

# IN3200/IN4200: High-Performance Computing & Numerical Projects

*Course overview & quick recap of serial  
programming*

Xing Cai

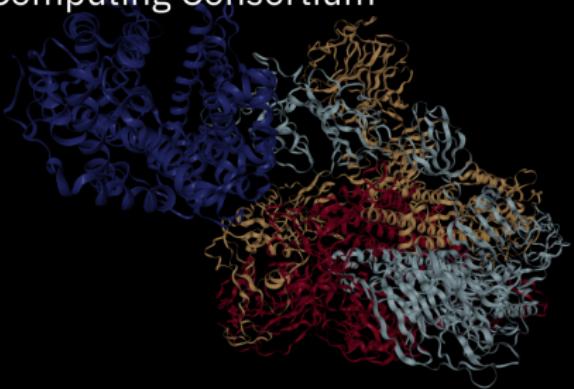
Simula Research Laboratory & University of Oslo, Norway

Spring 2022

# Motivation from the real life

COVID-19 HPC Consortium | Who We Are | Collaborations | Projects | News & Press | Blog | [Apply](#)

## The COVID-19 High Performance Computing Consortium



Bringing together the Federal government, industry, and academic leaders to provide access to the world's most powerful high-performance computing resources in support of COVID-19 research.

114 — Projects

603 — Petaflops

<https://covid19-hpc-consortium.org> (website last visited on 2022.01.17)

# Motivation from the real life (2)



UK Research  
and Innovation

Apply for funding   Manage your award   Our work   News and views

About us   Our councils

Search



Our main funds   What we've funded   Developing people and skills   International  
Infrastructure   Research culture   Impact   Investing across the UK   Public engagement  
**Tackling COVID-19   Responding to climate change   101 jobs that change the world**

[Home](#) > [Our work](#) > [Tackling the impact of COVID-19](#) > [Technological solutions](#) > [How high performance computing's power helped fight COVID-19](#)

## How high performance computing's power helped fight COVID-19

<https://www.ukri.org/> (website last visited on 2022.01.17)

# Real-life example 1

PHYS.ORG

≡  
Topics

Nanotechnology Physics Earth Astronomy & Space Technology

Home / Chemistry / Biochemistry  
Home / Chemistry / Analytical Chemistry



11



49



Share

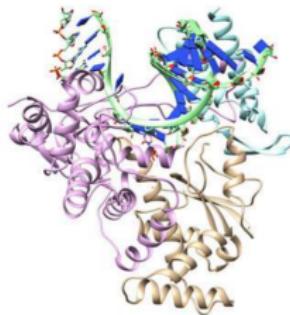


Email

SEPTEMBER 19, 2020

## Pulling the plug on the coronavirus copy machine

by Jorge Salazar, Texas Advanced Computing Center



<https://phys.org/news/2020-09-coronavirus-machine.html>

## Real-life example 1 (cont'd)

- To model key proteins used by coronavirus for its reproduction
- Use basic math and physics of Newton's equations and quantum mechanics to calculate the properties of these proteins
- HPC & supercomputers allow much faster simulations
- Goal: Finding ways to improve COVID-19 drugs
- Research team from University of North Texas



Simulations done on Frontera: No. 9 supercomputer in the world (according to TOP500 ranking in Nov. 2020)

<https://phys.org/news/2020-09-coronavirus-machine.html>

# Real-life example 2



FEATURES

## Fighting COVID-19 With the Power of Genomics and HPC

By Janet Morss | June 17, 2020

**Researchers at Cardiff University are using the power of genomic sequencing and high performance computing to unlock the secrets of COVID-19.**

In scientific laboratories around the world, efforts are under way to put the power of genomic sequencing and [high performance computing](#) (HPC) to work in the fight against COVID-19. At Cardiff University in Wales, a team of scientists is working with the COVID-19 Genomics UK Consortium (COG-UK), to unlock the secrets of the coronavirus that causes COVID-19.



YOU MAY ALSO LIKE

FEATURES

<https://www.delltechnologies.com/en-us/blog/fighting-covid-19-with-the-power-of-genomics-and-hpc/>

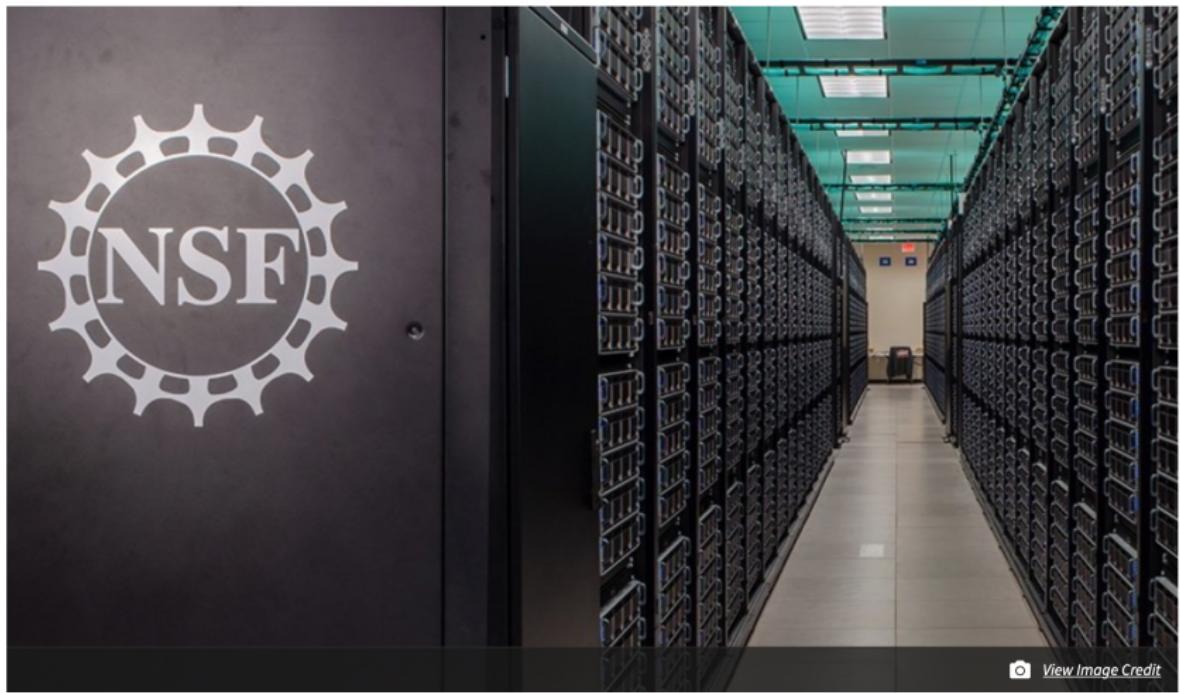
## Real-life example 2 (cont'd)

- Large-scale, rapid genomic sequencing and analysis of the coronavirus
- Relying on a large shared system that provides 2.5 petabytes of HPC data storage, also a huge amount of memory (78 terabytes)
- Goal: Unlocking the secrets of the coronavirus
- Research team from Cardiff University in Wales



<https://www.delltechnologies.com/en-us/blog/fighting-covid-19-with-the-power-of-genomics-and-hpc/>

# More about fighting COVID19 with HPC



 View Image Credit

## Why are supercomputers so important for COVID-19 research?

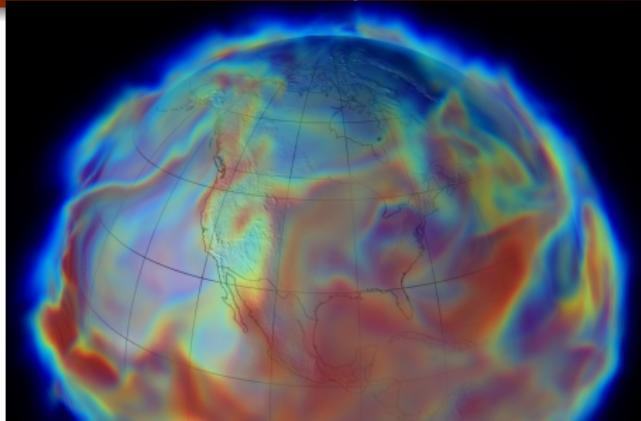
Spread the Word



# General motivations for HPC

- Many problems in natural sciences can benefit from large-scale or huge-scale computations
  - more details
  - better accuracy
  - more advanced models
- The need for computing is ever-increasing
- However, standard laptop PCs or desktop computers are not powerful enough!

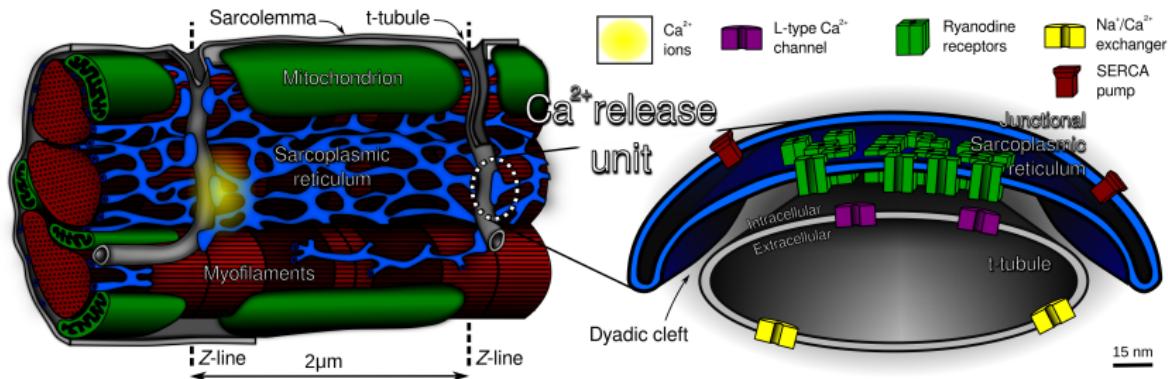
# Huge computation example 1 (Climate Simulation)



NASA Center for Climate Simulation

- Earth surface area:  $510,072,000 \text{ km}^2$
- If a spatial resolution of  $1 \times 1\text{km}^2$  is adopted  $\rightarrow 5.1 \times 10^8$  (510 million) small patches
- If a spatial resolution  $100 \times 100\text{m}^2$  is adopted  $\rightarrow 5.1 \times 10^{10}$  (51 billion) small patches
- Additional layers in the vertical direction
- High resolution in the time direction

## Example 2 (Subcellular Calcium Dynamics Simulation)



- Size of one cardiac muscle cell:  $100\mu\text{m} \times 10\mu\text{m} \times 10\mu\text{m}$
- Width of calcium release channels: 1 nanometer (nm)
- Ideal computational mesh resolution: 1 nm
- Computational mesh required:  $10^5 \times 10^4 \times 10^4$  (in total  $10^{13}$  computational voxels)
- Number of simulation time steps needed:  $\sim 10^6$

# Motivations (cont'd)

- Parallel computers are now everywhere!
  - CPUs nowadays have multiple “cores” on a chip
  - One computer may have several multicore chips
  - There are also accelerator-based parallel architectures — GPGPU (general-purpose graphics processing unit)
  - Clusters of different kinds



# What do we learn in IN3200/IN4200?

## High-performance computing (HPC) – an introduction

- Proper implementation of numerical algorithms
- Effective use of the hardware for numerical computations

After finishing the course, you should

- be able to write simple parallel programs with sufficiently good performance
- be able to learn more about advanced computing later on your own

# Part 1 of the course: Serial programming

- A brief architectural overview of modern cache-based microprocessors
- Inherent performance limitations of microprocessors
- Basic C programming
- Optimization strategies of serial code

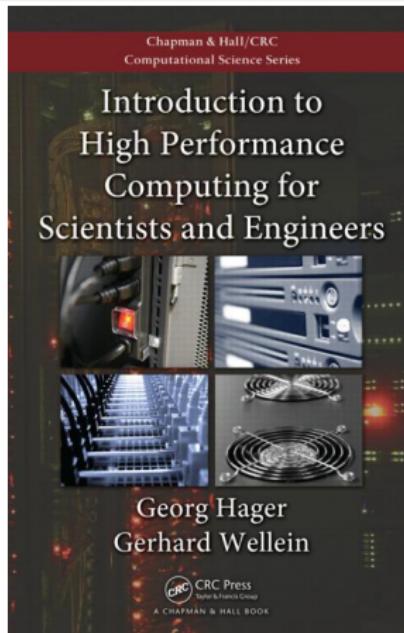
## Part 2 of the course: Parallel programming

- Parallel computer architecture
- Theoretical considerations of parallel computing
- Shared-memory parallel programming (OpenMP)
- Distributed-memory parallel programming (MPI)

# Why learning parallel programming?

- Parallel computing – a form of parallel processing by concurrently utilizing multiple computing units for one computational problem
  - shortening computing time
  - solving larger problems
- However ...
  - modern multicore-based computers are good at multi-tasking, but not good at automatically computing one problem in parallel
  - automatic parallelization compilers have had little success
  - special parallel programming languages have had little success
  - serial computer programs have to be modified or completely rewritten to utilize parallel computers
- Learning parallel programming is thus important!

# Textbook



Georg Hager, Gerhard Wellein

**Introduction to High Performance Computing for Scientists  
and Engineers**

1st Edition, CRC Press, ISBN 9781439811924

# Teaching approaches

- Focus on fundamental issues
  - parallel programming = serial programming + finding parallelism + enforcing work division and collaboration
- Use of examples relevant for natural sciences
  - mathematical details are not required
  - understanding basic numerical algorithms is needed
  - implementing basic numerical algorithms is essential
- Hands-on programming exercises and tutoring
- English is the “official language” of the course, but students should feel free to ask questions, write emails/reports in Norwegian

# **Recapitulation of serial programming + some difficult issues in C programming**

A tutorial in C programming will be given in the next lecture

# What is serial programming?

- Roughly speaking, a computer program executes a sequence of operations applied to data structures
- A program is normally written in a programming language
- Data structures:
  - variables of primitive data types (`char`, `int`, `float`, `double` etc.)
  - variables of composite and abstract data types (`struct` in C, `class` in Java & Python)
  - array variables
- Operations:
  - statements and expressions
  - functions

# Variables

- In a dynamically typed programming language (e.g. Python) variables can be used without declaration beforehand

```
a = 1.0
```

```
b = 2.5
```

```
c = a + b
```

- In statically typed languages (e.g. Java and C) declaration of variables must be done first

```
double a, b, c;
```

```
a = 1.0;
```

```
b = 2.5;
```

```
c = a + b;
```

## Simple example

- Suppose we have temperature measurement for each hour during a day
- $t_1$  is the temperature at 1:00 o'clock,  $t_2$  is the temperature at 2:00 o'clock, and so on.
- How to find the average temperature of the day?
- We need to first add up all the 24 temperature measurements:

$$T = t_1 + t_2 + \dots + t_{24} = \sum_{i=1}^{24} t_i$$

- The average temperature can then be calculated as  $\frac{T}{24}$ .

## Simple example (cont'd)

- How to implement the calculations as a computer program?
- First, create an array of 24 floating-point numbers to store the 24 temperatures. That is,  $t[0]$  stores  $t_1$ ,  $t[1]$  stores  $t_2$  and so on. Note that array index starts from 0!
- Sum up all the values in the array  $t$

- Same syntax for the computational loop in Java & C:

```
T = 0;  
for (i=0; i<24; i++)  
    T = T + t[i];
```

- Syntax for Python:

```
T = 0  
for i in range(0,24):  
    T = T + t[i]
```

- Finally,  $t\_average = T/24.0;$

# Similarities and differences between languages

- For scientific applications, arrays of numerical values are the most important basic building blocks of data structure
- Extensive use of `for`-loops for doing computations
- Different syntax details
  - allocation and deallocation of arrays
    - Java: `double[] v=new double[n];`
    - C: `double *v=malloc(n*sizeof(double));`
    - Python: `v=zeros(n,dtype=float64)` (using NumPy)
  - definition of composite and abstract data types
  - I/O

# C as the main choice of programming language

- C is one of the dominant programming languages in computational sciences
- Syntax of C has inspired many newer languages (C++, Java, Python)
- Good computational efficiency
- C is ideal for using MPI and OpenMP (also GPU programming)
- We will thus choose C as the main programming language
- (Most of the textbook's coding examples are in Fortran, but many of the “performance-engineering” principles are the same.)

## Some words about pointers in C

- A variable in a program has a name and type, its value is stored somewhere in the memory of a computer
- Type `*p` declares a pointer to a variable of datatype `Type`
- A pointer is actually a special type of variable, used to hold the memory address of a variable
- From a variable to its pointer: `int a; int *p; p = &a;`
- We can use a pointer to change the variable value `*p = 2;`  
(The value of `a` is now 2.)
- We can use several pointers (if needed) to work with an array:

```
int *p = (int*)malloc(10*sizeof(int));  
int *p2 = p + 3; /* p2 is now pointing to p[3] */
```

# Allocating multi-dimensional arrays

- Let's allocate a 2D array for representing a  $m \times n$  matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

- Java:

```
double[][] A = new double[m][n];
```

- C:

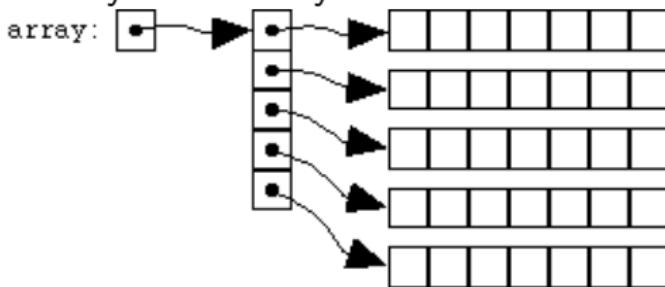
```
double **A = (double**)malloc(m*sizeof(double*));
for (i=0; i<m; i++)
    A[i] = (double*)malloc(n*sizeof(double));
```

- Same syntax in Java and C for indexing and traversing a 2D array

```
for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        A[i][j] = i+j;
```

## More about two-dimensional arrays in C (1)

- C doesn't have true multi-dimensional arrays, a 2D array is actually an array of 1D arrays



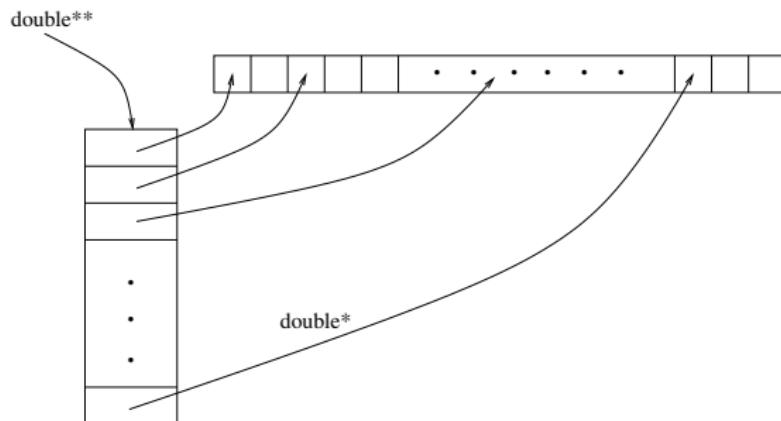
- $A[i]$  is a pointer to row number  $i+1$
  - It is also possible to use static memory allocation of fix-sized 2D arrays, for example:
- ```
double A[10][8];
```

However, the size of the array is decided at compiler time (not runtime)

## More about two-dimensional arrays in C (2)

- Dynamic memory allocation of 2D arrays through e.g. malloc
- Another way of dynamic allocation, to ensure contiguous underlying data storage (for good use of cache):

```
double *A_storage=(double*)malloc(n*n*sizeof(double));  
double **A = (double**)malloc(n*sizeof(double*));  
for (i=0; i<n; i++)  
    A[i] = &(A_storage[i*n]);
```



# Deallocation of arrays in C

- If an array is dynamically allocated, it is important to free the storage when the array is not used any more
- Example 1

```
int *p = (int*)malloc(n*sizeof(int));  
/* ... */  
free(p);
```

- Example 2

```
double **A = (double**)malloc(m*sizeof(double*));  
for (i=0; i<m; i++)  
    A[i] = (double*)malloc(n*sizeof(double));  
/* ... */  
for (i=0; i<m; i++)  
    free(A[i]);  
free(A);
```

- Be careful! Memory allocation and deallocation can easily lead to errors

# Functions in C

- Function declaration specifies name, type of return value, and (optionally) a list of parameters
- Function definition consists of declaration and a block of code, which encapsulates some operation and/or computation

```
return_type function_name (parameter declarations)
{
    declarations of local variables
    statements
}
```

## Function arguments

- All arguments to a C function are passed by value
- That is, a copy of each argument is passed to the function

```
void test (int i) {  
    i = 10;  
}
```

The change of `i` inside `test` has no effect when the function returns

- Passing pointers as function arguments can be used to get output

```
void test (int *i) {  
    *i = 10;  
}
```

The change of `i` inside `test` now has effect

## Function example 1: swapping two values

```
void swap (int *a, int *b)
{
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

## Function example 2: smoothing a vector

- We want to smooth the values of a vector  $v$  by the following formula:

$$v_i^{\text{new}} = v_i + c(v_{i-1} - 2v_i + v_{i+1}), \quad 1 \leq i < n - 1$$

where  $c$  is a constant

```
void smooth (double *v_new, double *v, int n, double c)
{
    int i;
    for (i=1; i<n-1; i++)
        v_new[i] = v[i] + c*(v[i-1]-2*v[i]+v[i+1]);
    v_new[0] = v[0];
    v_new[n-1] = v[n-1];
}
```

- Similar computations occur frequently in numerical computations

## Function example 3: matrix-vector multiplication

- We want to compute  $\mathbf{y} = \mathbf{Ax}$ , where  $\mathbf{A}$  is a  $m \times n$  matrix,  $\mathbf{y}$  is a vector of length  $m$  and  $\mathbf{x}$  is a vector of length  $n$ :

$$y_i = A_{i1}x_1 + A_{i2}x_2 + \dots + A_{in}x_n = \sum_{j=1}^n A_{ij}x_j, \quad 1 \leq i \leq m$$

```
void mat_vec_prod (double **A, double *y, double *x,
                   int m, int n)
{
    int i,j;
    for (i=0; i<m; i++) {
        y[i] = 0.0;
        for (j=0; j<n; j++)
            y[i] += A[i][j]*x[j];
    }
}
```

# IN3200/IN4200: C Programming Tutorial

A drastically simplified version of <https://www.tutorialspoint.com/cprogramming/>

Target audience: new beginners of C programming

# First things first

- A program in C is made up of
  - Preprocessor commands
  - Variables
  - Statements and expressions
  - Functions
  - Comments
- A program in C can be as simple as having only 3 lines, or as comprehensive as being composed of millions of lines
- A program in C can be stored in one file with name extension .c (or spread over many .h and .c files)
- Use of libraries—groups of already-coded functions and declarations—actually happens all the time

# Hello-World example

```
#include <stdio.h>

int main() {
    /* my first program in C */
    printf("Hello, World! \n");

    return 0;
}
```

# Hello-World example explained

- `#include <stdio.h>` is a preprocessor command, which tells a C compiler to include the header file stdio.h
- `int main()` defines the main function where the program execution begins
- `/*...*/` is a comment
- `printf(...)` is a standard library function available in C (found in `<stdio.h>`, for sending formatted output to the standard output stream)
- `return 0;` terminates the `main()` function and returns the value 0

## **Demo of compilation and execution**

# Identifiers

- An identifier is a name used to identify a variable, function, or any other user-defined item
- Examples of acceptable identifiers

|          |       |     |           |        |
|----------|-------|-----|-----------|--------|
| mohd     | zara  | abc | move_name | a_123  |
| myname50 | _temp | j   | a23b9     | RetVal |

- C is a case-sensitive programming language

# Keywords – reserved words

|          |        |          |          |
|----------|--------|----------|----------|
| auto     | else   | long     | switch   |
| break    | enum   | register | typedef  |
| case     | extern | return   | union    |
| char     | float  | short    | unsigned |
| const    | for    | signed   | void     |
| continue | goto   | sizeof   | volatile |
| default  | if     | static   | while    |
| do       | int    | struct   | _Packed  |
| double   |        |          |          |

Keywords can not be used as identifiers.

# C data types

- **Basic Types**

They are arithmetic types and are further classified into: (a) integer types and (b) floating-point types

- **Derived types**

They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types

# Integer types

| Type           | Storage size | Value range                                          |
|----------------|--------------|------------------------------------------------------|
| char           | 1 byte       | -128 to 127 or 0 to 255                              |
| unsigned char  | 1 byte       | 0 to 255                                             |
| signed char    | 1 byte       | -128 to 127                                          |
| int            | 2 or 4 bytes | -32,768 to 32,767 or -2,147,483,648 to 2,147,483,647 |
| unsigned int   | 2 or 4 bytes | 0 to 65,535 or 0 to 4,294,967,295                    |
| short          | 2 bytes      | -32,768 to 32,767                                    |
| unsigned short | 2 bytes      | 0 to 65,535                                          |
| long           | 4 bytes      | -2,147,483,648 to 2,147,483,647                      |
| unsigned long  | 4 bytes      | 0 to 4,294,967,295                                   |

## The sizeof operator

To get the exact size of a type or a variable on a particular platform, you can use the `sizeof` operator. The expression `sizeof(type)` yields the storage size of the object or type in number of bytes.

```
#include <stdio.h>

int main() {
    printf("Storage size for int : %d \n", sizeof(int));

    return 0;
}
```

# Floating-point types

| Type        | Storage size | Value range            | Precision         |
|-------------|--------------|------------------------|-------------------|
| float       | 4 bytes      | 1.2E-38 to 3.4E+38     | 6 decimal places  |
| double      | 8 bytes      | 2.2E-308 to 1.8E+308   | 15 decimal places |
| long double | 16 bytes     | 3.4E-4932 to 1.2E+4932 | 18 decimal places |

Note: The actual values can be machine-dependent!

## Header file float.h

The header file float.h defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs

```
#include <stdio.h>
#include <float.h>

int main() {
    printf("Storage size for float : %lu \n", sizeof(float));
    printf("Minimum float positive value: %E\n", FLT_MIN );
    printf("Maximum float positive value: %E\n", FLT_MAX );
    printf("Precision value: %d\n", FLT_DIG );

    return 0;
}
```

# C variables

A variable is a name given to a storage area that a C program can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive.

