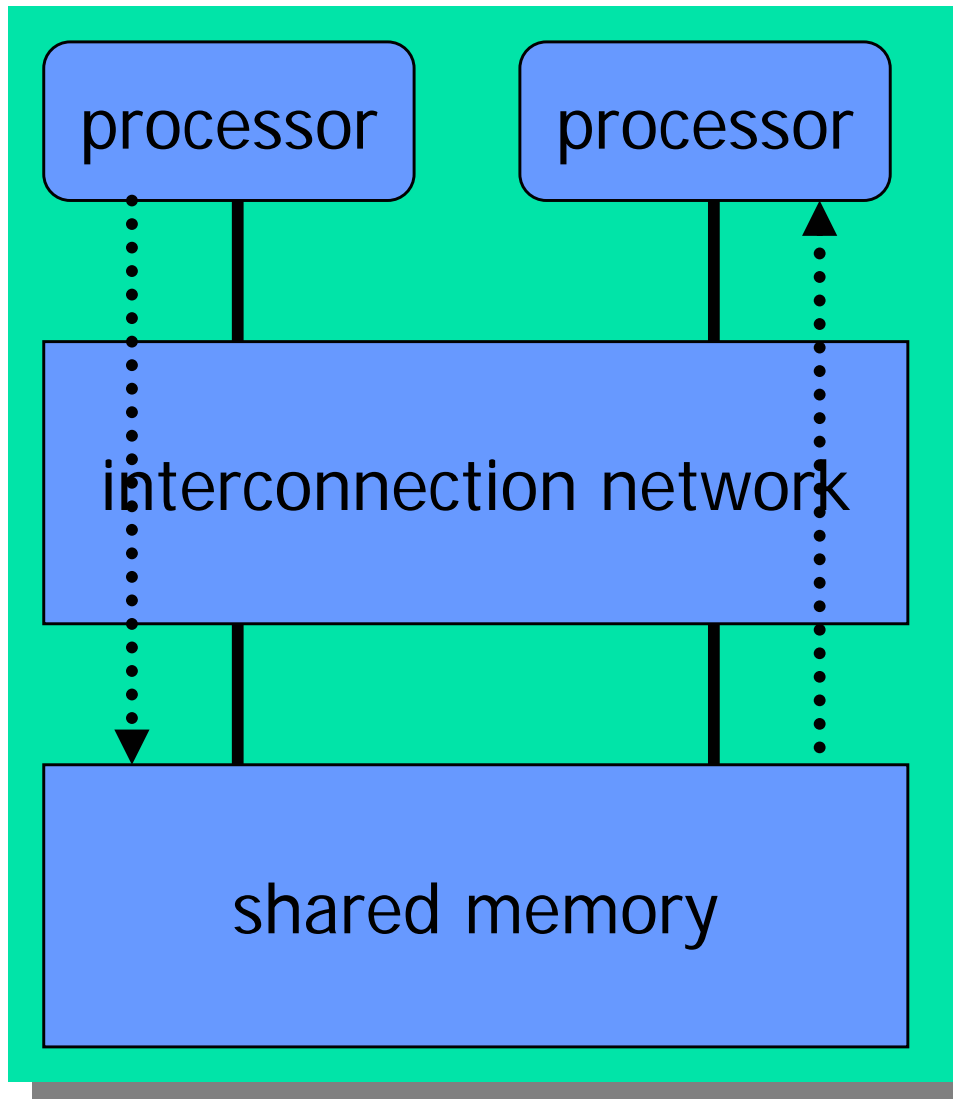
■ Advantages

- good scalability: just buy new nodes
 - Concept scales up to 100.000+ nodes
- you can use what you already have
- extend the system when you have money and need more power

■ Disadvantages

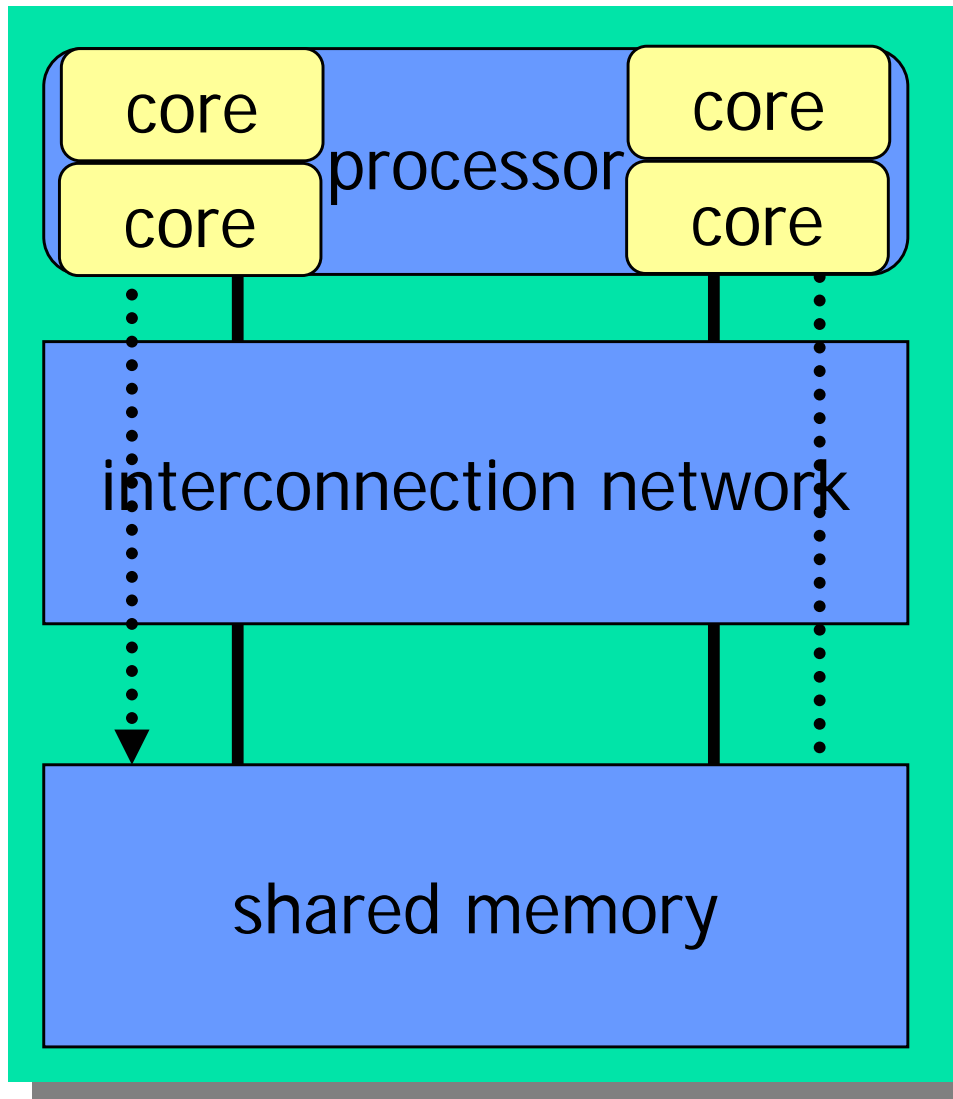
- complicated programming: parallelization of formerly sequential programs
(including complicated debugging, performance tuning, load blancing, etc.)

Shared Memory Architecture



- several processors in one box (e.g. multiprocessor mother-board)
- each processor has access to the complete address space
- called: multiprocessor system, symmetric multiprocessing system (SMP)
- now: multicore processors as standard

Shared Memory Architecture



- several processors in one box (e.g. multiprocessor mother-board)
- each processor has access to the complete address space
- called: multiprocessor system, symmetric multiprocessing system (SMP)
- now: multicore processors as standard



Shared Memory Architecture...

- Advantages
 - much easier programming
- Disadvantages
 - limited scalability: up to 64 processors (cores)
reason: interconnection network becomes bottleneck
(with multicore still unknown scalability)
 - limited extensibility
 - very expensive due to high performance interconnection network



Hybrid Architectures

- Use several SMP/multicore systems
 - combination of shared memory systems and distributed memory system
- The good thing: scalable performance according to your financial budget
- The bad thing: programming gets even more complicated (hybrid programming)
- The reality: vendors like to sell these systems, because they are easier to build



Supercomputers vs. Clusters

- Supercomputers (Distributed/shared memory)
 - constructed by a major vendor (IBM, HP, ...)
 - uses some proprietary components (processor, network, ...)
 - customized (Linux-like) operating systems
- Clusters (Network of workstations, NOWs)
 - assembled by vendor or users
 - commodity-off-the-shelf components (COTS)
 - Linux operating system



Supercomputers vs. Clusters...

- Supercomputers

- very expensive to buy
- usually high availability and scalability

- Clusters

- factor 10 cheaper to buy, but:
- very expensive to own
- lower overall availability and scalability



Scalability

„Scalability“ not a well-defined term, however frequently used with high performance computing

What it means: Extendability of the system with preservation of certain positive quality characteristics

- e.g. a program scales well when it produces high performance even with a high number of parallel processes
- e.g. a network scales well when we get double the performance for double the money that we invest



Interconnection Networks

- Most simple case
 - with shared memory: bus system
 - with distributed memory: ring or star topology
- Complex cases
 - many variations, however no full mesh

Problems

- latency, bandwidth
- network load, paket collisions



Background Storage

- Local disk at compute node
 - usually only for temporary data
- File server in the network
 - persistent file storage
 - file access is performance bottleneck
- Storage Area Network (SAN)
 - storage components in separated network connected to the cluster
- Tape systems



Operating Systems

- Cluster operating systems
 - variations of UNIX, mostly Linux
 - most of the systems
 - MS-Windows: now also product for clusters
 - almost no system with Windows yet
- Single system image
 - user sees only „one“ system
 - localization of services is hidden from user
 - e.g. MOSIX (not for high performance computing)



The TOP500-List

- Lists the world's 500 most powerful systems
www.top500.org
- Updates in June and November
- Ranking based on numerical algorithm
 - LINPACK-benchmark
 - R_{\max} is measured performance in GFlop/s
 - R_{peak} is theoretical maximum in GFlop/s
- In 6 months almost half of the systems fall off the list
- The majority of systems now are clusters

