

# Programming for High Performance

## An Introduction

**Thomas Ludwig**

Technische Universität München

[ludwig@in.tum.de](mailto:ludwig@in.tum.de)

<http://www.in.tum.de/~ludwig>

---

## Tell me why!

- High performance computing is essential in many fields of natural sciences: physics, chemistry, biology, etc.
- Real high performance can only be achieved with high performance computers
- Various hardware architectures exist
- Various software concepts are available
- A concise selection of hardware and software is inevitable for high efficiency of the parallel program
- Plus: lessons learnt here can also be applied to clusters of workstations (although they do not provide real high performance)

---

# Contents

- **Parallel Architectures**
- **Parallelization Concepts**
- **Shared Memory Programming Models**
- **Message Passing Programming Models**
- **Data Parallel Programming Models**

---

# Parallel Architectures

- Flynn's Classification
- Classification of MIMD Systems
- Shared Memory and Distributed Memory
- Distributed Shared Memory
- Communication Networks
- Performance Evaluation
- The Parallel Linpack-Benchmark
- The TOP500 List

# Concepts

## Parallel Computer

- Has computing units that work in a coordinated manner and in parallel

## Units

- Special units like processor internal pipeline units
- Calculation units (integer, floating point)
- Processor nodes
- Computers
- Parallel computers and workstation clusters

## Flynn's Classification (1972)

Computers work with instruction streams and data streams

The combinations of both result in 4 variants

- SISD     Single instruction stream, single data stream
- SIMD     Single instruction stream, multiple data stream
- MISD     Multiple instruction stream, single data stream
- MIMD     Multiple instruction stream, multiple data stream

# What is what with Flynn?

## **SISD**

- Classical von-Neumann-architecture

## **SIMD**

- Vector computers, array computers

## **MISD**

- Data flow machines ???

## **MIMD**

- All that interests us: multi processor systems

MIMD has to be divided into finer classes

# Classification of Flynn's MIMD Systems

Systems consist of *multiple processors* that are connected via an *interconnection network*

Via the interconnection network information is passed between processes on different processors. Synchronisation and control information is also transferred over the network

## Criteria

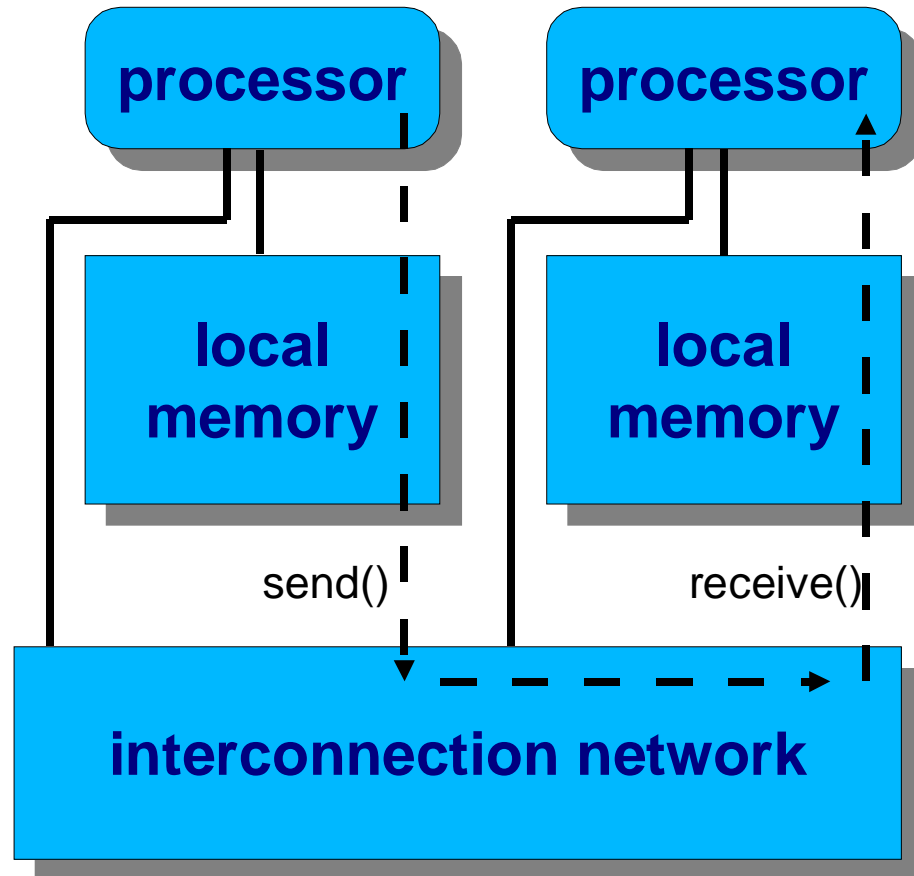
- How do processors see the memory's address space?
- How are memory components connected with the processors?

## MIMD Classification

We distinguish *three* classes:

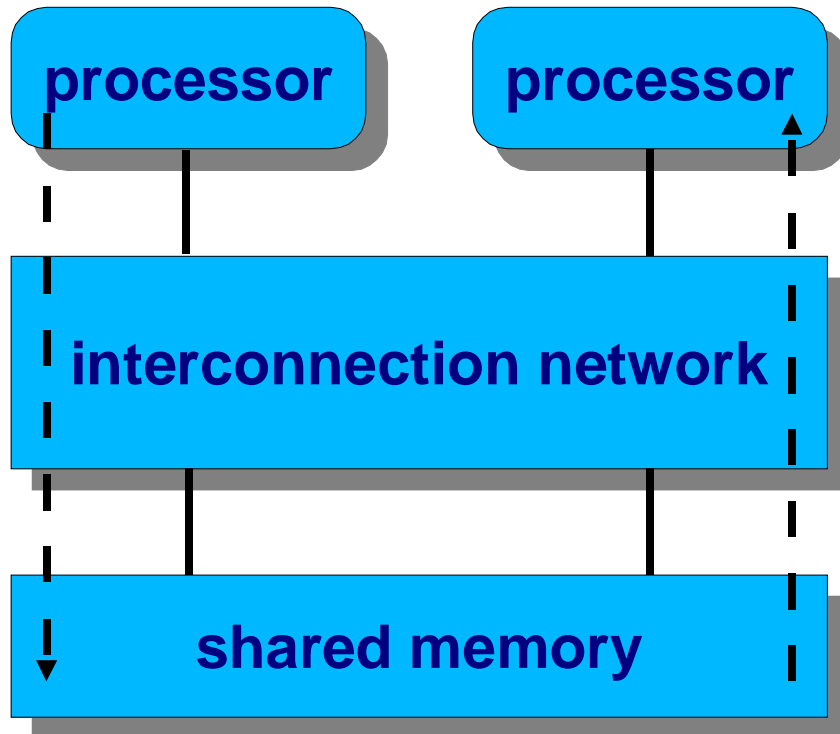
- Multi processor systems with distributed memory
- Multi processor systems with shared memory
- Multi processor systems with distributed shared memory

## Systems with Distributed Memory



- Processes see only local address space
- Access to remote memory via message passing
- Networks of workstations (NOWs) also belong to this category; their communication network is a local area network

## Systems with Shared Memory



- Each process can access the complete available address space
- Communication is handled via access to shared memory regions

# Comparison of Both Concepts

## Systems with distributed memory

- High degree of extensibility (10000+ processors)
- Complex programming and parallelization

## Systems with shared memory

- Low degree of extensibility (~100 processors)
- Easy programming and parallelization

# Synonyms for these architectures

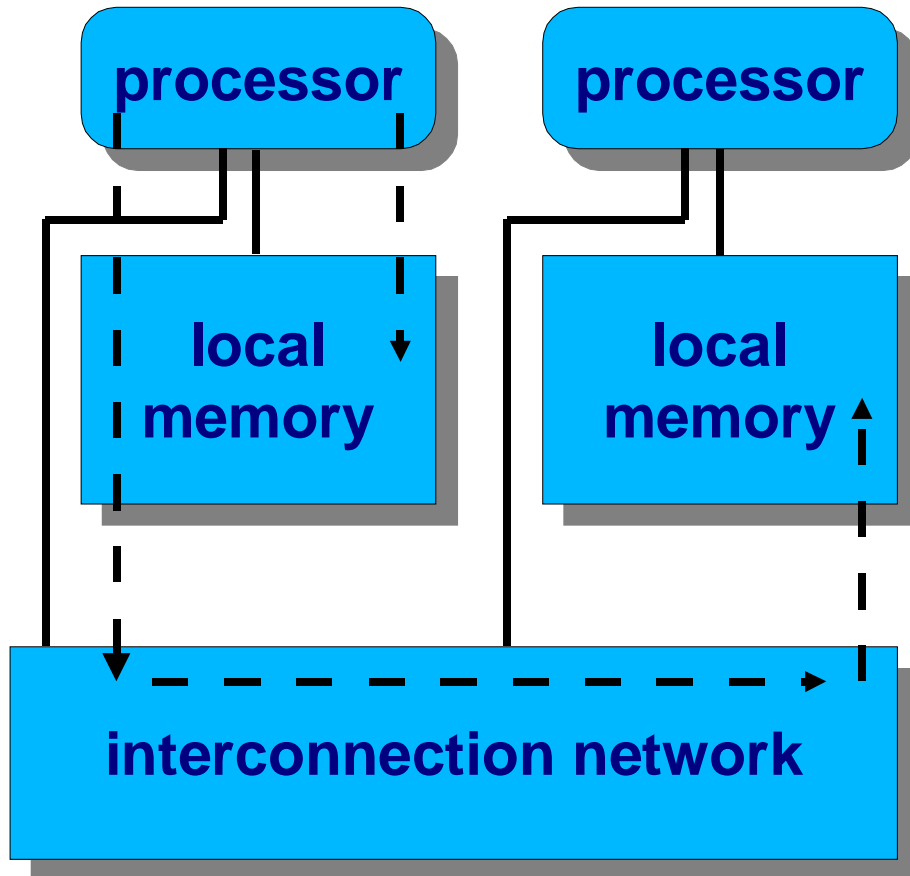
## Distributed Memory

- Loose coupling
- Weak coupling
- Multicomputer system
- Massively parallel system

## Shared Memory

- Tight coupling
- SMP (symmetric multi processing)
- Multi processor system

## Distributed Shared Memory



- Logically, each process can access the complete address space
- Physically, the address space is distributed
- Access to remotely located addresses via hardware and software support

## Classification with Respect to Memory Access

**UMA** - uniform memory access model

- Shared memory architecture

**NUMA** - non uniform memory access model

- Distributed shared memory architecture

**NORMA** - no remote memory access model

- Distributed memory architecture

**UCA** - uniform communication architecture model

**NUCA** - non uniform communication architecture model

- E.g. clusters of shared memory machines

## Interconnection Networks

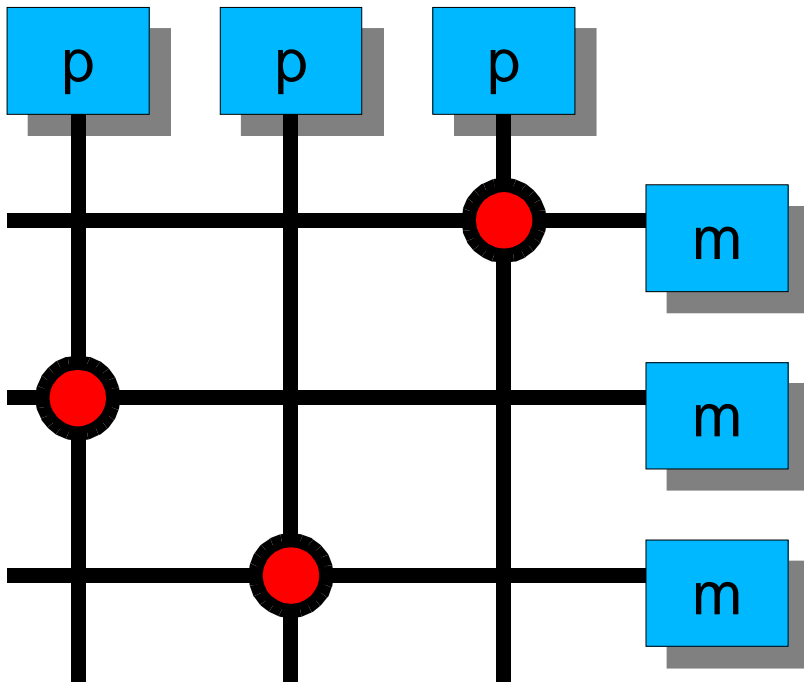
### Simplest case:

- Shared memory: bus system
- Distributed memory: ring interconnect

### Problems:

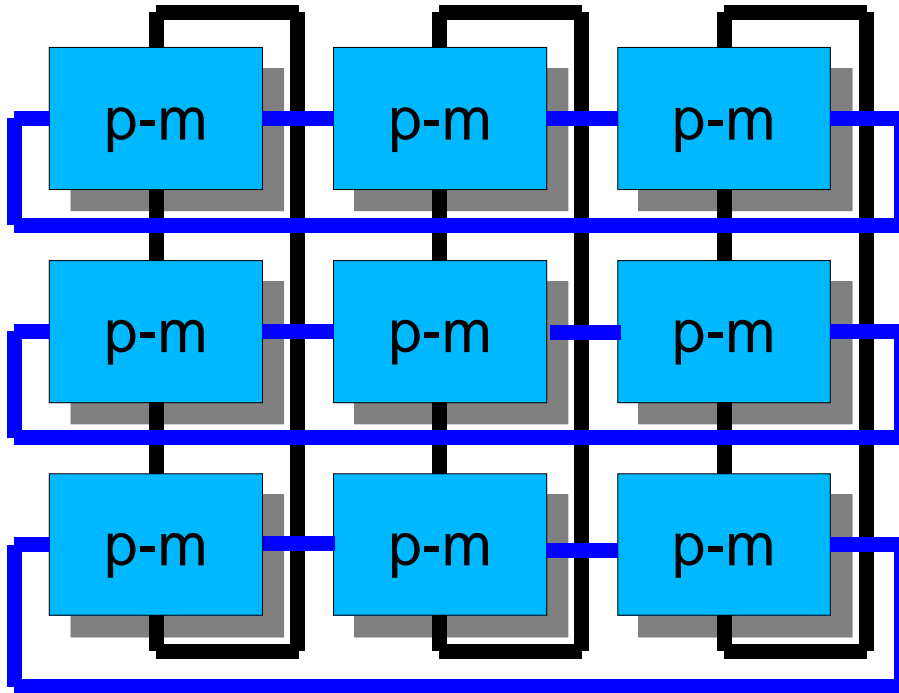
- Transfer times
- Congestions / Collisions

### Interconnection Network with Shared Memory



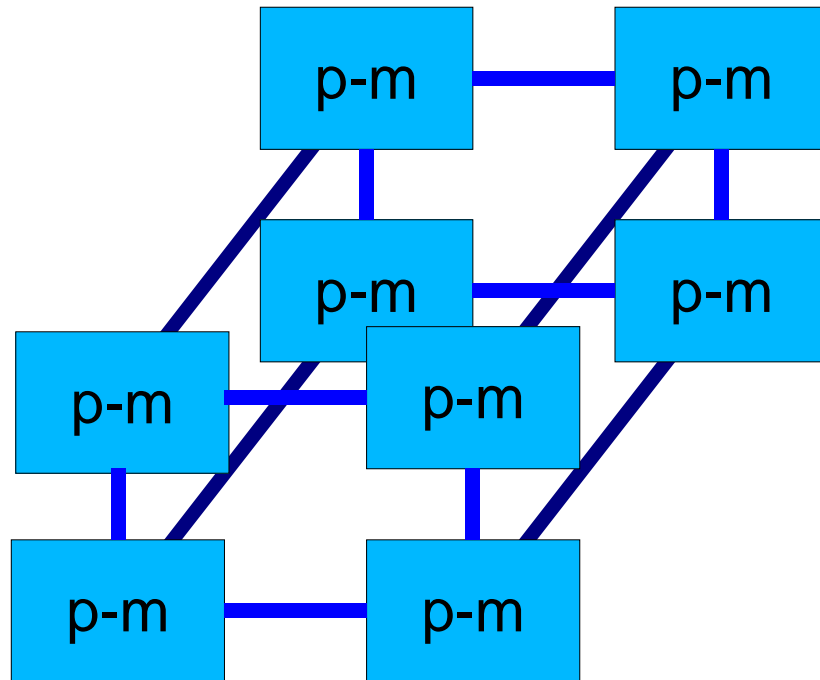
- Crossbar switch  $n \times m$
- Best case: like m-bus system
- High costs and technical effort
- Reduction of access conflicts