```
int    i, j, k;  
char   c, ch;  
float  f, salary;  
double d;
```

# C operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators:

- Arithmetic operators + - \* / % ++ --
- Relational operators == != > < >= <=
- Logical operators && || !
- Bitwise operators
- Assignment operators

# Bitwise operators

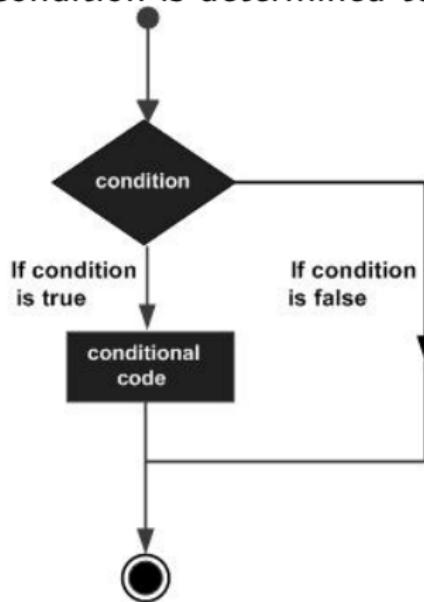
A bitwise operator works on bits and performs bit-by-bit operation

Some examples:

| p | q | p&q | p q | p^q |
|---|---|-----|-----|-----|
| 0 | 0 | 0   | 0   | 0   |
| 0 | 1 | 0   | 1   | 1   |
| 1 | 1 | 1   | 1   | 0   |
| 1 | 0 | 0   | 1   | 1   |

# Decision making

Decision making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.



# Loops

To execute a statement or a group of statements multiple times:

- `for`
- `while`
- `do ... while`

# Functions

A function is a group of statements that together perform a task. Every C program has at least one function, which is `main()`, and you can define additional functions.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The general form of a function definition in C programming language:

```
return_type function_name( parameter list ) {  
    body of the function  
}
```

## Function arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, the formal parameters get the values (that is, copies) of the **actual parameters**.

# One example

```
#include<stdio.h>
void func_1(int);

int main()
{
    int x = 10;

    printf("Before function call\n");
    printf("x = %d\n", x);

    func_1(x);

    printf("After function call\n");
    printf("x = %d\n", x);

    return 0;
}

void func_1(int a)
{
    a += 1;
    a++;
    printf("\na = %d\n\n", a);
}
```

## Scope rules

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language:

- Inside a function or a block: **local** variables.
- Outside of all functions: **global** variables.
- In the definition of function parameters: **formal** parameters.

## Global variables

Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program. **Should be used with care!**

```
#include <stdio.h>

/* global variable declaration */
int g;

int main () {

    /* local variable declaration */
    int a, b;

    /* actual initialization */
    a = 10;
    b = 20;
    g = a + b;

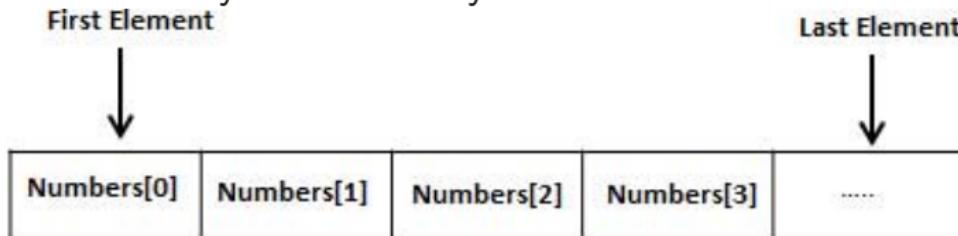
    printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

    return 0;
}
```

# Arrays

An array is one kind of data structure that can store a sequential collection of elements of the same type.

Instead of declaring individual variables, such as Number0, Number1, ..., and Number99, you can declare one array variable, named such as Numbers, and use Numbers[0], Numbers[1], ..., and Numbers[99] to represent individual variables. A specific element in an array is accessed by an index.



## Fix-sized arrays

A programmer can specify the type of the elements and the number of elements required by an array

```
type arrayName [ arraySize ];
```

arraySize must be an integer constant greater than zero.

## Address in memory

Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory.

```
#include <stdio.h>

int main () {

    int var1;
    char var2[10];

    printf("Address of var1 variable: %x\n", &var1);
    printf("Address of var2 variable: %x\n", &var2);

    return 0;
}
```

# Pointers

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer variable before using it to store any variable address.

```
int    *ip;    /* pointer to an integer */  
double *dp;    /* pointer to a double */  
float  *fp;    /* pointer to a float */  
char   *ch;    /* pointer to a character */
```

# How to use pointers?

- Define a pointer variable
- Assign the address of a variable to a pointer variable
- Access the value at the address stored in the pointer variable (via operator \*)

```
#include <stdio.h>

int main () {
    int var = 20;      /* actual variable declaration */
    int *ip;           /* pointer variable declaration */

    ip = &var;  /* store address of var in pointer variable*/

    printf("Address of var variable: %x\n", &var );

    /* address stored in pointer variable */
    printf("Address stored in ip variable: %x\n", ip );

    /* access the value using the pointer */
    printf("Value of *ip variable: %d\n", *ip );

    return 0;
}
```

## More pointer concepts

- **Pointer arithmetic:** Four arithmetic operators can be used on pointers: `++`, `--`, `+`, `-`
- **Array of pointers:** You can define an array to hold a sequence of pointers.
- **Pointer to pointer:** C allows you to have pointer on a pointer and so on.
- **Passing pointers to functions in C:** Passing an argument by address allows the passed argument to be changed.
- **Return pointer from functions in C:** C allows a function to return a pointer to the local variable, static variable, and dynamically allocated memory as well. (**Be very careful with such usage!!!!!!**)

# An example of function returning a pointer

```
#include <stdio.h>

int *getMax(int *m, int *n) {
    /* if the value pointed by pointer m is greater than n
     * then, return the address stored in the pointer variable m */
    if (*m > *n) {
        return m;
    }
    else {
        return n;
    }
}

int main(void) {
    // integer variables
    int x = 100;
    int y = 200;

    // pointer variable
    int *max = NULL;

    /* get the variable address that holds the greater value
     * for this we are passing the address of x and y
     * to the function getMax() */
    max = getMax(&x, &y);

    // print the greater value
    printf("Max value: %d\n", *max);

    return 0;
}
```

# C structures

**structure** is a user defined data type available in C that allows to combine data items of different kinds.

To define a structure, you must use the **struct** statement:

```
struct Books {  
    char title[50];  
    char author[50];  
    char subject[100];  
    int book_id;  
} book;
```

# Dynamic memory management

The C programming language provides several functions for memory allocation and management. These functions can be found in the `<stdlib.h>` header file.

- `void *calloc(int num, int size);` – allocates an array of num elements each of which size in bytes will be size.
- `void free(void *address);` – releases a block of memory block specified by address.
- `void *malloc(int num);` – allocates an array of num bytes and leave them uninitialized.
- `void *realloc(void *address, int newsize);` – re-allocates memory extending it upto newsize.

## Example of dynamic memory allocation

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n, i, *ptr, sum = 0;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    ptr = (int*) malloc(n * sizeof(int));
    if(ptr == NULL) {
        printf("Error! memory not allocated."); exit(-1);
    }

    printf("Enter elements: ");
    for (i = 0; i < n; ++i) {
        scanf("%d", &(ptr[i]));
        sum += ptr[i];
    }

    printf("Sum = %d\n", sum);
    free(ptr);
    return 0;
}
```

# Command-line arguments

Input arguments to the main function:

```
#include <stdio.h>

int main( int argc, char *argv[] )  {

    if( argc == 2 ) {
        printf("The argument supplied is %s\n", argv[1]);
    }
    else if( argc > 2 ) {
        printf("Too many arguments supplied.\n");
    }
    else {
        printf("One argument expected.\n");
    }
}
```

# Simple I/O

A file represents a sequence of bytes, regardless of it being a text file or a binary file. C programming language provides both high-level functions and low-level function calls to handle files.

- Opening a file

```
FILE *fopen(const char *filename, const char *mode);
```

- Closing a file

```
int fclose(FILE *fp);
```

- Writing to a file (many different functions available)

- Reading to a file (many different functions available)

- Binary I/O functions

```
size_t fread(void *ptr, size_t size_of_elements,
            size_t number_of_elements, FILE *a_file);
size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
```

# IN3200/IN4200: Chapter 1

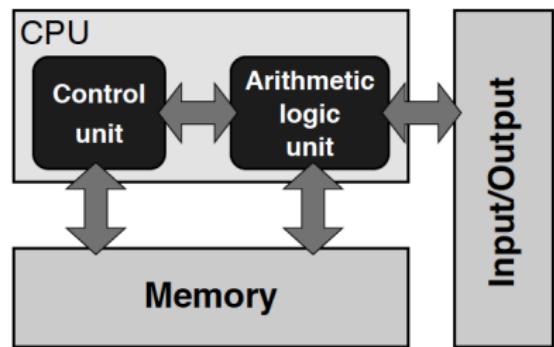
## Modern processors

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

# Objectives of Chapter 1

- A *high-level* overview of the architecture of modern cache-based microprocessors
- Introduction of important concepts, which will be useful for writing efficient code later
- Discussion of inherent performance limitations

# “Stored-program computer”



**Figure 1.1:** Stored-program computer architectural concept. The “program,” which feeds the control unit, is stored in memory together with any data the arithmetic unit requires.

## The “stored-program computer” concept

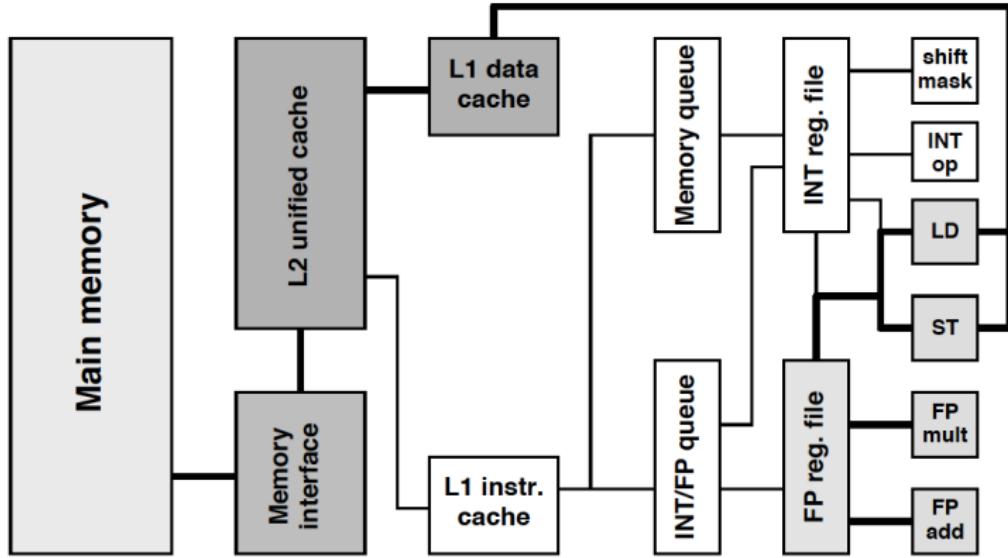
- Instructions (produced by a *compiler*) and data are stored in memory
- Instructions are read and executed by a control unit
- An arithmetic/logic unit “does the work”, which is coded in the instructions
- The speed of memory determines how fast instructions and data can be fed to the control and arithmetic units—limitation of performance
- I/O facilities enable interaction with users

**CPU** (central processing unit)—is the “brain” of a computer.

CPU incorporates control and arithmetic units (and many other components), together with appropriate interfaces to memory and I/O.

CPU has a “clock”, which at each *clock cycle* synchronizes the logic units within the CPU to process instructions.

# Cache-based microprocessor



**Figure 1.2:** Simplified block diagram of a typical cache-based microprocessor (one core). Other cores on the same chip or package (socket) can share resources like caches or the memory interface. The functional blocks and data paths most relevant to performance issues in scientific computing are highlighted.

# Important hardware components

- Arithmetic units for floating-point (FP) and integer (INT) operations
- Registers hold operands to be accessed by instructions
- Load (LD) and store (ST) units handle instructions that transfer data to and from registers
- Instructions are sorted into several queues, waiting to be executed (probably not in the order they were issued)
- Caches hold data and instructions to be (re-)used

## Pipelined functional units

Subdividing complex operations into simple components that can be executed using different functional units, it is possible to increase *instruction throughput*—the number of instructions executed per clock cycle.

This is the most elementary example of *instruction-level parallelism* (ILP).

Optimally pipelined execution leads to a throughput of one instruction per cycle per pipeline.

# Pipelining

- Pipelining in microprocessors follows the same principle of assembly lines in manufacturing: Workers (functional units) are highly skilled and specialized for a single task.
- Each worker executes the same step, over and over again, on different objects.
- If it takes  $m$  different steps to finish the product,  $m$  products are continuously worked on, in different stages of completion.
- If all tasks take the same amount of time, and all workers are continuously busy, eventually (after the initial  $m$  steps) one product will be finished per time step.

## A simple example of no pipelining

For simplicity, let us suppose every instruction has five stages, each taking one cycle.

The following picture shows the situation of no pipelining:



# The situation with instruction pipelining



## More about pipelining

Complex operations such as loading and storing data or performing floating-point arithmetic cannot be executed in a single cycle. The “fetch–decode–execute” pipeline is thus applicable, in which each stage can operate independently of the others.

These still complex tasks are usually broken down even further. The benefit of elementary subtasks is the potential for a higher clock rate as the functional units are kept simple.

## Example of “vector product”

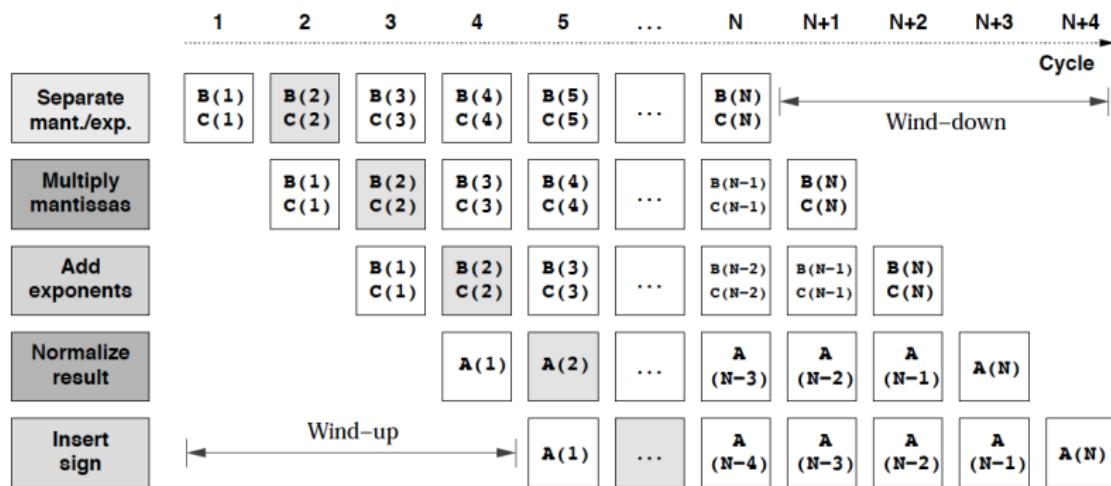
Arrays of floating-point values: A B C

```
for (i=0; i<N; i++)
    A[i] = B[i] * C[i];
```

Suppose a floating-point multiplication is decomposed into five subtasks. (A floating-point value is  $(\text{sign}) \times \text{mantissa} \times 2^{\text{exponent}}$ .)

- ① separation of mantissa and exponent on B[i] and C[i]
- ② multiply mantissas of B[i] and C[i] (recall that a mantissa is a binary fraction with non-zero leading bit)
- ③ add exponents of B[i] and C[i]
- ④ normalize result
- ⑤ insert sign

# Depiction of a pipeline



**Figure 1.5:** Timeline for a simplified floating-point multiplication pipeline that executes  $A(:) = B(:) * C(:)$ . One result is generated on each cycle after a four-cycle wind-up phase.

## Simple mathematical model for pipelining

An  $m$ -stage pipeline has *latency* (or *depth*) of  $m$  cycles. The *wind-up* and *wind-down* periods are both  $m - 1$  cycles.

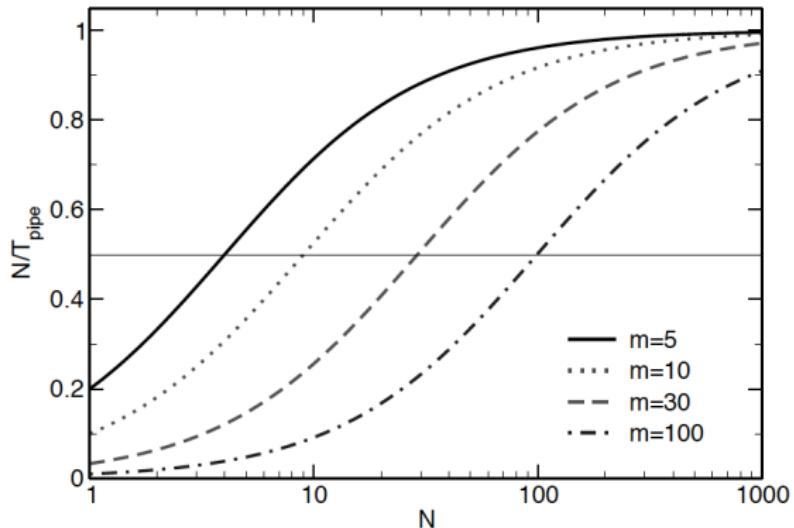
For a pipeline of depth  $m$ , executing  $N$  independent operations takes  $N + m - 1$  cycles. The speedup versus “no pipeling” is

$$\frac{T_{\text{seq}}}{T_{\text{pipe}}} = \frac{N \cdot m}{N + m - 1}$$

The throughput, average number of operations finished per cycle, can be calculated as

$$\frac{N}{T_{\text{pipe}}} = \frac{1}{1 + \frac{m-1}{N}}$$

# Example of pipeline throughput



**Figure 1.6:** Pipeline throughput as a function of the number of independent operations.  $m$  is the pipeline depth.

## Pipeline bubbles

Very complex calculations (like floating-point division or special math functions) tend to have very long latencies, and are only pipelined to a small level or not at all. In such cases, stalling the instruction stream becomes inevitable, leading to so-called “*pipeline bubbles*”.

Avoiding such complex functions, if possible, is a useful technique for code optimization (to be discussed in Chapter 2).

# Superscalarity

Goal: To produce more than one “result” per cycle.

- Multiple instructions are fetched and decoded concurrently
- Address and other integer calculations are performed in multiple integer (add, mult, shift, mask) units
- Multiple floating-point pipelines run in parallel
- Caches are fast enough to sustain more than one load or store operation per cycle

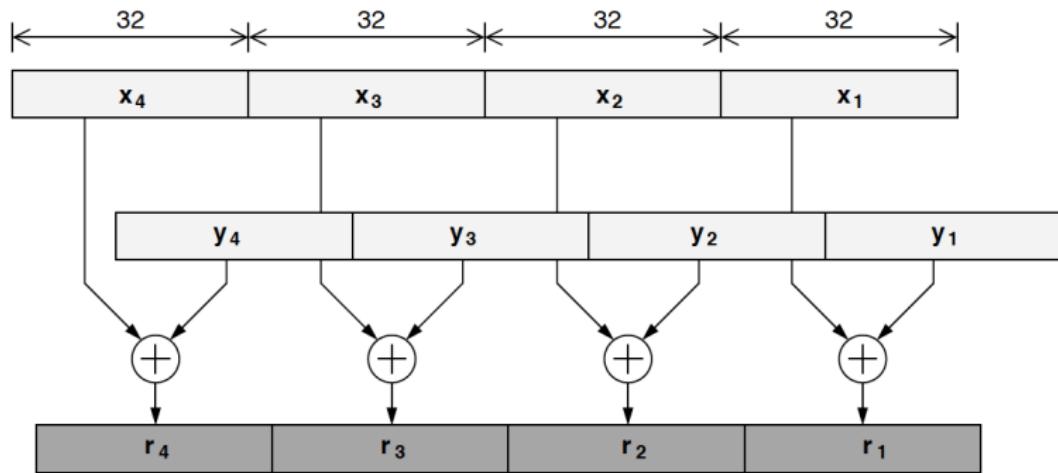
*Superscalarity* is a special form of parallel execution, and a variant of ILP.

Out-of-order execution and compiler optimization must work together to fully exploit superscalarity.

The SIMD (single-instruction-multiple-data) concept became widely known with the first vector supercomputers in 1970s.

Modern cache-based processors have instruction set extensions for both integer and floating-point operations. They allow the concurrent execution of arithmetic operations on “wide” registers, each holding multiple numerical values.

# Example of SIMD



**Figure 1.8:** Example for SIMD: Single precision FP addition of two SIMD registers ( $x,y$ ), each having a length of 128 bits. Four SP flops are executed in a single instruction.

## Memory hierarchy

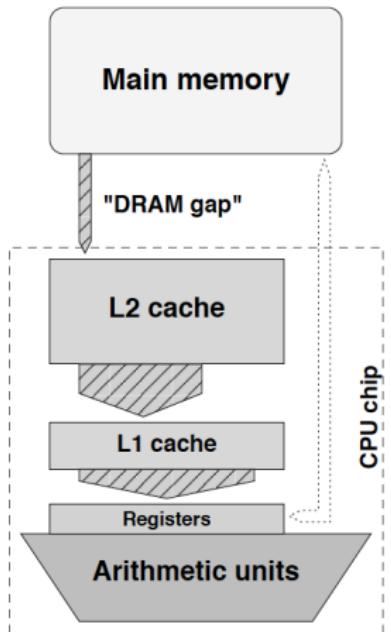
Data can be stored in a computer system in many different ways.

CPU has a set of registers, which can be accessed without delay.

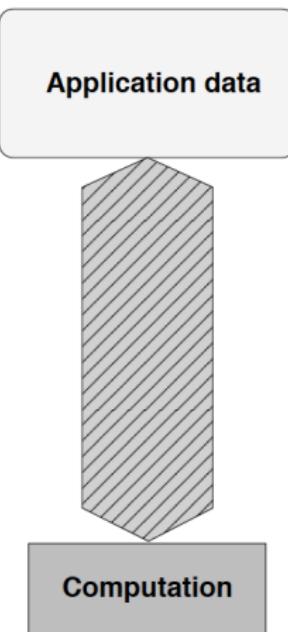
In addition, there are several levels of *cache*, holding copies of recently used data items.

Main memory of a computer is much slower (than the caches).

# Depiction of memory hierarchy



**Figure 1.3:** (Left) Simplified data-centric memory hierarchy in a cache-based microprocessor (direct access paths from registers to memory are not available on all architectures). There is usually a separate L1 cache for instructions. (Right) The “DRAM gap” denotes the large discrepancy between main memory and cache bandwidths. This model must be mapped to the data access requirements of an application.



# Cache

Caches are low-capacity, high-speed memories that are commonly integrated on the CPU die.

- L1 (level 1) data cache
- L1 instruction cache
- L2 and L3 (data & instruction) unified caches

The purpose of cache—reducing the impact of main memory's small bandwidth and high latency.

## Cache hit and miss

Whenever the CPU issues a read request ("load") for transferring a data item to a register, the L1 data cache is checked. If the wanted data item is found in L1, this is called a *cache hit*, otherwise a *cache miss* occurs.

In case of a cache miss in L1, data must be fetched from upper cache levels or, in the worst case, from main memory.

## Cache eviction

If a data item needs to be loaded into a cache where all cache entries are occupied. One of the occupant entries has to be *evicted* by a hardware-implemented algorithm (typically following the *least-recently used* strategy) to give space.

## Temporal locality

If data items loaded into a cache are to be used again “soon enough”, then this is called good *temporal locality*.

Suppose accessing a data item in cache is a factor of  $\tau$  faster than accessing the main memory. Let  $\beta$  denote the cache reuse ratio. Suppose access time to main memory is denoted by  $T_m$ , access time to cache thus  $T_c = T_m/\tau$ .

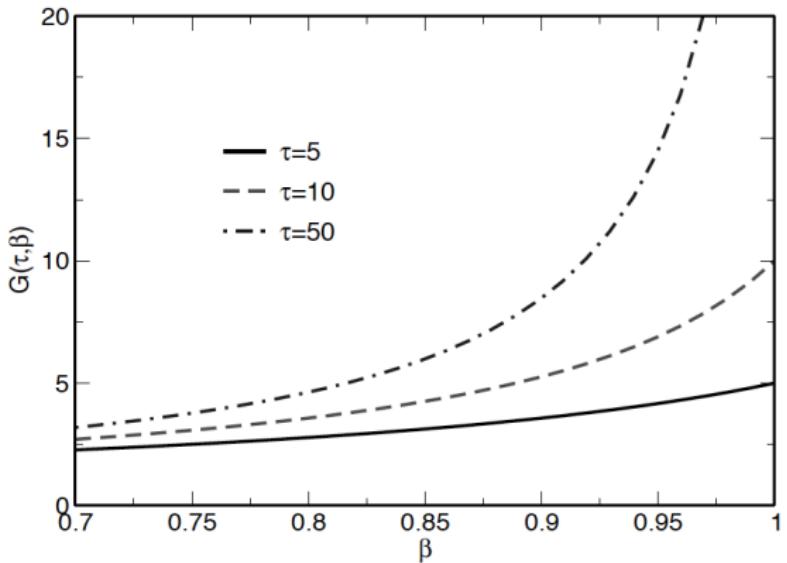
The average access time will be

$$T_{av} = \beta T_c + (1 - \beta) T_m$$

Performance gain due to cache can be calculated by

$$G(\tau, \beta) = \frac{T_m}{T_{av}} = \frac{\tau T_c}{\beta T_c + (1 - \beta)\tau T_c} = \frac{\tau}{\beta + (1 - \beta)\tau}$$

# Curves of performance gain



**Figure 1.9:** The performance gain from accessing data from cache versus the cache reuse ratio, with the speed advantage of cache versus main memory being parametrized by  $\tau$ .