Rank	Site	Computer/Year Vendor	Cores	R _{max}	R _{peak}	Power
1	DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz , Voltaire Infiniband / 2008 IBM	122400	1026.00	1375.78	2345.50
2	DOE/NNSA/LLNL United States	BlueGene/L - eServer Blue Gene Solution / 2007 IBM	212992	478.20	596.38	2329.60
3	Argonne National Laboratory United States	Blue Gene/P Solution / 2007 IBM	163840	450.30	557.06	1260.00
4	Texas Advanced Computing Center/Univ. of Texas United States	Ranger - SunBlade x6420, Opteron Quad 2Ghz, Infiniband / 2008 Sun Microsystems	62976	326.00	503.81	2000.00
5	DOE/Oak Ridge National Laboratory United States	Jaguar - Cray XT4 QuadCore 2.1 GHz / 2008 Cray Inc.	30976	205.00	260.20	1580.71
6	Forschungszentrum Juelich (FZJ) Germany	JUGENE - Blue Gene/P Solution / 2007 IBM	65536	180.00	222.82	504.00
7	New Mexico Computing Applications Center (NMCAC) United States	Encanto - SGI Altix ICE 8200, Xeon quad core 3.0 GHz / 2007 SGI	14336	133.20	172.03	861.63
8	Computational Research Laboratories, TATA SONS India	EKA - Cluster Platform 3000 BL460c, Xeon 53xx 3GHz, Infiniband / 2008 Hewlett-Packard	14384	132.80	172.61	786.00
9	IDRIS France	Blue Gene/P Solution / 2008 IBM	40960	112.50	139.26	315.00
10	Total Exploration Production France	SGI Altix ICE 8200EX, Xeon quad core 3.0 GHz / 2008 SGI	10240	106.10	122.88	442.00

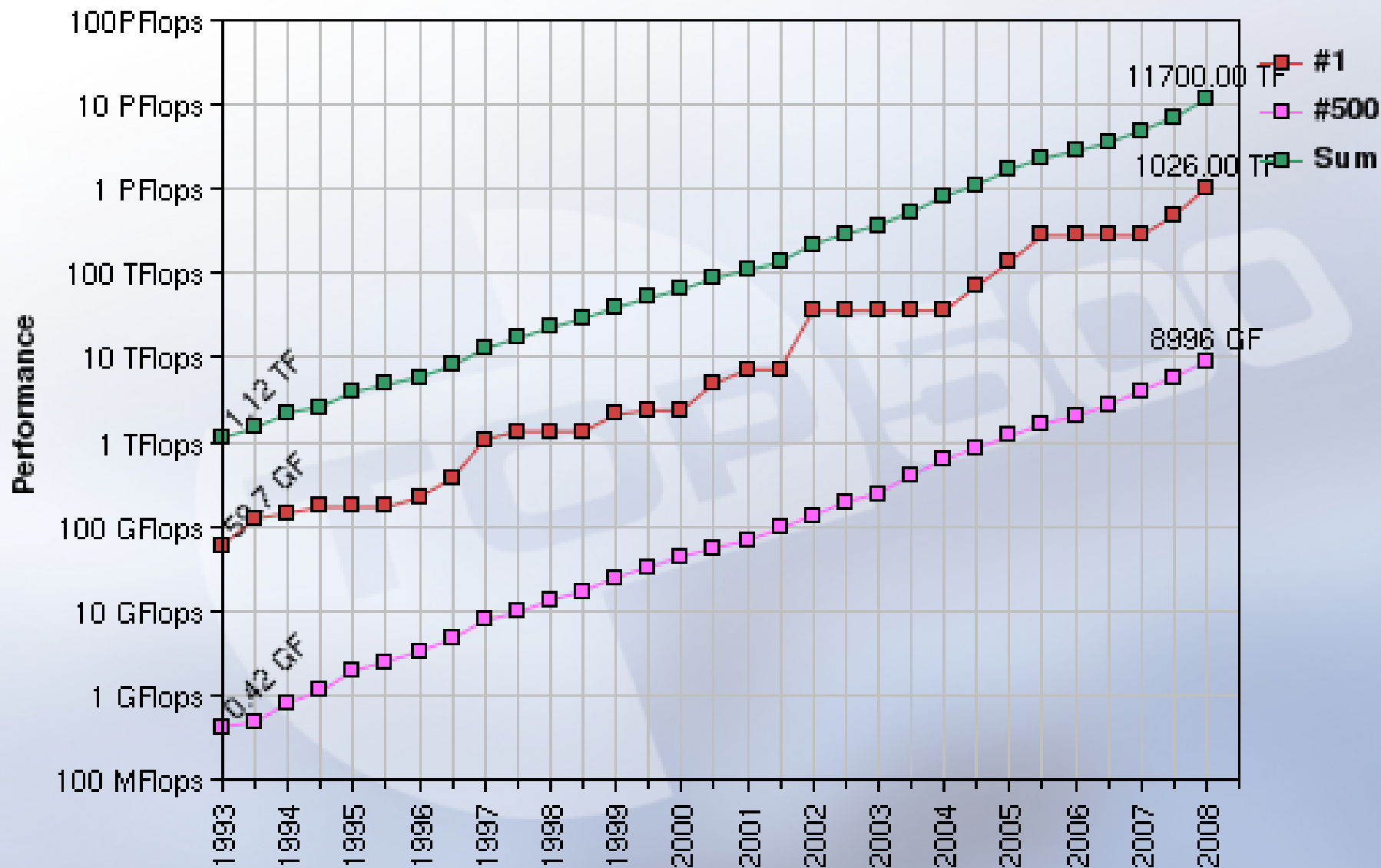
Russia in the TOP500 in June 2006

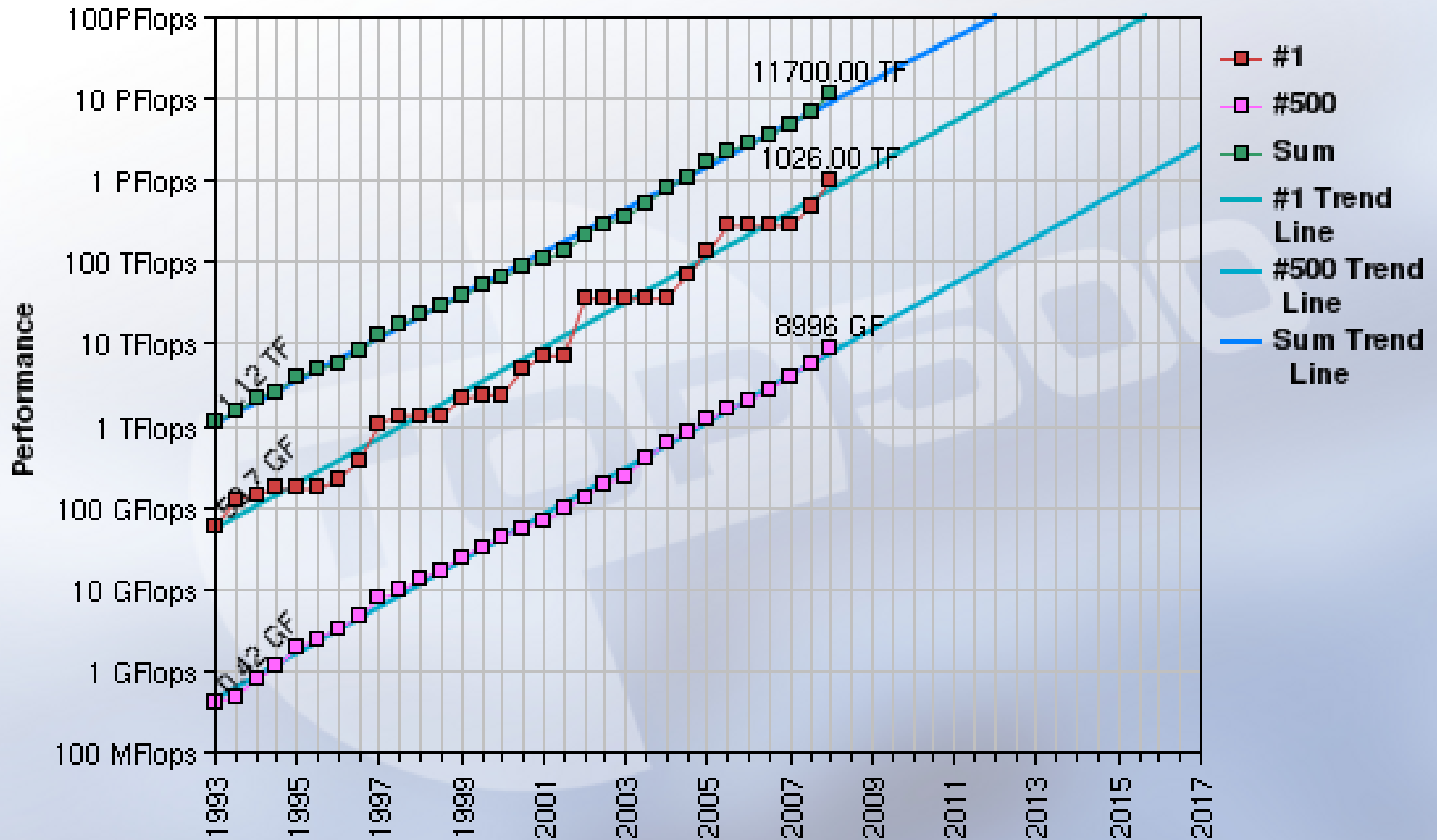
A good start! 😊

Rank	Site	System	Processors	R _{max}	R _{peak}
70	<u>Joint Supercomputer Center Russia</u>	<u>MVS-15000BM, eServer BladeCenter JS20 (PowerPC970 2.2 GHz), Myrinet IBM</u>	1148	6645.53	10102.4