### Interconnection Network with Distributed Memory (1)



- Two-dimensional torus or array interconnect
- Fixed neighbourhood, therefore scalable
- Transfer time depends on distance
- Maximum path length depends on node number

### Interconnection Network with Distributed Memory (2)



- Hypercube (n-dimensional binary cube)
- dimension = neighbourhood
- Short maximal distances
- High degree of connectivity
- Double nodes increments maximal distance

## Performance Evaluation

### Complex issue

- First approach: Mflops (millions of floating point operations per second)  
Theoretical maximum derived from number of cycles per floating point operation

### Evaluation with benchmark programs

- Synthetical benchmarks  
mostly assembler programs
- CPU benchmarks  
Real application programs, mostly numerical programs

# The Parallel LINPACK Benchmark

- Developed by Jack Dongarra (Univ. Tennessee, Knoxville)
- Is also a full mathematical library for linear algebra
- Benchmark: dense equation system
- Only relevant for numerical applications
- $R_{\max}$  is maximal performance with problem size  $N_{\max}$
- $N_{1/2}$  is problem size with performance  $1/2 R_{\max}$
- $R_{\text{peak}}$  is theoretical maximal performance

## The TOP500 List of Supercomputers

Maintained by

- Hans Meuer (Univ. Mannheim, Germany)
- Jack Dongarra (Univ. of Tennessee, Knoxville)
- Erich Strohmeier (Univ. of Tennessee, Knoxville)

Two yearly updates

- Supercomputing Conference in Mannheim (June)
- Supercomputing Conference in the USA (November)

Ranking based on the Linpack benchmark

### The TOP500 List in June 2000

**Rank 1:** Intel Paragon, ASCI Red, 9632 proc., 2.4 Tflops

**Rank 2:** IBM SP2, ASCI Blue-Pacific, 5808 proc., 2,1 Tflops

**Rank 3:** SGI, ASCI Blue-Mountain, 6144 proc., 1,6 Tflops

**Rank 5:** Hitachi SR8000, 112 proc., 1Tflop

- Leibniz Rechenzentrum München

**Rank 46:** Fujitsu VPP5000, 31 proc., 0.3 Tflops

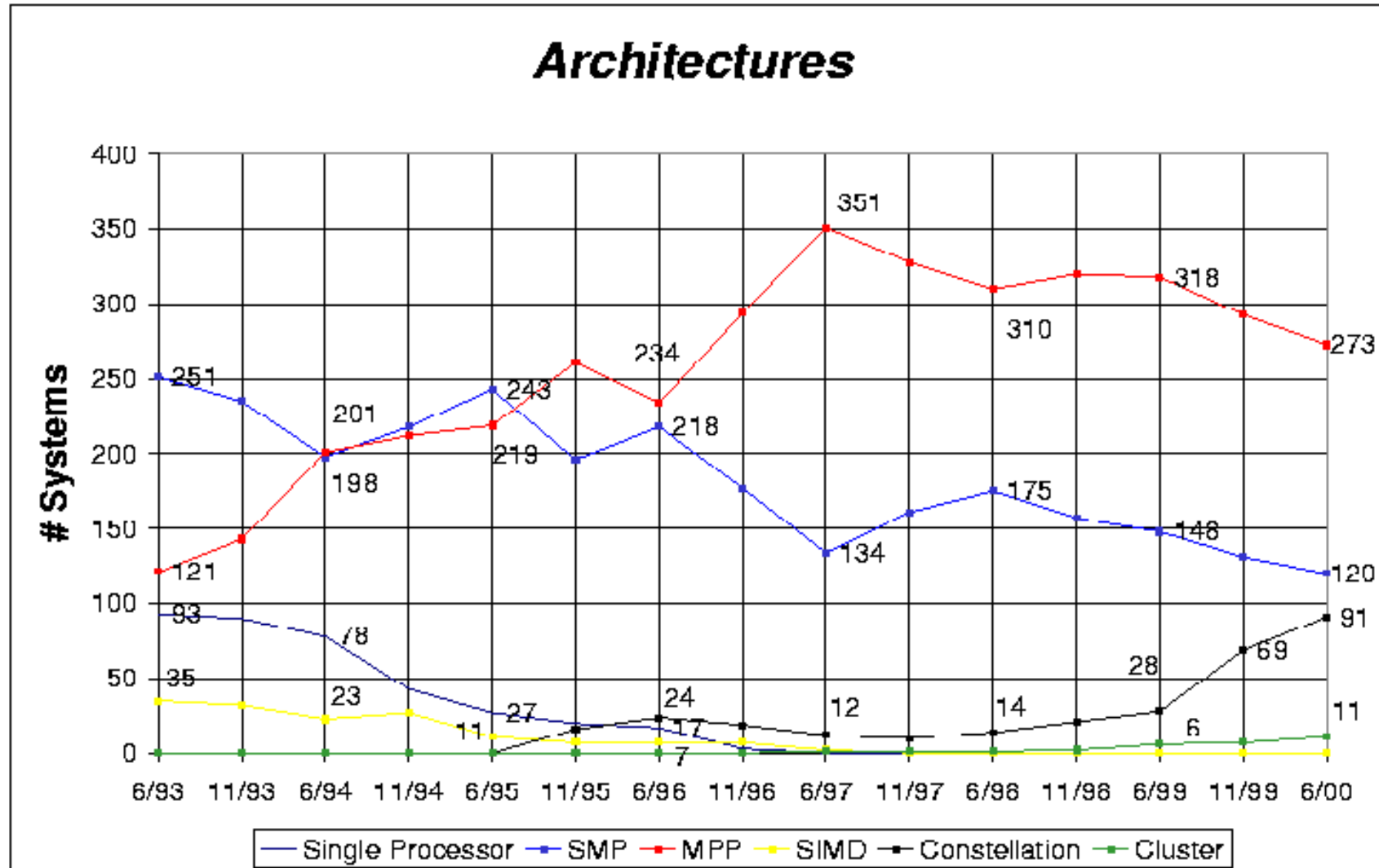
- Meteo-France, Toulouse

**Rank 62:** Cplant-Cluster (self-made), 580 proc., 0.2 Tflops

- Sandia National Laboratories, Albuquerque

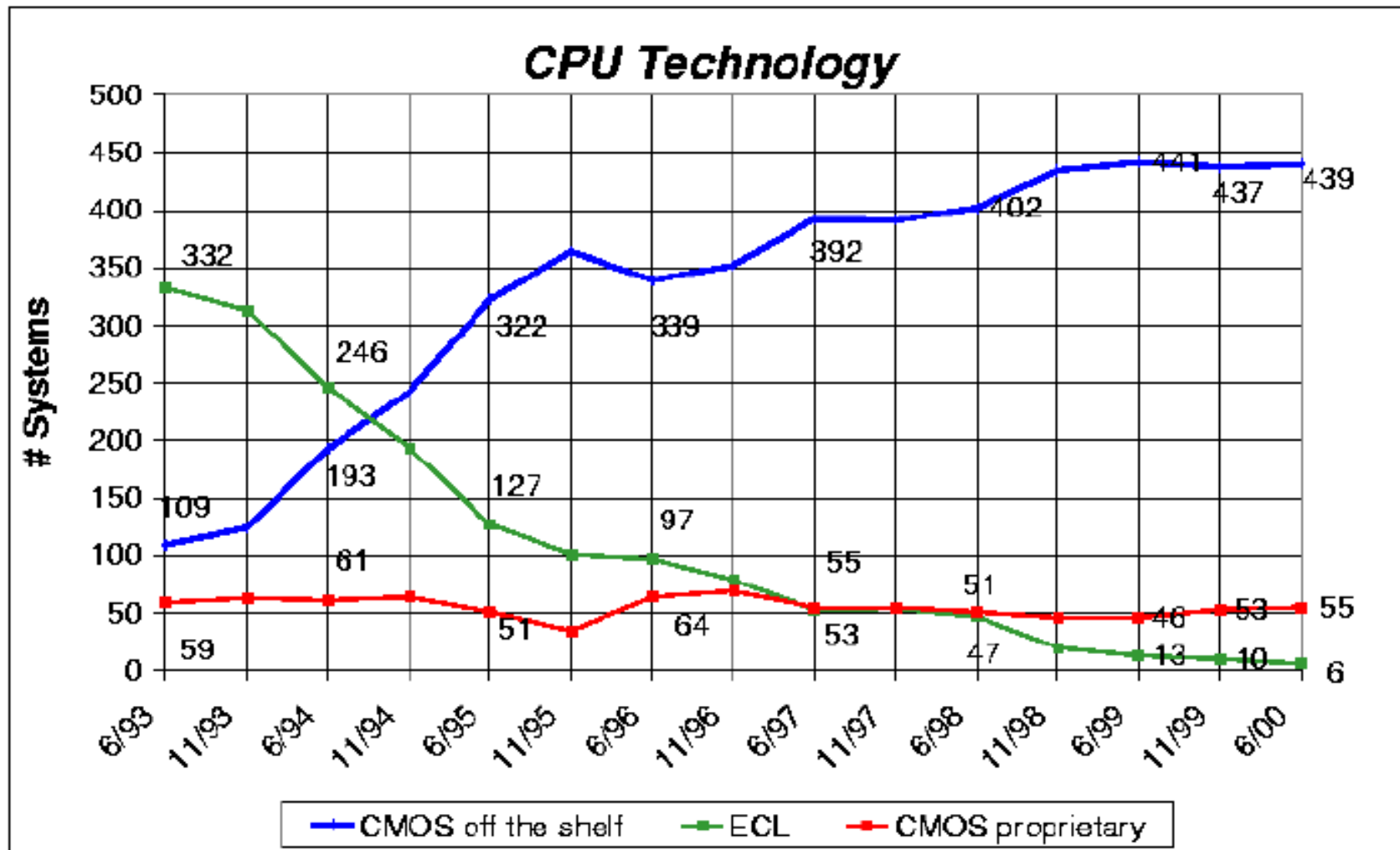
TOP500

6/00



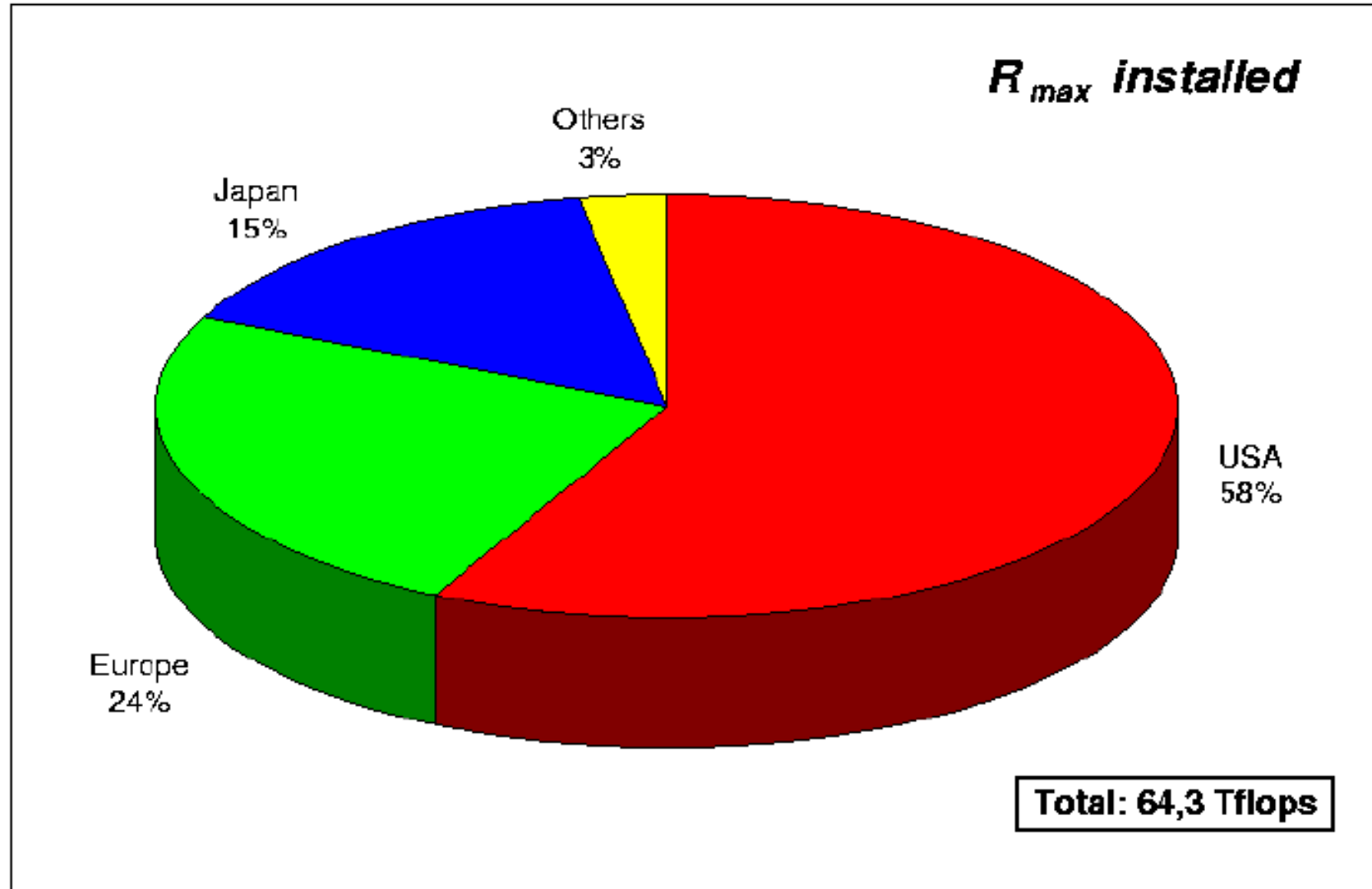
© Universität Mannheim

Diagrams by courtesy of [www.top500.org](http://www.top500.org)