## Cache lines

The content of a cache is organized as *cache lines*. (A cache line has space for multiple data items.) This is for reducing the latency penalty for *streaming*—large amounts of data are loaded into the CPU, modified, and written back without the potential of reuse “in time”.

All data transfers between caches and main memory happen on the cache line level.

If a code has good *spatial locality*, that is, the probability of successive accesses to neighboring items is high, the latency problem can be significantly reduced.

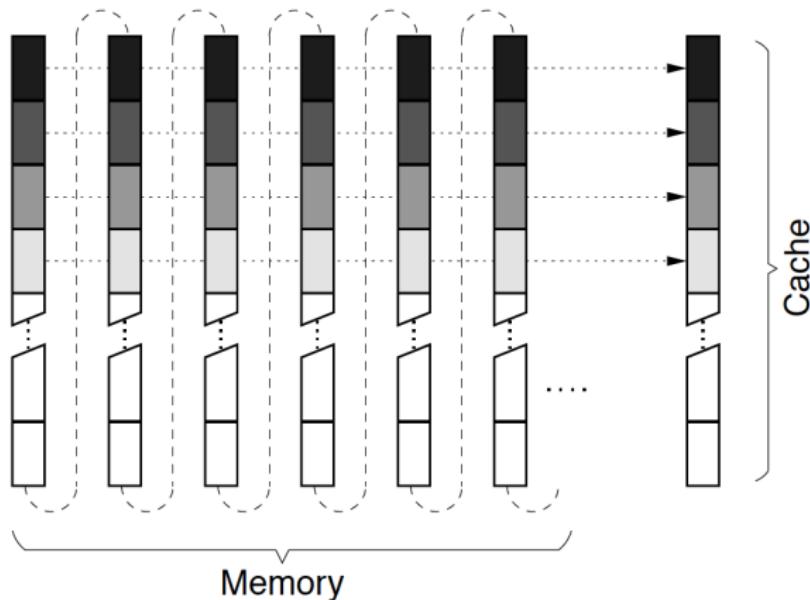
## Cache mapping

If a line of data items from main memory, to be loaded into cache, can be freely placed on any unoccupied cache line, it is called a *fully associative* mapping.

Unfortunately, it is hard to build large, fast and fully associative caches because of large bookkeeping overhead.

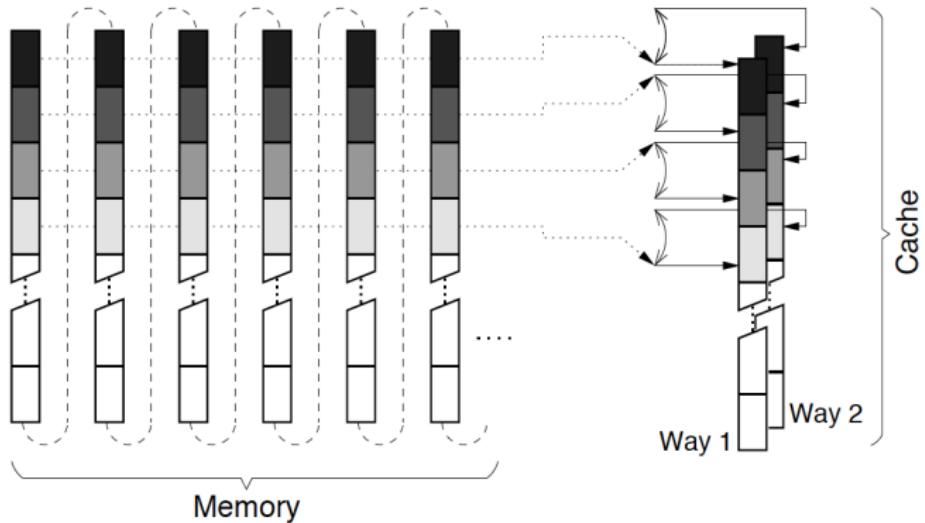
On the other end, a *directly-mapped* cache—a line of data items can be placed only on a prescribed cache line—runs the risk of low cache utilization.

## Directly-mapped cache



**Figure 1.10:** In a direct-mapped cache, memory locations which lie a multiple of the cache size apart are mapped to the same cache line (shaded boxes).

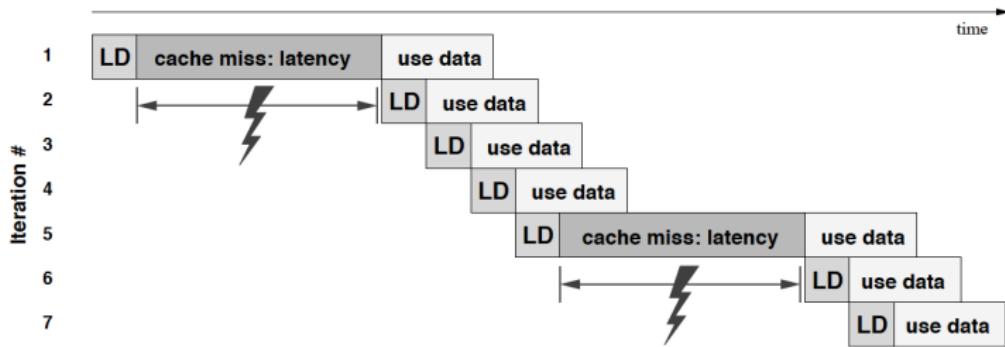
## $m$ -way associative



**Figure 1.11:** In an  $m$ -way set-associative cache, memory locations which are located a multiple of  $\frac{1}{m}$ th of the cache size apart can be mapped to either of  $m$  cache lines (here shown for  $m = 2$ ).

# The problem of “first cache miss”

Although exploiting spatial locality and cache lines can improve cache efficiency, there is still the problem of latency on the first miss.



**Figure 1.12:** Timing diagram on the influence of cache misses and subsequent latency penalties for a vector norm loop. The penalty occurs on each new miss.

# Prefetch

*Prefetching* supplies the cache with data ahead of the actual requirements from an application code.

Typically, a hardware pre-fetcher can detect regular access patterns and try to read ahead the needed data.

To completely hide the cache miss latency, the memory subsystem must be able to sustain a certain number of outstanding prefetch operations.

## How many outstanding prefetch operations needed?

If  $T_\ell$  is the cache miss latency and  $B$  is the bandwidth.

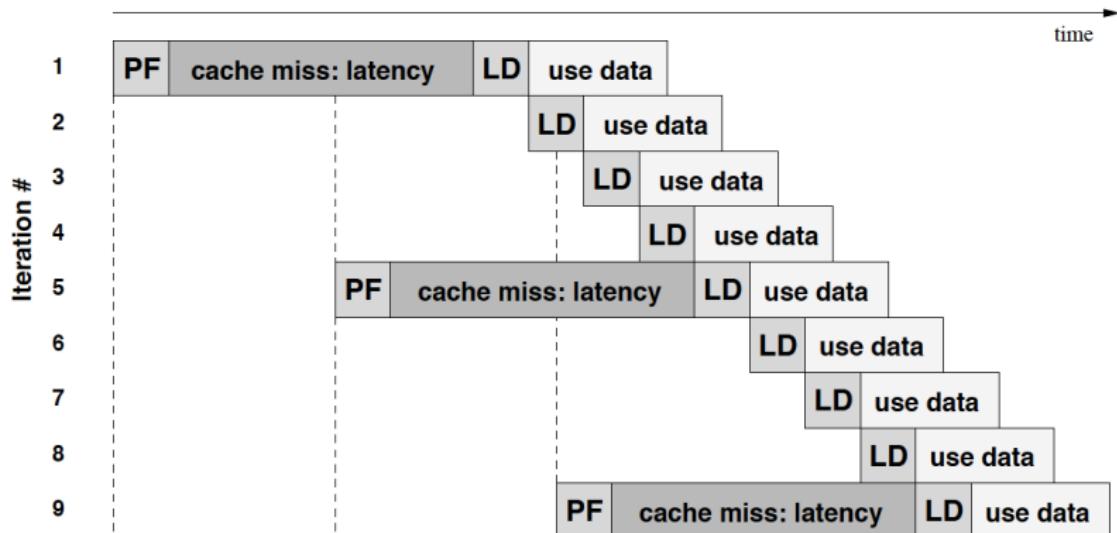
Suppose each cache line is of length  $L_c$  (in bytes), then loading a cache line due to cache miss takes a time of

$$T = T_\ell + \frac{L_c}{B}$$

The number of cache lines that can be transferred (without paying the latency penalty) during time  $T$  is the number of outstanding prefetches,  $P$ , that the processor must be able to sustain. So we have

$$P = \frac{T_\ell + \frac{L_c}{B}}{\frac{L_c}{B}} = 1 + \frac{T_\ell}{\frac{L_c}{B}}$$

# Prefetch helps to overlap computation with data transfer



**Figure 1.13:** Computation and data transfer can be overlapped much better with prefetching. In this example, two outstanding prefetches are required to hide latency completely.

## Multithreaded processors

All modern microprocessors are heavily pipelined. In case there are frequent “pipeline bubbles” caused by, for example,

- dependencies
- memory latencies
- insufficient loop length,
- branch mispredictions

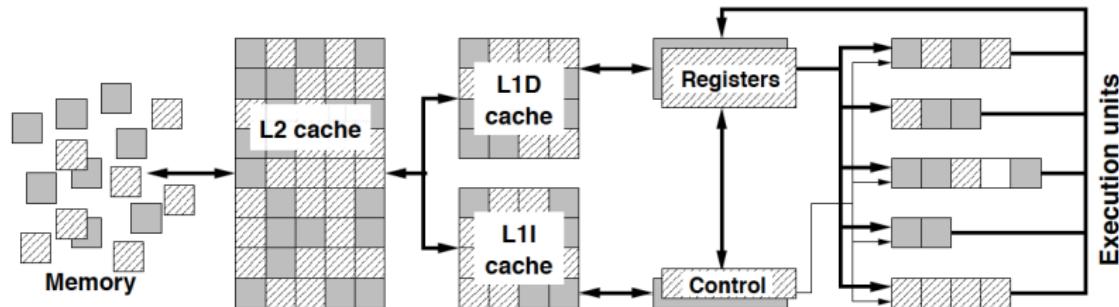
The consequence is that a large part of the execution resources is idle (wasted resources).

# Multithreading

*Hyper threading or simultaneous multithreading* (SMT) capabilities are thus built into modern processors.

- Multiple architectural states of a CPU core
- An architectural state comprises all data, stauts and control registers
- However, resources such as arithmetic units, caches, queues, memory interfaces are *not* duplicated

One CPU core “appears to be composed of several cores (also called *logical processors*). Multiple instruction streams, or threads, can be executed in parallel.



**Figure 1.20:** Simplified diagram of control/data flow in a (multi-)pipelined microprocessor with fine-grained two-way SMT. Two instruction streams (threads) share resources like caches and pipelines but retain their respective architectural state (registers, control units). Graphics by courtesy of Intel.

All threads share the same execution resources, so sometimes it is *possible* to fill pipeline bubbles that arise due to installs in one thread. SMT *may* enhance instruction throughput (instructions executed per cycle).

Whether the concept of SMT pays off is code-dependent and hardware-dependent!

# Performance metrics

Theoretically, the components of a CPU core can operate at some maximum speed called *peak performance*.

Whether this limit can be reached for a specific application code depends on many factors (one of the key topics of Chapter 3).

Performance metrics:

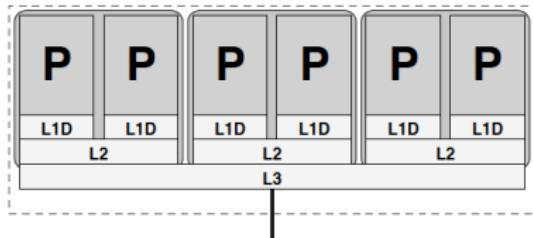
- The performance at which the floating-point units generate results for multiply and add operations is measured as *floating-point operations per second* (Flops/sec).
- The most important data paths are those to and from the caches and main memory. The performance, called *bandwidth*, of these paths is quantified in GBytes/sec.

## Multicore processors

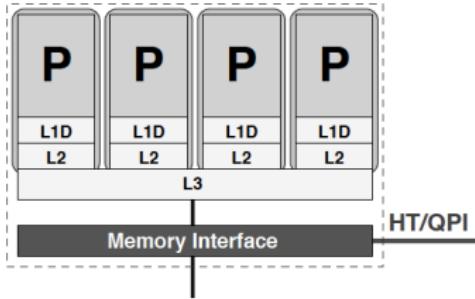
A higher clock frequency will allow a CPU to execute the instructions faster. However, Increasing the clock frequency can have a serious impact on the power dissipation.

On the other hand, reducing the clock frequency allows placing more than one CPU core on the same CPU die (or more generally, the same package), while keeping the same power envelope.

# More about multicore CPU



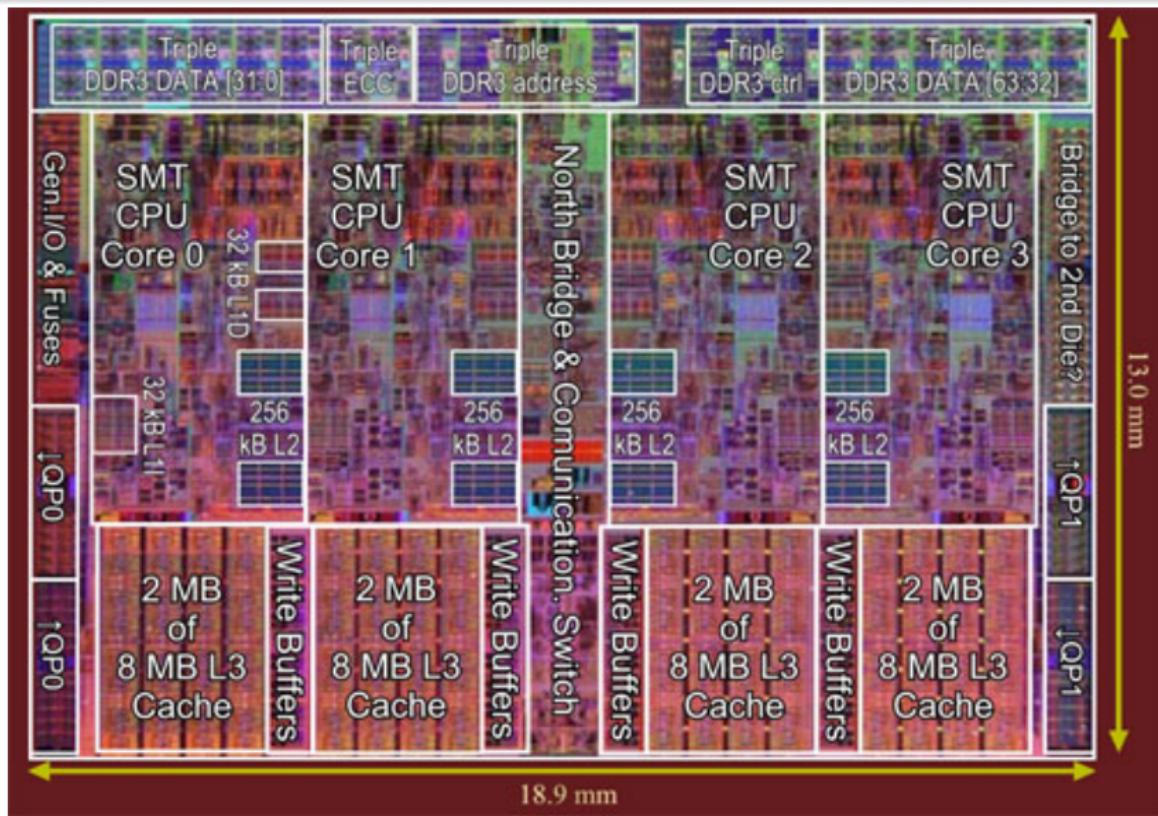
**Figure 1.17:** Hexa-core processor chip with separate L1 caches, shared L2 caches for pairs of cores and a shared L3 cache for all cores (Intel “Dunnington”). L2 groups are dual-cores, and the L3 group is the whole chip.



**Figure 1.18:** Quad-core processor chip with separate L1 and L2 and a shared L3 cache (AMD “Shanghai” and Intel “Nehalem”). There are four single-core L2 groups, and the L3 group is the whole chip. A built-in memory interface allows to attach memory and other sockets directly without a chipset.

The caches on the different levels can be private or shared. Sharing a cache enables superfast communication between the cores. An opposite effect of sharing can be cache bandwidth bottlenecks.

# Example of a 4-core CPU



Intel Nehalem (actually a very old CPU)

# Challenges with using multicore CPUs

- In order to use all the resources belonging to the multiple cores, parallel programming must be adopted. (This will be one of the topics for later lectures.)
- The memory bandwidth available per core can be a challenge. So programming techniques for memory traffic reduction will be even more important!

# Vector processors

- An very important processor architecture for HPC in the past
- However, some of the concepts and techniques related to *vectorization* are still used today

- Instructions operate on *vector registers* that can hold a large number of arguments
- The width of a vector register is called the *vector length*  $L_v$
- MULT and ADD pipelines are *multitrack*
- One or several load, store or combined load/store pipes are connected *directly to main memory*

The paradigm of SIMD

# Block diagram of vector processor

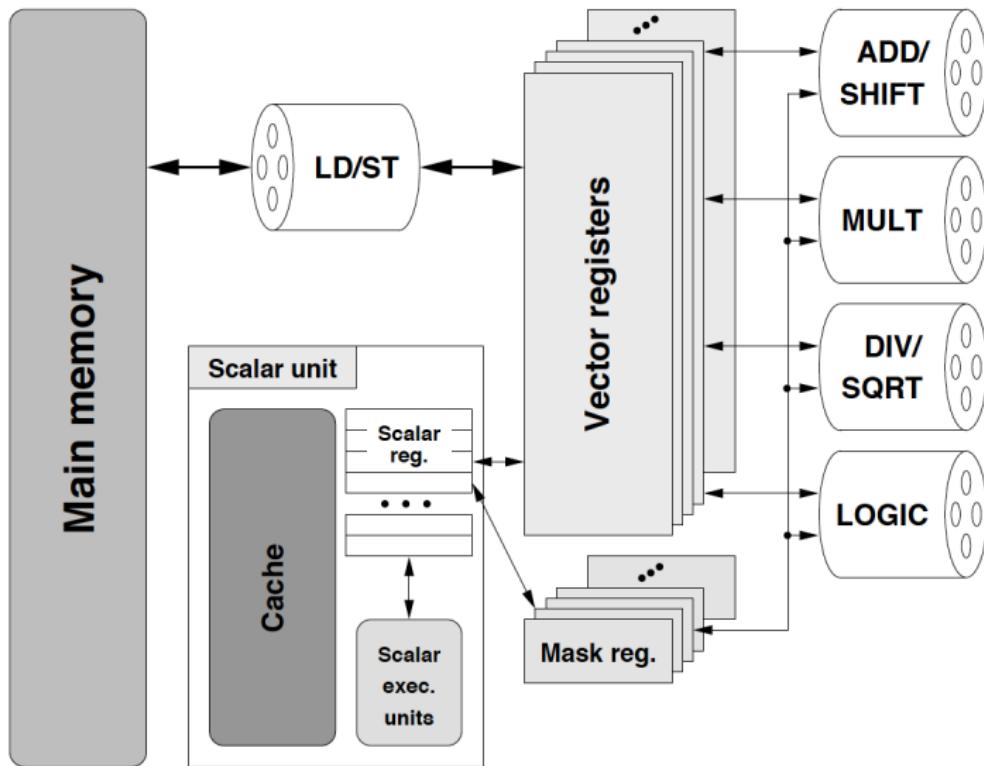


Figure 1.21: Block diagram of a prototypical vector processor with 4-track pipelines.

# SIMD for $A = B + C$

## Target calculation

```
for (s=0; s<N, s++)  
    A[s] = B[s] + C[s];
```

A vectorization-capable compiler will automatically translate into the following pseudocode:

```
for (s=0; s<N, s+=L) {  
    int E = min(N-1,s+L-1);  
    vload V1(0:L-1) = B(s:E);  
    vload V2(0:L-1) = C(s:E);  
    vadd V3(0:L-1) = V1(0:L-1) + V2(0:L-1);  
    vstore A(s:E) = V3(0:L-1);  
}
```

V1 V2 V3: vector registers, L: vector length  $L_v$

# Vectorization

Writing a program so that the compiler can generate effective SIMD vector instruction is called *vectorization*.

Sometime this requires reformulation of code or inserting directives to help the compiler identify SIMD parallelism.

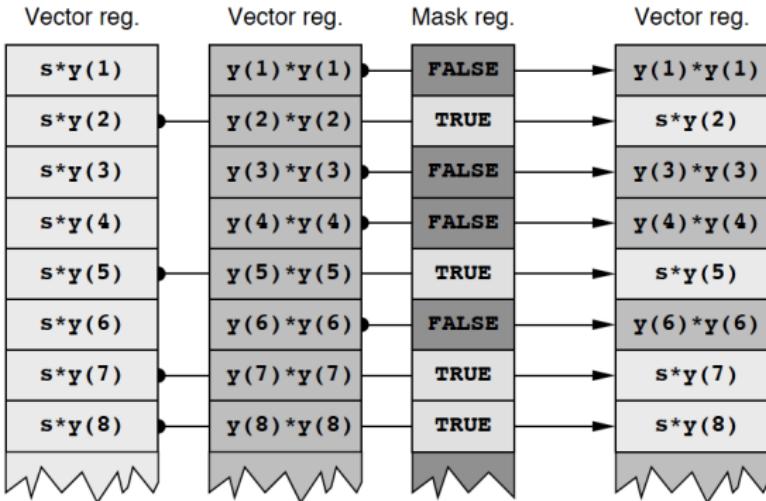
If a code cannot be vectorized, it makes no sense to use a vector computer!

A prerequisite for vectorization is true data independence across iterations of a loop. (Forward references are allowed, but not backward references.)

## Branches in vectorized loops (example 1)

```
for (i=0; i<N; i++) {  
    if (y[i] < 0.)  
        x[i] = s*y[i];  
    else  
        x[i] = y[i]*y[i];  
}
```

# Mask registers



**Figure 1.23:** On a vector processor, a loop with an if/else branch can be vectorized using a mask register.

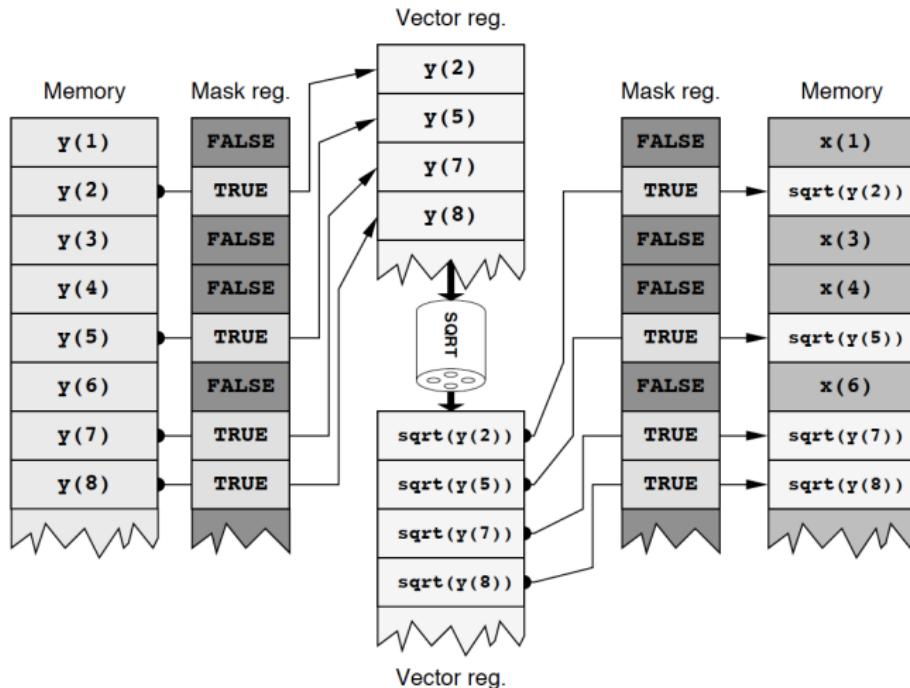
First, a vector of boolean values is generated by the logic pipeline. Then both branches are executed for all loop indices. Finally the boolean vector is used to choose the correct results.

## Branches in vectorized loops (example 2)

```
for (i=0; i<N; i++) {  
    if (y[i] > 0.)  
        x[i] = sqrt(y[i]);  
}
```

There is only the `if` branch (no `else` branch). Also, the `sqrt` calculation is expensive. Execution for all loop indices can be a huge waste of resource. What should be done in such a case?

# The gather/scatter method



**Figure 1.24:** Vectorization by the gather/scatter method. Data transfer from/to main memory occurs only for those elements whose corresponding mask entry is true. The same mask is used for loading and storing data.

# IN3200/IN4200: Chapter 2

## Basic optimization techniques for serial code

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

## Objectives of Chapter 2

- “Common sense” and simple optimization strategies for serial code
- (Data access optimization will be discussed in Chapter 3)
- The role of compilers
- Basics of performance profiling

## “Common sense” optimizations

Very simple code changes can sometimes lead to significant performance boost.