Russia in the TOP500 in June 2008

Rank	Site	System	Cores	R _{max}	R _{peak}
36	Moscow State University - Scientific & Research Computing Center Russia	T-Platforms T60, Intel Quadcore 3Mhz, Infiniband DDR T-Platforms	5000	47.17	60
56	Joint Supercomputer Center Russia	Cluster Platform 3000 BL460c, Xeon 53xx 3GHz, Infiniband Hewlett-Packard	3760	33.89	45.12
169	Ufa State Aviation Technical University Russia	BladeCenter HS21 Cluster, Xeon quad core 2.33 GHz, Infiniband IBM	2128	15.33	19.86
227	Roshydromet Russia	SGI Altix ICE 8200, Xeon quad core 2.83 GHz SGI	1416	13.68	16.03
252	Siberian National University Russia	BladeCenter HS21 Cluster, Xeon quad core 2.33 GHz, Infiniband IBM	1808	13.06	16.87
282	South Ural State University Russia	T16 Cluster, Intel E54xx 3Mhz, Infiniband DDR T-Platforms	1328	12.2	15.94
362	Roshydromet Russia	Altix 4700 1.66 GHz SGI	1664	10.29	11.05
372	IT Service Provider Russia	Cluster Platform 3000 BL460c, Xeon 53xx 1.86GHz, GigEthernet Hewlett-Packard	2440	10.01	18.15
484	Tomsk State University Russia	Xeon Cluster, QLogic InfiniPath T-Platforms	1128	9.01	12







NEC's Earth Simulator

- 640 nodes
- 8 vector processors each
- 5120 processors in total
- 0.15micron copper
- 10TByte main memory
- 700TByte disks
- 1.6PByte tapes
- 83.000 copper cables
- 2.800km / 220t of cabeling
- 3250m² foot print
- earth quake protected
- 7MW consumption
- 200MioUSD computer
- 200MioUSD building and power station
- application: climat research

Computenic-Shock

カートリッジテープ
ライブラリシステム

磁気ディスク

地球シミュレータ本体

結合ネットワーク部

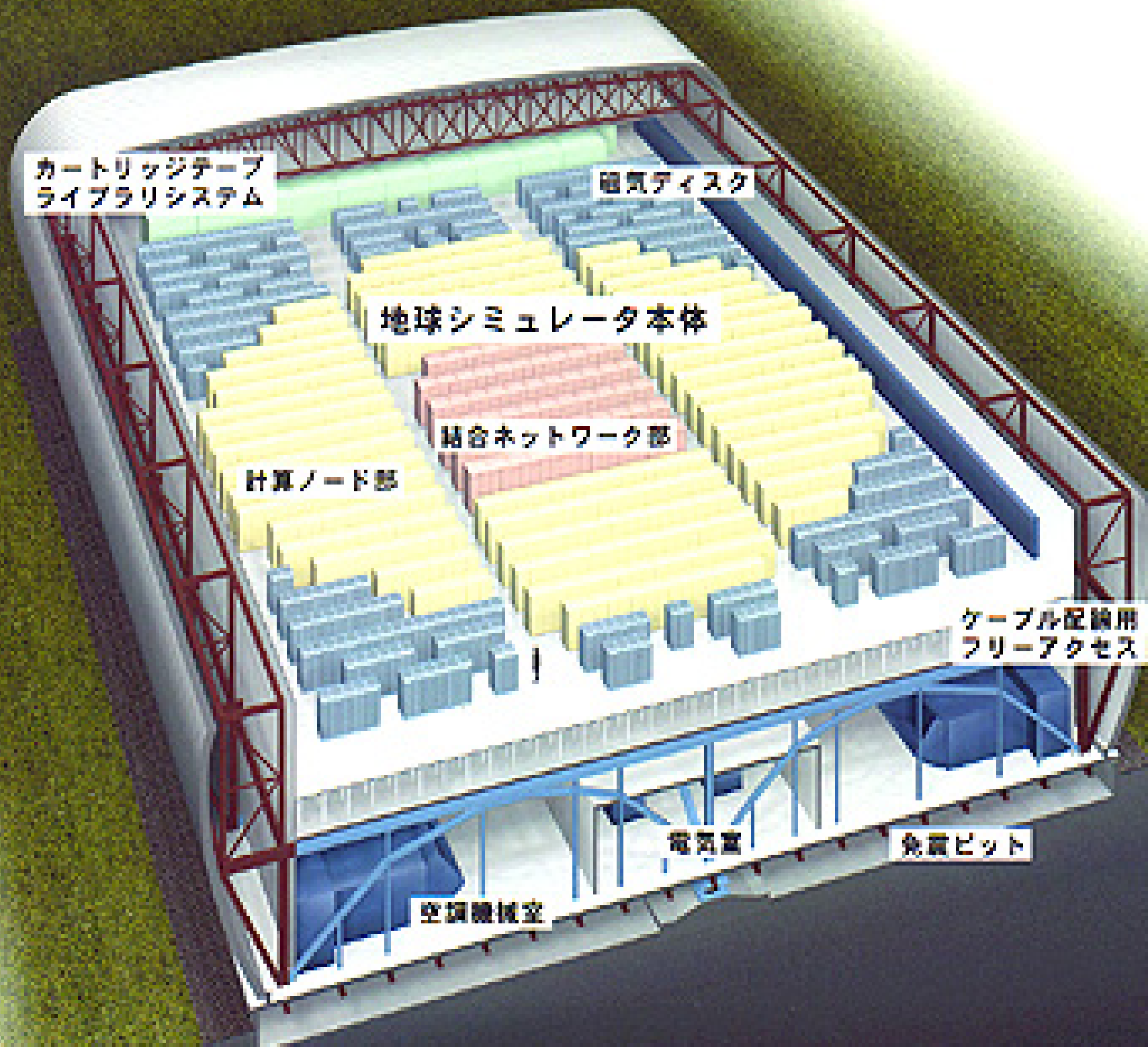
計算ノード部

ケーブル配線用
フリーアクセス

空調機械室

電気室

負荷ビット

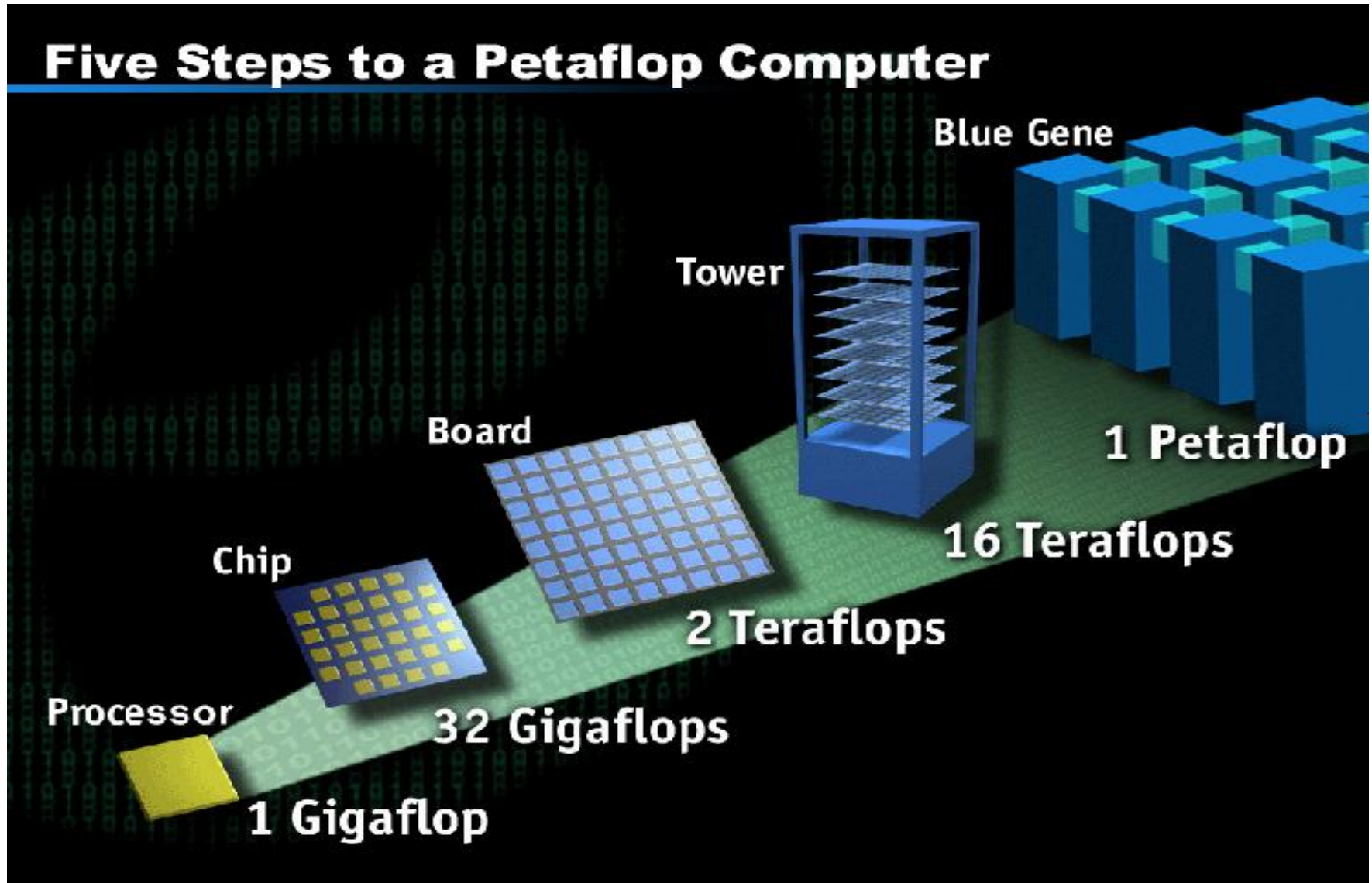




IBM's Blue Gene-Program

- Main application: protein folding
- Dense packing of chips
- Systems:
 - Blue Gene/L: 180TFlops/s, by end of 2004
 - by end of 2005 doubles to 370TFlops/s
 - Blue Gene/P: 1PFlop/s, 2007
 - Blue Gene/Q: 3PFlop/s, 2008

BlueGene Systems





Summary

- HPC ranges from factor 100 to 200.000 compared to single PC
- Application fields: everything that is computationally intensive, in particular also bioinformatics
- Main architectural concept: memory organization
- Distributed memory: scales well, is difficult to program
- Shared memory: does not scale but is easy to program
- Most wide spread concept: cluster architecture
- Nr. 1 supercomputer has now 1026 TFlop/s



Programming

Parallel Programming Principles

- What is parallelization?
- Paradigms of parallel programming
- Tools for parallelization
- Algorithmic aspects
- Examples



What is Parallelization?

Task

- Find implicit parallelization in the algorithms and make it explicit
- Means: distribute program and data onto the resources of the system
- Who? Programmer and/or compiler



What is Parallelization?...