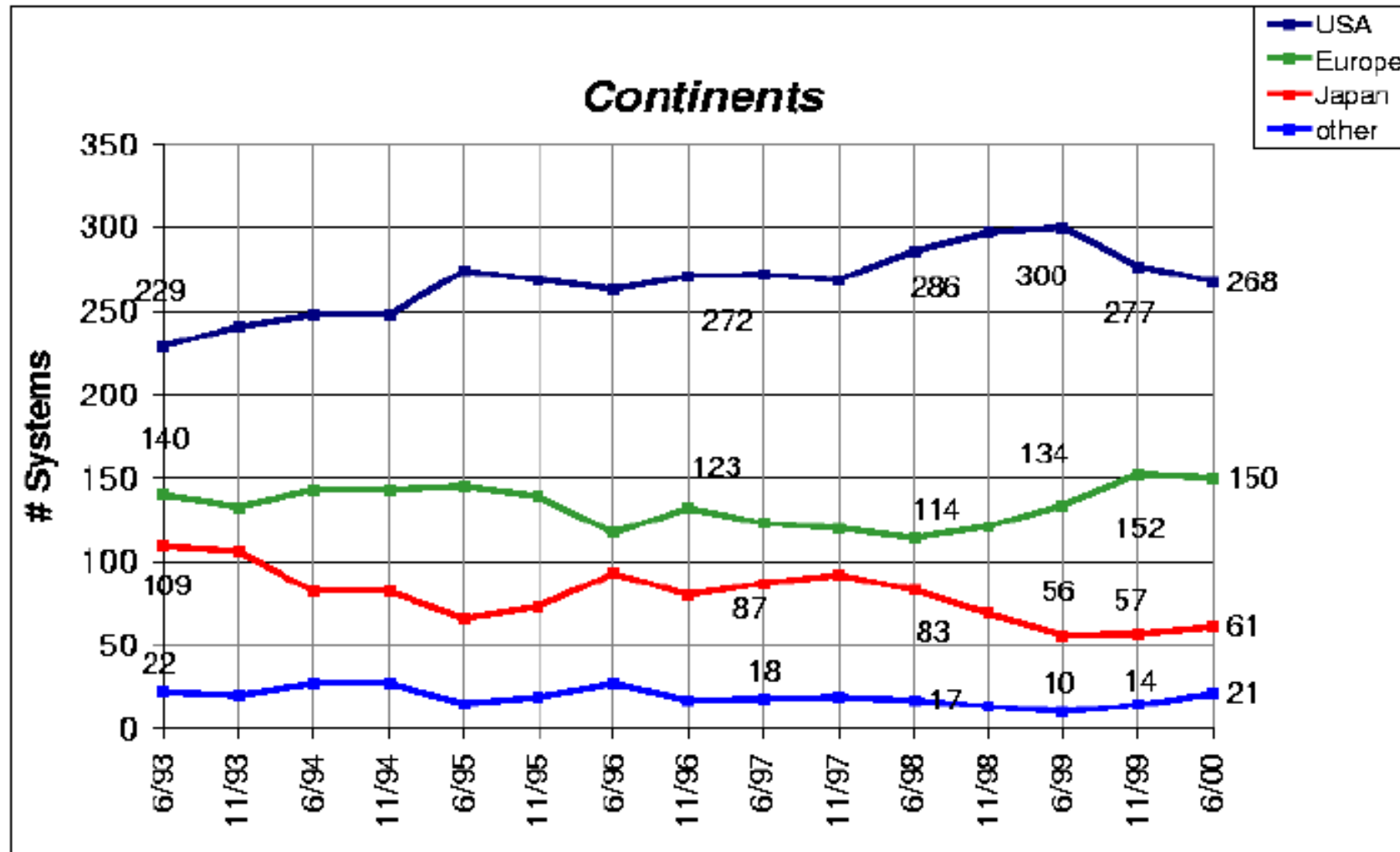
TOP500

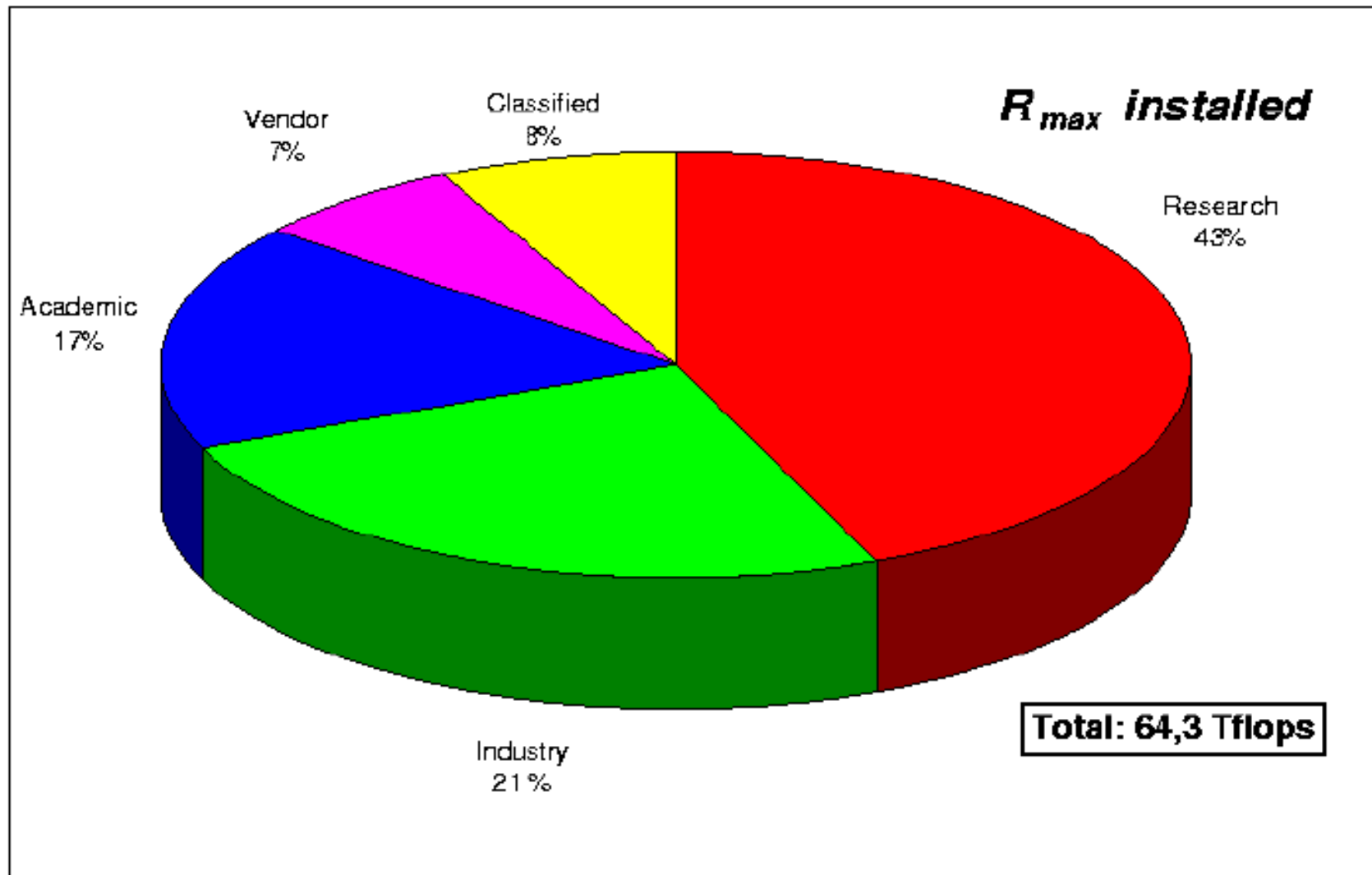
6/00

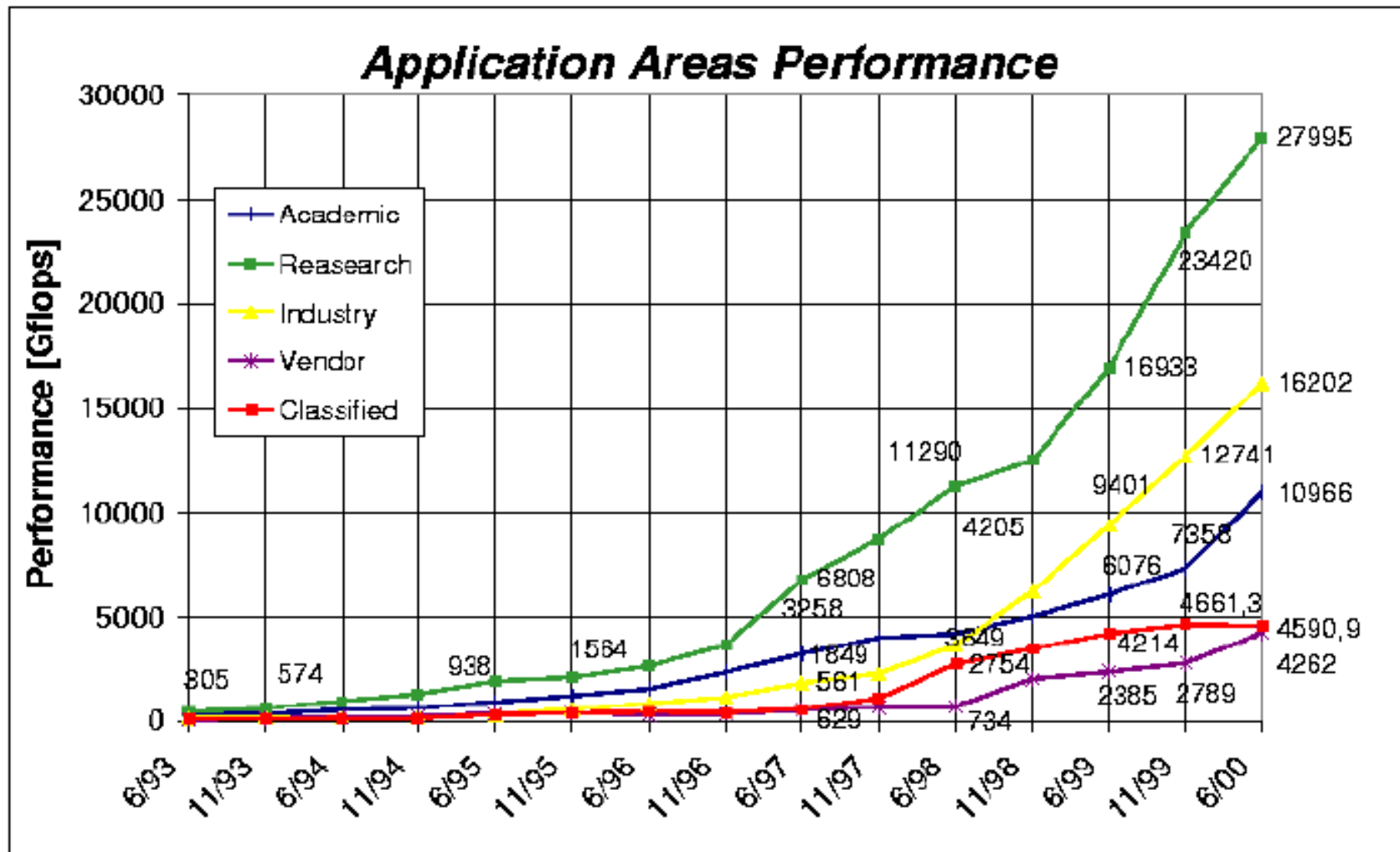


© Universität Mannheim

Diagrams by courtesy of [www.top500.org](http://www.top500.org)