The most important “common sense” principle: **avoiding performance pitfalls!**

## Do less work; example 1

Example: assume A is an array of numerical values, and a prescribed threshold value: threshold\_value.

```
int flag = 0;  
for (i=0; i<N; i++) {  
    if ( some_function(A[i]) < threshold_value )  
        flag = 1;  
}
```

Improvement: leave the loop as soon as flag becomes 1.

```
int flag = 0;  
for (i=0; i<N; i++) {  
    if ( some_function(A[i]) < threshold_value ) {  
        flag = 1;  
        break;  
    }  
}
```

## Do less work; example 2

```
for (i=0; i<500; i++)
    for (j=0; j<80; j++)
        for (k=0; k<4; k++)
            a[i][j][k] = a[i][j][k] + b[i][j][k]*c[i][j][k];
```

How many times is the k-indexed loop executed? And how many times for the j-indexed loop?

## Do less work; example 2 (cont'd)

If the 3D arrays a, b and c have **contiguous** memory storage for all their values, then we can re-code as follows:

```
double *a_ptr = a[0][0];
double *b_ptr = b[0][0];
double *c_ptr = c[0][0];

for (i=0; i<(500*80*4); i++)
    a_ptr[i] = a_ptr[i] + b_ptr[i]*c_ptr[i];
```

This technique is called *loop collapsing*. The main motivation is to reduce loop overhead, may also help other (compiler-supported) optimizations.

## Do less work; example 3

```
for (i=0; i<ARRAY_SIZE; i++) {  
    a[i] = 0.;  
    for (j=0; j<ARRAY_SIZE; j++)  
        a[i] = a[i] + b[j]*d[j]*c[i];  
}
```

Observation:  $c[i]$  is independent of the  $j$ -indexed loop.

## Do less work; example 3 (cont'd)

Improvement:

```
for (i=0; i<ARRAY_SIZE; i++) {  
    a[i] = 0.;  
    for (j=0; j<ARRAY_SIZE; j++)  
        a[i] = a[i] + b[j]*d[j];  
    a[i] = a[i]*c[i];  
}
```

Can we improve further?

## Do less work; example 3 (further simplification)

There is a common factor:

$b[0]*d[0]+b[1]*d[1]+\dots+b[\text{ARRAY\_SIZE}-1]*d[\text{ARRAY\_SIZE}-1]$

which is unnecessarily re-computed in every  $i$  iteration!

```
t = 0.;  
for (j=0; j<ARRAY_SIZE; j++)  
    t = t + b[j]*d[j];
```

```
for (i=0; i<ARRAY_SIZE; i++)  
    a[i] = t*c[i];
```

This technique is called *loop factoring* or *elimination of common subexpressions*.

## Another example of common subexpression elimination

```
for (i=0; i<N; i++)
    A[i] = A[i] + s + r*sin(x);
```

⇓

```
tmp = s + r*sin(x);
for (i=0; i<N; i++)
    A[i] = A[i] + tmp;
```

# Avoid expensive operations!

Special math functions (such as trigonometric, exponential and logarithmic functions) are usually very costly to compute.

An example from simulating non-equilibrium spins:

```
for (i=1; i<Nx-1; i++)
    for (j=1; j<Ny-1; j++)
        for (k=1; k<Nz-1; k++) {
            iL = spin_orientation[i-1] [j] [k];
            iR = spin_orientation[i+1] [j] [k];
            iS = spin_orientation[i] [j-1] [k];
            iN = spin_orientation[i] [j+1] [k];
            iO = spin_orientation[i] [j] [k-1];
            iU = spin_orientation[i] [j] [k+1];
            edelz = iL+iR+iS+iN+iO+iU;
            body_force[i] [j] [k] = 0.5*(1.0+tanh(edelz/tt));
        }
```

## Example continued

If the values of  $iL$ ,  $iR$ ,  $iS$ ,  $iN$ ,  $iO$ ,  $iU$  can only be  $-1$  or  $+1$ , then the value of  $edelz$  (which is the sum of  $iL$ ,  $iR$ ,  $iS$ ,  $iN$ ,  $iO$ ,  $iU$ ) can only be  $-6, -4, -2, 0, 2, 4, 6$ .

If  $tt$  is a constant, then we can create a lookup table:

```
double tanh_table[13];
for (i=0; i<=12; i+=2)
    tanh_table[i] = 0.5*(1.0+tanh((i-6)/tt));
```



```
for (i=1; i<Nx-1; i++)
    for (j=1; j<Ny-1; j++)
        for (k=1; k<Nz-1; k++) {
            ....
            edelz = iL+iR+iS+iN+iO+iU;
            body_force[i][j][k] = tanh_table[edelz+6];
        }
```

## Strength reduction

```
for (i=0; i<N; i++)
    y[i] = pow(x[i],3)/s;
```

⇓

```
double inverse_s = 1.0/s;
for (i=0; i<N; i++)
    y[i] = x[i]*x[i]*x[i]*inverse_s;
```

## Strength reduction (another example)

```
for (i=0; i<N; i++)
    y[i] = a*pow(x[i],4)+b*pow(x[i],3)+c*pow(x[i],2)
          +d*pow(x[i],1)+e;
```



```
for (i=0; i<N; i++)
    y[i] = (((a*x[i]+b)*x[i]+c)*x[i]+d)*x[i]+e;
```

Use of Horner's rule of polynomial evaluation:

$$ax^4 + bx^3 + cx^2 + dx + e = (((ax + b)x + c)x + d)x + e$$

## Shrinking the work set!

The *work set* of a code is the amount of memory it uses (or touches), also called *memory footprint*.

In general, shrinking the work set (if possible) is a good thing for performance, because it raises the probability of cache hit.

One example: The `spin_orientation` array should store values of type `char` instead of type `int`. (A factor of 4 in the difference of memory footprint.)

# Avoiding branches

“Tight” loops: few operations per iteration, typically optimized by compiler using some form of pipelining. In case of conditional branches in the loop body, the compiler optimization will easily fail.

```
for (j=0; j<N; j++)
    for (i=0; i<N; i++) {
        if (i>j)
            sign = 1.0;
        else if (i<j)
            sign = -1.0;
        else
            sign = 0.0;

        C[j] = C[j] + sign * A[j][i] * B[i];
    }
```

## Avoiding branches (cont'd)

```
for (j=0; j<N-1; j++)
    for (i=j+1; i<N; i++)
        C[j] = C[j] + A[j][i] * B[i];

for (j=1; j<N; j++)
    for (i=0; i<j; i++)
        C[j] = C[j] - A[j][i] * B[i];
}
```

We have got rid of the if-tests completely!

## Another example of avoiding branches

```
for (i=0; i<n; i++) {  
    if (i==0)  
        a[i] = b[i+1]-b[i];  
    else if (i==n-1)  
        a[i] = b[i]-b[i-1];  
    else  
        a[i] = b[i+1]-b[i-1];  
}
```

## Another example of avoid branches (cont'd)

Using the technique of *loop peeling*, we can re-code as follows:

```
a[0] = b[1]-b[0];  
for (i=1; i<n-1; i++)  
    a[i] = b[i+1]-b[i-1];  
a[n-1] = b[n-1]-b[n-2];
```

## Yet another example of avoiding branches

```
for (i=0; i<n; i++) {  
    if (j>0)  
        x[i] = x[i] + 1;  
    else  
        x[i] = 0;  
}
```



```
if (j>0)  
    for (i=0; i<n; i++)  
        x[i] = x[i] + 1;  
else  
    for (i=0; i<n; i++)  
        x[i] = 0;
```

## Using SIMD instructions

A “vectorizable” loop can potentially run faster if multiple operations can be performed with a single instruction.

Using SIMD instructions, register-to-register operations will be greatly accelerated.

**Warning:** if the code is strongly limited by memory bandwidth, no SIMD technique can bridge this gap.

## Ideal scenario for applying SIMD to a loop

- All iterations are independent
- There is no branch in the loop body
- The arrays are accessed with a stride of one

Example:

```
for (i=0; i<N; i++)
    r[i] = x[i] + y[i];
```

(We assume here that the memory regions pointed by `r`, `x`, `y` do not overlap—no aliasing)

# An example of applying SIMD

**Pseudocode** of applying SIMD (assuming that each SIMD register can store 4 values):

```
int i, rest = N%4;
for (i=0; i<N-rest; i+=4) {
    load R1 = [x[i],x[i+1],x[i+2],x[i+3]];
    load R2 = [y[i],y[i+1],y[i+2],y[i+3]];
    R3 = ADD(R1,R2);
    store [r[i],r[i+1],r[i+2],r[i+3]] = R3;
}
for (i=N-rest; i<N; i++)
    r[i] = x[i] + y[i];
```

# Beware of loop dependency!

If a loop iteration depends on the result of another iteration—**loop-carried dependency**

```
for (i=start; i<end; i++)
    A[i] = 10.0*A[i+offset];
```

If  $\text{offset} < 0 \rightarrow \text{real dependency (read-after-write hazard)}$

If  $\text{offset} > 0 \rightarrow \text{pseudo dependency (write-after-read hazard)}$

## When there is loop-carried dependency...

In case of real dependency, SIMD cannot be applied if the negative offset size is smaller than the SIMD width. For example,

```
for (i=start; i<end; i++)
    A[i] = 10.0*A[i-1];
```

In case of pseudo dependency, SIMD can be applied. For example when `offset>0`,

```
for (i=start; i<end; i++)
    A[i] = 10.0*A[i+offset];
```

## Risk of aliasing

Is it safe to vectorize the following function?

```
void compute(int start, int stop, double *a, double *b) {  
    for (int i=start; i<stop; i++)  
        a[i] = 10.0*b[i];  
}
```

## Risk of aliasing (cont'd)

A problem of “aliasing” will arise if the `compute` function is called as follows

```
compute(0, N-1, &(array_a[1]), array_a);
```

If a programmer can guarantee that aliasing won't happen, this hint can be provided to the compiler.

# The role of compilers

A compiler translates a program, which is implemented in a programming language, to machine code.

A compiler can carry out code optimization of various degrees, dictated by the compiler options provided by the user. (-O0, -O1, -O2, ....)

Different compilers probably allow different compiler options, should refer to the user manual!

Numerical accuracy **may** suffer from too aggressive compiler optimizations.

# Profiling

Profiling—gather information about a program's behavior, especially its use of resources. The purpose is to pinpoint the “hot spots”, and more importantly, to identify any performance optimization opportunities (if any) and/or bugs.

Two approaches of “information gathering”:

- Instrumentation—compiler automatically inserts some code to *log* each function call during the actual execution
- Sampling—the program execution is interrupted at periodic intervals, with information being recorded

## GNU gprof

One well-known profiler: GNU gprof

<https://sourceware.org/binutils/docs/gprof/>

- Step 1: compile and link the program with profiling enabled;
- Step 2: execute the program to generate a profile data file;
- Step 3: run gprof to analyze the profile data.

(There are other profilers, of course.)

# Hardware performance counters

Knowing how much time is spent where is the first step. But what is the actual reason for “a slow code” or by which resource is the performance limited?

Modern processors feature a small number of *performance counters*, which are special on-chip registers that get incremented each time a certain event occurs.

Possible events that can be monitored:

- number of cache line transfers
- number of loads and stores
- number of floating-point operations
- number of branch mispredictions
- number of pipeline stalls
- number of instructions executed

# IN3200/IN4200: Chapter 3

## Data access optimization

### (Part 1)

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

# Objectives

- What is the maximumly achievable performance?
  - Balance analysis and “lightspeed” estimates
- Data access optimization techniques

# Importance of data access

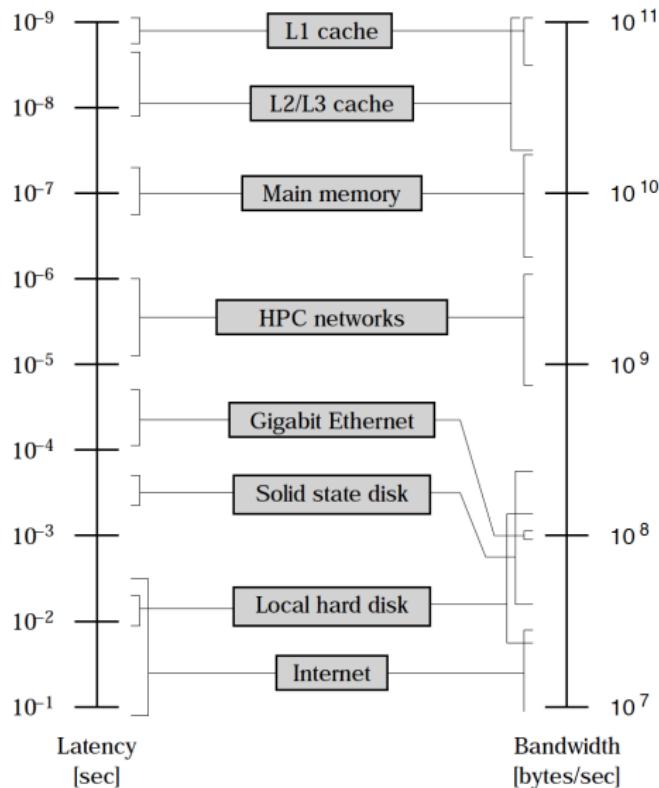
Applications in science and engineering mostly consist of **loop-based** code that moves large amounts of data in and out of the CPU.

Accessing data in the memory hierarchy (from L1 cache to main memory) is often the most prominent performance limiter.

Modern microprocessors have a very impressive theoretical peak performance (in number of FP operations *maximumly* executable per second), but the memory system is “**too slow**”.

# Typical latency and bandwidth numbers

$$\text{time usage} = \text{latency} + \frac{\text{data volume}}{\text{bandwidth}}$$



## Rule of the thumb

**Any optimization attempt, with respect to data access, should first aim at reducing traffic over slow data paths, or, making the data transfer as efficient as possible.**

## Understanding the “limitation”

**Bandwidth-based performance modeling**—to get a rough idea about the maximum performance for a code.

One can *estimate* the theoretically achievable performance of loop-based code, if it is bound by bandwidth limitations.

# The concept of “machine balance”

**Machine balance**,  $B_m$ , of a processor is the ratio between the maximum memory bandwidth and the peak FP performance:

$$B_m = \frac{\text{memory bandwidth [GWords/sec]}}{\text{peak FP performance [GFlops/sec]}} = \frac{b_{\max}}{P_{\max}}$$

Access latency is assumed to be hidden completely (for example thanks to prefetch).

“Word” = one DP value (8 bytes)

“Memory bandwidth” could also be substituted by the bandwidth to caches or even network bandwidth.

## Example values of machine balance

| data path                    | balance [W/F] |
|------------------------------|---------------|
| cache                        | 0.5–1.0       |
| <b>machine (memory)</b>      | 0.03–0.5      |
| interconnect (high speed)    | 0.001–0.02    |
| interconnect (GBit ethernet) | 0.0001–0.0007 |
| disk (or disk subsystem)     | 0.0001–0.01   |

**Table 3.1:** Typical balance values for operations limited by different transfer paths. In case of network and disk connections, the peak performance of typical dual-socket compute nodes was taken as a basis.

The above values are somewhat outdated.

The increase of memory bandwidth typically falls behind the increase of FP performance—the ever-increasing **DRAM gap**.

## A more recent example of machine balance



Intel Xeon Skylake Platinum 28-core CPU (model 8180)

- Peak memory bandwidth:  
6 memory channels  $\times$  2.666 GT/sec  $\times$  1 word/transfer =  
16 GWords/sec (G: giga ( $10^6$ ) T: transfer)
- Peak double-precision FP performance:  
28 cores  $\times$  2.3 GHz AVX-512 clock rate  $\times$  32 Flops/cycle =  
2061 GFlops/sec (Each core: 2 FP pipelines, 512-bit SIMD,  
fused multiply-add)
- So the machine balance is only  $\frac{16}{2061} = 0.00776$

## The concept of “code balance”

To characterize a loop, we can calculate the **code balance**  $B_c$ :

$$B_c = \frac{\text{data traffic [Words]}}{\text{floating-point operations [Flops]}}$$

That is, you should count the number of FP operations (easy), and also count (or estimate) the amount of data transferred over the performance-limiting data path (can be difficult).

Note:  $\frac{1}{B_c}$  is called **computational intensity**.

## What is the expected maximum performance of a loop?

When you know the machine balance  $B_m$  of a CPU, and you want to run a loop that has  $B_c$  as its code balance.

What will be the maximum achievable performance  $P$  (in Flops/sec)?

$$P = \min \left( P_{\max}, \frac{b_{\max}}{B_c} \right)$$

Recall:  $P_{\max}$  denotes the maximum FP performance,  $b_{\max}$  denotes the maximum bandwidth of the performance-limiting data path.

## Comparing $P$ with $P_{\max}$

- In case  $P \approx P_{\max}$ : the achievable performance is not limited by bandwidth (so data access optimization is **not** needed).
- In case  $P \ll P_{\max}$ : more analysis is needed to find out whether the code balance  $B_c$  can be improved, that is, making  $B_c$  smaller by data access optimization. (Note: smaller  $B_c \rightarrow$  higher  $P = \frac{b_{\max}}{B_c}$ )

## “Lightspeed” of a loop

$$\frac{P}{P_{\max}} = \min \left( 1, \frac{B_m}{B_c} \right)$$

is the maximum achievable fraction of peak performance for a code with balance  $B_c$  on a machine with balance  $B_m$ —also called the **lightspeed** of a loop.

## Example of “balance analysis”

```
for (i=0; i<N; i++)  
    A[i] = B[i] + C[i]*D[i];
```

- Each iteration has three loads ( $B[i]$ ,  $C[i]$ ,  $D[i]$ ), one store ( $A[i]$ ) and two floating-point operations
- Code balance:  $B_c = \frac{3+1}{2} = 2$
- If a CPU has machine balance  $B_m = 0.1$ , then the maximumly achievable performance is  $\frac{B_m}{B_c} P_{\max}$ , that is, 5% of the peak FP performance
- On cache-based microprocessors, each store miss may incur a *cache line write allocate*, if non-temporal stores are not used. In that case, each store of  $A[i]$  in effect must be counted as a load plus a store,  $B_c$  thus becomes 2.5 → only 4% of  $P_{\max}$  is maximumly achievable.

## How realistic is $b_{\max}$ ?

In reality, even the simplest memory-intensive loops are not able to achieve the theoretical hardware maximum memory bandwidth  $b_{\max}$ .

The well-known stream micro-benchmarks can be used to measure the realistically achievable maximum memory bandwidth.

# STREAM micro-benchmarks

Four micro-benchmarks (<https://www.cs.virginia.edu/stream/>)

| type  | kernel                   | DP words | flops | $B_c$     |
|-------|--------------------------|----------|-------|-----------|
| COPY  | $A(:) = B(:)$            | 2 (3)    | 0     | N/A       |
| SCALE | $A(:) = s * B(:)$        | 2 (3)    | 1     | 2.0 (3.0) |
| ADD   | $A(:) = B(:) + C(:)$     | 3 (4)    | 1     | 3.0 (4.0) |
| TRIAD | $A(:) = B(:) + s * C(:)$ | 3 (4)    | 2     | 1.5 (2.0) |

**Table 3.2:** The STREAM benchmark kernels with their respective data transfer volumes (third column) and floating-point operations (fourth column) per iteration. Numbers in brackets take write allocates into account.

## More realistic “balance analysis”

We will from now on use the realistically achievable memory bandwidth,  $b_S$ , which is measured by STREAM.

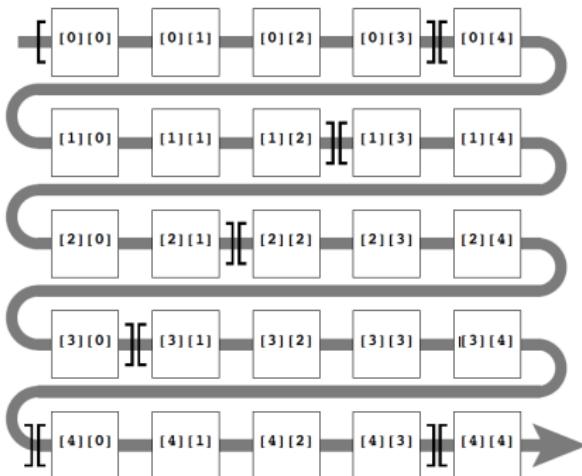
Then, the realistically achievable maximum FP performance is estimated as

$$P = \min \left( P_{\max}, \frac{b_S}{B_c} \right)$$

# Storage order of multi-dimensional arrays

Multi-dimensional arrays normally have an underlying contiguous 1D storage.

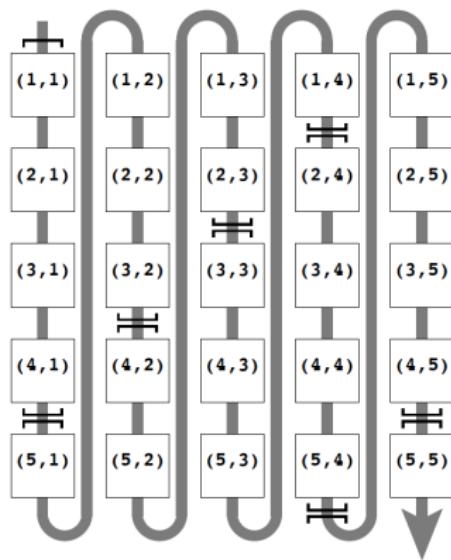
C program typically adopts a **row-major** storage order.



**Figure 3.3:** Row major order matrix storage scheme, as used by the C programming language. Matrix rows are stored consecutively in memory. Cache lines are assumed to hold four matrix elements and are indicated by brackets.

## Storage order of multi-dimensional arrays (cont'd)

Fortran program typically adopts a **column-major** storage order.



**Figure 3.4:** Column major order matrix storage scheme, as used by the Fortran programming language. Matrix columns are stored consecutively in memory. Cache lines are assumed to hold four matrix elements and are indicated by brackets.

(Read the textbook with care, because most coding examples are in Fortran.)

## Use unit-stride to access arrays, if possible

Assume that 2D array A has row-major storage order.

```
for (i=0; i<N; i++)
    for (j=1; j<N; j++)
        A[i][j] = i*j;    // stride-1 access, good
```

```
for (i=0; i<N; i++)
    for (j=1; j<N; j++)
        A[j][i] = i*j;    // stride-N access, bad!!!
```

## Cache lines (repetition of knowledge from Chapter 1)

The content of a cache is organized as **cache lines**. (Each cache line has space for multiple data items.)

All data transfers between caches and main memory happen on the cache line level. (**Must load/store an entire cache line, cannot only load/store a single data item.**)

When a new cache line is loaded into the cache, but all its possible locations are occupied, one of the old occupant cache lines needs to be “kicked out” (evicted). The most commonly used policy is to evict the **least-recently used** cache line.

## Case study: The 2D Jacobi algorithm

Skipping the mathematical and numerical details (which are given in the textbook), let us focus on the following computation:

```
for (it=0; it<itmax; it++) {  
    for (k=1; k<kmax-1; k++)  
        for (i=1; i<imax-1; i++)  
            phi_new[k][i] = (phi[k-1][i]+ph[k][i-1]  
                            +phi[k][i+1]+phi[k+1][i])*0.25;  
    /* pointer swapping */  
    temp_ptr = phi_new;  
    phi_new = phi;  
    phi = temp_ptr;  
}
```

Note: both phi\_new and phi are 2D arrays (**row-major storage**)

## 2D Jacobi: performance prediction

Balance analysis applied to 2D Jacobi:

- 4 floating-point operations per  $(k, i)$
- 1 store to memory per  $(k, i)$
- **How many loads from memory per  $(k, i)$ ?**  
*(It depends on the cache size.)*

## 2D Jacobi: performance prediction (cont'd)

Suppose the cache is very small, that is, not enough to even store one row of phi. Then, memory load traffic needed for computing  $\text{phi\_new}[k][i]$  is as follows:

- The  $\text{phi}[k-1][i]$  value has to be loaded from memory again (although it was loaded from memory and evicted twice already);
- The  $\text{phi}[k][i-1]$  value is guaranteed to be in cache (it was latest loaded from memory for computing  $\text{phi\_new}[k][i-2]$ );
- The  $\text{phi}[k][i+1]$  has to be loaded from memory again (and will be immediately reused for computing  $\text{phi\_new}[k][i+2]$ );
- The  $\text{phi}[k+1][i]$  value has to be loaded from memory (and it will be evicted from the cache before needed again);

Therefore, 3 memory loads per  $(k, i) \rightarrow B_c = \frac{3 \text{ loads} + 1 \text{ store}}{4 \text{ FPs}}$

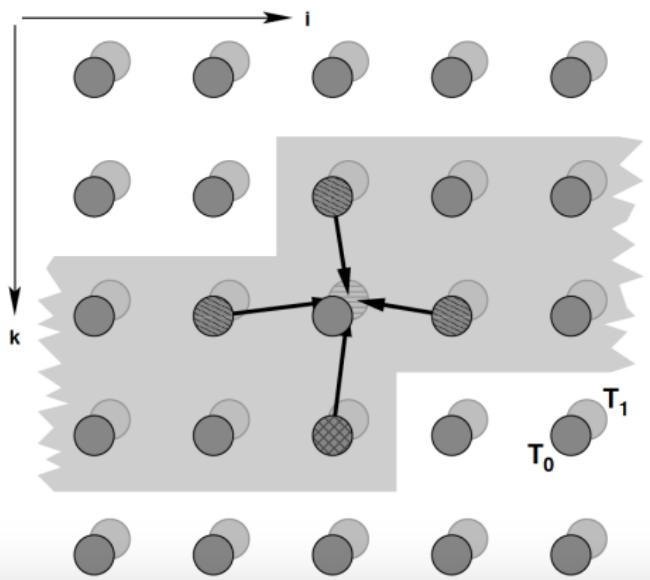
## 2D Jacobi: performance prediction (cont'd)

Suppose the cache can store at least two rows of  $\phi$ , but not enough to store the entire array  $\phi$ . Then, memory load traffic needed for computing  $\phi_{\text{new}}[k][i]$  is as follows:

- The  $\phi[k-1][i]$  value is still in cache (it was first loaded from memory for computing  $\phi_{\text{new}}[k-2][i]$ );
- The  $\phi[k][i-1]$  value is in cache;
- The  $\phi[k][i+1]$  value is also in cache;
- The  $\phi[k+1][i]$  value has to be loaded from memory (and it will be reused twice during computation on row  $k+1$ , and reused once on row  $k+2$ );

In effect, 1 memory load per  $(k, i) \rightarrow B_c = \frac{1 \text{ load} + 1 \text{ store}}{4 \text{ FPs}}$

## The case of (a little more than) 2 rows fit in cache



**Figure 3.5:** Stencil update for the plain 2D Jacobi algorithm. If at least two successive rows can be kept in the cache (shaded area), only one  $T_0$  site per update has to be fetched from memory (cross-hatched site).