- Distribution generates new load (*overhead*)
 - overhead is minimal if no distribution at all
- Distribution uses resources optimally
 - maximum performance if fully distributed

Objective

use all resources and minimize overhead



Requirements

- In addition to sequential software
 - partitioning of the program into small tasks
 - add coordination and communication
 - map these parts onto the components of the computer

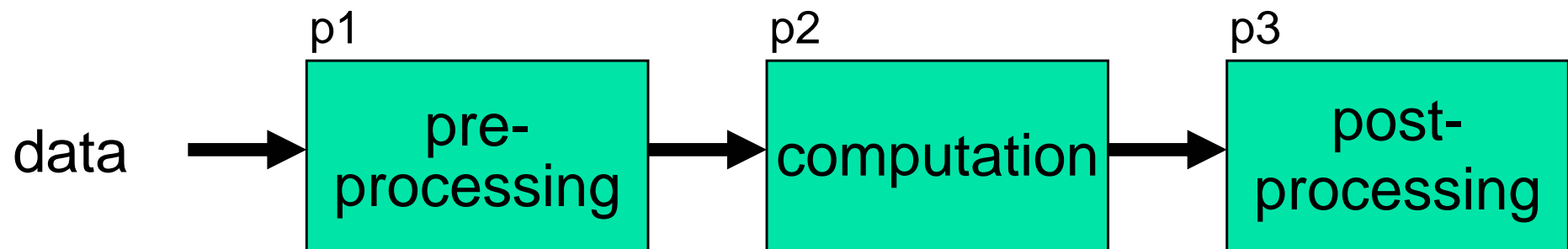
- Problems
 - debugging (new types of errors)
 - performance analysis (program slow-down)
 - load balancing (for maximum performance)

Parallelization Paradigms

Code-partitioning (also: macro-pipelining)

distribute program code onto the nodes

- different code on each node
- data varies according to flow of computation
- coordinator: first/last process





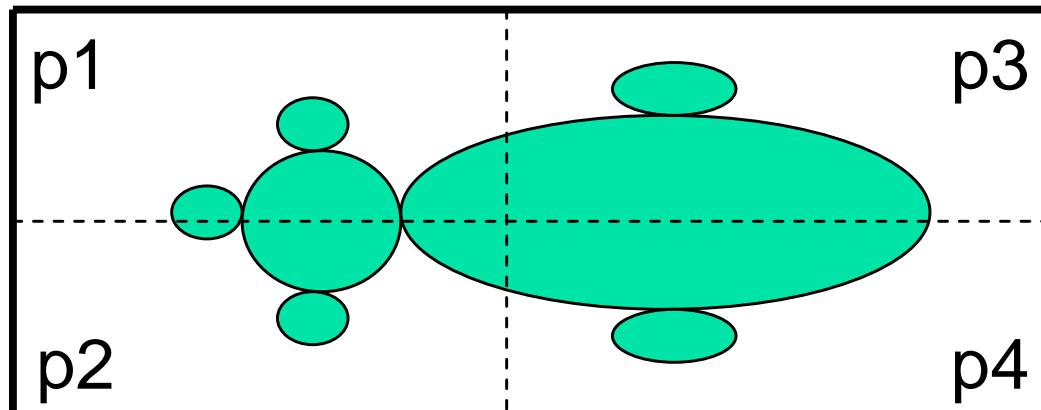
Parallelization Paradigms...

- **Advantage** of code-partitioning
 - sometimes easy to design
 - sometimes appropriate algorithms (e.g. FFT)
- **Disadvantage** of code-partitioning
 - multiple source code files
 - difficult to adapt to target machine
 - complex communication schemes
 - complicated debugging
 - complicated load balancing

Parallelization Paradigms...

Data-partitioning

- distribute data over the nodes
 - identical code on each node
 - data varies from node to node
 - selected process coordinates





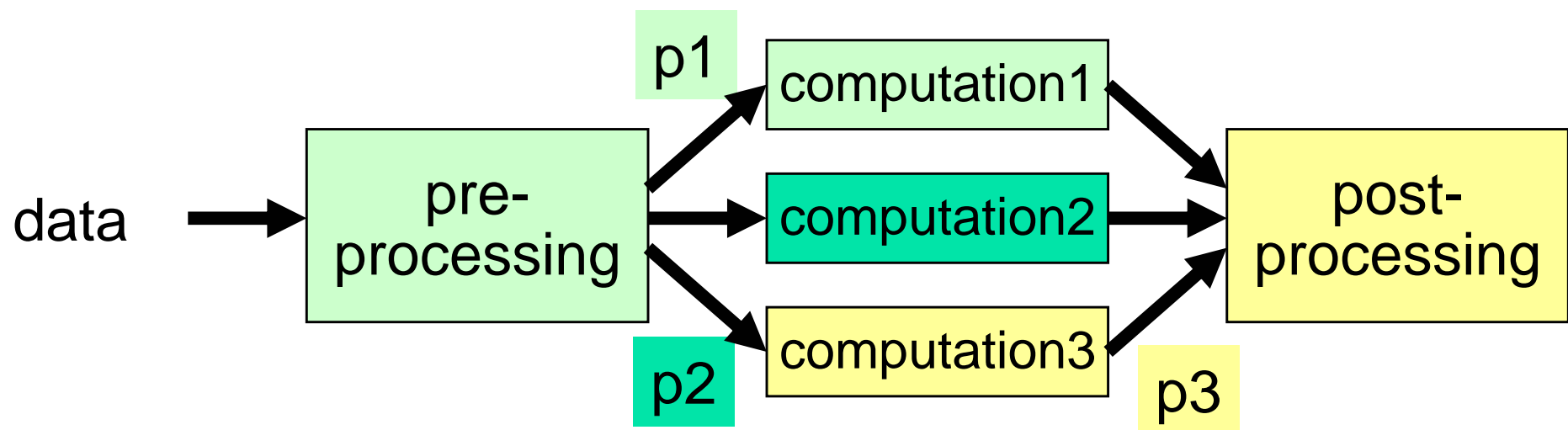
Parallelization Paradigms...

- **Advantage** of data-partitioning
 - easy to program: only one source code
 - easy to adapt to target machine
 - often regular data structures
 - relatively easy debugging
- **Disadvantage** of data-partitioning
 - sometimes not appropriate for the algorithm
- Data-partitioning is de facto standard
 - called **SPMD** (single program, multiple data)

Parallelization Paradigms...

Mixed code- and data-partitioning

- this is our final goal
 - has advantages of both approaches
 - needs multiprocessing on each node





Tools for Parallelization

- Automatically parallelizing compilers
- Manually parallelizing compilers
- Parallel languages and language extensions
- Parallel extensions for existing sequential languages
- Parallel programming libraries for existing sequential languages

Automatically Parallelizing Compilers

- Parallelisation at loop level (Fortran)
 - compiler analyses data dependencies
 - detects loop indices that can be computed in parallel and distributes them
 - compiler pragmas control compiler
 - usually bad performance



Manually Parallelizing Compilers

- Important new approach: OpenMP
- Parallelization for systems with shared memory
- Compilation controlled by special comments introduced by the programmer
- Runtime library necessary

Parallel Languages and Language Extensions



- Approach: High Performance Fortran (HPF)
- Designed 1990+ by a consortium
- No longer of importance for high performance computing
- Problem: quality of parallelization by the compiler



Parallel Programming Libraries

Standard with high performance computing

- Concept

- Spawn independent processes
- Integrate cooperation via message passing

- Examples

- Parallel Virtual Machine (PVM)
- Message Passing Interface (MPI)



Software/Hardware-Relation

- All programming concepts are applicable on all architectural types of high performance computers
- In reality for reasons of efficiency
 - libraries for message passing with distributed memory architectures
 - threads and automatic parallelization with shared memory architectures



Algorithmical Aspects

Divided world of the programmers

- numerical algorithms
 - Grand Challenge Algorithms:
weather forecast, protein design, crash simulation...
- non-numerical algorithms
 - search algorithms: theorem prover, game programs etc.
 - database applications



Numerical Algorithms

- Computational fluid dynamics (CFD), numerical computations, optimizations, simulations etc.
 - iterative algorithms
 - complete on a global condition
 - regular datastructures (vectors, fields, ...)
 - regular communication structure
 - static process structure



Non-Numerical Algorithms

- Database applications, artificial intelligence
 - search tree algorithms
 - irregular communication structures
 - irregular data structures (dynamic, garbage collection)
 - dynamic process/thread structure



A First Summary

- Paradigms of parallelization
 - data partitioning, code partitioning
- Tools for parallel programming
 - compilers and libraries
 - most important: natural intelligence
- Divided world
 - numerical / non-numerical algorithms



Parallelization Examples

Three examples

- numerical application
 - discussion of partitioning
 - computational fluid dynamics (CFD)
 - discussion of objects to be distributed
 - tree search algorithm
 - general discussion
-
- all examples manually parallelized



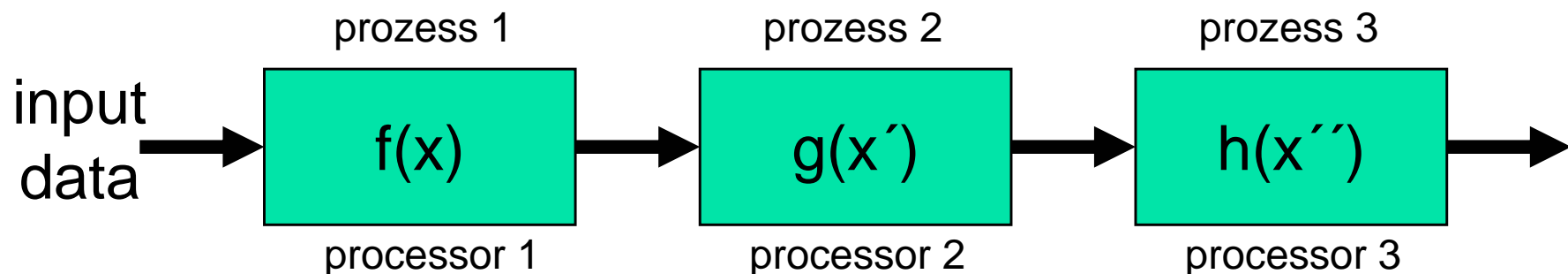
Example 1: Numerical Program

- Three functions $f()$, $g()$ and $h()$
- Apply functions to a set of values and compute result $h(g(f(x)))$
- We consider two cases:
 - code partitioning / data partitioning
 - both for distributed memory systems