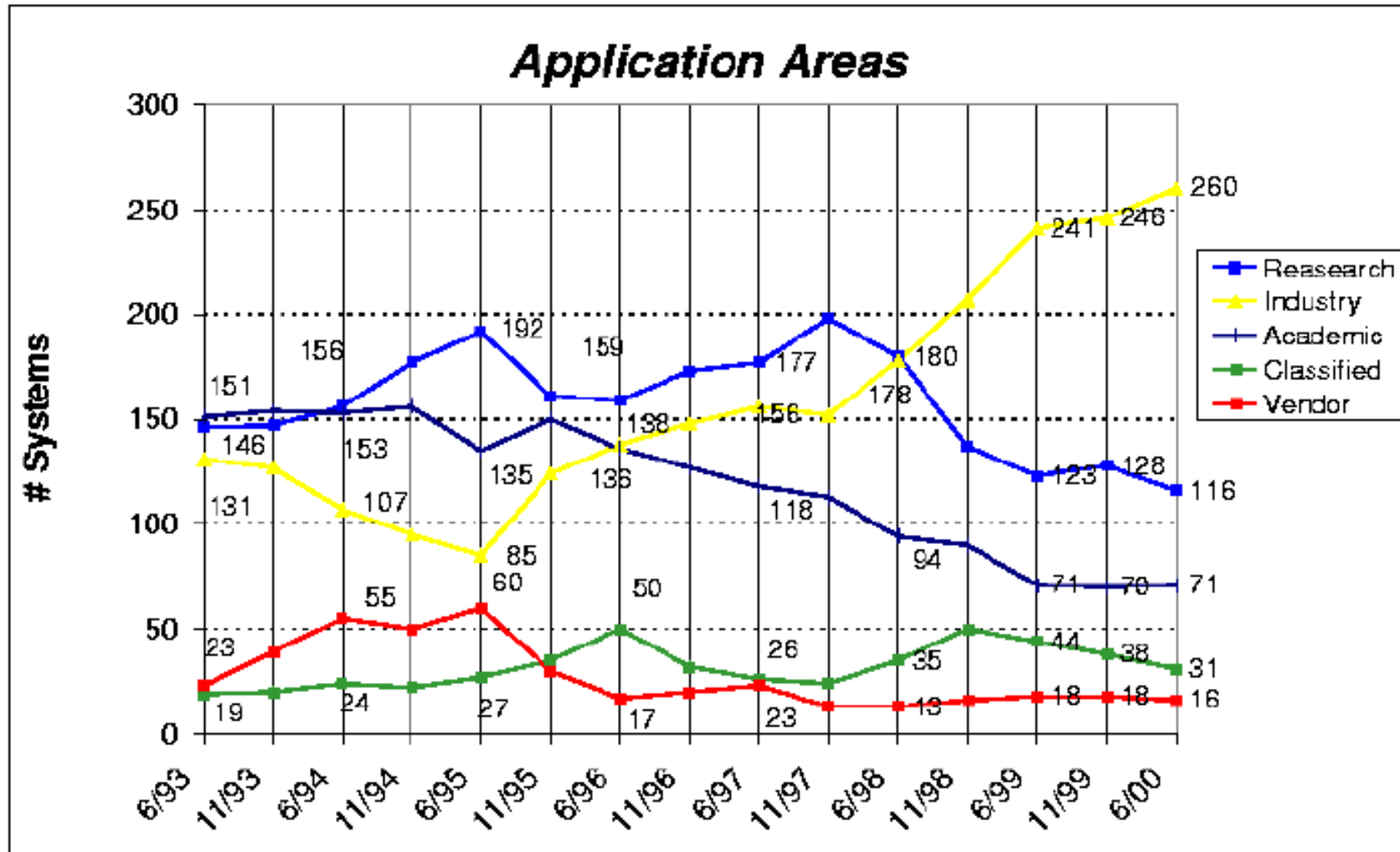


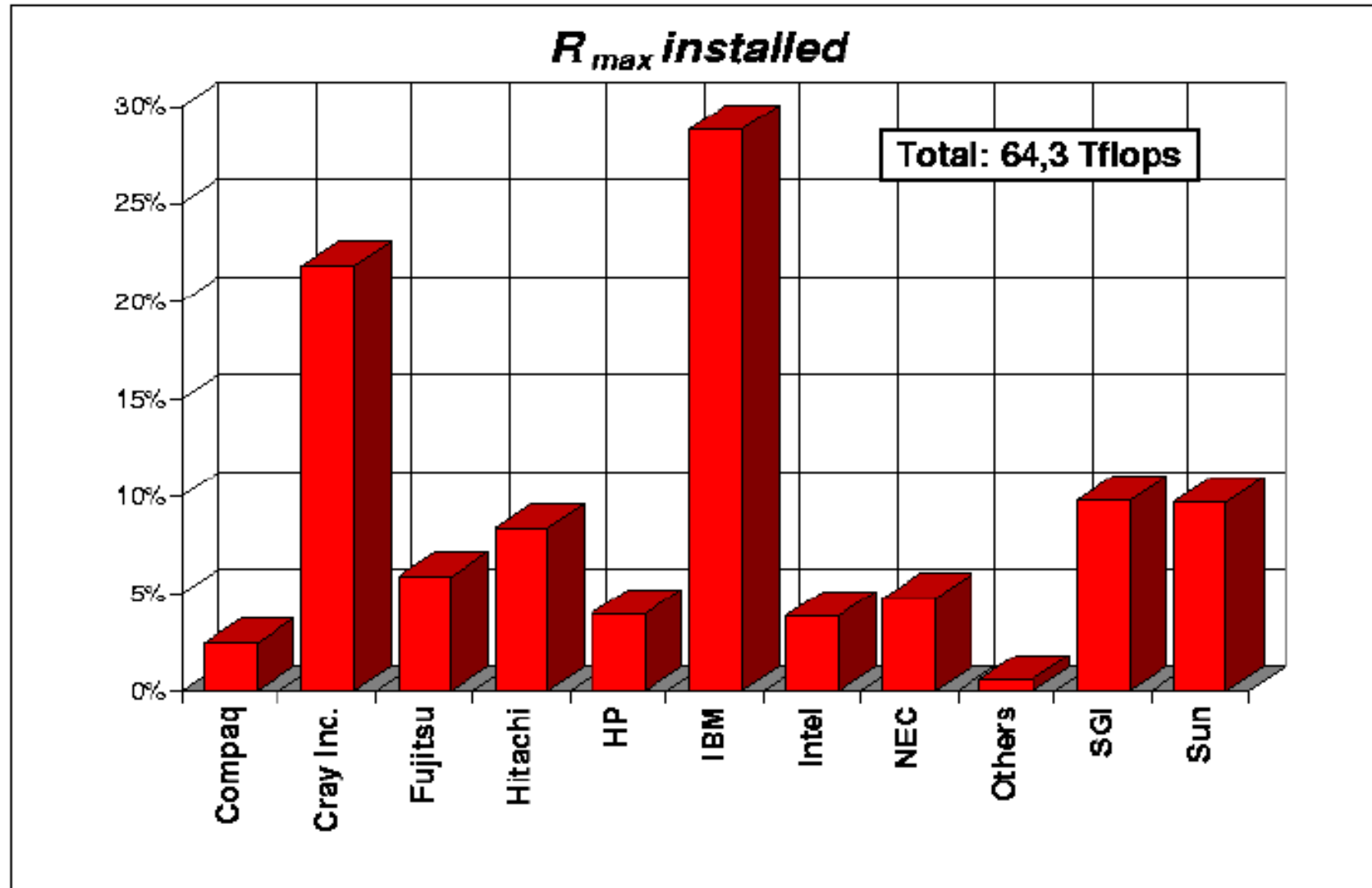




© Universität Mannheim

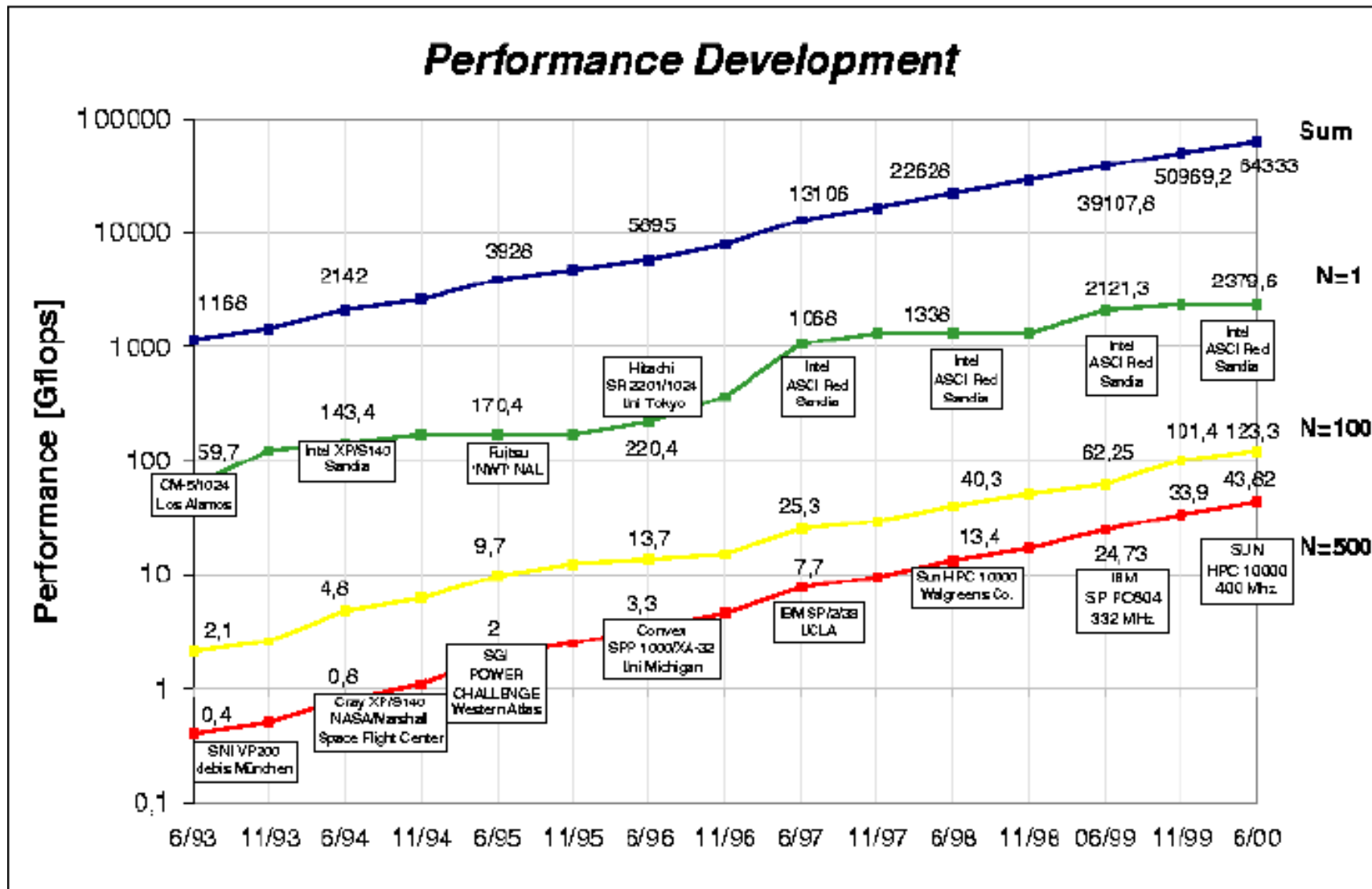
Diagrams by courtesy of [www.top500.org](http://www.top500.org)





TOP500

6/00



© Universität Mannheim

Diagrams by courtesy of [www.top500.org](http://www.top500.org)

# TOP500 Results

## Performance growth

- Factor 40 for  $R_{\max}$  in 7 years
- Moore's law: factor 2 in transistors every 18 months  
7 years correspond to factor 26

## System architectures

- Mainly distributed memory

## CPU technology

- CMOS off-the-shelf technology
- Fewer special processors and vector processors

# Summary Parallel Architectures

- We distinguish distributed memory and shared memory as basic architectural principles
- Mixtures like clusters of shared memory machines become popular
- Clusters of workstations also provide high performance
- Interconnection networks show various degrees of complexity, performance, and costs
- Performance evaluation is done with synthetic and real world benchmark programs
- The TOP500 list records all changes world wide and gives insight into change in technology

---

# Parallelization Concepts

- What is Parallelization?
- Parallelization Paradigms
- Algorithmic Aspects
- Example 1: Numerics
- Example 2: Computational Fluid Dynamics
- Example 3: Search Tree Algorithms

# What is Parallelization?

## Task

- Find parallelism that implicitly exists in the application and make it explicit
- Means: Distributed data and programs over the nodes
- Who? Programmer and/or compiler

Distribution produces new load (overhead)

→ Minimal overhead if not distributed

Distribution uses resources in an optimal way

→ Optimal performance if fully distributed

**Goal:** Use all resources and minimize overhead

# Requirements

In comparison with sequential software we need also

- Partitioning of the program into small parts
- Introduction of coordination and communication
- Mapping of the individual parts onto the components of the computer

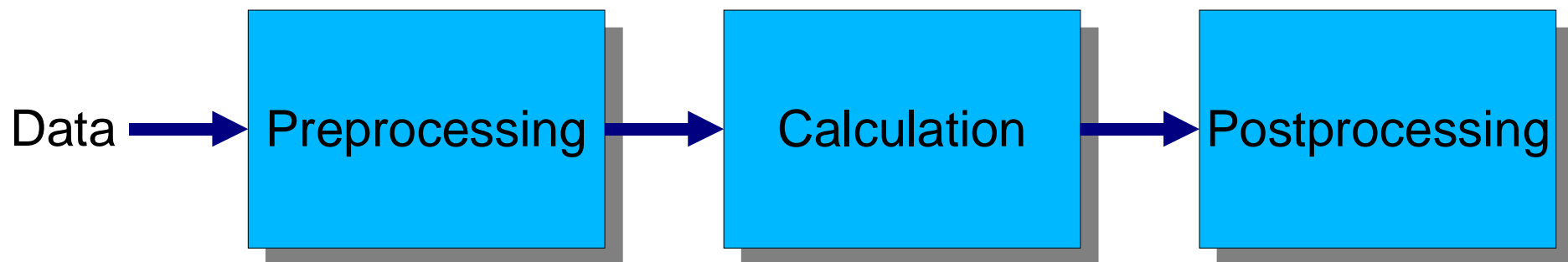
**Problems** (not covered in this lecture)

- Debugging (new types of errors)
- Performance evaluation (intrusion effects)
- Load balancing (for optimal performance)

# Parallelization Paradigms

## Code partitioning (also called macro pipelining)

- Distribute the program's code onto the nodes
  - Different code per node
  - Data varies according to flow of computation
  - Organizer: first/last process



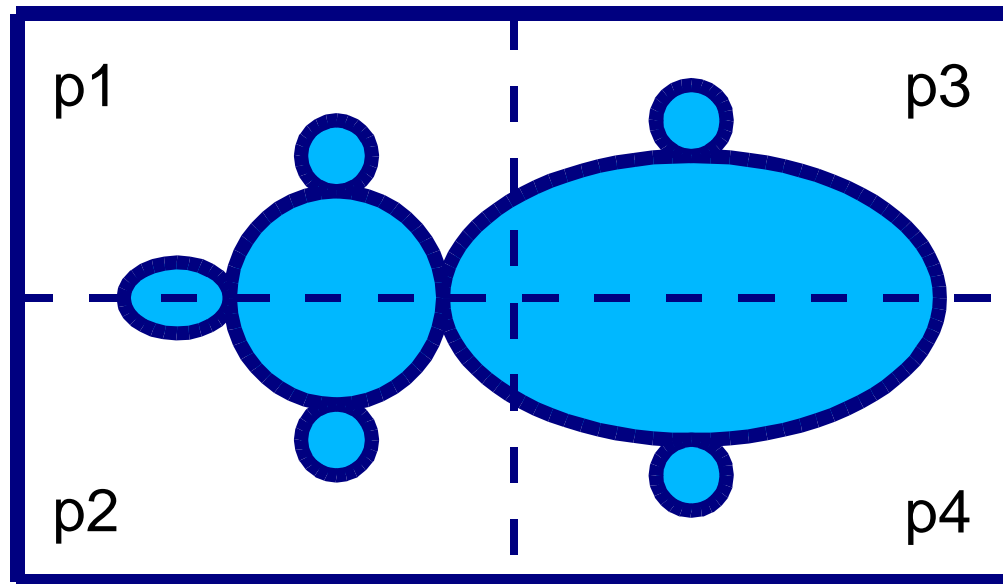
# Parallelization Paradigms (2)

- **Advantages** of code partitioning
  - Sometimes easy to be specified
  - Appropriate algorithms exist (e.g. FFT)
- **Disadvantages** of code partitioning
  - Multiple code files to be organized
  - Difficult to adapt to other target machines
  - Complex data communication schemes
  - Complex debugging
  - Complex load balancing

# Parallelization Paradigms (3)

## Data partitioning

- Distributed the data structures onto the compute nodes
  - Identical code on each node
  - Data regions vary from node to node
  - Organized by selected process



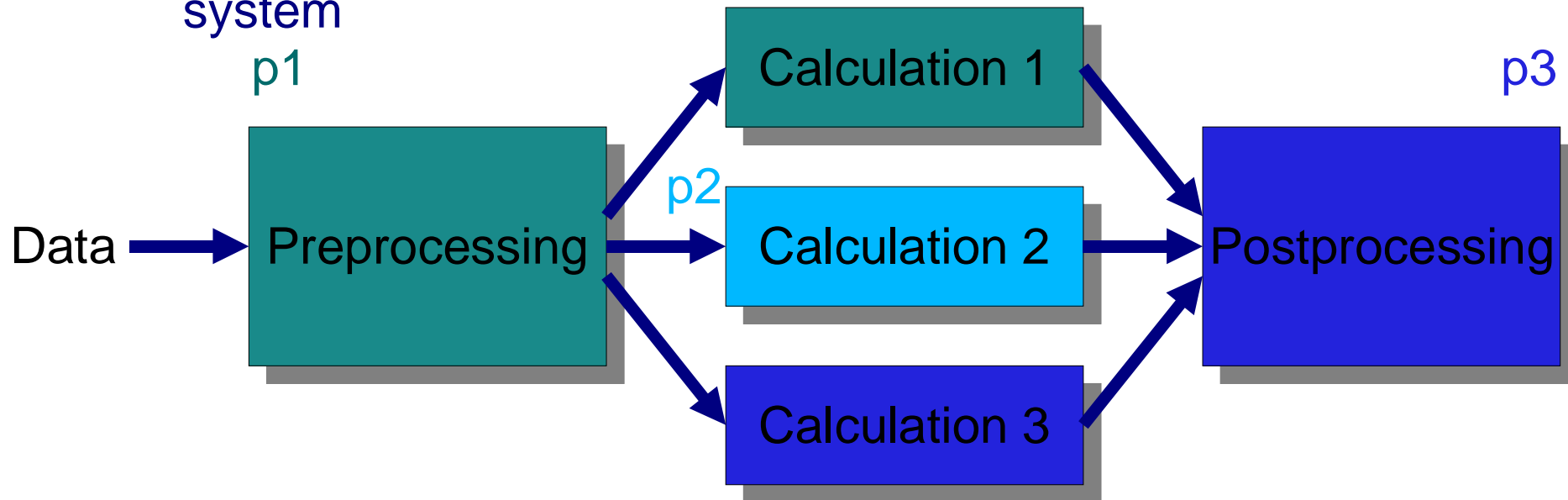
# Parallelization Paradigms (4)

- **Advantages** of data partitioning
  - Easy to program; only one source code
  - Easy to adapt to other target machines
  - Very often regular data exchange schemes
  - Not so complex debugging
- **Disadvantages** of data partitioning
  - Sometimes not appropriate for the application
  
- Data partitioning is the quasi standard of parallel programming  
Also called **SPMD** (single program, multiple data)

# Parallelization Paradigms (5)

## Mixed code and data parallelization

- This is where we want to go
  - Not yet standard
  - Adds advantages of both approaches
  - Needs multi processing in the target machine's operating system



# Algorithmic Aspects

## Bi-partite world of the programmer

- Numerical algorithms
  - Grand challenge algorithms: weather forecast, protein design, etc.
- Non-numerical algorithms
  - Search algorithms: theorem prover, game programs, etc.

# Numerical Algorithms

Computational fluid dynamics (CFD), numerical calculations, simulations, etc.