## A question to ponder

What will be the code balance if the cache has space for one row, but not two rows?

# IN3200/IN4200: Chapter 3

## Data access optimization

### (Part 2)

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

## Quick repetition of Part 1

**Bandwidth-based performance modeling/prediction**—to get a rough idea about the maximumly achievable performance of a code.

One can *estimate* the maximumly achievable performance of a code, if we know a characteristic ratio that describes the processor (*machine balance*) and a characteristic ratio that describes the code (*code balance*).

## Repetition; The concept of “machine balance”

**Machine balance**,  $B_m$ , of a processor is the ratio between the maximum memory bandwidth and the peak FP (*floating-point*) performance:

$$B_m = \frac{\text{memory bandwidth [GWords/sec]}}{\text{peak FP performance [GFlops/sec]}} = \frac{b_{\max}}{P_{\max}}$$

“Word” = one DP (*double-precision*) value (8 bytes)

The machine balance for a modern processor has typically a very small value (meaning the memory is “slow” relative to floating-point operations).

## Repetition; The concept of “code balance”

To characterize a code, we can calculate the **code balance**  $B_c$ :

$$B_c = \frac{\text{data traffic [Words]}}{\text{floating-point operations [Flops]}}$$

That is, you need to count the number of FP operations (easy), and also count (or estimate) the number of data words transferred over the performance-limiting data path. (Counting the actual amount of transfers can be non-trivial.)

## Repetition; To estimate maximumly achievable performance

When you know the machine balance  $B_m$  of a CPU, and you want to run a code with  $B_c$  as its code balance.

What will be the maximumly achievable performance  $P$  (in Flops/sec)?

$$P = \min \left( P_{\max}, \frac{b_{\max}}{B_c} \right)$$

Recall:  $P_{\max}$  denotes the theoretical peak FP performance,  $b_{\max}$  denotes the theoretical maximum bandwidth of the performance-limiting data path. (To be more realistic,  $b_{\max}$  can be replaced by  $b_S$ , which denotes realistically achievable memory bandwidth.)

In case  $P \ll P_{\max}$ : more analysis is needed to find out whether the code balance  $B_c$  can be improved by data access optimization (that is, decreasing memory traffic).

## Revisiting the first example of “balance analysis”

```
for (i=0; i<N; i++)  
    A[i] = B[i] + C[i]*D[i];
```

- Each iteration has three loads ( $B[i]$ ,  $C[i]$ ,  $D[i]$ ), one store ( $A[i]$ ) and two floating-point operations  $\Rightarrow B_c = \frac{3+1}{2} = 2$
- Indeed, data traffic to/from memory is always in **cachelines**, but this doesn't change the code balance for this example:
  - Suppose each cacheline has space for 8 words
  - One cacheline containing 8 values of array A is stored to memory every 8th iteration
  - One cacheline containing 8 values of array B is loaded from memory every 8th iteration
  - One cacheline containing 8 values of array C is loaded from memory every 8th iteration
  - One cacheline containing 8 values of array D is loaded from memory every 8th iteration
  - For every 8 iterations, 4 cachelines are stored/loaded to/from memory, that is, 32 words of data traffic
  - During 8 iterations, 16 floating-point operations executed
  - Code balance is still  $2 = \frac{32}{16}$

## Case study: The 2D Jacobi algorithm

Skipping the mathematical and numerical details (given in Section 3.3 of the textbook), let us focus on the following computation:

```
for (it=0; it<itmax; it++) {  
    for (k=1; k<kmax-1; k++)  
        for (i=1; i<imax-1; i++)  
            phi_new[k][i] = (phi[k-1][i]+ph[k][i-1]  
                            +phi[k][i+1]+phi[k+1][i])*0.25;  
    /* pointer swapping */  
    temp_ptr = phi_new;  
    phi_new = phi;  
    phi = temp_ptr;  
}
```

Note: both `phi_new` and `phi` are 2D arrays (row-major storage,  
**different from the Fortran code example used in the textbook!**)

## 2D Jacobi: performance prediction

Balance analysis applied to 2D Jacobi:

- 4 floating-point operations per  $(k, i)$  per it iteration
- 1 store to memory per  $(k, i)$  per it iteration
- **How many loads from memory per  $(k, i)$  per it iteration?**  
*(It depends on the cache size.)*

## 2D Jacobi: performance prediction (cont'd)

Suppose the (last-level) cache is very small, that is, not enough to even store one row of  $\phi$ . Then, memory load traffic needed for computing  $\phi_{\text{new}}[k][i]$  is as follows:

- The  $\phi[k-1][i]$  value has to be loaded from memory again (although it was loaded from memory twice already);
- The  $\phi[k][i-1]$  value is guaranteed to be already in cache (it was recently loaded again from memory for computing  $\phi_{\text{new}}[k][i-2]$ );
- The  $\phi[k][i+1]$  has to be loaded again from memory for computing  $\phi_{\text{new}}[k][i]$  (and will be immediately reused for computing  $\phi_{\text{new}}[k][i+2]$ );
- The  $\phi[k+1][i]$  value has to be loaded from memory (and it will be evicted from the cache before needed again);

Therefore, 3 memory loads per  $(k, i)$   $\rightarrow B_c = \frac{3 \text{ loads} + 1 \text{ store}}{4 \text{ FPs}}$

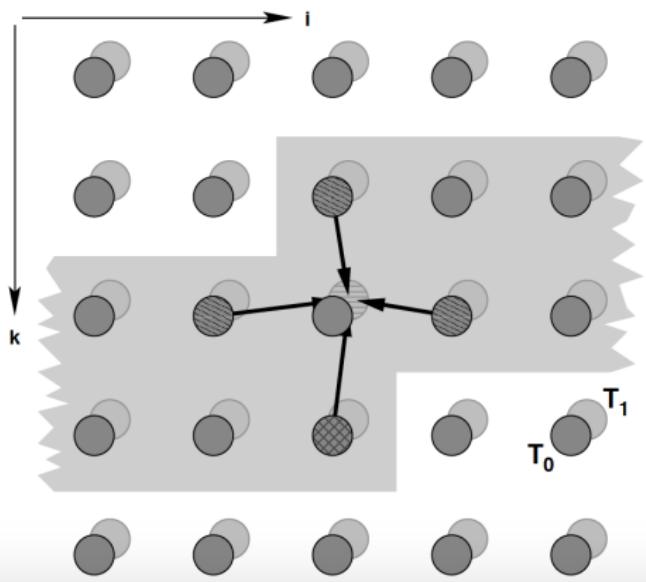
## 2D Jacobi: performance prediction (cont'd)

Suppose the cache can store at least two rows of phi, but not enough to store the entire array phi. Then, memory load traffic needed for computing  $\text{phi\_new}[k][i]$  is as follows:

- The  $\text{phi}[k-1][i]$  value is still in cache (it was first loaded from memory for computing  $\text{phi\_new}[k-2][i]$ );
- The  $\text{phi}[k][i-1]$  value is still in cache;
- The  $\text{phi}[k][i+1]$  value is also still in cache;
- The  $\text{phi}[k+1][i]$  value has to be loaded from memory (and it will be reused during computation on rows  $k+1$  and  $k+2$ );

In effect, 1 memory load per  $(k, i) \rightarrow B_c = \frac{1}{4} \frac{\text{load} + 1}{\text{store}} \text{FPs}$

## The case of 2 rows fit in cache



**Figure 3.5:** Stencil update for the plain 2D Jacobi algorithm. If at least two successive rows can be kept in the cache (shaded area), only one  $T_0$  site per update has to be fetched from memory (cross-hatched site).

# Access optimization for algorithm class $O(N)/O(N)$

## Algorithm class $O(N)/O(N)$

- 1D loops ( $N$ : loop length)
- 1D arrays ( $N$ : array length)

Normally not much room for data access optimization, but *loop fusion* can **sometimes** help.

## Example of loop fusion

Original code: two loops after each other:

```
for (i=0; i<N; i++) {  
    A[i] = B[i] + C[i];  
}
```

```
for (i=0; i<N; i++) {  
    Z[i] = B[i] + E[i];  
}
```

- Number of floating-point operations:  $2N$
- Number of memory loads & stores:  $4N + 2N$

Code balance:  $B_c = \frac{6}{2}$ , can we improve?

## Example of loop fusion (cont'd)

Loop fusion:

```
for (i=0; i<N; i++) {  
    A[i] = B[i] + C[i];  
    Z[i] = B[i] + E[i];  
}
```

- Now each  $B[i]$  value is only loaded once instead of twice!
- New code balance:  $B_c = \frac{5}{2}$
- Loop fusion will also reduce looping overhead
- Beware of the limited register resources: The code body of each iteration shouldn't be too large. (Otherwise, *register spilling* can lead to performance degradation.)

## Access optimization for algorithm class $O(N^2)/O(N^2)$

### Algorithm class $O(N^2)/O(N^2)$

- Two-level loop nests ( $N$ : loop length on each level)
- Number of floating-point operations:  $O(N^2)$
- Number of memory loads & stores:  $O(N^2)$

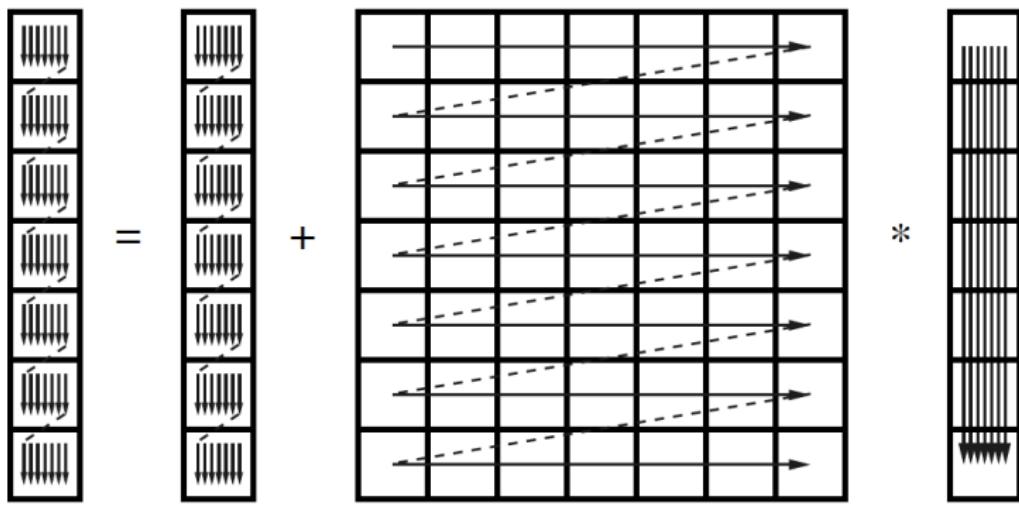
There is more room for data access optimization (than the class of  $O(N)/O(N)$ )

## Dense matrix-vector multiply

```
for (i=0; i<N; i++) {  
    double tmp = C[i];  
    for (j=0; j<N; j++)  
        tmp += A[i][j]*B[j];  
    C[i] = tmp;  
}
```

- Number of FP:  $2N^2$
- Number of loads & stores:  $N^2$  for 2D array A,  $2N$  for 1D array C
- But, how many loads are associated with 1D array B?
  - Small cache → array B is loaded  $N$  times →  $N^2$  memory loads
  - Large cache → array B is loaded only once →  $N$  memory loads

## Illustration of array B being loaded $N$ times



**Figure 3.11:** Unoptimized  $N \times N$  dense matrix vector multiply. The RHS vector is loaded  $N$  times.

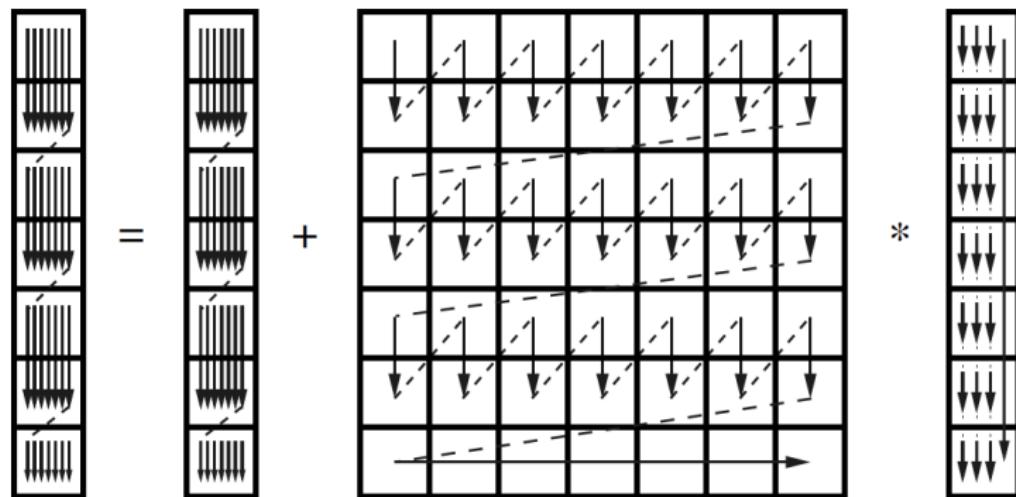
## Loop unrolling

*m*-way unroll and jam:

```
for (i=0; i<N; i+=m) {  
    for (j=0; j<N; j++) {  
        C[i+0] += A[i+0][j]*B[j];  
        C[i+1] += A[i+1][j]*B[j];  
        // ...  
        C[i+m-1] += A[i+m-1][j]*B[j];  
    }  
}  
// remainder code in case (N%m)>0 ....
```

- *m*-fold reuse of each  $B[j]$  from register
- Total number of memory loads and stores:  $N^2 + N^2/m + 2N$  (for small cache size)
- Size of *m* shouldn't be too large, to avoid too high *register pressure*

## Illustration of the effect of unrolling



**Figure 3.12:** Two-way unrolled dense matrix vector multiply. The data traffic caused by reloading the RHS vector is reduced by roughly a factor of two. The remainder loop is only a single (outer) iteration in this example.

## Another $O(N^2)/O(N^2)$ algorithm: matrix transpose

$$A = B^T$$

```
for (j=0; j<N; j++)
    for (i=0; i<N; i++)
        A[j][i] = B[i][j];
```

Both A and B are assumed to be 2D arrays with row-major storage.

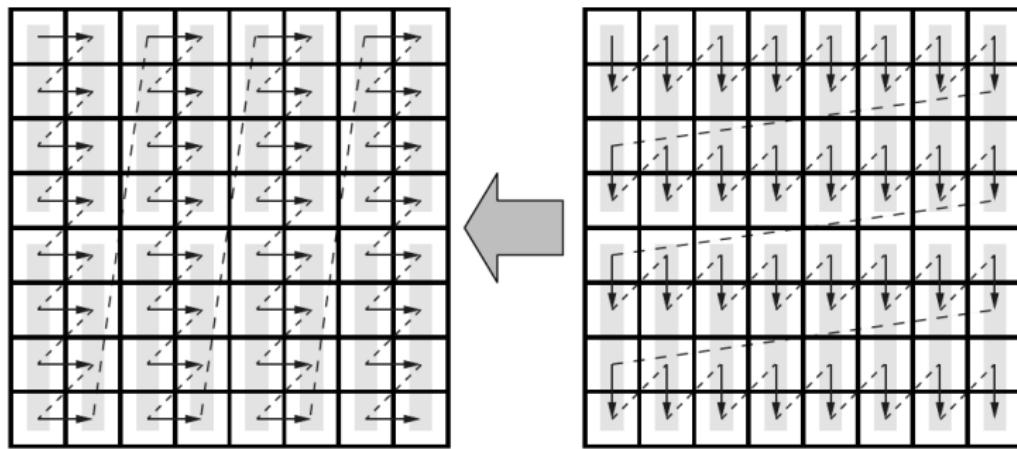
(**Note: The matrix-transpose example in the textbook, Section 3.4, is programmed in Fortran and assumes column-major storage!**)

Very large jumps in memory associated with loading  $B[i][j] \rightarrow$  very bad cache line utilization.

## Loop unrolling applied to matrix transpose

```
for (j=0; j<N; j+=m)
    for (i=0; i<N; i++) {
        A[j+0][i] = B[i][j+0];
        A[j+1][i] = B[i][j+1];
        // ....
        A[j+m-1][i] = B[i][j+m-1];
    }
```

# Illustration



**Figure 3.13:** Two-way unrolled “flipped” matrix transpose (i.e., with strided load in the original version).

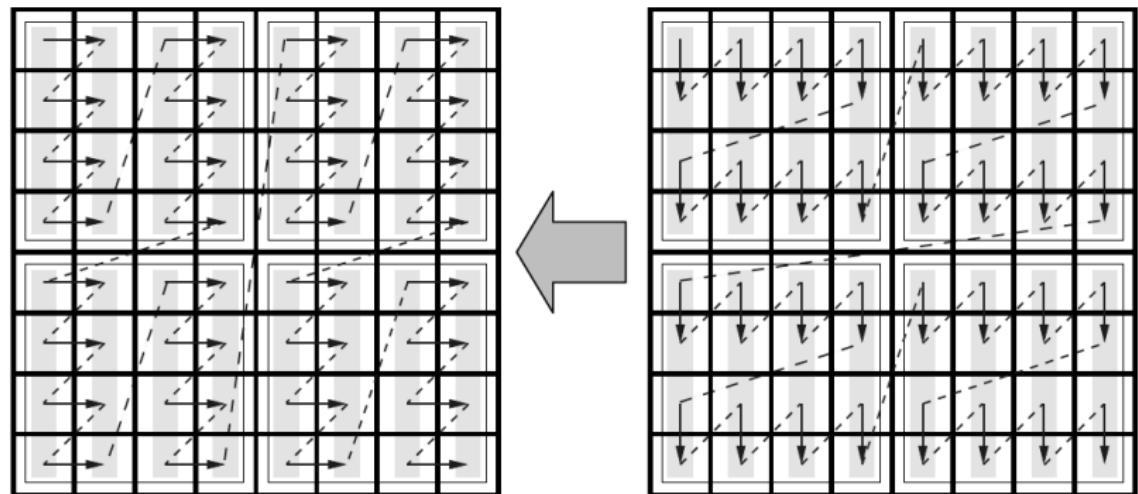
*B*

*A*

## Loop blocking + unrolling

```
for (jj=0; jj<N; jj+=b) {  
    jstart = jj; jstop = jj+b-1;  
    for (ii=0; ii<N; ii+=b) {  
        istart = ii; istop = ii+b-1;  
  
        for (j=jstart; j<=jstop; j+=m)  
            for (i=istart; i<=istop; i++) {  
                A[j+0][i] = B[i][j+0];  
                A[j+1][i] = B[i][j+1];  
                // ....  
                A[j+m-1][i] = B[i][j+m-1];  
            }  
    }  
}
```

## Illustration



**Figure 3.14:**  $4 \times 4$  blocked and two-way unrolled “flipped” matrix transpose.

B

A

## Access optimization for algorithm class $O(N^2)/O(N)$

Example:

```
double sum = 0.;  
  
for (i=0; i<N; i++) {  
    for (j=0; j<N; j++)  
        sum = sum + foo(A[i],B[j])  
}
```

- Array B has the risk of being loaded  $N$  times (when  $N$  is very large)
- Total number of memory loads:  $N + N^2$  (first part for array A, second part for array B)

## Applying loop blocking

```
double sum = 0.;

for (jj=0; jj<N; jj+=b) {
    jstart = jj; jstop = jj+b-1;

    for (i=0; i<N; i++) {
        for (j=jstart; j<=jstop; j++)
            sum = sum + foo(A[i],B[j])
    }
}
```

- Appropriate choice of  $b$  will allow array B to be loaded from memory only once.
- Array A will now be loaded  $N/b$  times (instead of only once).
- Total number of memory loads:  $N^2/b + N$

# IN3200/IN4200: Chapter 3

## Data access optimization

### (Part 3)

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

Two cases of code balance analysis (and data access optimization):

- Dense matrix-vector multiply (repetition)
- Sparse matrix-vector multiply

# Matrix-vector multiply

A square matrix **A**:  $N$  rows and  $N$  columns of numerical values

Vector **B**:  $N$  numerical values

Vector **C**:  $N$  numerical values

Mathematical definition of matrix-vector multiply:  $\mathbf{C} = \mathbf{C} + \mathbf{A} * \mathbf{B}$   
such that each value in vector **C** is calculated as

$$C_i = C_i + \sum_{0 \leq j < N} A_{i,j} * B_j \quad 0 \leq i < N$$

## Dense matrix-vector multiply (repetition)

Here, we consider the case of **A** being a “dense” matrix: all its  $N \times N$  numerical values are nonzero.

Storage on a computer:

- Dense matrix **A** as a 2D array,  $N$  rows and  $N$  columns, row-major storage (in C language)
- Vectors **B** and **C** each as a 1D array of length  $N$

## Straightforward implementation & balance analysis

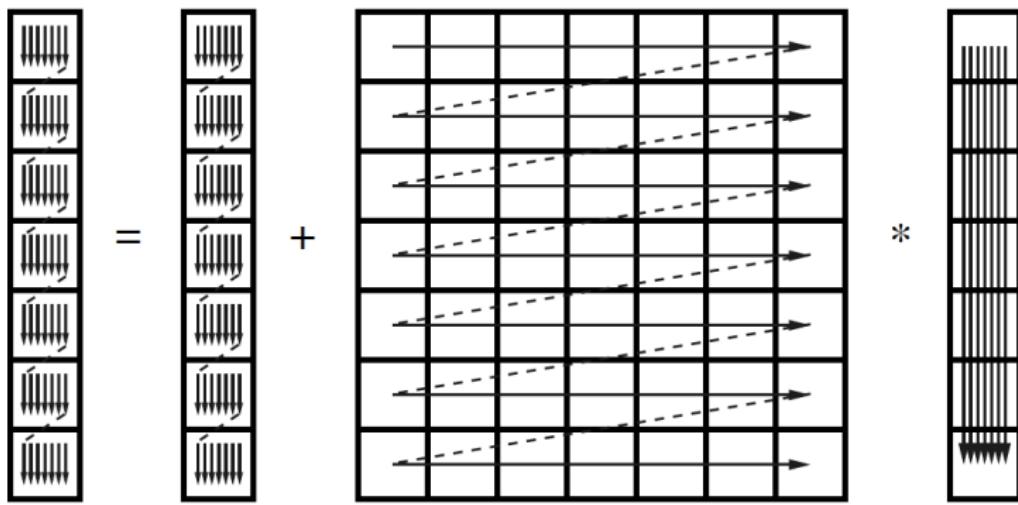
```
for (i=0; i<N; i++) {  
    double tmp = C[i];  
    for (j=0; j<N; j++)  
        tmp = tmp + A[i][j]*B[j];  
    C[i] = tmp;  
}
```

- Total number of floating-point (FP) operations:  $2N^2$
- Memory traffic:  $N^2$  loads for 2D array A,  $N$  loads &  $N$  stores for 1D array C
- How many loads are associated with 1D array B?
  - Small cache → array B is loaded  $N$  times →  $N^2$  memory loads
  - Large cache → array B is loaded only once →  $N$  memory loads

Code balance for the small-cache case:

$$\frac{N^2 + N^2 + 2N}{2N^2} = 1 + \frac{1}{N}$$

## Illustration of array B being loaded $N$ times



**Figure 3.11:** Unoptimized  $N \times N$  dense matrix vector multiply. The RHS vector is loaded  $N$  times.

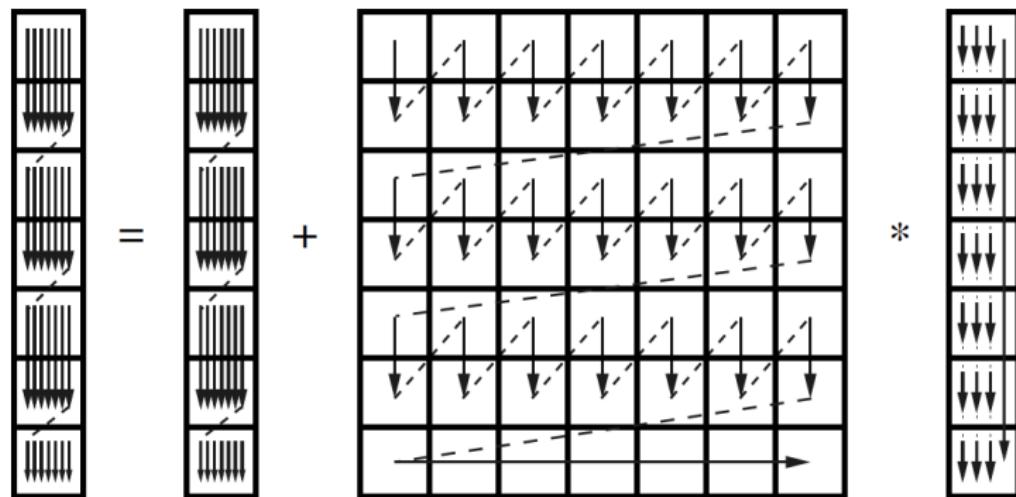
## How to reduce memory traffic for small-cache case?