Example 1: Numerical Program (2)

Code partitioning

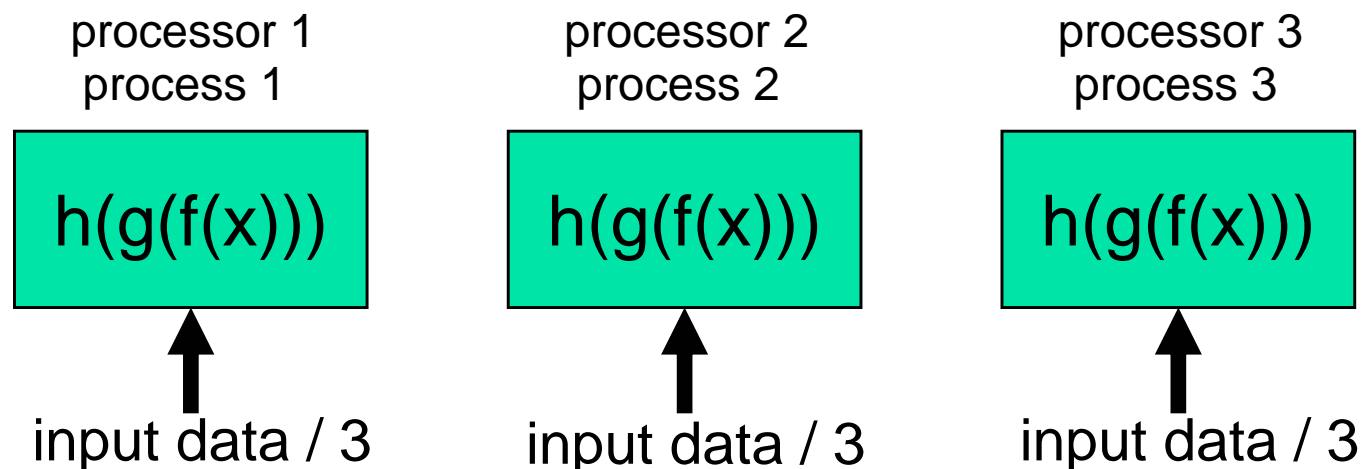
- distribute three functions onto three nodes
- works in so-called macro pipeline mode
- use an input set of values
- only three processors can be used efficiently



Example 1: Numerical Program (3)

Data partitioning

- replicate functions on nodes
- distribute input data over nodes



Example 1: Numerical Program (4)

Code partitioning

- basic implementation
 - store values in vector
 - three processes, one on each processor
 - each process computes one function of f , g , h
 - process i computes intermediate results and sends vector to process $i+1$
- problem
 - this is **not** a parallel program!

Example 1: Numerical Program (5)

- improving the basic implementation
 - process i sends computed values immediately to process $i+1$
- advantage
 - good parallel implementation
- disadvantage
 - bad communication/computation ratio: frequent sending of values

Example 1: Numerical Program (6)

- second improvement
 - increase granularity
 - send values in blocks of e.g. 1000 values
- advantage
 - good parallel implementation
 - better communication/computation ratio
- disadvantage
 - filling and emptying of pipeline take some time

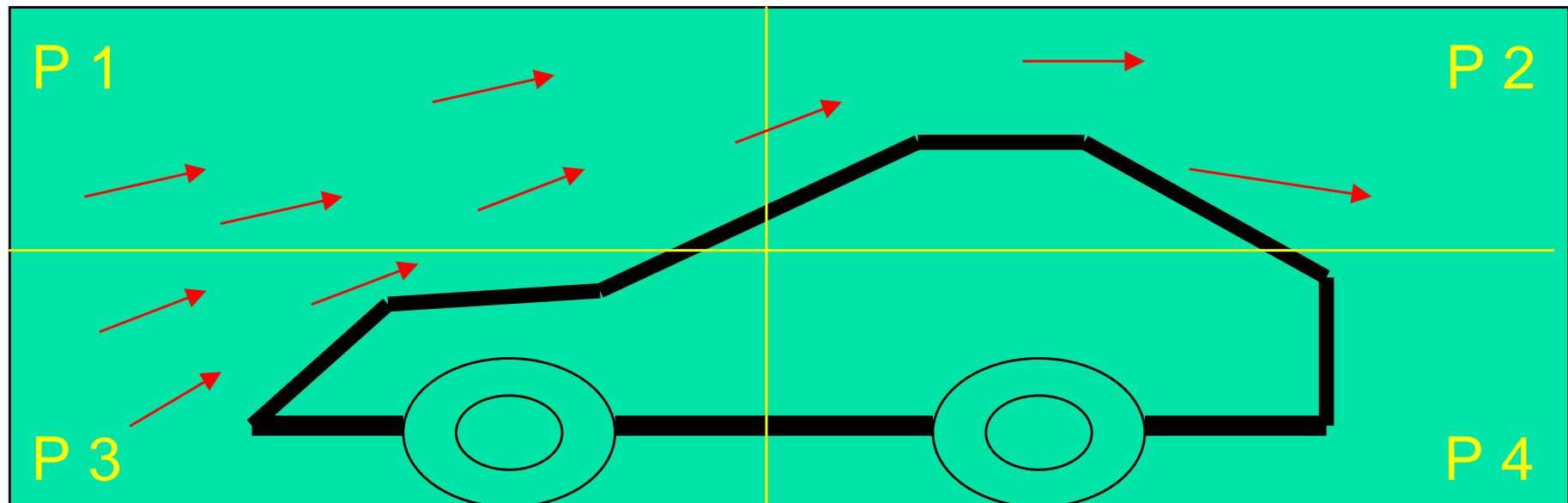
Example 1: Numerical Program (7)

Data partitioning with distributed memory

- basic implementation
 - distribute values over nodes
 - three processes on three nodes compute $h(g(f(x)))$
 - send results to process 0
- advantage
 - good parallel implementation
 - good communication/computation ratio
- disadvantage
 - distribution of data must be programmed

Example 2: CFD

- Simulation of a wind tunnel
- Iterative computation with time step t
 - Microscopic approach: compute particles
 - Macroscopic approach: compute distribution of pressure, temperature etc.





Example 2: CFD (2)

We consider only the microscopic approach

- Also called molecular dynamics

First option: distribute particles

- each process computes one part of all particles

- disadvantage
 - difficult to find neighbouring particles for collisions
- advantage
 - equal distribution of particles onto processors usual results in good load balance



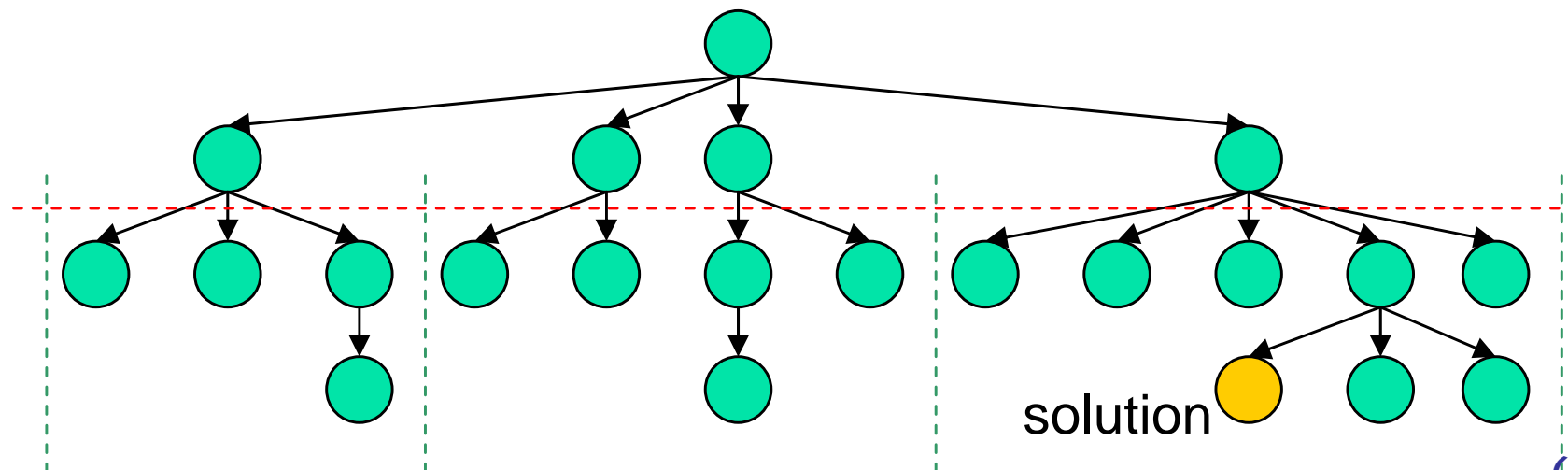
Example 2: CFD (3)

Second option: distribute volume segments

- each process computes its own segment of the complete volume
- disadvantage
 - varying number of particles leads to bad load imbalance
- advantage
 - neighbouring particles can be easily found

Example 3: Tree Search

- Each position has several possible continuations
- Problems
 - level of solution unknown
 - load balance between processes
 - detection of program completion



Example 3: Tree Search (2)

- Algorithms
 - process i computes tree until level j and puts descriptions into a waiting queue
 - idle processes contact process i , receive an element from the queue via email, and compute results
- Good parallel implementation
 - load balance no problem, but must be programmed
 - completion detection: send completion-message regularly to all processes; they check for completion



Conclusion from the Examples

- There are always different alternatives to parallelize code
- The chosen variant influences the maximal achievable performance
- You cannot derive the parallel program's efficiency from the sequential program
- Usually data partitioning is easier to be programmed
- Tree search algorithms are often trivial to be parallelized



Summary

- Program parallelization is a complex task
- We see code and data partitioning
- Data partitioning is often easy and efficient
- Tools: programming libraries and own experience
- Significant differences between numerical and non-numerical algorithms
- Efficiency of parallelization can usually not be predicted