- Iterative algorithms
- Complete on a global condition
- Regular data structures (vectors, arrays, etc.)
- Regular communication structures
- Static process structure

# Non-numerical Algorithms

Data-base applications, artificial intelligence

- Search tree algorithms
- Irregular communication structures
- Irregular data structures (dynamic, garbage collection)
- Dynamic process structures

### Example 1: Numerics

We have three functions  $f()$ ,  $g()$ , and  $h()$

We have to apply the functions to a set of test candidates to compute the result  $h(g(f(x)))$

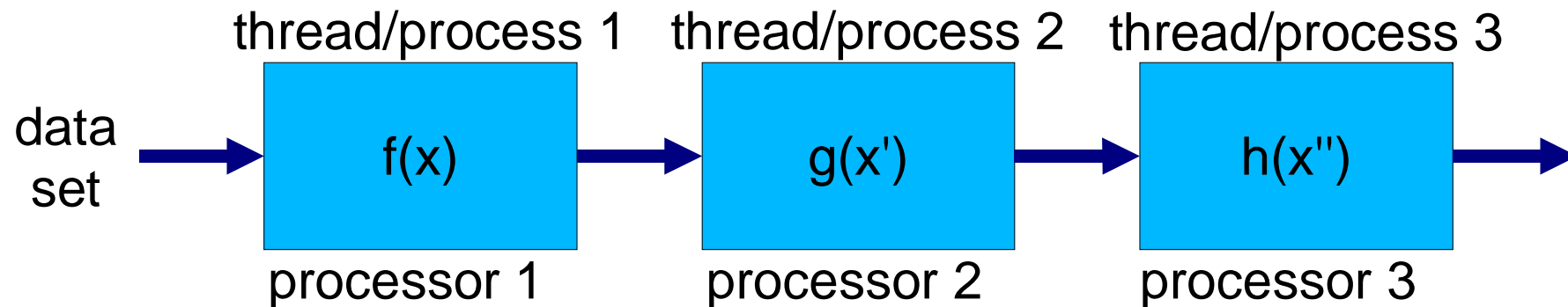
We will consider four variants:

- Code partitioning / data partitioning
- Shared memory / distributed memory

### Example 1: Numerics (2)

#### Code partitioning

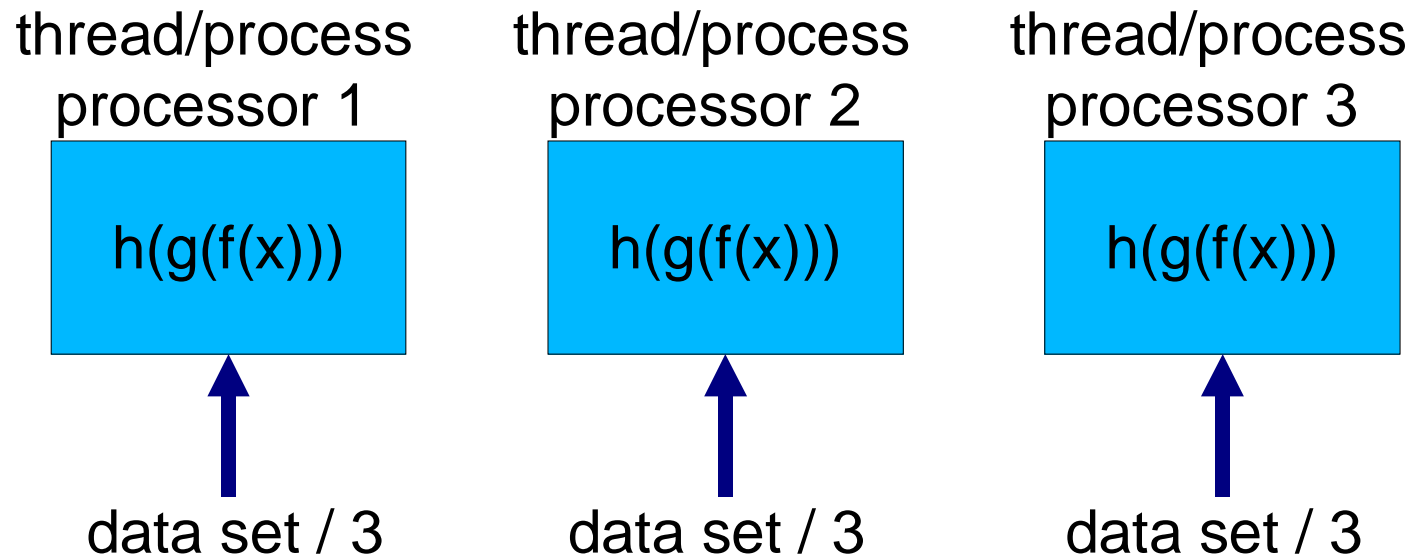
- Distribute the 3 functions onto three nodes
- Works in macro pipelining mode
- The set of test candidates is the data to be computed
- The example already shows that the only number of processors that can reasonably applied for this example is three



### Example 1: Numerics (3)

#### Data partitioning

- Replicate the functions on each node
- Distribute the set of test candidates onto the nodes



### Example 1: Numerics (4)

#### Code partitioning with shared memory

- Basic implementation

- Test candidates are stored in a vector
- We have three threads each running on its own processor
- Each thread computes one of the functions f, g, h
- The vector entries are replaced by the results of the functions being applied
- A variable indicates the active thread

- Disadvantage

→ This is **NOT** a parallel program (might sound obvious)

### Example 1: Numerics (5)

- Improvement to basic implementation
  - Counter for thread  $i$ , that indicates its position
  - Thread  $i$  may advance to counter of thread  $(i-1)$  minus 1
  
- **Advantage**
  - Good parallel implementation
  
- **Disadvantage**
  - Bad coordination/computation ratio: frequent inspection of the counter variable

### Example 1: Numerics (6)

- Second improvement to (a)
  - Improve the granularity of computation
  - Increment counter by 100, not by 1
  
- **Advantage**
  - Good parallel implementation
  - Better coordination/computation ratio
  
- **Disadvantage**
  - Longer phase to fill and empty pipeline

### Example 1: Numerics (7)

## Code partitioning with distributed Memory

### ■ Basic implementation

- Test candidates are stored in a vector
- We have three processes each running on its own processor
- Each process computes on of the functions f, g, h
- Process i computes all candidates, stores results in vector, and sends vector to process i+1

### ■ Disadvantage

→ This is again **NOT** a parallel program!

### Example 1: Numerics (8)

- Improvement to basic implementation
  - Process  $i$  sends computed values immediately to process  $i+1$
  
- **Advantage**
  - Good parallel implementation
  
- **Disadvantage**
  - Bad communication/computation ratio: frequent sending of candidates

### Example 1: Numerics (9)

- Second improvement to basic implementation
  - Improve the granularity
  - Send candidates in blocks of 100
  
- **Advantage**
  - Good parallel implementation
  - Better communication/computation ratio
  
- **Disadvantage**
  - Longer phases to fill and empty pipeline

### Example 1: Numerics (10)

#### Data partitioning with shared memory

- Basic implementation
  - Candidates are distributed in three blocks
  - We have three threads that compute  $h(g(f(x)))$
  - One variable per block signals end of computation
  - A dedicated thread organizes the output of results
- Advantage
  - Good parallel implementation
  - No access conflict with test candidates
- Disadvantage
  - Potential access conflict with thread object code

### Example 1: Numerics (11)

#### Data partitioning with distributed memory

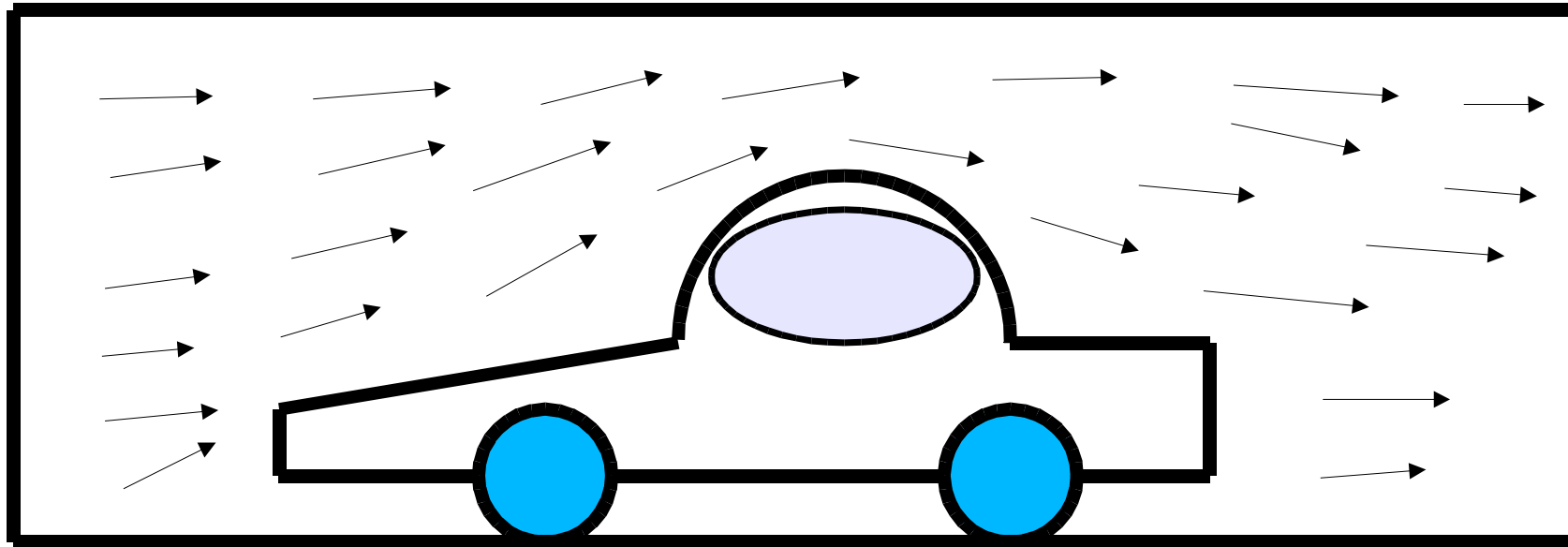
- Basic implementation
  - Test candidates are distributed onto the nodes
  - We have three processes on three processors that compute  $h(g(f(x)))$
  - Results are sent individually to processor 0
- Advantages
  - Good parallel implementation
  - Good communication/computation ratio
- Disadvantages
  - Programming of data distribution

### Example 2: Computational Fluid Dynamics

Simulation of wind tunnel experiments

Iterative computation with time step  $t$

- Microscopic approach: calculate particles
- Macroscopic approach: calculate pressure, temperature



### Example 2: Computational Fluid Dynamics (2)

First variant: distribute particles

- Each processor computes part of the particles
- **Disadvantage**
  - Difficult to determine neighbouring particles
- **Advantage**
  - Equal number of particles of implies good load balance

### Example 2: Computational Fluid Dynamics (3)

Second variant: distribute volume regions

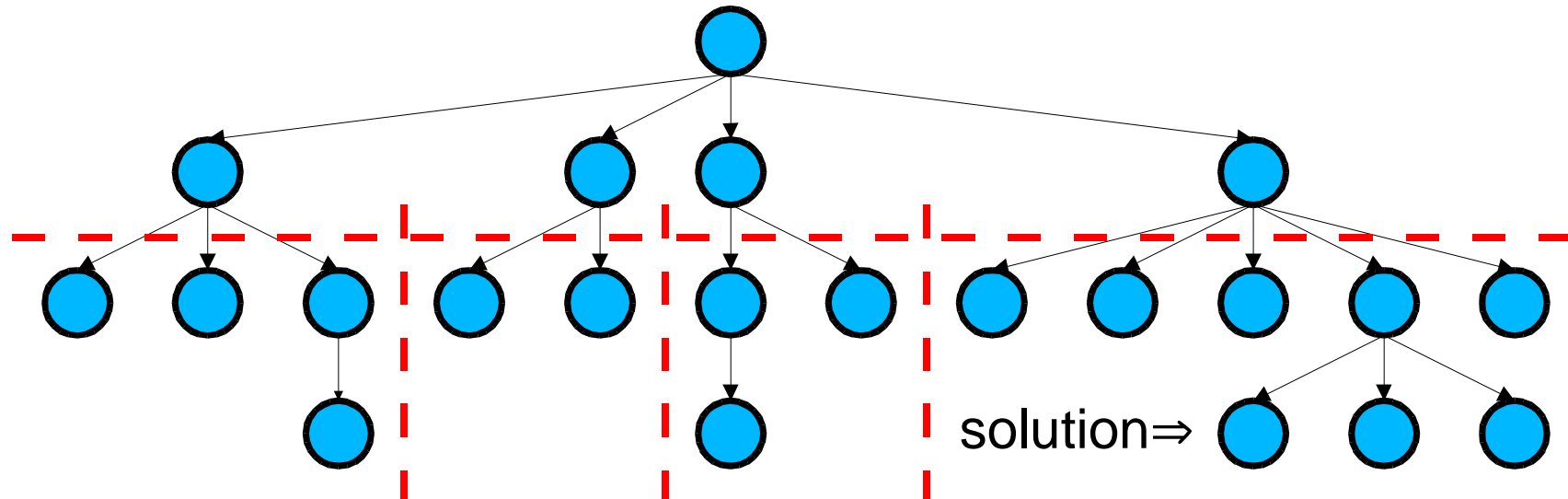
- Each processor computes part of the volume
- **Disadvantage**
  - Varying number of particles often implies bad load balance
- **Advantage**
  - Easy to determine neighbouring particles

### Example 3: Search Tree Algorithms

Several possibilities to continue at each position

Problems

- Level of solution is unknown
- Load balance between nodes
- Detection of program completion



### Example 3: Search Tree Algorithms (2)

#### Tree search with shared memory

- Processor computes tree up to level  $j$  and puts descriptions of possible continuations into a queue
- Available processors take descriptions and calculate the remaining tree

#### Good parallel algorithm

- Load balance: no problem
- Completion detection: set completion bit; other processor check regularly

### Example 3: Search Tree Algorithms (3)

#### **Tree search with distributed memory**

- Processor  $i$  computes tree up to level  $j$  and puts descriptions of possible continuations into a local queue
- Available processor contact processor  $i$ , get elements of the queue as messages and calculate the remaining tree

#### Good parallel algorithm

- Load balance: no problem, though more effort
- Completion detection: send completion message to all other processor; other processor check regularly

# Parallelization Concepts Summary

- By program partitioning and mapping onto the individual components we use the machine efficiently
- Parallelization by code and/or data partitioning
- We often distinguish numerical and non-numerical algorithms
- An efficient parallelization needs a lot of experience
- The optimal parallel program can not be derived from the listing of the sequential program
- The optimal parallelization depends on the target hardware and its characteristics

---

# Shared Memory Programming Models

- A View to Architecture
- Problems
- Programming Concepts
- ANSI X3H5 Shared Memory Model
- The POSIX Thread Model
- The OpenMP Standard
- A Comparison of Approaches

# A View to Architecture

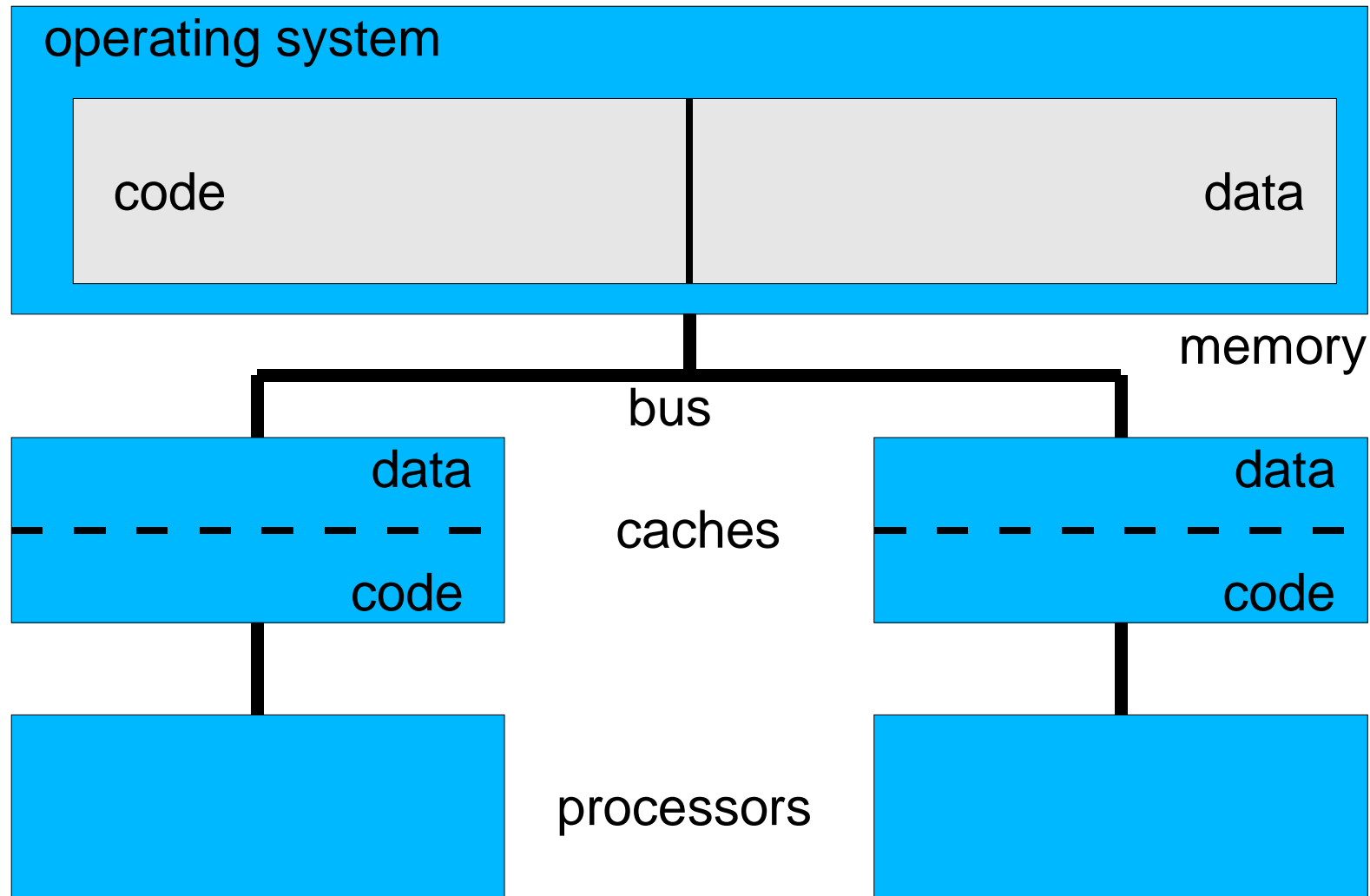
## Shared Memory

- Programming: single address space  
(shared memory programming)
- Architecture: real or virtual shared memory  
(shared memory architecture)