$m$ -way unroll and jam:

```
for (i=0; i<N; i+=m) {  
    for (j=0; j<N; j++) {  
        C[i+0] += A[i+0][j]*B[j];  
        C[i+1] += A[i+1][j]*B[j];  
        // ...  
        C[i+m-1] += A[i+m-1][j]*B[j];  
    }  
}  
// remainder code in case (N%m)>0 ....
```

- $m$ -fold reuse of each  $B[j]$  from register
- Total number of memory loads for array B:  $N^2/m$  (for small-cache case)
- Size of  $m$  shouldn't be too large, to avoid too high *register pressure*

## Illustration of the effect of unrolling



**Figure 3.12:** Two-way unrolled dense matrix vector multiply. The data traffic caused by reloading the RHS vector is reduced by roughly a factor of two. The remainder loop is only a single (outer) iteration in this example.

## Improved code balance

For the small-cache case, unroll and jam will result in the following improved code balance:

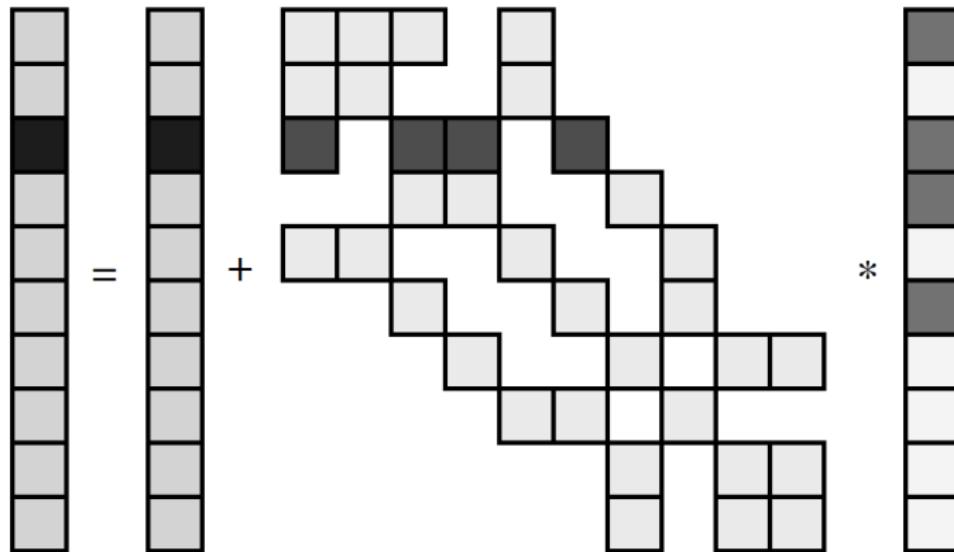
$$\frac{N^2 + \frac{N^2}{m} + 2N}{2N^2} = \frac{1}{2} + \frac{1}{2m} + \frac{1}{N}$$

## Sparse matrix

When most of the numerical values of matrix **A** are zero, it is called a *sparse* matrix.

- It will be a waste of float-point operations if we still use the straightforward implementation
- It will also be a waste of storage if we store a sparse matrix as a 2D array

# Illustration of sparse matrix-vector multiply



**Figure 3.15:** Sparse matrix-vector multiply. Dark elements visualize entries involved in updating a single LHS element. Unless the sparse matrix rows have no gaps between the first and last nonzero elements, some indirect addressing of the RHS vector is inevitable.

## Basic idea for saving storage and computation

- Store only the nonzero values of  $\mathbf{A}$ 
  - 2D-array format can no longer be used, requires an efficient storage format
- Avoid multiplications with zero
  - If  $N_{\text{nz}} (\ll N^2)$  denotes the number of nonzero values in a sparse matrix  $\mathbf{A}$ , then we only need  $2N_{\text{nz}}$  floating-point operations (instead of  $2N^2$  FP) for a sparse matrix-vector multiply

# Compressed row storage (CRS) format

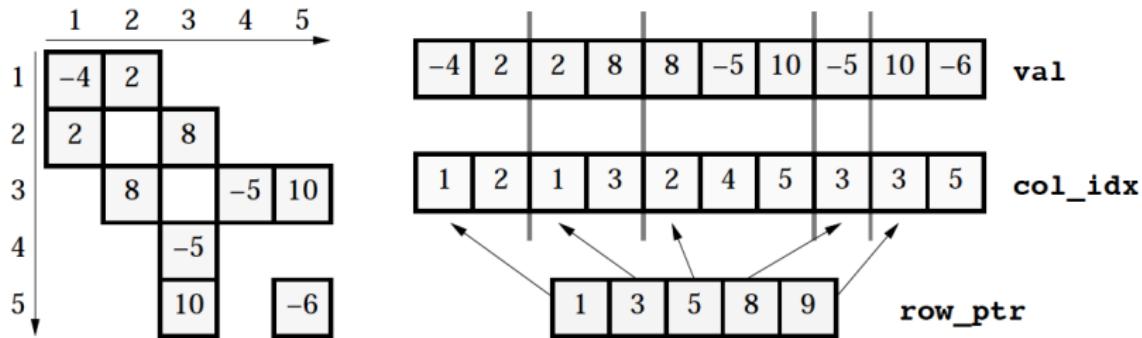


Figure 3.16: CRS sparse matrix storage format.

Three arrays:

- 1D array **val**, of length  $N_{nz}$ , stores all the nonzero values of the sparse matrix
- 1D array **col\_idx**, of length  $N_{nz}$ , records the original column positions of the all nonzero values
- 1D array **row\_ptr**, of length  $N + 1$ , contains the indices at which new rows start in array **val**

## Implementation of matrix-vector multiply using CRS format

```
for (i=0; i<N; i++) {  
    tmp = C[i];  
    for (j=row_ptr[i]; j<row_ptr[i+1]; j++)  
        tmp = tmp + val[j]*B[col_idx[j]];  
    C[i] = tmp;  
}
```

- There is a long outer loop (of length  $N$ )
- The inner loop can be very short
- Access to array C will be well optimized by compiler
- Access to array val is with stride one
- Access to array B is indirect (via col\_idx) and can be irregular

## Code balance analysis of matrix-vector multiply with CRS

Best-case scenario (entire B array is cached, needing only  $N$  loads), each entry in `row_ptr` and `col_idx` is half a word:

$$\frac{N_{\text{nz}}(1 + 0.5) + 0.5N + N + 2N}{2N_{\text{nz}}}$$

Worst-case scenario ( $B[\text{col\_idx}[j]]$  needs to be loaded from memory every single time, and only one value is used per cacheline):

$$\frac{N_{\text{nz}}(1 + 0.5) + 0.5N + N_{\text{nz}} \frac{\text{cacheline size}}{\text{word size}} + 2N}{2N_{\text{nz}}}$$

## Main ideas for improvement

- Continue using CRS format, but with suitable permutations (to reduce the actual memory traffic associated with array B)
- Use the JDS format with further optimization (see Sections 3.6.1 & 3.6.2)

# IN3200/IN4200: Chapter 4

## Parallel computers

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

# Objectives

- An introduction to the fundamental variants of parallel computers
  - The *shared-memory* type
  - The *distributed-memory* type
- A glimpse at basic design rules and performance characteristics for communication networks

## What is *parallel computing*?

**Parallel computing**—using multiple “*compute elements*” (processor cores) to solve a problem in a cooperative way.

All modern supercomputer architectures depend heavily on parallelism—a large number of interconnected compute elements.

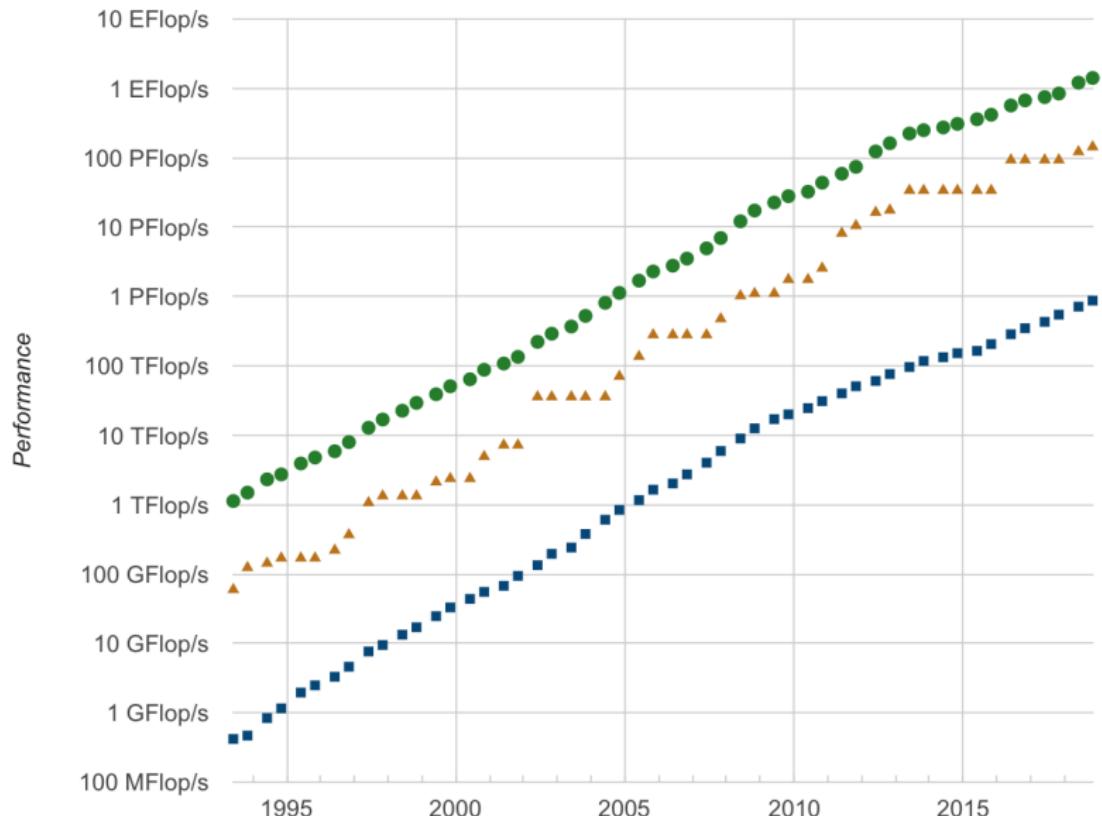
# A “peek” into supercomputers through Top500

## The Top500 list (<https://www.top500.org/>)

- A list of the world's 500 most powerful supercomputers
- Ranking by the measured performance of the LINPACK benchmark
  - Solve a dense system of linear equations (the system size freely adjustable)
  - Metric: number of floating-point operations executed per second
  - Mostly reflect the floating-point capability of a supercomputer
  - *Relevance of LINPACK is debatable*
- The list is updated twice a year

# History of Top500

## Performance Development



# Top supercomputers of today (November 2021)

| Rank | System                                                                                                                                                                        | Cores      | Rmax<br>(TFlop/s) | Rpeak<br>(TFlop/s) | Power<br>(kW) |
|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------|-------------------|--------------------|---------------|
| 1    | <b>Supercomputer Fugaku</b> - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan                               | 7,630,848  | 442,010.0         | 537,212.0          | 29,899        |
| 2    | <b>Summit</b> - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States | 2,414,592  | 148,600.0         | 200,794.9          | 10,096        |
| 3    | <b>Sierra</b> - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States     | 1,572,480  | 94,640.0          | 125,712.0          | 7,438         |
| 4    | <b>Sunway TaihuLight</b> - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China                                                | 10,649,600 | 93,014.6          | 125,435.9          | 15,371        |
| 5    | <b>Selene</b> - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States                                     | 555,520    | 63,460.0          | 79,215.0           | 2,646         |

# Some of the previous Top1 systems



**Summit:** DOE/SC/Oak Ridge National Laboratory  
**No.1 in Jun 2018**



**Sunway TaihuLight:** National Supercomputing Center in Wuxi  
**No.1 from Jun 2016 until Nov 2017**



**Tianhe-2 (MilkyWay-2)** : National University of Defense Technology  
**No.1 from Jun 2013 until Nov 2015**



**Titan:** Oak Ridge National Laboratory  
**No.1 in Nov 2012**

# Taxonomy of parallel computing paradigms

Dominating concepts:

- **SIMD** (*Single Instruction, Multiple Data*)—A single instruction stream, either on a single processor (core) or on multiple computing elements, provides parallelism by operating on multiple data streams concurrently. (Hardware examples: vector processors, SIMD-capable modern superscalar microprocessors and GPUs.)
- **MIMD** (*Multiple Instruction, Multiple Data*)—Multiple instructions streams on multiple processor (cores) operate on different data items concurrently. (Hardware examples: shared-memory and distributed-memory parallel computers.)

The focus of this chapter is on multiprocessor MIMD parallelism.

# Shared-memory computers

A *shared-memory parallel computer* has a number of CPUs (cores) that work on a shared physical address space.

Two varieties:

- *Uniform Memory Access* (UMA) systems have a “flat” memory model: latency and bandwidth are the same for all processors and all memory locations. (Typically, single multicore processor chips are “UMA machines”.)
- *Cache-coherent Nonuniform Memory Access* (ccNUMA) systems have a physically distributed memory that is *logically shared*. The aggregated memory appears as one single address space. Memory access performance depends on which CPU (core) accesses which parts of memory (“local” vs. “remote” memory access).

## Caches are not (completely) shared

A shared-memory system, no matter UMA or ccNUMA, has multiple CPU cores.

Although there is a single address space (shared memory), there are private caches, or partially shared caches, for the different CPU cores.

Therefore, copies of the same cache line **may** reside in several local caches.

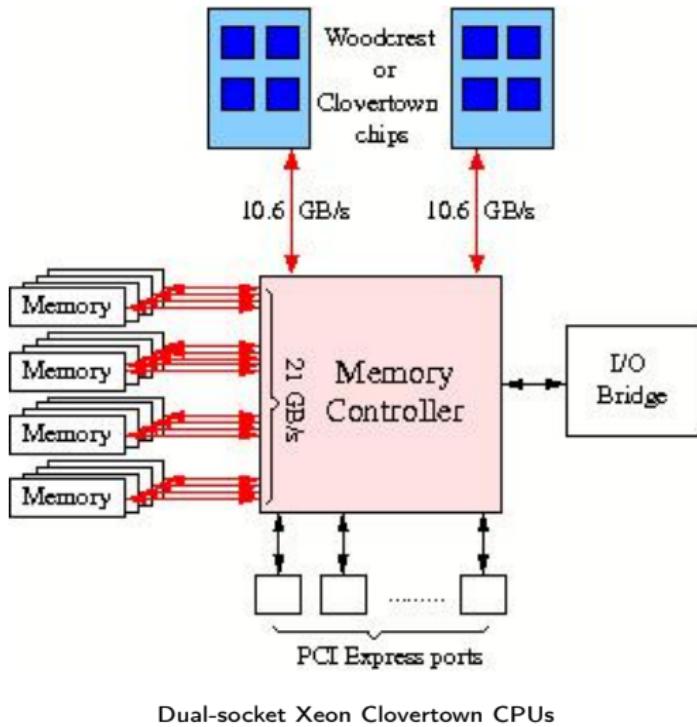
# Cache coherence

Problematic situations when the same cache line resides in several caches:

- If the cache line in one of the caches is modified, the other caches' contents are *outdated* (thus invalid).
- If different parts of the same cache line are modified by different processors in their local caches → no one has the correct cache line anymore.

**Cache coherence** protocols (supported in hardware) guarantee *consistency* between cached data and data in the shared memory at all times.

# Example of UMA

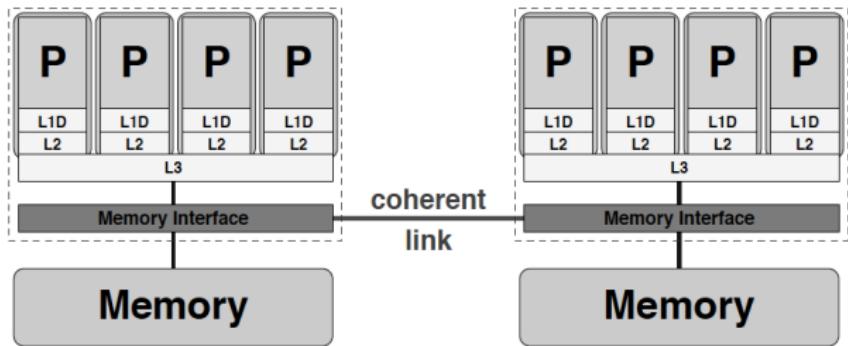


# ccNUMA for scalable memory bandwidth

- A *locality domain* (LD) is a set of processor cores together with locally connected memory. This “local” memory can be accessed by the set of processor cores in the most efficient way, without resorting to a network of any kind.
- Each LD is a UMA building block.
- Multiple LDs are linked via a coherent interconnect, which can mediate cache-coherent memory accesses. (This mechanism is transparent for the programmer.)
- The whole ccNUMA system has a shared address space (memory), runs a single OS instance.

# Example of ccNUMA

**Figure 4.5:** A ccNUMA system with two locality domains (one per socket) and eight cores.



## Penalty for non-local transfers

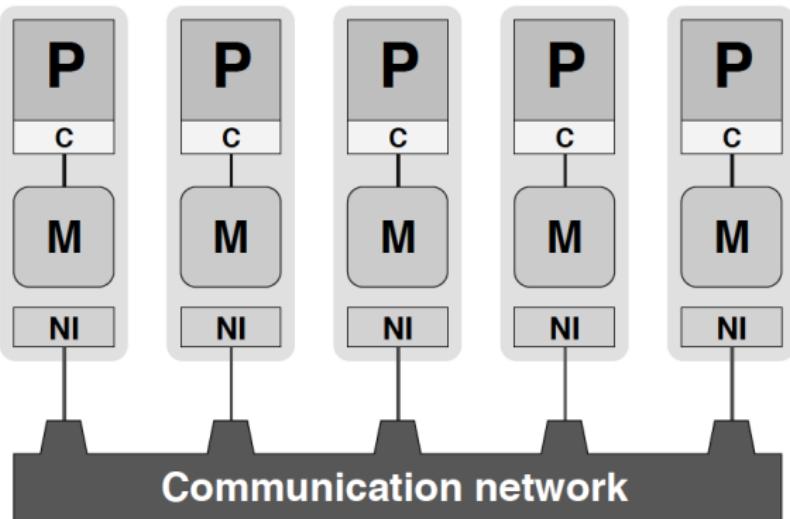
The *locality problem*: Non-local memory transfers (between LDs) are more costly than local transfers (within a LD).

The *contention problem*: If two processors from different LDs access memory in one LD, they will fight for the same memory bandwidth.

Both problems can be “solved” (alleviated) by carefully observing the data access patterns of an application and restricting data access of each processor (mostly) to its own LD, through proper programming.

# A “purely” distributed-memory computer

**Figure 4.7:** Simplified programmer’s view, or “programming model,” of a distributed-memory parallel computer: Separate processes run on processors (P), communicating via interfaces (NI) over some network. No process can access another process’ memory (M) directly, although processors may reside in shared memory.



“A programmer’s view”: Each processor is connected to its exclusive local memory (not shared by any other CPUs).

No such “purely” distributed-memory computer today.

## Typical modern distributed-memory systems

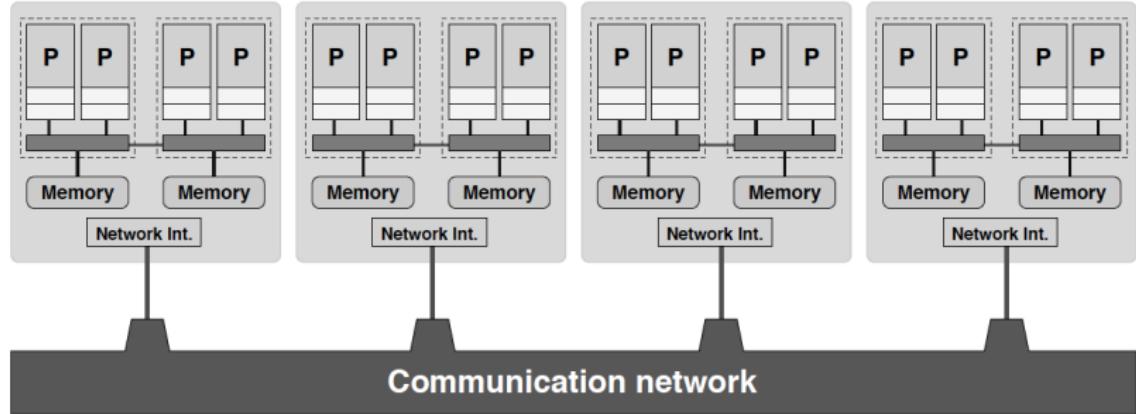
A cluster of shared-memory “*compute nodes*”, interconnected via a *communication network*.

Each node comprises at least one network interface (NI) that mediates the connection to the communication network.

A serial process runs on each CPU (core). Between the nodes, processes can communicate by means of the network.

The layout and speed of the network has a considerable impact on application performance.

# Hierarchical hybrid systems



**Figure 4.8:** Typical hybrid system with shared-memory nodes (ccNUMA type). Two-socket building blocks represent the price vs. performance “sweet spot” and are thus found in many commodity clusters.

# Networks

There are different network technologies and topologies for connecting the compute elements.



The following is a *very* brief overview of the topological and performance aspects of different types of communication networks.

# Basic performance characteristics of networks

- Point-to-point communication (from one compute element to another)
- Bisection bandwidth (a measure of the “whole” network)

## Simple model of point-to-point communication

Time spent on transferring a message of size  $N$  [bytes] from a “sender” process to a “receiver” process:

$$T = T_\ell + \frac{N}{B}$$

This is a simplified performance model:

- $T_\ell$ : latency
- $B$ : maximum network point-to-point bandwidth [bytes/sec]

$T_\ell$  and  $B$  are considered as constants, but in reality they can both depend on  $N$  (message size), as well as on the locations of the two processes.

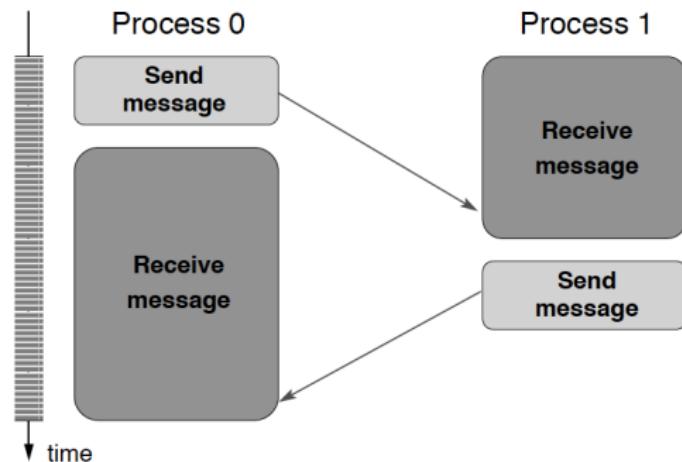
## Effective bandwidth

Due to latency  $T_\ell$ , the actual data transfer rate will be lower than  $B$ :

$$B_{\text{eff}} = \frac{N}{T_\ell + \frac{N}{B}}$$

The effective bandwidth  $B_{\text{eff}}$  approaches  $B$  when  $N$  is large enough.

# “Ping-pong” benchmark



**Figure 4.9:** Timeline for a “Ping-Pong” data exchange between two processes. PingPong reports the time it takes for a message of length  $N$  bytes to travel from process 0 to process 1 and back.

## “Ping-pong” benchmark (cont'd)

Pseudo code:

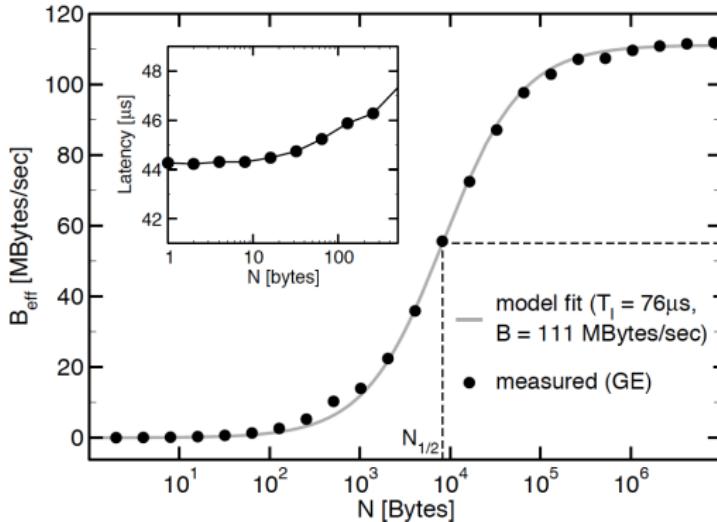
---

```
1 myID = get_process_ID()
2 if(myID.eq.0) then
3     targetID = 1
4     S = get_walltime()
5     call Send_message(buffer,N,targetID)
6     call Receive_message(buffer,N,targetID)
7     E = get_walltime()
8     MBYTES = 2*N/(E-S)/1.d6      ! MBytes/sec rate
9     TIME      = (E-S)/2*1.d6      ! transfer time in microsecs
10                            ! for single message
11 else
12     targetID = 0
13     call Receive_message(buffer,N,targetID)
14     call Send_message(buffer,N,targetID)
15 endif
```

---

The same code is simultaneously run by two processes.

## Example of “ping-pong” measurements



**Figure 4.10:** Fit of the model for effective bandwidth (4.2) to data measured on a GigE network. The fit cannot accurately reproduce the measured value of  $T_\ell$  (see text).  $N_{1/2}$  is the message length at which half of the saturation bandwidth is reached (dashed line).

$B_{\text{eff}}$  is measured for different values of  $N$ ; The values of  $T_\ell$  and  $B$  can be deduced by “fitting” the measurements with the theoretical model.