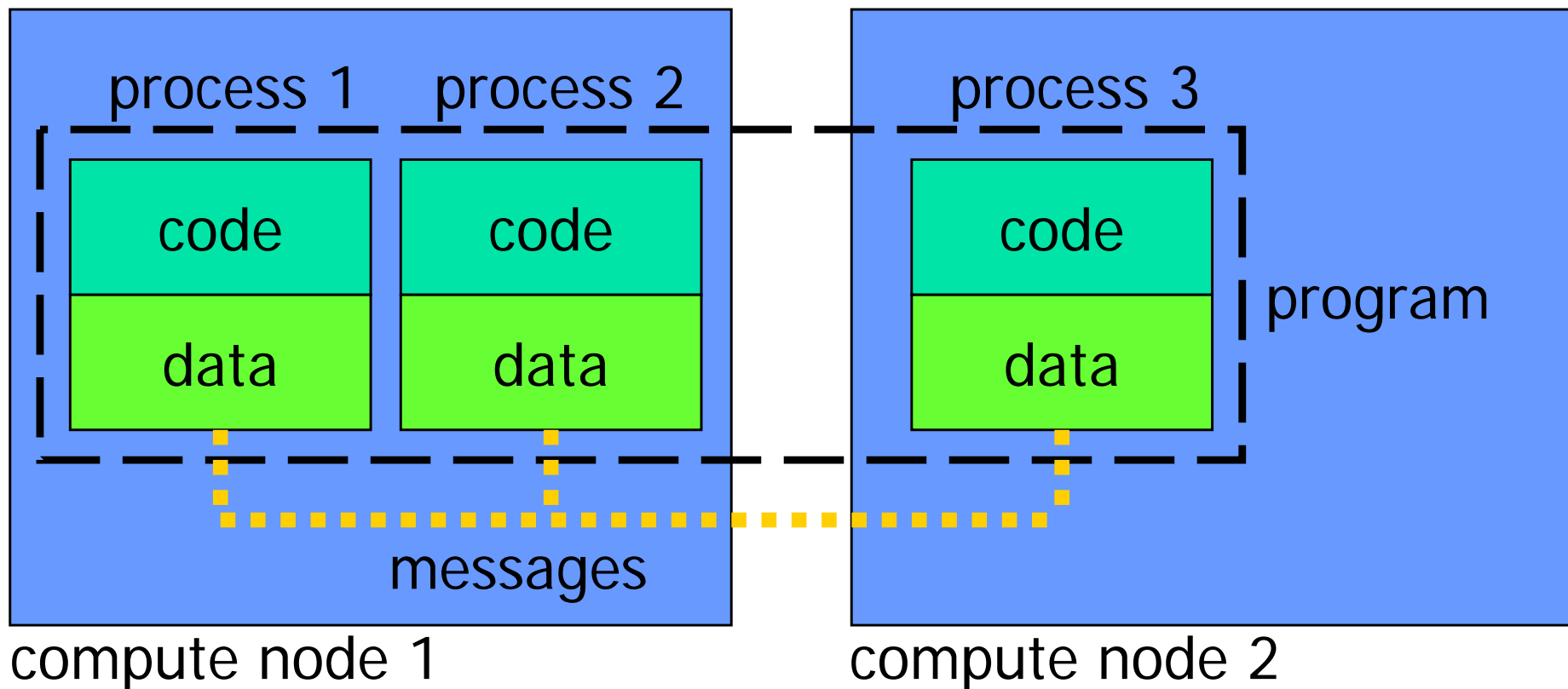
Message Passing

Message Passing with MPI

- The Problem
- The Message Passing Interface (MPI)
- Goals and Content of the Specification
- Point-to-Point Communication
- Derived Datatypes
- Collective Communication
- Groups and Contexts
- Evaluation

The Problem

Program code of the processes can be identical or may be different





The Problem...

- Compile for different architectures
- Load code onto different nodes
- Start processes on the nodes
- Bring processes in contact with each other
- Information exchange between processes
- Optimization of communication
- Relation of processes with respect to communication

Load and Start Code

- Similar to creation of threads

```
spawn(<binary_name>, <node_list>, ...);
```

- If we do have only one program code

```
if (myid()==0)
then /* I'm the first */
  spawn(...);
  send(init_data);
else /* I was spawned */
  receive(init_data);
fi
```

Not necessarily only one process per processor

Information Exchange

- Sending of messages

```
send(<to_proc_id>, <data>);
```

```
broadcast(<data>);
```

- Receiving of messages

```
receive(<from_proc_id>, <data>);
```

```
testreceive(<from_proc_id>);
```

What we do: integration of communication calls into the program source code

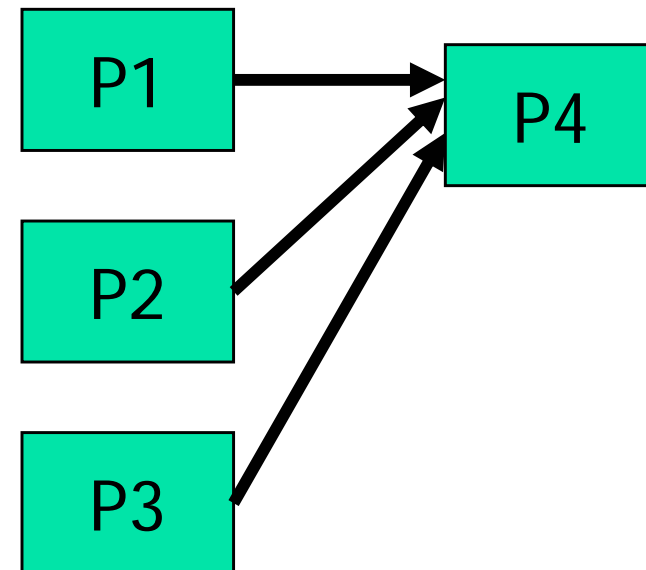
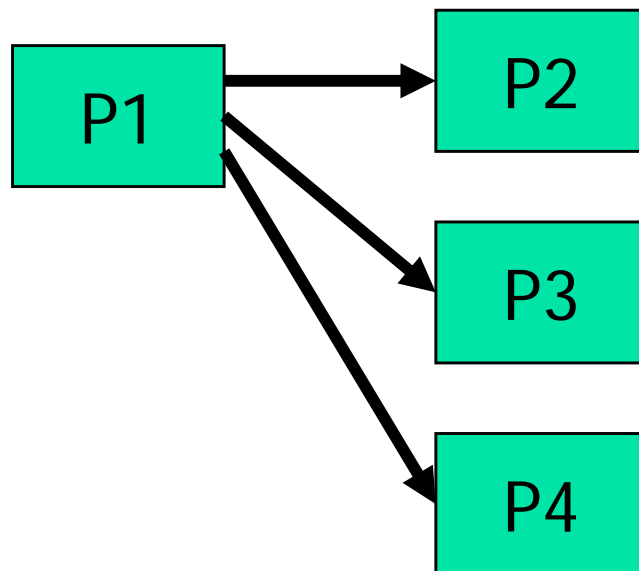
Big effort – but also big performance

Communication Schemes

direct communication



broadcast / multicast

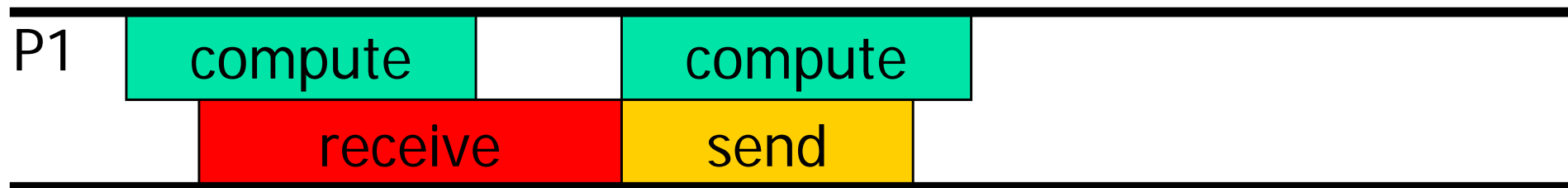


indirect communication

Optimization of Communication Efficiency



Try to do sending and receiving concurrently
Can only be done efficiently with a combination of hardware and software





Existing Approaches

- P4, Parmacs, Chameleon, NX, ...
Historical message passing libraries
- Parallel Virtual Machine (PVM)
A library available for almost all computer architectures
Was for a long time the de facto standard
- Message Passing Interface (MPI)
Specification of an API for message passing
De facto standard for all high performance computers and cluster architectures

Message Passing Interface (MPI)

- Driven by the MPI-Forum
(companies, universities, ...)
- Started in 1992
- MPI Standard 1995 (communication only)
- MPI-2 Standard 1997 (the rest)
- Advantages of this standard:
 - portability
 - ease of use

Before we had 10+ different competing approaches



Goals of MPI

- Design of an API (application programming interface)
- Support for efficient communication methods
- Support for heterogeneous environments
- Supported languages: Fortran77 and C/C++ (now also Java and script languages)
- Specification close to already existing approaches
- Language independent semantics
- Provide for a thread-safe implementation



What is in MPI?

- Point-to-point communication
- Collective operations
- Process groups
- Communication contexts
- Process topologies
- Profiling interface



What is **not** in MPI?

- Shared memory communication
- Support from the operating system
e.g. interrupt-driven communication
- Process management
e.g. start of application
- Parallel input/output (I/O)

MPI essentially only process communication

MPI-2 covers further important aspects

MPI Specification Method

- Definition of calls are language independent
- Arguments are annotated IN, OUT, INOUT

E.g. `MPI_WAIT(request, status)`

`INOUT request`

`OUT status`

C: `int MPI_Wait(MPI_Request *request,
MPI_Status *status)`

F77: `MPI_WAIT(REQUEST, STATUS, IERROR)
INTEGER REQUEST,
STATUS(MPI_STATUS_SIZE), IERROR`



MPI Definitions

MPI very careful with language aspects

Important terms are well defined

- *Nonblocking* : a call returns before the operation completes and before local resources may be re-used
- *Locally blocking* : on return local resources may be re-used
 - depends only on the local process
- *Globally blocking* : on return the communication has completed
 - depends on other processes
- *Collective* : all processes in a group must execute the call



Point-to-Point Communication

Send operation

```
MPI_SEND(buf, count, datatype, dest, tag, comm)
```

```
IN buf          address of send buffer
```

```
IN count        #elements in buffer
```

```
IN datatype     data type of elements
```

```
IN dest         rank of target process
```

```
IN tag          message label
```

```
IN comm        communicator (group, context)
```

Datatypes: int, long int, float, char, ...