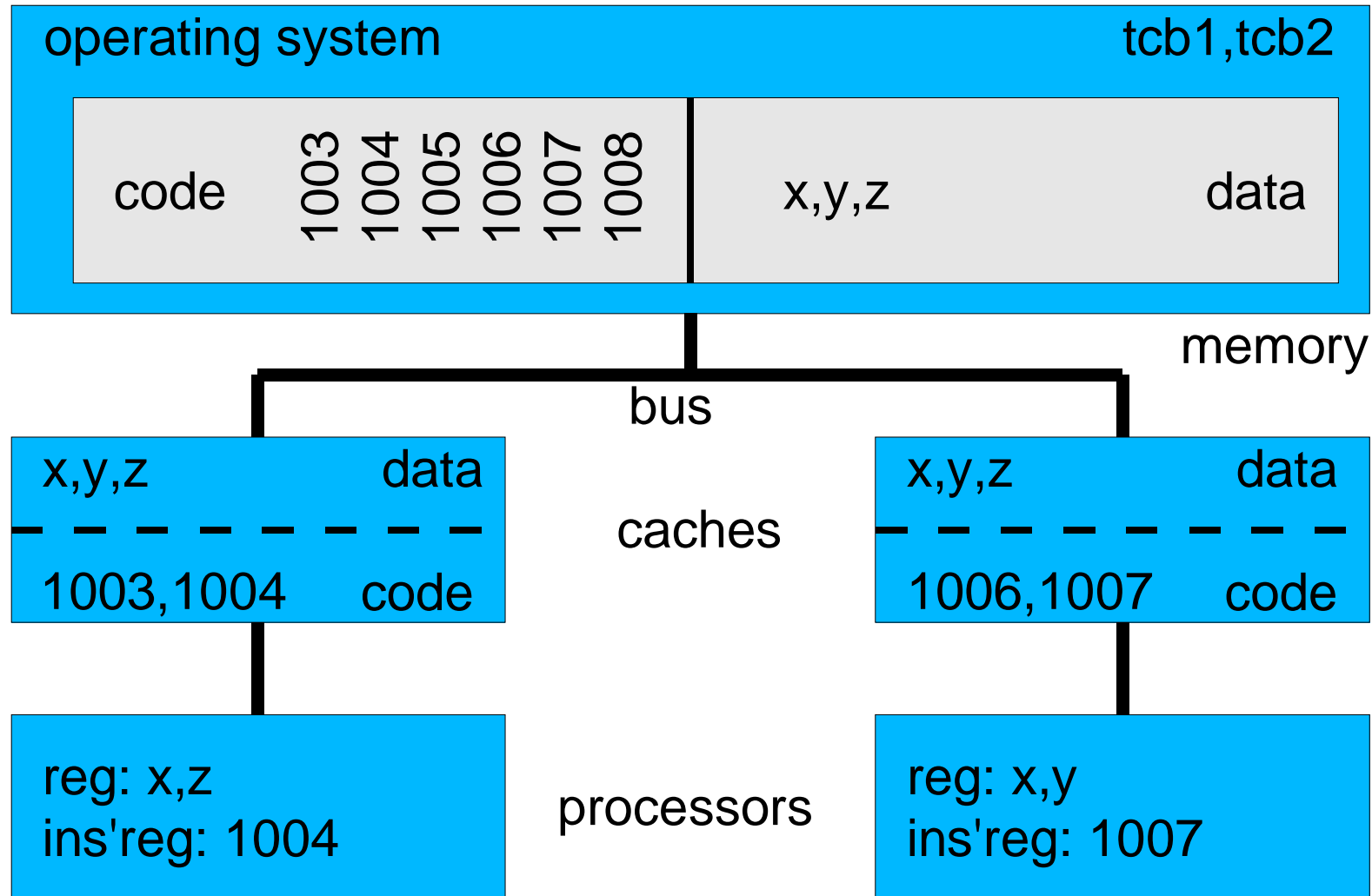
Often, both concepts appear together

Optimal efficiency for physically shared memory

## A View to Architecture: Static View



## A View to Architecture: Dynamic View



# Problems

- Processes access same program code
  - Not critical as access is read only
  - However, bus access conflicts appear
- Processes access same variables
  - Write accesses have to be synchronized
- Variables are replicated on several locations
  - Consistency problem (see SMP architectures)

# Programming Concepts

Shared memory

- No partitioning and distribution of data

Asynchronous program flow

Explicit synchronisation necessary

- Error prone and difficult to tune

No widely accepted standard

- Programs are difficult to port

Good for irregular data structures and pointer constructs

# Programming Concepts (2)

## Language standards

- X3H5: ANSI standard
- POSIX Threads standard
- OpenMP: future vendor standard (?)
  
- Proprietary approaches: SGI Power C, Cray MPP

# ANSI X3H5 Shared Memory Model

- Standardized 1993
- Never became a de facto standard  
however, many vendors borrowed concepts of X3H5
- X3H5 defines a conceptional programming model
- X3H5 defines 3 language bindings:  
C, Fortran 77, Fortran 90

### X3H5 Constructs for Parallelism

- X3H5 uses a set of specific language constructs
- No explicit definition of the number of threads that execute
- Programs start in sequential mode with a master thread
- Constructs are `parallel` `psection` `psingle` `pdo`

## X3H5 Example 1

```
program main
```

```
  A
```

```
  parallel
```

```
  B
```

```
  psection
```

```
  section
```

```
    C
```

```
  section
```

```
    D
```

```
  end sections
```

```
  psingle
```

```
    E
```

```
  end psingle
```

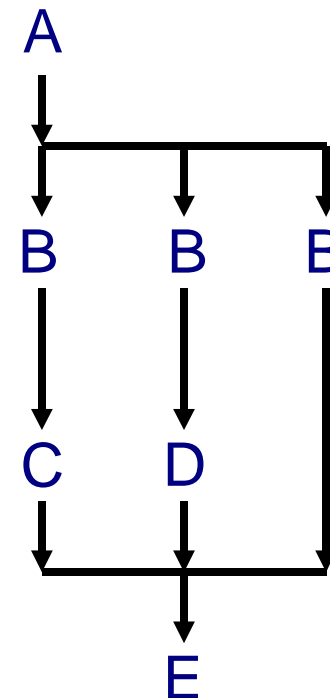
```
  end parallel
```

```
end
```

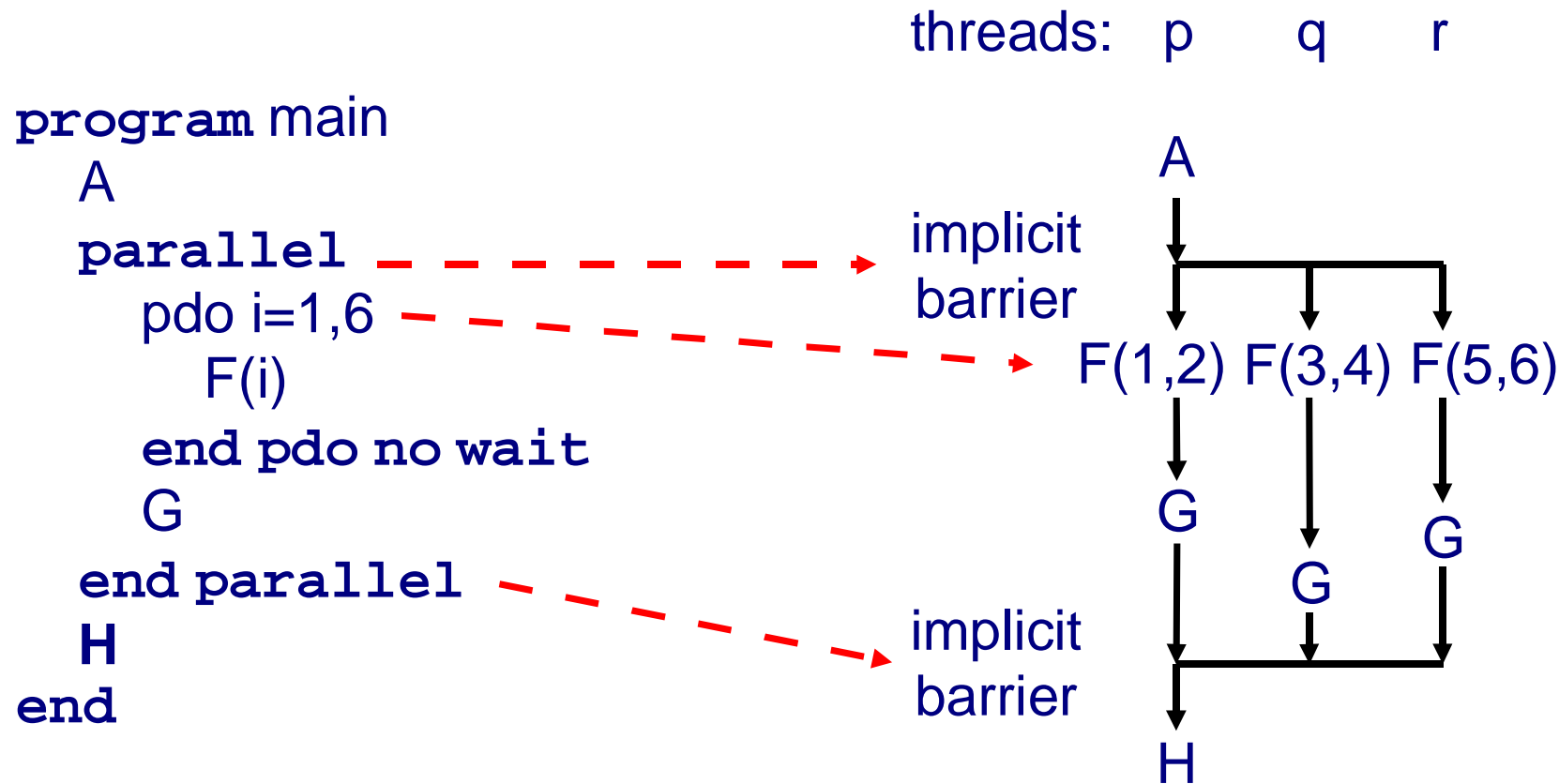
threads: p q r

implicit  
barrier

implicit  
barrier



## X3H5 Example 2



### X3H5 Semantics

#### `parallel`

- a number of threads are started
- change to parallel execution mode
- further constructs are executed in parallel

#### Worksharing constructs

- `psection`: multiple code multiple data parallelism  
corresponds to code partitioning
- `psingle`: only one thread executes (e.g. input/output)
- `pdo`: single program multiple data parallelism

### X3H5 Semantics (2)

Worksharing constructs support load balancing

- e.g. with `pdo` threads execute indices according to availability

Implicit barriers

- `parallel`, `end parallel`, `end psections`,  
`end pdo`, `end psingle`
- enforces memory consistency

Explicit barriers

- `no wait`-construct; faster, more error prone

### Thread Interaction

Four classes of synchronization constructs

- **latch, lock, event, ordinal**

Attributes shared/private for variables

- **private**: variable is thread local and invisible to others
- **shared**: variable is visible to other threads

Synchronization objects

- **Ordinal**: thread *i* can enter critical region only after threads 1,...,*i*-1
- **Latch**: **critical region** [<latch\_variable>]  
code  
**end critical region**

# Special Constructs for Critical Regions

We need atomic test-and-set

Thread A

**while (x>0) do done**

**x=1**

<critical region>

x=0

Thread B

**while (x>0) do done**

**x=1**

<critical region>

x=0

Operating system support

- a) block requesting thread
- b) continue with other ready to compute thread
- c) resume thread when access is possible

### Small Example to Critical Regions

- Program completion time is 10,000s
- There is a critical region with 10s
- We work with 100 threads
  
- With 100 threads we expect 10,000s/100 run time
  - Completion time for the critical region:  $100 * 10s = 1,000s$
  - Remaining execution time parallelized:  $9,990s / 100 = 99.9s$
  - Speed-up:  $10000s / 1099.9s \sim 9$
  - **Critical region too long - number of threads too high**

### Generation of the Parallel Program

**Compiler** knows syntactical constructs and re-arranges the source code

- Variable replication, insertion of barriers

**Libraries** with special functions get linked

- Functionality of synchronization constructs, etc.

**Execution** on computer with shared memory model and any number of processors and threads

**Mapping** of threads to processors by runtime system

# The POSIX Thread Model

- IEEE standardisation 1995
- Slightly different implementations in different operating systems
- Similar to threads in Solaris
- Designed for SMP systems with low number of processors
- Pure library approach
- No compiler support necessary/available

# Threads Programming Model

- Manual thread creation
  - `pthread_create ( ... (* myroutine) ... )`
  - low abstraction level
- Synchronisation
  - Mutex variable (mutual exclusion)
  - Condition variable
- No high level abstraction constructs
  - no `parallel`, `psection`, `pdo` or similar

# The OpenMP Standard

- By vendors like DEC, Intel, IBM, Potland Group, and others
- OpenMP initially for Fortran in 1997
- OpenMP has compiler directives, libraries, and environment variables
- OpenMP defines API for shared memory programming under Unix and WindowsNT
- Similiar to X3H5 but more constructs

### OpenMP Orphan Directive

```
subroutine compute(field, ispectrum)
!$OMP DO
  do i=1,npoints
    index = field(i)*nzone+1
!$OMP ATOMIC
    ispectrum(index) = ispectrum(index)+i
  enddo
!$OMP END DO NOWAIT
```

- Orphan constructs do not belong to the lexical structure of the programs
- Supports step-wise parallelization

# OpenMP Data Environments

- Useful to define what is shared and what is exclusive
- More sophisticated as with X3H5
  
- E.g.
  - `!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(x,y)`  
all variables private except x and y
  - `!$OMP PARALLEL REDUCTION(+:sum)`  
create thread local reduction variable sum

### A Comparison of Concepts

	X3H5	pthread	OpenMP
scalability	no	maybe	yes
Fortran binding	yes	no	yes
C binding	yes	yes	planned
high abstraction	yes	no	yes
performance oriented	no	no	yes
data parallelism	yes	no	yes
portability	yes	yes	yes
incremental parallelization	yes	no	yes
vendor support	no	UNIX/SMP	starting

# Shared Memory Programming Models Summary

- Shared memory programming model for architectures with shared and distributed memory
- Consistency problems here not discussed but important
- Synchronization with language constructs
- Library oriented approach POSIX threads
- Language oriented approaches X3H5 and OpenMP
- Constructs for incremental parallelization of sequential programs
- OpenMP is vendors' future standard
- Today's situation: heterogeneous, despite OpenMP