## Bisection bandwidth

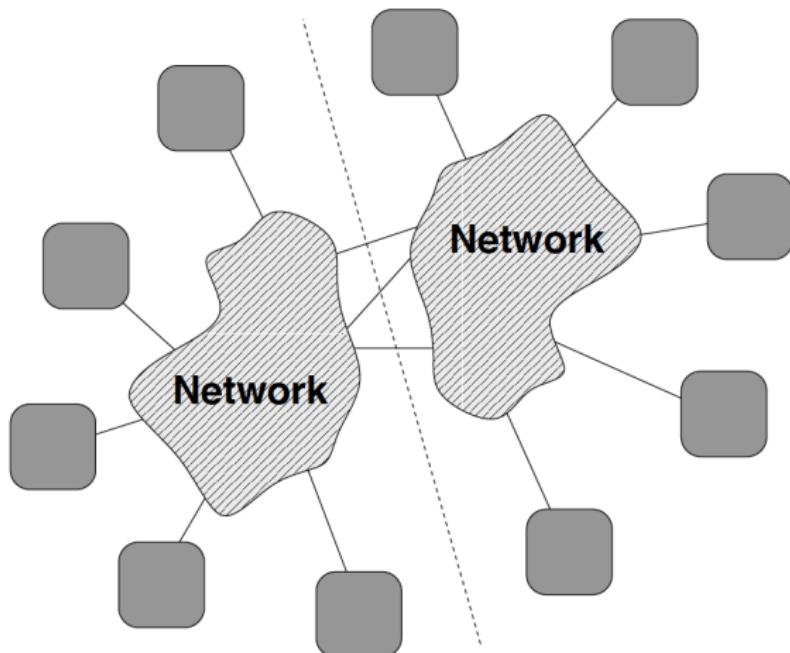
How to quantify the “total” communication capacity of a network?

When all the compute elements are sending or receiving data at the same time:

- “competition” (even collision) may lead to that the *aggregated bandwidth*, the sum of all effective bandwidths for all point-to-point connections, is lower than the theoretical limit.

**Bisection bandwidth** of a network,  $B_b$ , is the sum of the bandwidths of the minimal number of connections cut when splitting the system into two *equal-sized* parts.

# Illustration of bisection bandwidth

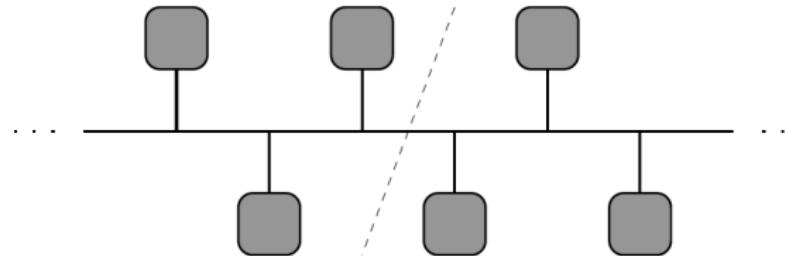


**Figure 4.12:** The bisection bandwidth  $B_b$  is the sum of the bandwidths of the minimal number of connections cut (three in this example) when dividing the system into two equal parts.

# Different types of a communication network

- Buses
- Switched and fat-tree networks
- Mesh networks

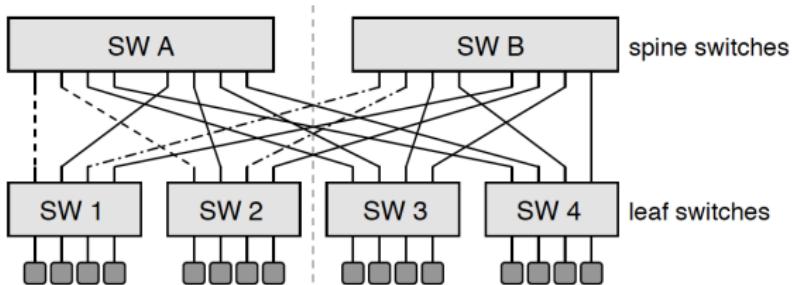
## Buses



**Figure 4.13:** A bus network (shared medium). Only one device can use the bus at any time, and bisection bandwidth is independent of the number of nodes.

- Can be used by exactly **one** communicating device at a time.
- Easy to implement, featuring lowest latency at small utilization.
- The most important drawback is *blocking*.
- Buses are susceptible for failures.

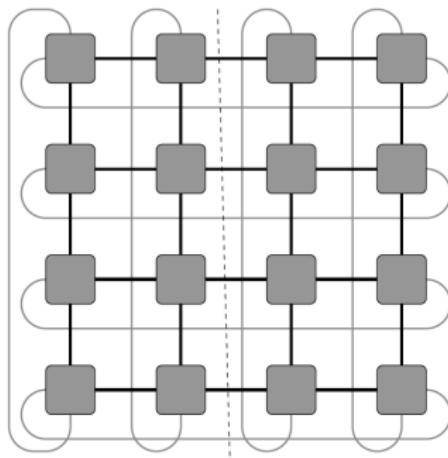
# Switched and fat-tree networks



**Figure 4.15:** A fully nonblocking full-bandwidth fat-tree network with two switch layers. The switches connected to the actual compute elements are called *leaf switches*, whereas the upper layers form the *spines* of the hierarchy.

- All communicating devices are organized into groups.
- The devices in one group are connected to a *switch*.
- Switches are connected with each other (as a *fat-tree* hierarchy)
- The “distance” between two communicating devices—number of “hops”.

# Mesh networks



**Figure 4.18:** A two-dimensional (square) torus network. Bisection bandwidth scales like  $\sqrt{N}$  in this case.

- In form of a multidimensional (hyper)cubes.
- Each compute element is located at a Cartesian grid intersection.
- Connections can be wrapped around the boundaries, to form a torus topology.

## IN3200/IN4200: Chapter 5 Basics of parallelization

Sections 5.3.5, 5.3.6, 5.3.7, 5.3.8 are not required

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

# Objectives

*High-performance computing* = efficient serial computing + effective parallel processing → needs parallel programming

But before actually engaging in parallel programming, it is vital to know some fundamental things in parallelization:

- The most common strategies for parallelization
- Simple theoretical insights into the factors that can hamper parallel performance (to be covered in part 2)

## Why parallelize?

- We want to solve the problems faster, but the speed of a single CPU core has “saturated”.
- We want to solve larger problems, but the main memory available on a single system is not large enough.

So, we need to identify parallelism in a given computational problem, so that parallel programming can produce a parallel implementation that can efficiently use many processor cores, on a shared- or distributed-memory system.

## Identifying parallelism

The first step is to identify the parallelism inherent in the algorithm at hand—how can *multiple* compute elements collaborate to solve the computational problem?

Different variants of parallelism induce different methods of parallelization.

## Data parallelism

Most problems in scientific computing involve processing of large quantities of data stored on a computer.

If this can be performed in parallel by multiple processors concurrently working on different parts of the data—**data parallelism**.

This is the dominant parallelization concept in scientific computing, also goes under the name *SPMD* (single program multiple data).

## Example: medium-grained loop parallelism

Processing of array data by loops or loop nests is a central component in most scientific codes.

When computations performed on the individual array elements are independent of each other—typical candidates for parallel execution by multiple “workers” with help of shared memory. (Also possible on distributed-memory systems after appropriate data partitioning.)

|    |                                                |                                              |
|----|------------------------------------------------|----------------------------------------------|
| P1 | <pre>do i=1,500     a(i)=c*b(i) enddo</pre>    | <pre>do i=1,1000     a(i)=c*b(i) enddo</pre> |
| P2 | <pre>do i=501,1000     a(i)=c*b(i) enddo</pre> |                                              |

**Figure 5.1:** An example for medium-grained parallelism: The iterations of a loop are distributed to two processors P1 and P2 (in shared memory) for concurrent execution.

## More examples of data parallelism (matrix-matrix multiplication)

Multiplying two matrices  $A$  and  $B$  to yield matrix  $C$ . The output matrix  $C$  can for example be partitioned into four blocks (where each block is a sub-matrix):

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \times \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

$$\text{Process 1: } C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$\text{Process 2: } C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$\text{Process 3: } C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$\text{Process 4: } C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

More examples of data parallelism (counting occurrences)

Counting the occurrences of given itemsets in a database of transactions. For example, the output (itemset frequencies) can be partitioned across processes.

(a) Transactions (input), itemsets (input), and frequencies (output)

| Database Transactions | Itemsets | Itemset Frequency |
|-----------------------|----------|-------------------|
| A, B, C, E, G, H      | A, B, C  | 1                 |
| B, D, E, F, K, L      | D, E     | 3                 |
| A, B, F, H, L         | C, F, G  | 0                 |
| D, E, F, H            | A, E     | 2                 |
| F, G, H, K,           | C, D     | 1                 |
| A, E, F, K, L         | D, K     | 2                 |
| B, C, D, G, H, L      | B, C, F  | 0                 |
| G, H, L               | C, D, K  | 0                 |
| D, E, F, K, L         |          |                   |
| F, G, H, L            |          |                   |

**(b) Partitioning the frequencies (and itemsets) among the tasks**

| Database Transactions |  | Itemsets | Itemset Frequency | Itemsets         | Itemset Frequency |
|-----------------------|--|----------|-------------------|------------------|-------------------|
| A, B, C, E, G, H      |  | A, B, C  | 1                 | A, B, C, E, G, H | C, D              |
| B, D, E, F, K, L      |  | D, E     | 3                 | B, D, E, F, K, L | D, K              |
| A, B, F, H, L         |  | C, F, G  | 0                 | A, B, F, H, L    | B, C, F           |
| D, E, F, H            |  | A, E     | 2                 | D, E, F, H       | C, D, K           |
| F, G, H, K,           |  |          |                   | F, G, H, K,      | 0                 |
| A, E, F, K, L         |  |          |                   | A, E, F, K, L    | 0                 |
| B, C, D, G, H, L      |  |          |                   | B, C, D, G, H, L | 0                 |
| G, H, L               |  |          |                   | G, H, L          | 0                 |
| D, E, F, K, L         |  |          |                   | D, E, F, K, L    | 0                 |
| F, G, H, L            |  |          |                   | F, G, H, L       | 0                 |

## More examples of data parallelism (counting occurrences)

We have the following observations:

- If the database is shared (on a shared-memory system) or replicated across the processes (on a distributed-memory system), each process can be independently accomplished (no need for communication).
- If the database is partitioned across processes as well (for best use of distributed memory), each process can first compute partial counts. These counts then have to be aggregated (by communication).

More examples of data parallelism (counting occurrences)

**Input Data Partitioning:** For the database counting example, we can choose to partition the input (i.e., the transaction set).

## Partitioning the transactions among the tasks

| Database Transactions | Itemsets | Itemset Frequency |
|-----------------------|----------|-------------------|
| A, B, C, E, G, H      | A, B, C  | 1                 |
| B, D, E, F, K, L      | D, E     | 2                 |
| A, B, F, H, L         | C, F, G  | 0                 |
| D, E, F, H            | A, E     | 1                 |
| F, G, H, K,           | C, D     | 0                 |
|                       | D, K     | 1                 |
|                       | B, C, F  | 0                 |
|                       | C, D, K  | 0                 |

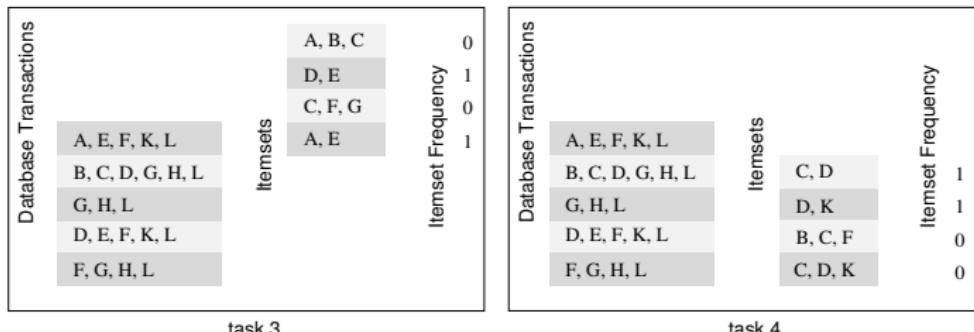
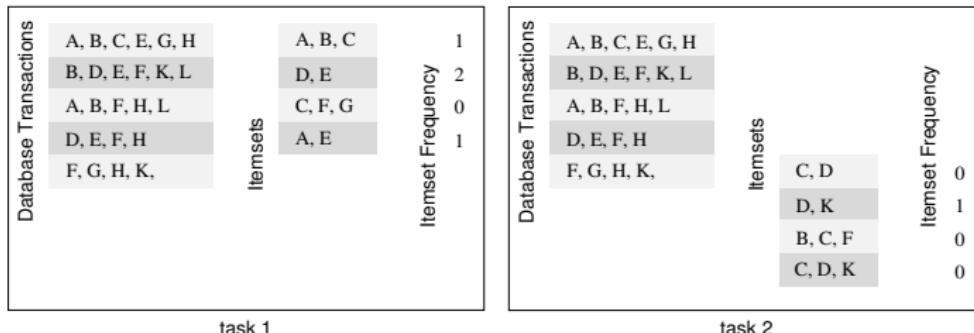
| Database Transactions | Itemsets | Itemset Frequency |
|-----------------------|----------|-------------------|
|                       | A, B, C  | 0                 |
|                       | D, E     | 1                 |
|                       | C, F, G  | 0                 |
| A, E, F, K, L         | A, E     | 1                 |
| B, C, D, G, H, L      | C, D     | 1                 |
| G, H, L               | D, K     | 1                 |
| D, E, F, K, L         | B, C, F  | 0                 |
| F, G, H, L            | C, D, K  | 0                 |

The two partial counting results of itemset frequency need to be aggregated.

# More examples of data parallelism (counting occurrences)

Input and Output Data Partitioning (aggregation needed afterwards):

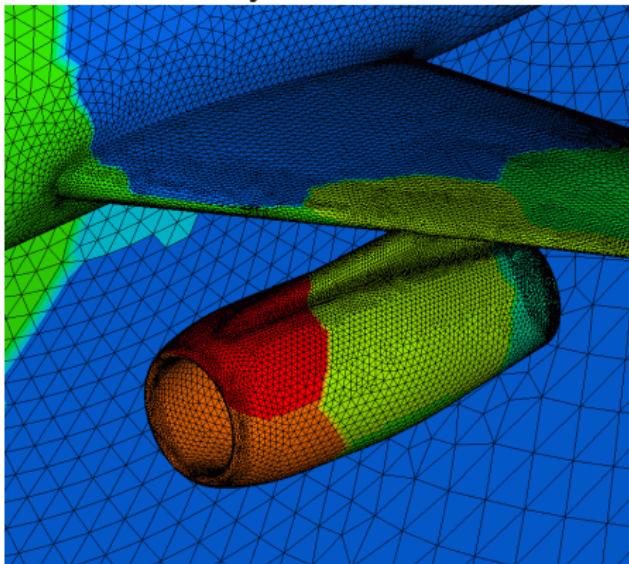
Partitioning both transactions and frequencies among the tasks



## Example: Coarse-grained parallelism by domain decomposition

In case of a computational domain represented as a grid, a straightforward way to distribute the computation among “workers” is to assign a part of the grid to each worker—*domain decomposition*.

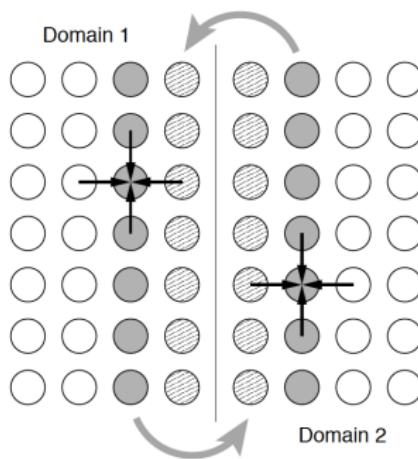
Works on both shared-memory and distributed-memory computers.



# Domain decomposition & distributed memory

If domain decomposition is to be implemented for distributed memory, grid & data are explicitly partitioned. Updating one subdomain's boundary points requires data from one or more adjacent subdomains.

Explicit communication is thus needed, together with *halo* or *ghost* layers in the data structure.



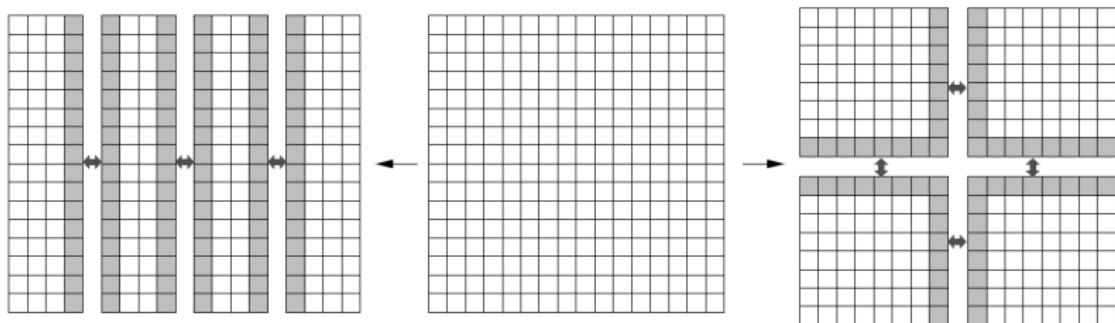
**Figure 5.2:** Using halo ("ghost") layers for communication across domain boundaries in a distributed-memory parallel Jacobi solver. After the local updates in each domain, the boundary layers (shaded) are copied to the halo of the neighboring domain (hatched).

## More about domain decomposition

Unless the computational grid is regular, domain decomposition can be a non-trivial task:

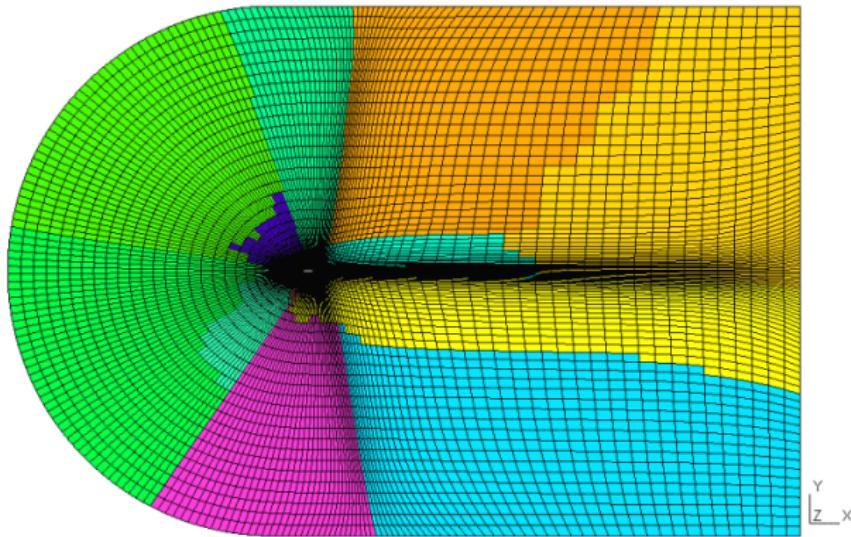
- Load balance—all “workers” get roughly the same amount of computation;
- Communication overhead—must be kept low;

# Impact of decomposition on communication overhead



**Figure 5.3:** Domain decomposition of a two-dimensional Jacobi solver, which requires next-neighbor interactions. Cutting into stripes (left) is simple but incurs more communication than optimal decomposition (right). Shaded cells participate in network communication.

# Example of general decomposition



<https://nutscfd.wordpress.com/2017/03/06/mesh-partitioning-using-parmetis/>

## Functional parallelism

When the solution of a “big” problem can be split into disparate subtasks, which work together by data exchange and synchronization. The subtasks execute completely different code on different data items, so **functional parallelism** becomes an option.

Functional parallelism is also called *MPMD* (multiple program multiple data). Each subtask may contain data parallelism and be further parallelized by SPMD.

- If different parts of the “big” problem have different performance properties and hardware requirements, bottlenecks and load imbalance can easily arise.
- Overlapping subtasks—in the “design” of functional parallelism—can give performance benefits.

## Example: master-worker scheme

Reserving one “master” compute element for administrating the other compute elements (as “workers”).

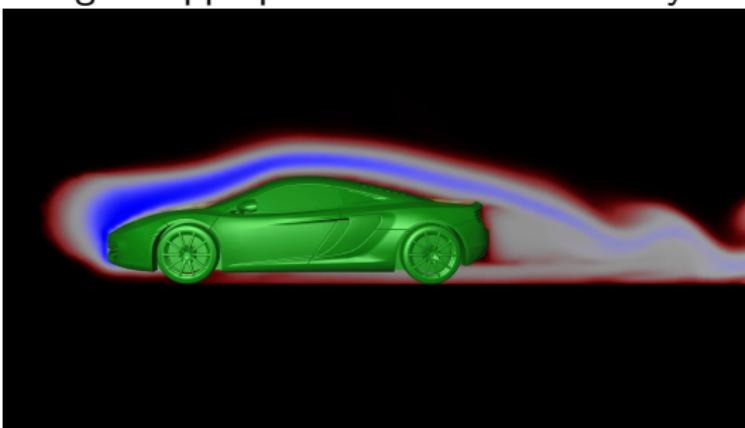
- From a pool of subtasks, the master dynamically assigns the workers with computational work.
- The master is also responsible for collecting results from the workers.

A drawback of the master-worker scheme is the potential communication and performance bottleneck that may appear with a single master that administers a large number of workers.

## Example: functional decomposition

Multiphysics simulations are possible candidates for parallelization by functional decomposition (combined with data parallelism).

For instance, the air flow around a racing car can be simulated using a parallel CFD (computational fluid dynamics) code. On the other hand, a parallel finite element simulation can compute the reaction of the structure of the car to the air flow. The two parts are coupled using an appropriate communication layer.



## IN3200/IN4200: More about parallelization

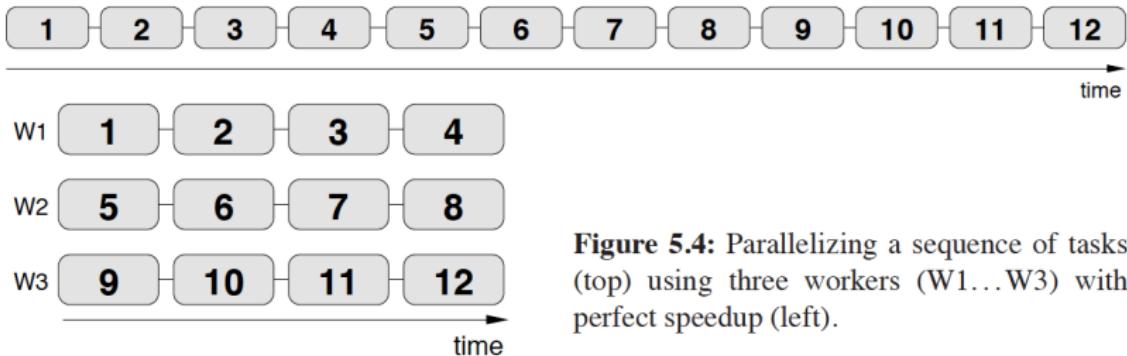
Chapter 5 in textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

Plus examples from A. Grama, A. Gupta, G. Karypis, and V. Kumar: "Introduction to Parallel Computing", Addison Wesley, 2003

- Simple theoretical insights into the factors that can hamper parallel performance
- More examples of identifying parallelism
- Simple design of parallel algorithms

# Parallel scalability

The *ideal* goal: If a problem takes time  $T$  to be solved by one worker, we expect the solution time by using  $N$  identical workers to be  $T/N$ —a perfect **speedup** of  $N$ .



**Figure 5.4:** Parallelizing a sequence of tasks (top) using three workers (W1...W3) with perfect speedup (left).

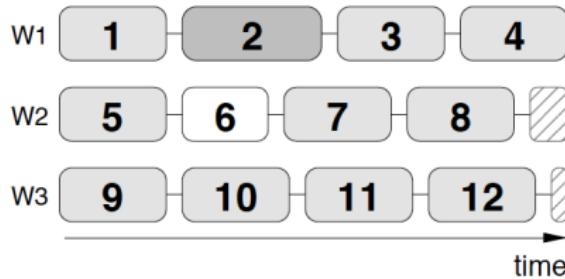
However, perfect speedup is often not achievable in reality, why?

# Factors that limit parallel execution

Reasons for non-perfect speedup:

- Not all workers might execute their tasks equally fast, because the problem was not (or could not be) partitioned into equal pieces—**load imbalance**;
- There might be shared resources which can only be used by one worker at a time—**serialization**;
- New tasks may arise due to parallelization, such as communication between workers—**overhead**.

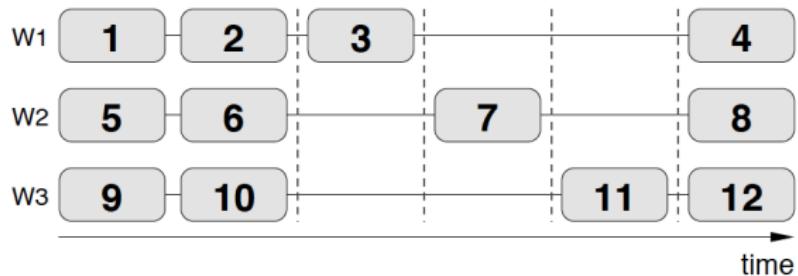
## Example of load imbalance



**Figure 5.5:** Some tasks executed by different workers at different speeds lead to *load imbalance*. Hatched regions indicate unused resources.

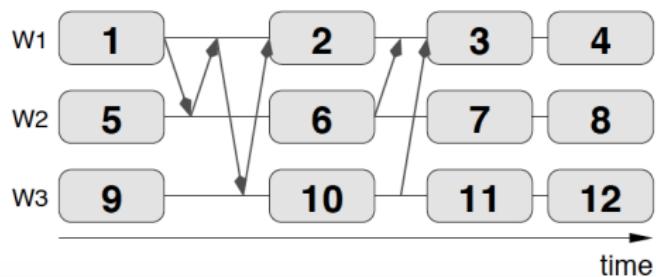
# Example of serialization

**Figure 5.6:** Parallelization with a bottleneck. Tasks 3, 7 and 11 cannot overlap with anything else across the dashed “barriers.”