Messages consist of envelope and contents

Point-to-Point Communication... (2)

Receive operation

```
MPI_RECV(buf, count, datatype, source, tag, comm,  
          status)
```

OUT buf	address of receive buffer
IN count	#elements to receive
IN datatype	data type of elements
IN source	rank of sending process
IN tag	message label
IN comm	communicator (group, context)
OUT status	result of operation

Point-to-Point Communication... (3)

Receive operation...

- controlled by envelope
`MPI_ANY_SOURCE`, `MPI_ANY_TAG` (wildcard)
- enquiry of details
`MPI_GET_SOURCE()`, `MPI_GET_TAG()`

Point-to-Point Communication... (4)

- Communication semantics
 - preservation of sending order
- Data conversion
 - Automatically in heterogeneous networks
- Variants
 - normal: locally blocking
 - ready communication: sending allowed only after receive operation was posted (allows a more efficient implementation)
 - synchronous communication: globally blocking
completes when receiver starts receiving

Point-to-Point Communication... (5)

- Non-blocking communication
 - higher efficiency because of overlapping computation and communication
- Important concepts
 - blocking / non-blocking
(when does the call return?)
 - synchronous / asynchronous
(when is the operation completed?)
 - each call gets a unique reference
check for call completion with this reference

Point-to-Point Communication... (6)

Non-blocking

`MPI_ISEND(...)`

immediate send

`MPI_IRecv(...)`

immediate receive

`MPI_TEST(request, flag, status)`

nonblocking

`MPI_WAIT(request, status)`

blocking

`MPI_CANCEL(request, status)`

Blocking

`MPI_SEND(...)`

`MPI_RECV(...)`



MPI „Hello World“

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

int main (int argc, char *argv[])
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf("Hello World from process %d of %d\n",
           rank, size );
    MPI_Finalize();
    return 0;
}
```



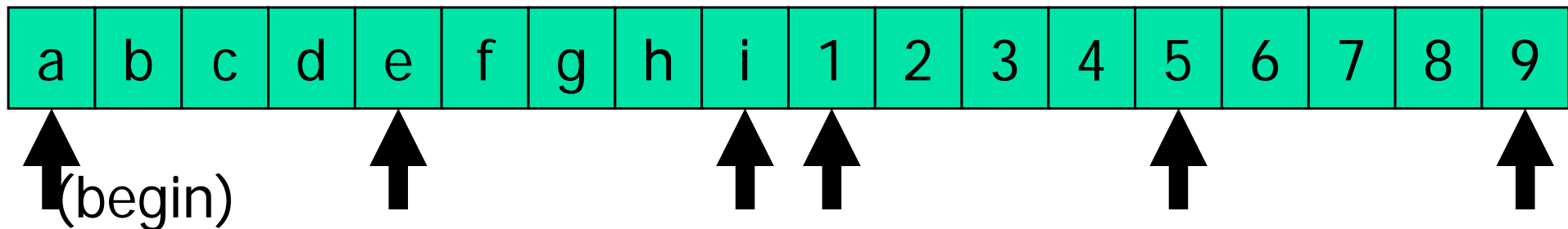
Derived Data Types

- Goal
 - messages with mixed data types
 - messages with non-contiguous data areas
- Packaging of messages needs CPU resources
- Efficiency depends on hardware
(in particular Direct Memory Access, DMA)

Derived Data Types...

Example: two matrices with complex numbers

Task: send (and receive) only the two diagonals



```
MPI_TYPE_VECTOR(3 blocks, 1 element/block,  
                4 blockstride, MPI_COMPLEX, diag)
```

```
MPI_TYPE_CREATE_HVECTOR(2 blocks, 1 element/block,  
                        9*sizeof(MPI_COMPLEX), diag, doublediag)
```

```
MPI_TYPE_COMMIT(doublediag)
```

```
MPI_SEND(begin, 1, doublediag, me, other, comm)
```



Collective Communication

Collective communications must always be performed by all members of the respective group

- Broadcast from one to all
- Barrier synchronization
- Collect and distribute data
- Global computation of functions

Possible support by specialized hardware
Space for optimizations

Collective Communication...

Collective calls for data movement

data

processes	A0		

one-all broadcast

data

processes	A0		
	A0		
	A0		

data

processes	A0		
	B0		
	C0		

all gather

data

processes	A0	B0	C0
	A0	B0	C0
	A0	B0	C0

Collective Communication...

Collective calls for data movement

data

processes	A0	A1	A2

one-all scatter



one-all gather

data

processes	A0		
	A1		
	A2		

data

processes	A0	A1	A2
	B0	B1	B2
	C0	C1	C2

all-all gather



data

processes	A0	B0	C0
	A1	B1	C1
	A2	B2	C2

Collective Computation

- Frequent situation: processes have to apply the same function to the data, e.g. to sum them up
- Use MPI-function **MPI_REDUCE**(. . . , op , . . .)
 - each process contributes with its local data
 - after completion each process has the global result
max, min, sum, product, AND, OR, XOR
- Any evaluation order must be correct
- Can be supported by special hardware in parallel computers
- Own functions are possible (be careful)

Groups, Contexts, Communicators

- New concept in MPI (not available in PVM)
- Problem:
 - third party implementors provide libraries with message passing
 - message tags and ranks can conflict with those of the application program
- Solution
 - MPI groups integrate processes that logically belong together
 - MPI contexts differentiate between program parts
 - MPI communicator: combines group and context
 - default-communicator: **MPI_COMM_WORLD**



Review of MPI

- API only for message exchange (communication)
- A big set of function calls
- Process management is missing
- No dynamic process management
 - no programs with a varying number of processes



Outlook to MPI-2

- MPI-2 is an extension, not a new version
- Includes also clarifications for MPI
- Important enhancement: process management (many different solutions before MPI-2)
- Important enhancement: parallel input/output (idea: I/O equivalent to message sending and receiving)
- Again: an even bigger set of new function calls



MPI-Implementations

- MPICH (Argonne National Laboratory)
 - current version: MPICH2 v1.0.7
 - <http://www-unix.mcs.anl.gov/mpi/mpich/>
 - available for parallel computers and PCs
- Alternatives
 - LAM/MPI (<http://www.lam-mpi.org/>)
was popular before, now replaced by
 - Open-MPI (<http://www.open-mpi.org/>)



Summary

- Important problems with message passing: communication schemes, efficiency, process management
- MPI specifies an API for message passing
- Point-to-point communication with many different alternatives:
synchronous / asynchronous, blocking / non-blocking
- Derived data types ease communication
- MPI groups and contexts support separation of regions of influence in different parts of a program
- MPI-2 extends MPI by important issues



Advanced Issues

Advanced issues with message passing

- Introduction, Concepts, Definitions
- Simple I/O
- Non-Contiguous Accesses
- Collective Calls
- Nonblocking I/O
- Shared File Pointers
- File Formats
- Performance Aspects
- Implementation



What is MPI-2 I/O?

- Extension to the MPI-Standard: parallel input/output (I/O)
- Defined in the MPI-2-Standard document
- Semantics analogous to message passing
 - e.g. collective, nonblocking also for I/O
 - I/O equivalent to sending and receiving



Why Use Parallel I/O in MPI?

- Higher performance
 - e.g. by using collective calls
 - e.g. by making asynchronous calls
- Easier data access
 - e.g. derived data types with irregular data
 - as a consequence also better portability in heterogeneous environments



MPI-I/O Concepts

- File pointer
 - individual / shared file pointer
- Non-contiguous access
 - manage data at different locations with one call
- Collective call
- File view
 - process oriented view to the data in the file
- Hints
 - pass information on to the implementation layer




Some Definitions

- file
 - a collection of typed data
 - random or sequential access
 - collective opening by a group of processes
- displacement
 - an absolute byte position in the file where the individual views of the processes start
- etype
 - basic unit in file; used for positioning

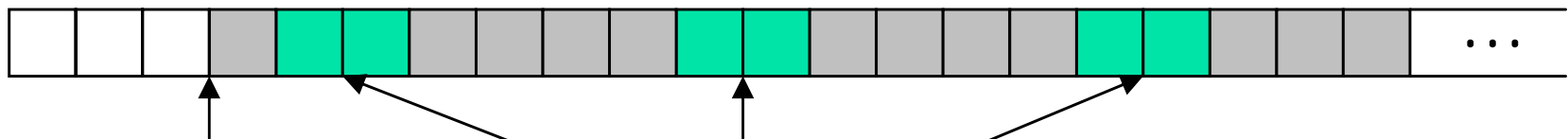
Some Definitions...

- filetype
 - template for a file
 - consists of etypes and holes of equal size

etype 

filetype  ← hole

construction of a file



displacement

data

Some Definitions...

- view (of a process)
 - defined by displacement, etype and filetype

etype



process 0 filetype



process 1 filetype



process 2 filetype



file structure



displacement



Some Definitions...

- offset
 - position in file in relation to the current view specified in number of etypes
- file size
 - total number of bytes in a file
- file pointer
 - position in file managed by MPI
 - individual file pointer: each process has own pointer
 - shared file pointer: all processes have one single pointer
 - file handle
 - reference to the file (as with Unix)

Simple I/O: Multiple Processes read/write a File

- processes open collectively(!) a file ...
MPI_FILE_OPEN
- ... each process positions with its own pointer ...
MPI_FILE_SEEK
- ... and reads from file / writes to file ...
MPI_FILE_READ
MPI_FILE_WRITE
- ... and closes the file
MPI_FILE_CLOSE



Simple I/O Function Prototypes

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

```
int MPI_File_seek (MPI_File fh, MPI_Offset,  
    int whence)
```

```
int MPI_File_read (MPI_File fh, void *buf, int  
    count, MPI_Datatype datatype, MPI_Status *status)
```

```
int MPI_File_write (MPI_File fh, void *buf, int  
    count, MPI_Datatype datatype, MPI_Status *status)
```

```
int MPI_File_close (MPI_File *fh)
```



File Access: Positioning

- Three alternatives
 - Explicit offsets
 - Individual file pointers
 - Shared file pointers
- Mixed use in one program
- Syntax
 - explicit offsets: **MPI..._AT**
 - shared: **MPI..._SHARED, MPI..._ORDERED**

Non-Contiguous Access and Collective Calls



- Until here I/O could also be managed with regular Unix-I/O: one file, contiguous data
- However: parallel programs access files independently at non-contiguous positions from different processes
- MPI-2 I/O offers functionality to access non-contiguous parts in files from different processes with one single call

Non-Contiguous Access: File View

- By using „views“ each process sees only its own part of the file
- view defined by
 - displacement, etype, filetype
 - etype and filetype are standard data types or derived data types
- Specify view with
MPI_FILE_SET_VIEW
- holes need to be defined too
MPI_TYPE_CREATE_RESIZED

Non-Contiguous Access: Example