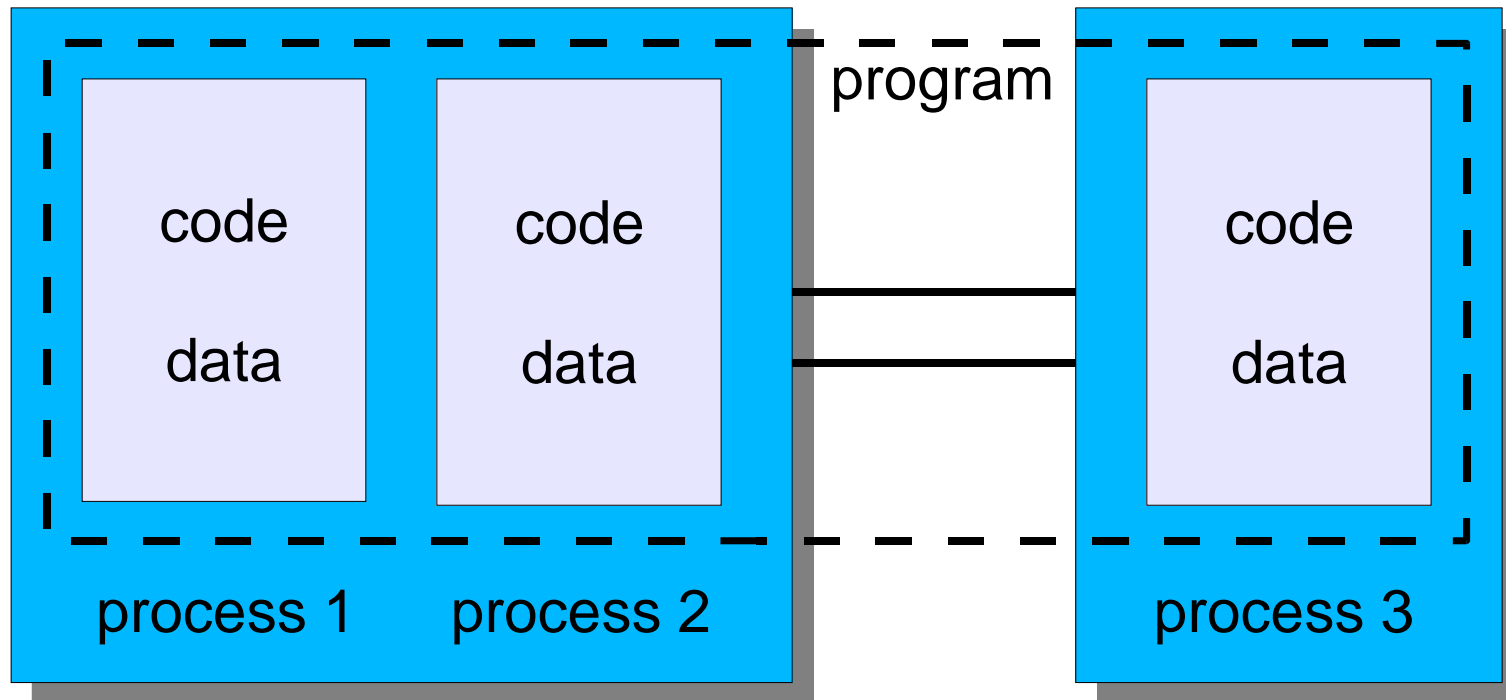
---

# Message Passing Programming Models

- A View to Architecture
- Problems
- The Message Passing Interface (MPI)
- Goals and Contents of the Specification
- Point to point communication
- Derived Data Types, Global Computations, Groups, Contexts, Process Topologies
- Evaluation of MPI
- MPI-2
- Comparison with OpenMP

# A View to Architecture

The code of the processes of the program may be identical or different. Usually it's identical but has some if-statements for the organizing process



# Problems

- Loading the program to many nodes
  - With workstations clusters also: binary incompatibility
- Starting the program at the same time
- Processes must know each other
- Communication between processes
- Optimization of communication
- Communication schemes. e.g. ring communication

### Loading and Starting

Similar to thread creation

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

If there is 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 ( <receiver_id>, <data> );`
- `broadcast ( <data> );`

#### Receiving of messages

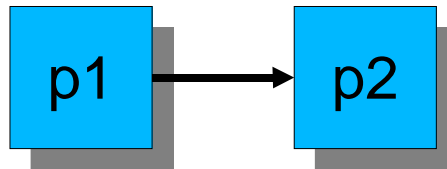
- `receive ( <sender_id>, <data> );`
- `testreceive ( <sender_id> );`

Characteristics: Programmer adds communication calls;

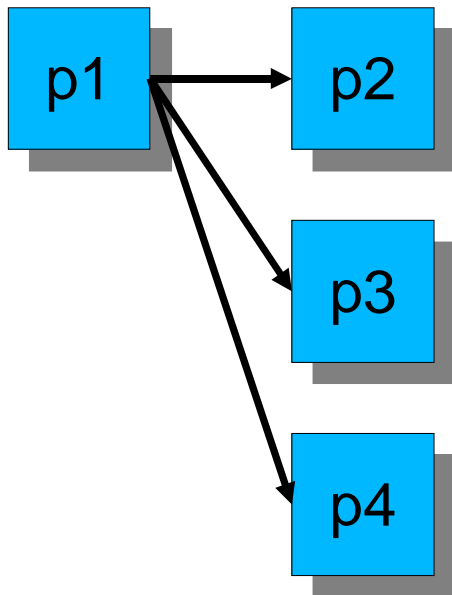
- High effort; high efficiency possible; highly error prone

## Communication Schemes

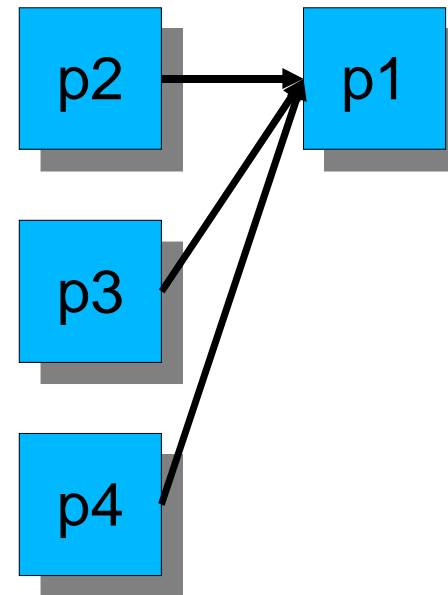
direct communication



broadcast / multicast



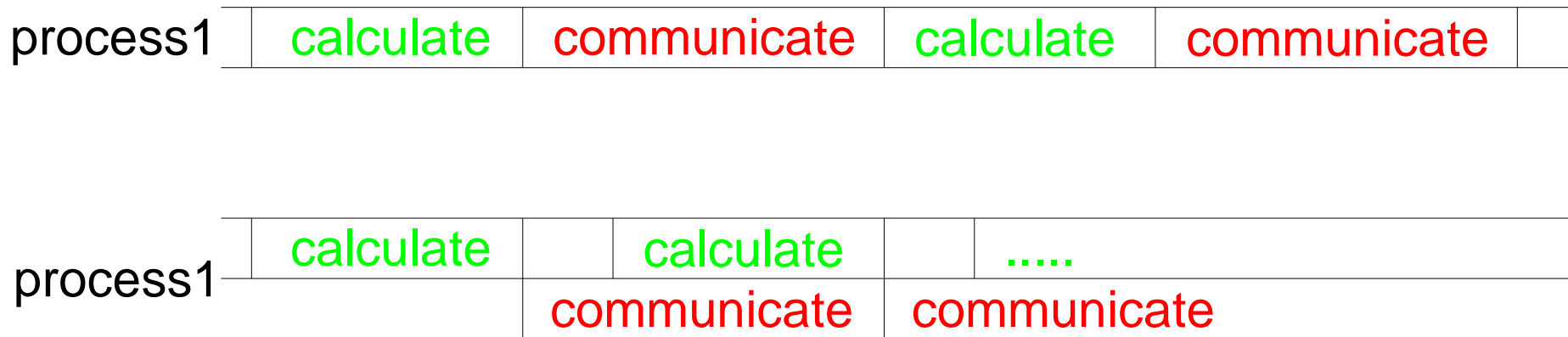
(indirect communication)



# Communication Optimization

Handle communication and calculation concurrently

Needs software and hardware support



# Existing Approaches

- P4, PARMACS and others
  - This is the history of message passing libraries
- Parallel Virtual Machine (PVM)
  - **Library** approach
  - Exists for all machines, also regular workstations
  - De facto standard for workstation clusters
- Message Passing Interface (MPI)
  - **Specification** of a library for any architecture
  - De facto standard on super computers with distributed memory
  - Implementations for workstations also exist (MPICH)

# The Message Passing Interface (MPI)

- Driven by the MPI forum (vendors, academia, ...)
- Started 1992
- First MPI standard 1995
- MPI-2 standard 1997
- Advantage of a standard: portability, simplicity (?)
  - Before we had one dozen of competing approaches

### MPI's Goals

- Specification of an API (application programming interface)
- Support of efficient communication methods
- Support of heterogeneous environments
- Language bindings for Fortran 77 and C/C++
- Constructs that are already known to users
- Interface semantics must be independent of the language binding
- Must support a thread-safe implementation

### What's in MPI?

- Point to point communication
- Collective communications
- Process groups
- Communication contexts
- Process topologies
- Inquiry functions for the environment
- Profiling interface

### What's **not** in MPI?

- Explicit operations for shared memory
- Support by the operating system
  - E.g. interrupt driven communication
- Explicit support for process management
- Parallel input/output

MPI is message passing **exclusively**

MPI-2 will cover some of above issues

### MPI Specification Method

- Function calls are specified in a language independent way
- Function arguments are annotated IN, OUT, and INOUT

E.g. MPI\_WAIT (request, status)

INOUT request

OUT status

C:       int MPI\_WAIT (MPI\_Comm\_request \*request,  
                      MPI\_Status \*status)

F77: MPI\_WAIT (REQUEST, STATUS, IERROR)

INTEGER REQUEST,

STATUS (MPI\_STATUS,SIZE), IERROR

# Point to Point Communication

## Sending

MPI\_SEND (buf, count, datatype, dest, tag, comm)

IN buf /\* start address of send buffer \*/

IN count /\* number of elements in buffer \*/

IN datatype /\* data type of elements to be sent \*/

IN dest /\* receiver id: rank in process group \*/

IN tag /\* message marker \*/

IN comm /\* communicator (group, context) \*/

- Data types: int, long int, float, char
- Messages consist of contents and envelope
  - envelope: (source, dest, tag, context)

# Point to Point Communication (2)

## Receiving

MPI\_RECV (buf, count, datatype, source, tag, comm, status)

OUT buf /\* start address of receive buffer \*/

IN count /\* number of elements in buffer \*/

IN datatype /\* data type of elements to be received \*/

IN source /\* sender id: rank in process group \*/

IN tag /\* message marker \*/

IN comm /\* communicator (group, context) \*/

OUT status /\* result of receive operation \*/

### ■ Reception is envelope driven

- Wildcards: MPI\_ANY\_SOURCE, MPI\_ANY\_TAG

- Get info with MPI\_GET\_SOURCE(), MPI\_GET\_TAG()

### Point to Point Communication (3)

#### Semantics of communication

- Message sequence is maintained

#### Data conversion

- In heterogeneous environments automatic conversion

#### Modi

- Usually: local blocking
- Ready communication: sending may only be invoked when receiver is already waiting (more efficient implementation possible)
- Synchronous communication: global blocking (SEND) completes when receiving operation starts

### Point to Point Communication (4)

#### Non-blocking communication

- better efficiency by communication/calculation overlap

#### Important aspects

- blocking / non-blocking  
when does the call return?
- synchronous / asynchronous  
when is the call completed?

#### Concept

- each call gets a reference
- inquiry functions can check the reference

### Point to Point Communication (5)

#### Non-blocking communication

- `MPI_ISEND (...)`
- `MPI_IRECV (...)`
- `MPI_TEST (request, flag, status)`      non-blocking
- `MPI_WAIT (request)`                      blocking
- `MPI_CANCEL (request)`                      cancel call

# Derived Data Types

## Usage

- Messages with mixed data types
- Messages with disconnected regions

Packing / unpacking needs computational effort

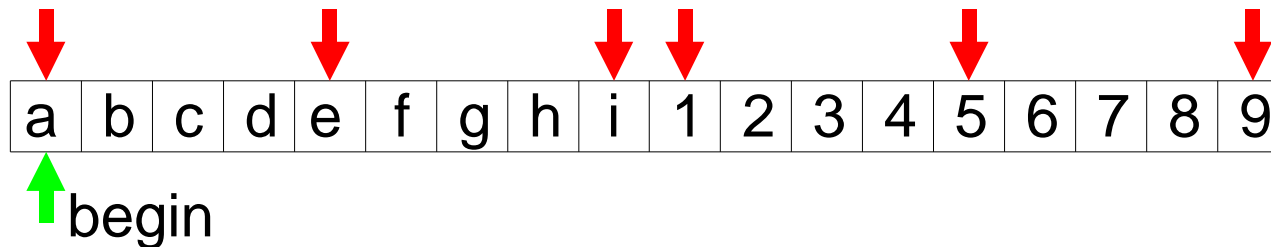
Efficiency depends of hardware support

- E.g. direct memory access (DMA)

### Derived Data Types (2)

Example: two matrices with complex numbers

Task: send the two diagonals of the matrices



MPI\_TYPE\_VECTOR (3 blocks, 1 element/block, 4 blockstride,  
MPI\_COMPLEX, diag)

MPI\_TYPE\_HVECTOR (2 blocks, 1 element/block,  
9\*sizeof(MPI\_COMPLEX),diag, doubleddiag)

MPI\_TYPE\_COMMIT (doublediag)

MPI\_SEND (begin, 1, doubleddiag, other, mytag, comm)

# Global Computations

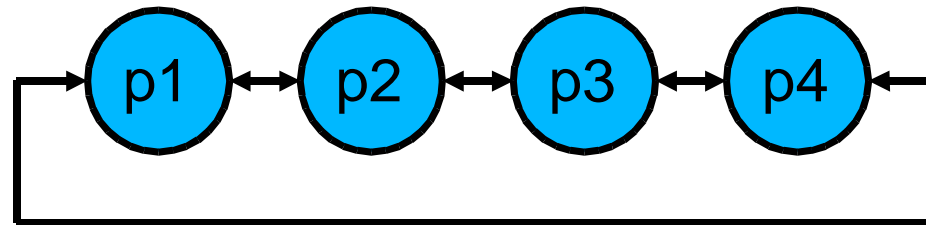
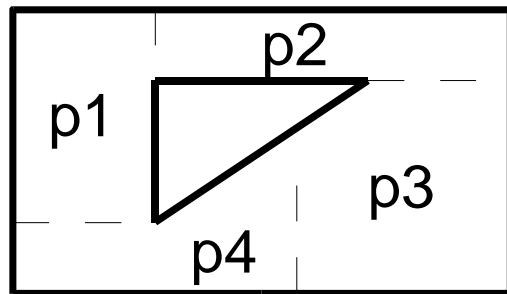
- Often all processes need to apply the same function, e.g. sum
- Function `MPI_REDUCE` (... , op,...)
  - Each process adds its own operands
  - In the end each process knows the final result
  - Operations: max, min, sum, product, AND, OR, XOR
- Any evaluation sequence allowed
- Sometimes supported by hardware of a super computer
- Own operations may be linked in

# Group, Context, Communicator

- New concept that did not exist before
- Problem
  - Third party software providers implement parallel libraries that use message passing
  - Tags and sender/receiver ids must not interfere with the user program
- Solution
  - Groups compose sets of processes that belong together
  - Contexts distinguish different logical parts of the program
  - Communicators combine group and context for a message passing operation

# Process Topologies

- Problem: Rank says nothing about logical relation



- User view
  - Only specific message passing schemes appear
  - Access to neighbours via symbolic names: left, right
- System view
  - Systems require optimized mapping of process to processor  
process graph to processor graph with minimal communication
- MPI supports topology management

# Evaluation of MPI

- Specification exclusively for message passing
- Many functions
- No process management
- No dynamic process concept (static rank)
  - No programs with varying number of processes possible (in contrast to PMV)

# MPI-2

- MPI-2 is an extension to MPI, not a new version
- Comprises some clarifications to MPI and extensions
- Defines important new feature: process management (before each vendor did it as it wanted)
- Important extension: input/output (idea: make it corresponding to send/receive)
- Disadvantage: many new functions

# Comparison of MPI with Pthreads and OpenMP

	MPI	pthread	OpenMP
scalability	yes	maybe	yes
Fortran binding	yes	no	yes
C binding	yes	yes	planned
high abstraction	no	no	yes
performance oriented	yes	no	yes
data parallelism	no	no	yes
portability	yes	yes	yes
incremental parallelization	no	no	yes
vendor support	yes!	UNIX/SMP	starting

# Message Passing Programming Models Summary

- Relevant problems with message passing:  
Communication schemes, process management, efficiency
- MPI specifies an API for message passing
- Point to point communication in many variants
  - blocking/non-blocking
  - synchronous/asynchronous
- Derived data types ease communication
- Groups and contexts separate program parts from each other
- MPI-2 extends MPI with additional concepts like I/O

---

# Data Parallel Programming Models

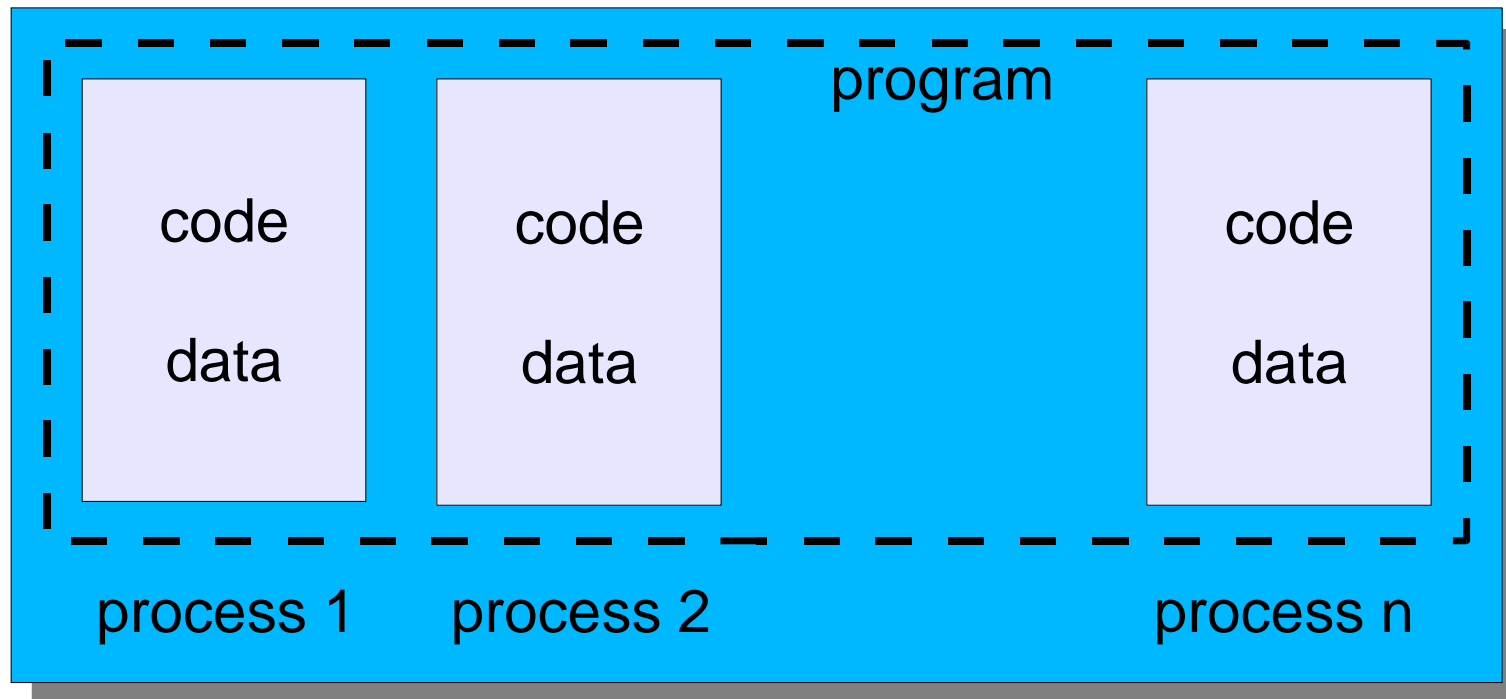
- A View to Architecture
- Existing Approaches
- High Performance Fortran (HPF)
- The HPF Model
- Examples for Data Distribution
- Data Alignment and Distribution
- Data Parallel Assignments
- Evaluation of HPF
- Comparison of HPF with MPI and OpenMP
- Comparison of the three Programming Paradigms

# A View to Architecture

Data parallel programming aims at message passing

A mapping to shared memory architectures is also possible

parallel computer / workstation cluster



### Characteristics of Data Parallelism

- Programmer's view: one control flow (program)
- Parallel operations on data structures
- Weak synchronization
- Single address space
- Implicit interactions
- Implicit data distribution and arrangement

# Existing Approaches

- Fortran90
  - Improved Fortran77
  - Exists since 1991
- High Performance Fortran (HPF)
  - Becoming the de facto standard
- Further approaches
  - Fortran D, Vienna Fortran
  - HyperC, pC++

# High Performance Fortran (HPF)

- 40+ institutions from industry, research, and academia
- HPF 1.0: May 1993
- HPF 1.1: November 1994
- HPF 2.0: January 1997
- Leading idea:
  - Improvement of standard Fortran to make it suited for high performance applications

### HPF Goals

- Language extension for Fortran for a data parallel programming style
- High performance on MIMD and SIMD systems
- Code optimization for various architectures
- Close relation to Fortran77 and Fortran90
- Simple language extension
- Open interface to other programming languages

### New Concepts in HPF

- FORALL-construct
- Intrinsic functions
- Data distribution
- Extrinsic procedures to connect with other programming languages

### Fortran90 Binding

HPF extends Fortran90: Fortran90 already has field computation and dynamic memory management

Four new categories

- New compiler directives (comments); manipulate performance, not semantics
- New language constructs with own semantics
- Useful library routines
- Restrictions to Fortran90: some constructs are incompatible with HPF

### What is in HPF?

- Data distribution: optimal efficiency only with locality
  - ALIGN statement to align data sets
  - DISTRIBUTE statement to distribute data to nodes
- Data parallel execution
  - FORALL statement for parallel assignments
  - INDEPENDENT statement to control program flow
- Intrinsic functions
  - E.g. to control the environment
- Extrinsic functions
  - E.g. to add message passing to the HPF program

# The HPF Model

- Main goal: Portability across many architectures
  - Highly efficient programs from one machine should at least be efficient programs on other machines
- Factors of influence
  - Parallelism in the application / necessary communication
  - Communication depends from data distribution
- HPF compiler generates programs for the nodes
  - Access to non-local data is mapped to message passing operations (in most cases MPI)

# Examples for Data Distribution

### Code fragment

```
REAL a(1000), b(1000), c(1000), x(500), y(0:501)
INTEGER inx(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: a, b, inx
!HPF$ DISTRIBUTE (CYCLIC) ONTO procs :: c
!HPF$ ALIGN x(i) WITH y(i+1)
```

### Examples Data Distribution (2)

#### Data partitioning

a(1)...a(100)  
b(1)...b(100)  
inx(1)...inx(100)

c(1),c(11),c(21)...

x(i),y(i+1)

processor 1

a(401)...a(500)  
b(401)...b(500)  
inx(401)...inx(500)

c(5),c(15),c(25)...

x(j),y(j+1)

processor 5

a(901)...a(1000)  
b(901)...b(1000)  
inx(901)...inx(1000)

c(10),c(20),c(30)...

x(k),y(k+1)

processor 10

### Examples Data Distribution (3)

#### Assignments and their execution

- $a(i) = b(i)$ 
  - all elements with identical index on same node
  - no communication
- $x(i) = y(i+1)$ 
  - all elements for assignment on same node
  - no communication

$a(1) \dots a(100)$   
 $b(1) \dots b(100)$   
 $x(i), y(i+1)$

$a(401) \dots a(500)$   
 $b(401) \dots b(500)$   
 $x(j), y(j+1)$

$a(901) \dots a(1000)$   
 $b(901) \dots b(1000)$   
 $x(k), y(k+1)$

### Examples Data Distribution (4)

#### Assignments and their execution

- $a(i) = c(i)$ 
  - $a$  is distributed blockwise,  $c$  cyclic
  - 10% of all elements are located on the same node
  - almost always communication necessary

$a(1)\dots a(100)$   
 $c(1), c(11), c(21)\dots$

$a(401)\dots a(500)$   
 $c(5), c(15), c(25)\dots$

$a(901)\dots a(1000)$   
 $c(10), c(20), c(30)\dots$

### Examples Data Distribution (5)

#### Assignments and their execution

- $a(i) = a(i-1) + a(i) + a(i+1)$ 
  - 98% of all elements are located on the same node
  - few communication necessary
- $c(i) = c(i-1) + c(i) + c(i+1)$ 
  - neighbouring elements always on different node
  - the two neighbouring elements require communication

### Examples Data Distribution (6)

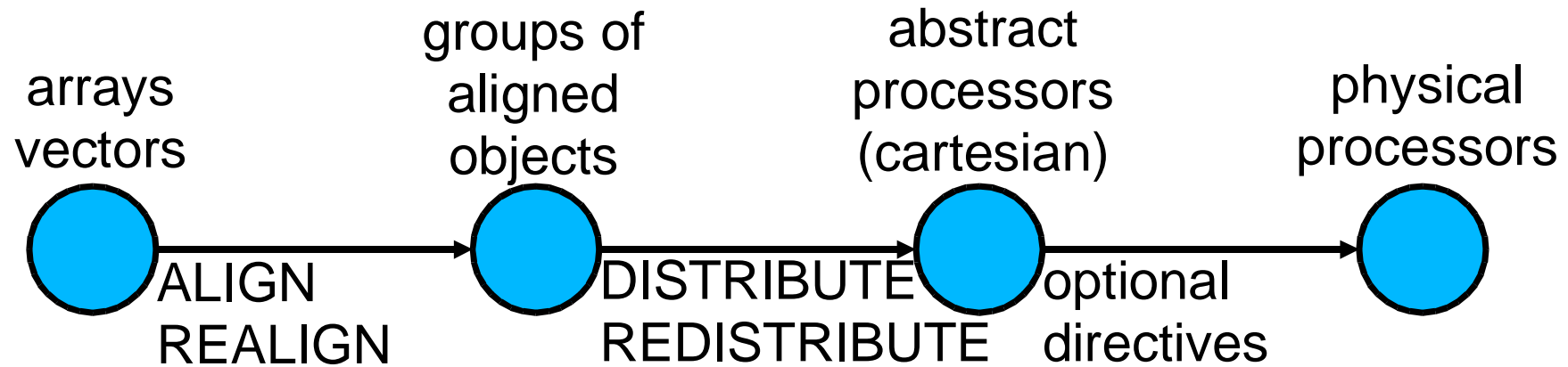
#### Assignments and their execution

- $x(i) = y(i)$ 
  - No rule for identical index given
  - No statement about communication possible
- $a(i) = a(\text{inx}(i)) + b(\text{inx}(i))$ 
  - $a(\text{inx}(i))$  and  $b(\text{inx}(i))$  are always on the same processor
  - Relation between  $a(i)$  and  $a(\text{inx}(i))$  unknown
  - No statement about communication possible

Important result: The mapping of the data must correspond with the main access scheme to minimize communication

# Data Alignment and Distribution

## ■ Model



- Instructions for the compiler how to map arrays to processor local memory blocks
- Goal: maximize local access, minimize remote access
- Variants: blockwise, cyclic

### Data Alignment and Distribution (2)

REAL field(14)

!HPF\$ PROCESSORS procs(6)

!HPF\$ DISTRIBUTE field (BLOCK) ONTO procs

1	2	3	4	5	6
1	4	7	10	13	
2	5	8	11	14	
3	6	9	12		

each processor gets  $\lceil 14/6 \rceil$  elements  
load imbalance

!HPF\$ DISTRIBUTE field (BLOCK(4)) ONTO procs

1	2	3	4	5	6
1	5	9	13		
2	6	10	14		
3	7	11			
4	8	12			

### Data Alignment and Distribution (3)

`!HPF$ DISTRIBUTE field (CYCLIC) ONTO procs`

1	2	3	4	5	6
1	2	3	4	5	6
7	8	9	10	11	12
13	14				

`!HPF$ DISTRIBUTE field (CYCLIC(2)) ONTO procs`

1	2	3	4	5	6
1	3	5	7	9	11
2	4	6	8	10	12
13					
14					

### Multi Dimensional Arrays

```
INTEGER CHESS_BOARD(8,8), GO_BOARD(19,19)
!HPF$ DISTRIBUTE CHESS_BOARD(BLOCK,BLOCK)
!HPF$ DISTRIBUTE GO_BOARD(CYCLIC,*)
```

- The chess board is mapped to a two dimensional processor structure
- The go board has the line distributed cyclically  
no distribution along the second dimension: lines are distributed as one object

# Data Parallel Assignments

- FORALL statement
  - Parallel assignment of field elements with masking

FORALL (i=1:n, j=1:n, y(i,j).NE.0.0) x(i,j)=1.0/y(i,j)  
compute reciprocal value if defined

FORALL (i=2:4) x(i)=x(i-1)+x(i)+x(i+1)

x' = (1.0 , 20.0 , 300.0 , 4000.0 , 50000.0)

x = (1.0 , 321.0 , 4320.0 , 54300.0 , 50000.0)

Not identical to: DO i=2,4 x(i)=x(i-1)+x(i)+x(i+1) END DO

### Data Parallel Assignments

- The INDEPENDENT directive precedes DO loops or FORALL statements
- Tells the compiler that iterations can be handled independently
- Good compilers recognize many variants automatically
- Bad programmers do not recognize anything
- If iterations are not really independent wrong results *might* be the consequence

### Intrinsic Functions

- Inquire system with NUMBER\_OF\_PROCESSORS and PROCESSOR\_SHAPE
- Compute maximum and minimum of a given field
- Library functions
  - Inquire data mapping
  - Field reductions like sum, product, etc.
  - Field distribution functions like scatter etc.  
More general field reduction with distribution of the result

# Evaluation of HPF

- Consequent development of existing Fortran dialects
- Data dependencies: manually or automatically with a good compiler
- Distribution and Alignment
  - Optimal solution is hard to find
  - Equally difficult as with message passing
- Tool support
  - Difficult: requires visualization of data and communication
  - Semantic gap between running MPI program and HPF source code

# Comparison of HPF with MPI and OpenMP

	HPF	MPI	OpenMP
scalability	yes	yes	yes
Fortran binding	yes	yes	yes
C binding	no	yes	planned
high abstraction	yes	no	yes
performance oriented	yes	yes	yes
data parallelism	yes	no	yes
portability	yes	yes	yes
incremental parallelization	no	no	yes
vendor support	yes!	yes!	starting

# Data Parallel Programming Models Summary

- Data parallel programming is optimal with distributed memory
- De facto standard HPF 2.0
- HPF has constructs for data distribution and data parallel assignments
- Data distribution optimized for arrays
- Data alignment supports locality during execution
- Support compiler via INDEPENDENT statement

---

# Comparison of Programming Paradigms

	data parallelism (HPF)	messages passing (MPI)	shared memory (OpenMP)
control flow	single	multiple	single
synchronism	lose	asynchronous	asynchronous
address space	single	multiple	single
interaction	implicit	explicit	explicit
data assignment	implicit	explicit	implicit

---

# Lessons Learnt

Hardware Aspects of Super Computers

Parallelization Paradigms

Programming Model

Parallel Input/Output

Debugging and Performance Tuning

Run-time Environments of Super Computers

---

# Abbreviations

- ❑ DSM - distributed shared memory
- ❑ HPF - High Performance Fortran
- ❑ MPI - Message Passing Interface
- ❑ NORMA - no remote memory access model
- ❑ NOW - network of workstations
- ❑ NUCA - non uniform communication architecture model
- ❑ NUMA - non uniform memory access model
- ❑ PVM - Parallel Virtual Machine
- ❑ SMP - symmetric multi processing
- ❑ SPMD - single program multiple data model
- ❑ UCA - uniform communication architecture model
- ❑ UMA - uniform memory access model

---

# Weblinks

- ❑ The top 500 list of computers in the world  
<http://www.top500.org>
- ❑ ParaScope - a listing of parallel computing sites  
<http://computer.org/parascope/>
- ❑ Accelerated Strategic Computing Initiative (ASCI)  
<http://www.llnl.gov/asci>
- ❑ OpenMp  
<http://www.openmp.org>  
<http://www.openmp.org/index.cgi?faq>
- ❑ A public domain OpenMP implementation  
<http://www.it.lth.se/odinmp>
- ❑ The Message Passing Interface (MPI)  
<http://www.mpi-forum.org/>

---

## Weblinks (2)

- A public domain MPI implementation  
<http://www-unix.mcs.anl.gov/mpi/mpich>
- High Performance Fortran (HPF)  
<http://www.vcpv.univie.ac.at/information/mirror/HPFF>

---

## Literature

- Kai Hwang, Zhiwei Xu: Scalable Parallel Computing: Technology, Architecture, Programming. Boston: WCB/McGraw-Hill, 1998. ISBN 0-07-031798-4. Comprehensive standard book; huge list of web references; 800 pages