```
/* 2 MPI_INT contiguous as derived data type */
MPI_Type_contiguous(2,MPI_INT,&contig);

/* append 4 holes; makes size 6 */
lower_boundary=0;
extent=6*sizeof(int);
MPI_Type_create_resized(contig,lower_boundary,extent,
    &filetype);

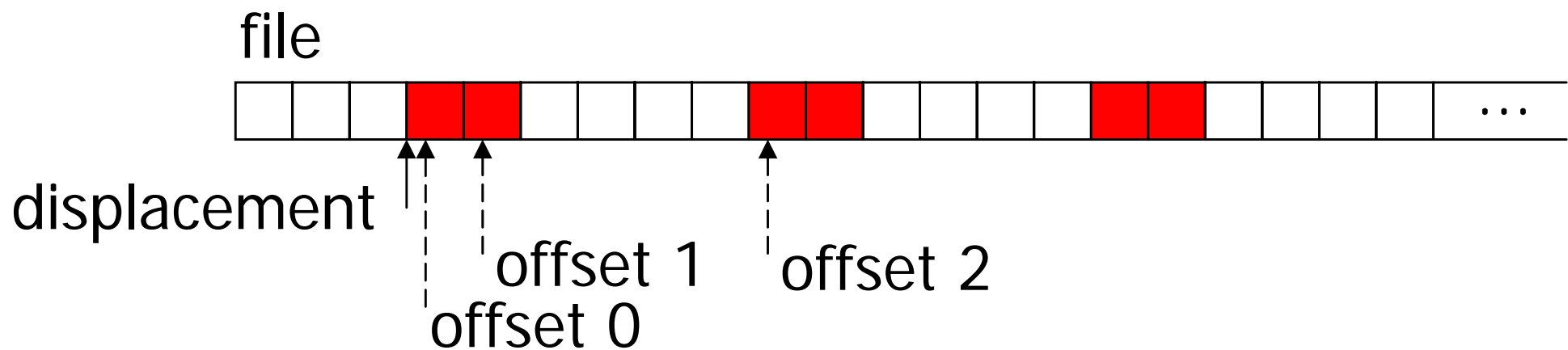
/* publish the new data type ...*/
MPI_Type_commit(&filetype);

/* ... and set the file view */
MPI_File_set_view(filehandle,displacement,etype,filetype,
    "native",MPI_INFO_NULL);
```

Non-Contiguous Access: Example

etype = MPI_INT 

filetype = 2*MPI_INT resized to size 6





Collective Calls

- For further optimization all processes can concurrently access the file
- Specification of a view just like before but now additional functions
 - allows the MPI-implementation to optimize accesses from multiple processes
- Even if each process reads only small non-contiguous sections of the file, the MPI-implementation can possibly compose a single big file access from them
- **MPI_FILE_READ_ALL, MPI_FILE_WRITE_ALL**



Nonblocking I/O

- Use it to overlap I/O with communication and/or computation
- All non-collective(!) read and write calls have corresponding non-blocking calls
 - test for completion with standard MPI-test calls
- Naming convention: **MPI_FILE_I...**
e.g. **MPI_FILE_IREAD**



Shared File Pointer

- Until now only individual pointers and offsets
- Also supported: shared pointers
 - used by all accessing processes
 - every access manipulates pointer position
 - next accessing process sees new position

- Functions

MPI_FILE_SEEK_SHARED

MPI_FILE_READ_SHARED

MPI_FILE_WRITE_SHARED



Shared File Pointer...

- With collective calls we can have a serialization according to the process rank
 - `MPI_FILE_READ_ORDERED`
 - `MPI_FILE_READ_ORDERED_BEGIN`
- Typical application
 - shared protocol files (log files)



Hints

- hints give the user the chance to pass information on to the MPI-implementation
- Examples for hints are
 - number of disks to use to stripe the file (striping)
 - width of each stripe
- hints are optional
 - also the implementation may ignore hints



File Formats

- Files are a sequence of bytes
 - physical storage is implementation dependent
- MPI defines three data representations for different degrees of portability
 - „native“: no conversion (= memory representation)
quick and non-portable
 - „internal“: portable between platforms with identical MPI-implementation
 - „external32“: 32-bit big endian; portable between each MPI-implementation on every architecture

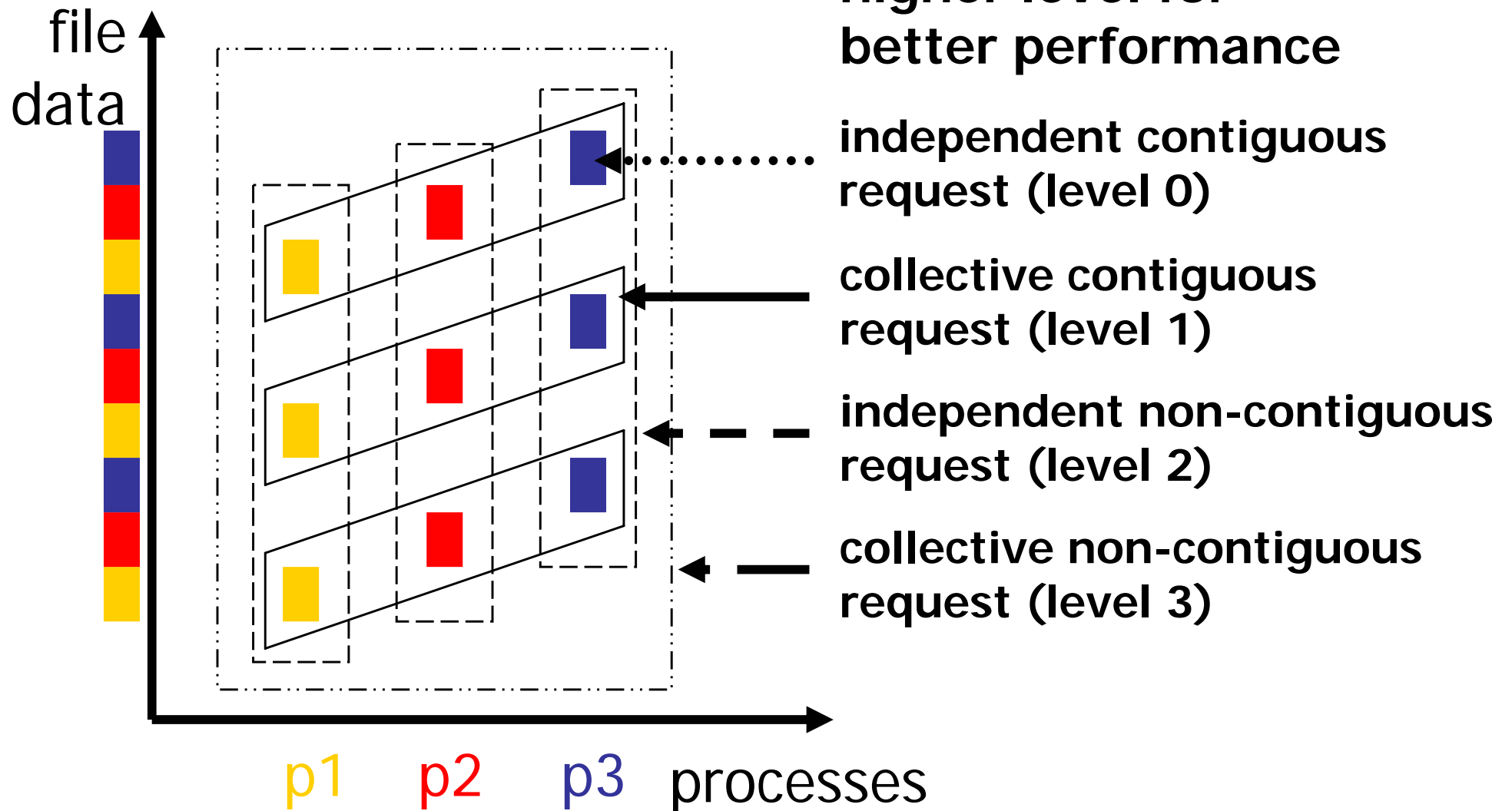


Performance Aspects

- Selection of optimal I/O-method determines achievable I/O bandwidth
 - contiguous / non-contiguous
 - collective / non-collective

- Example
 - file with 3x3-matrix of a complex data type

Performance Aspects...





Implementation ROMIO

- ROMIO ist the standard open-source implementation of MPI-2 I/O
 - part of MPICH but can be used separately with other MPI-implementations
- ROMIO supports several I/O hardware architectures and also file systems
- ROMIO supports all characteristics of MPI-2 I/O



Summary

- Parallel I/O is defined just like communication
- Also uses derived data types
- Files are a sequence of elementary data type elements
- Each process has its own file view
- We position explicitly, with individual file pointers, or shared file pointers
- Non-contiguous accesses improve performance
- Collective calls improve performance
- ROMIO is the standard implementation



Literature

- Mark Snir et al.: MPI – The Complete Reference. Volume 1, The MPI Core. Second Edition. MIT-Press, 1998.
- W. Gropp, B. Nitzberg, E. Lusk: MPI – The Complete Reference Volume 2. MIT-Press, 1998.
- William Gropp et al.: Using MPI – Portable Parallel Programming with the Message-Passing Interface. Second Edition. MIT-Press, 1999.
- W. Gropp, R. Thakur, E. Lusk: Using MPI-2 – Advanced Features of the Message-Passing Interface. MIT-Press, 1999.
- Joseph D. Sloan: High Performance Linux Clusters – with OSCAR, Rocks, openMosix & MPI, O´Reilly, 2004
- William Gropp, Ewing Lusk, Thomas Sterling: Beowulf Cluster Computing with Linux, MIT Press, 2003



Links

- MPI: www-unix.mcs.anl.gov/mpi
- MPI-Forum: www.mpi-forum.org
- MPICH: www-unix.mcs.anl.gov/mpi/mpich
- TOP500: www.top500.org