## Example of communication overhead

**Figure 5.7:** Communication processes (arrows represent messages) limit scalability if they cannot be overlapped with each other or with calculation.



# Scalability metrics

How well can a computational problem be parallelized?

Scalability metrics help to answer the following questions:

- How much faster can a given problem be solved with  $N$  workers instead of one?
- How much more work can be done with  $N$  workers instead of one?
- What impact do the communication requirements have on performance and scalability?
- What fraction of the resources is actually used productively?

## Strong and weak scaling

Starting point: The overall problem size ("amount of work") is *normalized* as

$$s + p = 1$$

where  $s$  is the serial, non-parallelizable fraction,  $p$  is the perfectly parallelizable fraction.

We can now define *strong scaling* and *weak scaling*, and study the relationship between single-worker serial runtime and multi-worker parallel runtime.

## Strong scaling

Single-worker (serial) normalized runtime for a fixed-size problem:

$$T_f^s = s + p$$

Solving the same problem using  $N$  workers will require a runtime of

$$T_f^p = s + \frac{p}{N}$$

This is called **strong scaling**, because the total amount of work stays constant no matter how many workers are used.

Here, the goal of parallelization is minimization of time-to-solution for a given problem.

## Weak scaling

For **weak scaling**, the goal is to solve an increasingly larger problem with more workers  $N$ .

More specifically, the total amount of work is scaled with some power of  $N$

$$s + pN^\alpha \quad (\alpha \text{ is a positive parameter})$$

which means that single-worker runtime for the variable-sized problem **would have been**  $T_v^s = s + pN^\alpha$ .

Using  $N$  workers, the parallel runtime is

$$T_v^p = s + pN^{\alpha-1}$$

Here, we have also assumed that  $s$  doesn't grow with  $N$ .

The most typical choice is  $\alpha = 1$ , then  $T_v^s = s + pN$  and  $T_v^p = s + p$ .

# Simple scalability laws

How to calculate speedup?

$$\text{application speedup} = \frac{\text{serial runtime}}{\text{parallel runtime}}$$

or equivalently

$$\text{application speedup} = \frac{\text{parallel performance}}{\text{serial performance}}$$

where “performance” is defined as “work over time”.

## Amdahl's law

For a fixed problem size  $s + p = 1$ , the application speedup (“scalability”) is

$$S_f = \frac{T_f^s}{T_f^p} = \frac{s + p}{s + \frac{p}{N}} = \frac{1}{s + \frac{1-s}{N}}$$

This is “Amdahl’s law”—maximum speedup is  $1/s$  when  $N \rightarrow \infty$ .

## Gustafson's law

The problem size is scaled with the number of workers  $N$ .

Recall that for  $\alpha = 1$  we have  $T_v^s = s + pN$  and  $T_v^p = s + p$ .  
Therefore the application speedup is

$$S_v = \frac{T_v^s}{T_v^p} = \frac{s + pN}{s + p} = \frac{s + (1 - s)N}{1} = s + (1 - s)N$$

This is “Gustafson’s law”—speedup can be arbitrarily large when  $N \rightarrow \infty$ .

# Parallel efficiency

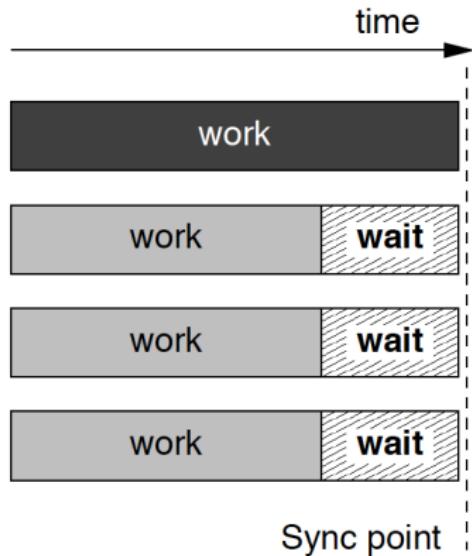
How effectively is the resource used by parallel program?

Parallel efficiency is defined as

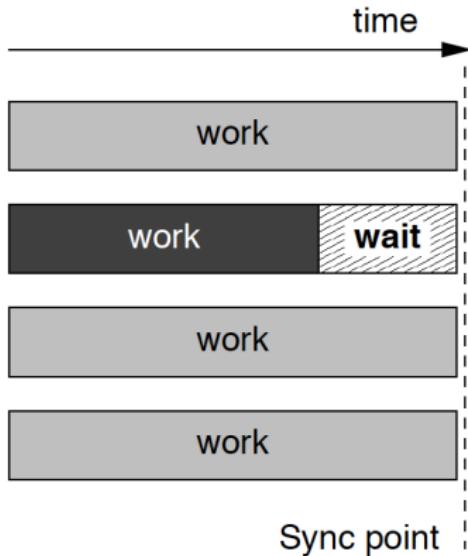
$$\varepsilon = \frac{\text{speedup}}{N}$$

This will be a value between 0 and 100%.

# Negative impact of load imbalance



**Figure 5.13:** Load imbalance with few (one in this case) “laggers”: A lot of resources are underutilized (hatched areas).



**Figure 5.14:** Load imbalance with few (one in this case) “speeders”: Underutilization may be acceptable.

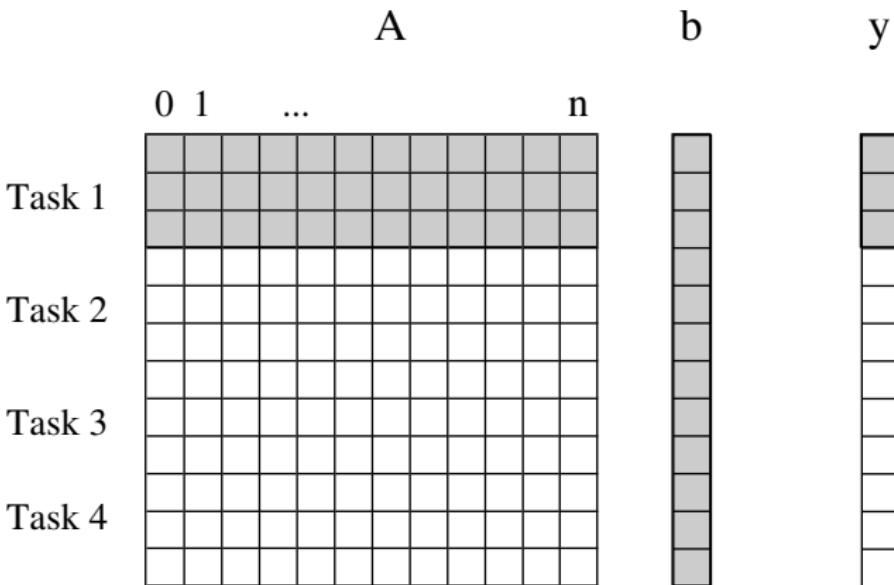
## Example: dense matrix-vector multiply

Dense matrix-vector multiply

$$\mathbf{y} = \mathbf{Ab}$$

```
for (i=0; i<N; i++) {  
    double tmp = 0.;  
    for (j=0; j<N; j++)  
        tmp += A[i][j]*b[j];  
    y[i] = tmp;  
}
```

# Parallelization



Decomposition of the outer loop (index  $i$ ) into  $P$  chunks, each as the computational task for a processor core. All the tasks are *completely independent*.

## Work decomposition

Let  $N$  denote the number of entries in vector  $\mathbf{y}$  (same as the number of rows in matrix  $\mathbf{A}$ ). If  $N$  is divisible by the number of processor cores  $P$ , then work decomposition will be perfectly even.

For example: processor core number  $k$  ( $0 \leq k < P$ ) can be responsible for computing the following entries of vector  $\mathbf{y}$ :

```
y[k*chunk_size],  
y[k*chunk_size+1],  
...  
y[(k+1)*chunk_size-1]
```

where  $\text{chunk\_size} = N/P$

## Danger for severe load imbalance

What if  $N$  is not divisible by  $P$ ?

Integer division `chunk_size=N/P` will result in

$$\text{chunk\_size} = \lfloor \frac{N}{P} \rfloor = \frac{N - \text{modulo}(N, P)}{P}$$

That can easily lead to that  $P - 1$  processor cores compute each `chunk_size` entries of vector  $\mathbf{y}$ , whereas one processor core computes  $\text{modulo}(N, P)$  entries **extra**.

An extreme case of load imbalance arises when  $N = 2P - 1$ . It will mean that the amount of work for the “heavy-load” processor core is  $P$  times of the other processor cores!

## Remedy for load balance

The following work decomposition will guarantee that the maximum difference between “heavy-load” and “light-load” tasks is at most 1.

Processor core number  $k$  computes

```
y[start_k],  
y[start_k+1],  
...  
y[stop_k-1]
```

where  $\text{start}_k = k * N/P$  and  $\text{stop}_k = (k+1) * N/P$  (integer divisions are used to compute both values).

## Example: summing an array of values

```
sum=0. ;  
for (i=0; i<N; i++)  
    sum += y[i];
```

Basic strategy of parallelization:

- Divide the entries of array  $y$  into as equal-sized chunks as possible

$$\text{start\_k} = k * N/P \text{ and } \text{stop\_k} = (k+1) * N/P$$

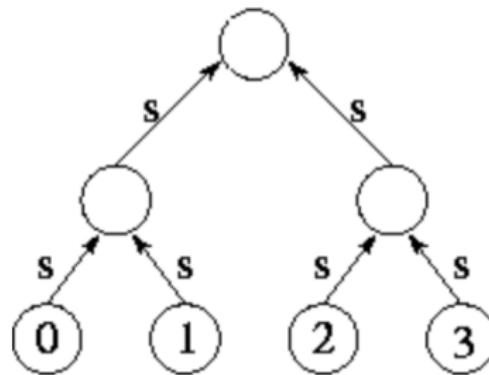
- Each processor core *independently* computes a partial sum as  
 $y[\text{start\_k}] + y[\text{start\_k}+1] + \dots + y[\text{stop\_k}-1]$
- When all the  $P$  partial sums are computed, they are added up to produce the correct value of  $\text{sum}$

## How to sum up $P$ values from $P$ processor cores?

Approach 1: Pick a “master” processor core, and let the master add the  $P$  values together.

Downside of this approach: The master core can become a bottleneck if  $P$  is large.

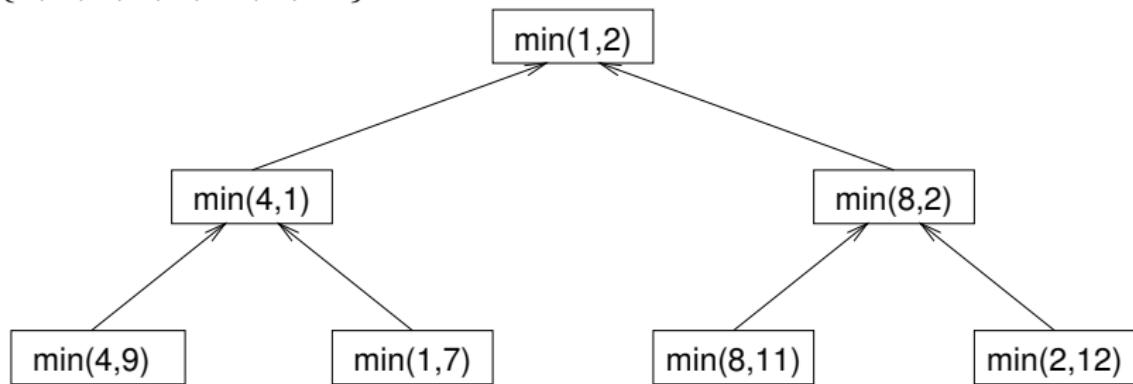
Approach 2: *reverse recursive decomposition*



The “bottom” tasks represent individual partial sums on the processor cores, the other tasks are pair-wise additions until sum is computed at the “top”.

## Another example of reverse recursive decomposition

Suppose we want to find the minimum value in the set  $\{4, 9, 1, 7, 8, 11, 2, 12\}$ .



## Example: Database Query Processing

Consider the execution of the query:

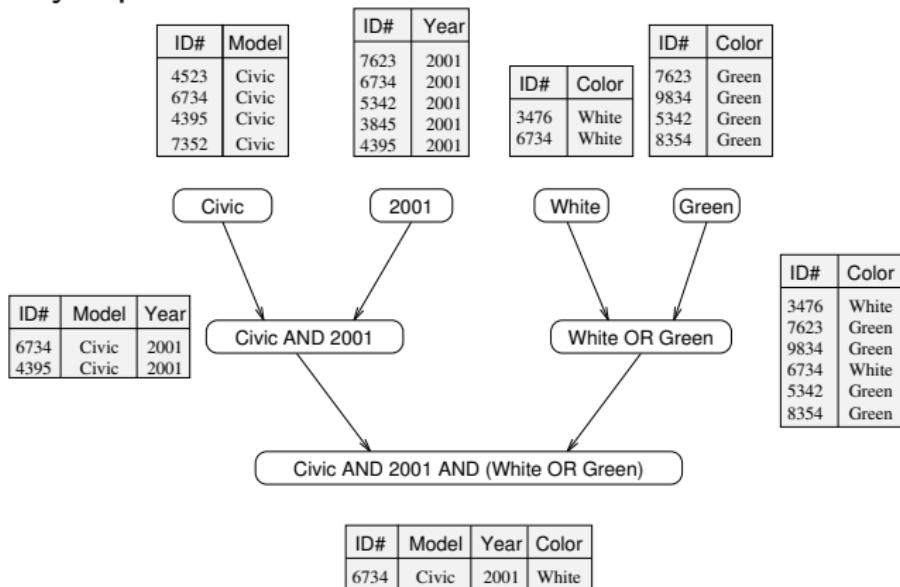
MODEL = “CIVIC” AND YEAR = 2001 AND  
(COLOR = “GREEN” OR COLOR = “WHITE”)

on the following database:

| ID#  | Model   | Year | Color | Dealer | Price    |
|------|---------|------|-------|--------|----------|
| 4523 | Civic   | 2002 | Blue  | MN     | \$18,000 |
| 3476 | Corolla | 1999 | White | IL     | \$15,000 |
| 7623 | Camry   | 2001 | Green | NY     | \$21,000 |
| 9834 | Prius   | 2001 | Green | CA     | \$18,000 |
| 6734 | Civic   | 2001 | White | OR     | \$17,000 |
| 5342 | Altima  | 2001 | Green | FL     | \$19,000 |
| 3845 | Maxima  | 2001 | Blue  | NY     | \$22,000 |
| 8354 | Accord  | 2000 | Green | VT     | \$18,000 |
| 4395 | Civic   | 2001 | Red   | CA     | \$17,000 |
| 7352 | Civic   | 2002 | Red   | WA     | \$18,000 |

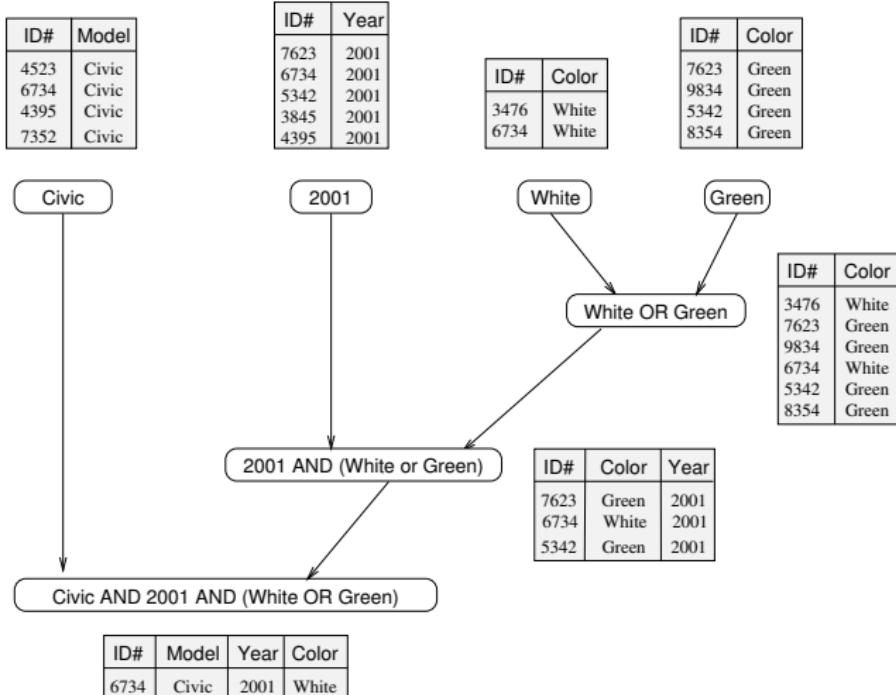
# Decomposition into tasks

The execution of the query can be divided into tasks. Each task can be thought of as generating an intermediate table of entries that satisfy a particular clause.



Decomposing the given query into several tasks. Edges denote that the output of one task is needed to accomplish the next.

# Another decomposition

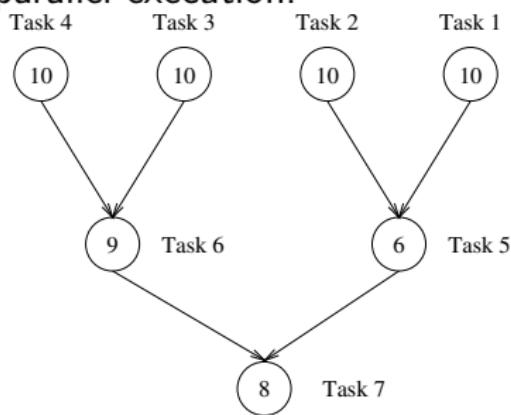


Different task decompositions may lead to significant differences with respect to their eventual parallel performance.

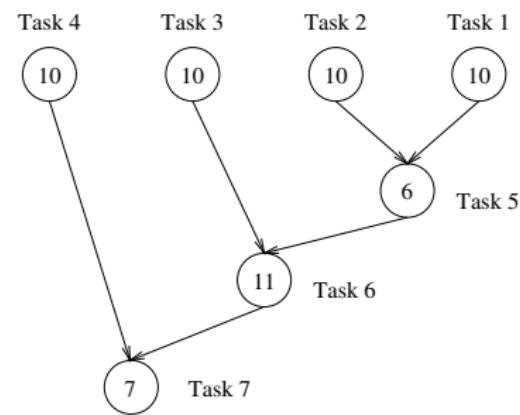
# Task dependency graph & critical path

Task dependency graph: A directed path in the task dependency graph represents a sequence of tasks that must be processed one after the other.

The length of the longest path in a task dependency graph is called the critical path length. It also gives the minimum time needed by parallel execution.



(a)



(b)

# IN3200/IN4200: Chapter 6

## Shared-memory parallel programming with OpenMP (Part 1)

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

# Objectives

- A very brief introduction to **OpenMP**
  - An application programming interface (API) based mostly on a set of compiler directives
  - Today's most widely used approach for shared-memory parallel programming
- Basic OpenMP programming through examples

(More advanced OpenMP programming will be taught in later chapters.)

# First things first

The applicable hardware context: **shared memory**

- All processors can directly access all data in a shared memory, no need for explicit communication between the processors
- OpenMP: A parallel programming standard for shared-memory parallel computers
  - A set of compiler directives (with additional clauses)
  - A small number of library functions
  - A few environment variables

## Advantages of compiler directives

- An OpenMP compiler directive is a “parallelization hint” intended for a compatible compiler to automatically create parallel code, will be ignored by non-OpenMP compilers otherwise.

```
#pragma omp ...
```

- It is **possible** to maintain a same code for both the serial implementation and the parallel OpenMP implementation.

```
#ifdef _OPENMP
```

```
...
```

```
#endif
```

- Allows incremental parallelization—simplifies coding effort—easier to debug

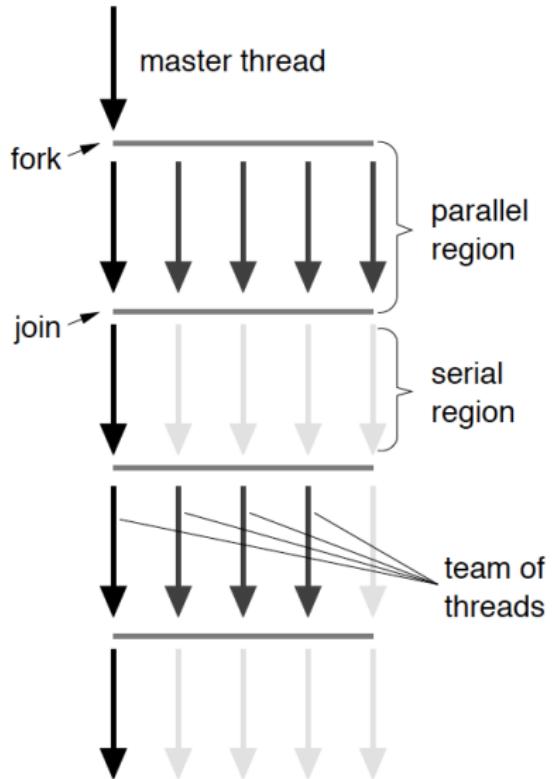
# Threads in OpenMP

- The central execution entities in an OpenMP program are **threads**—lightweight processes.
- The OpenMP threads share a common address space and mutually access data.
- Spawning a thread is much less costly than forking a new process, because threads share everything except the instruction pointer, stack pointer and register state.
  - If wanted, each thread can have a few “private variables” (by means of the local stack pointer).

## “Fork-join” model

- In any OpenMP program, a single thread, called **master thread**, runs immediately after startup.
- The master thread can spawn (also called **fork**) a number of additional threads when entering a so-called **parallel region**.
- Inside a parallel region, the master thread and the spawned threads execute instruction streams **concurrently**.
  - Each thread has a unique ID.
  - Different threads work on different parts of the shared data or carry out different tasks.
  - OpenMP has compiler directives for dividing the work among the threads.
- At the end of a parallel region, the threads are **joined**: all the threads are terminated but the master thread.
- There can be multiple parallel regions in an OpenMP program.

# OpenMP's parallel execution model



**Figure 6.1:** Model for OpenMP thread operations: The master thread “forks” team of threads, which work on shared memory in a parallel region. After the parallel region, the threads are “joined,” i.e., terminated or put to sleep, until the next parallel region starts. The number of running threads may vary among parallel regions.

# Hello world in OpenMP

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

int main (int nargs, char **args) {

#pragma omp parallel
{
    printf("Hello world!\n");
}

    return 0;
}
```

- Example of compilation: gcc -fopenmp hello\_world.c
- Parallel execution: ./a.out
- What do you get?

## Control the number of threads at runtime

Example on a Linux system:

```
gcc -fopenmp hello_world.c
export OMP_NUM_THREADS=6
./a.out
export OMP_NUM_THREADS=8
./a.out
```

OMP\_NUM\_THREADS is an environment variable (understood by OpenMP)

It is also possible to hard-code the number of threads inside an OpenMP program:

```
#pragma omp parallel num_threads(6)
{
    // .....
}
```

But this approach is normally **not** recommended!

## Hello world in OpenMP (a bit more interesting example)

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

int main (int nargs, char **args) {
    printf("I'm the master thread, I'm alone.\n");

#pragma omp parallel
{
    int num_threads, thread_id;
    num_threads = omp_get_num_threads();
    thread_id = omp_get_thread_num();
    printf("Hello world! I'm thread No.%d out of %d threads.\n",
           thread_id, num_threads);
}

    return 0;
}
```

## “Manual” loop parallelization

Now, let's try to “manually” parallelize a for-loop, that is, divide the iterations evenly among the threads.

Example:

```
for (i=0; i<N; i++)  
    a[i] = b[i] + c[i];
```

**Important observation:** Assuming that the arrays a, b and c do not overlap in memory (that is, no aliasing), then the iterations of this for-loop are independent of each other, thus safe to be executed by multiple threads concurrently.

## “Manual” loop parallelization (2)

How to divide the iterations evenly among the threads?

Given `num_threads` as the total number of threads, **one** way to divide  $N$  iterations for thread with ID `thread_id` is as follows:

```
int blen, bstart;

blen = N/num_threads;

if (thread_id < (N%num_threads)) {
    blen = blen + 1;
    bstart = blen * thread_id;
}
else {
    bstart = blen * thread_id + (N%num_threads);
}
```

Why is this a fair division?

# “Manual” loop parallelization (3)

## OpenMP coding

```
#pragma omp parallel
{
    int num_threads, thread_id;
    int blen, bstart, bend, i;

    num_threads = omp_get_num_threads();
    thread_id = omp_get_thread_num();

    blen = N/num_threads;
    if (thread_id < (N%num_threads)) {
        blen = blen + 1;
        bstart = blen * thread_id;
    }
    else {
        bstart = blen * thread_id + (N%num_threads);
    }
    bend = bstart + blen;

    for (i=bstart; i<bend; i++)
        a[i] = b[i] + c[i];

} // end of parallel region
```

## Data scoping

Any variables that existed before a parallel region still exist inside the parallel region, and are by default **shared** between all threads.

Often it will be necessary for the threads to have some *private* variables.

- Each thread can either declare new local variables inside the parallel region, these variables are private “by birth”;
- Or, each thread can “privatize” some of the shared variables that already existed before a parallel region (using the **private** clause):

```
int blen, bstart, bend;  
#pragma omp parallel private(blen, bstart, bend)  
{  
    // ...  
}
```

- Each “privatized” variable has one (*uninitialized*) instance per thread;
- The private variables’ scope is until the end of the parallel region.

# Actually, parallelizing a for-loop is easy in OpenMP

- Parallelizing for-loops is OpenMP's main work-sharing mechanism.
- OpenMP has several built-in strategies for dividing the iterations among the threads.
  - No need to manually calculate each thread's loop bounds

```
#pragma omp parallel
{
#pragma omp for
    for (i=0; i<N; i++)
        a[i] = b[i] + c[i];
} // end of parallel region
```

or simply

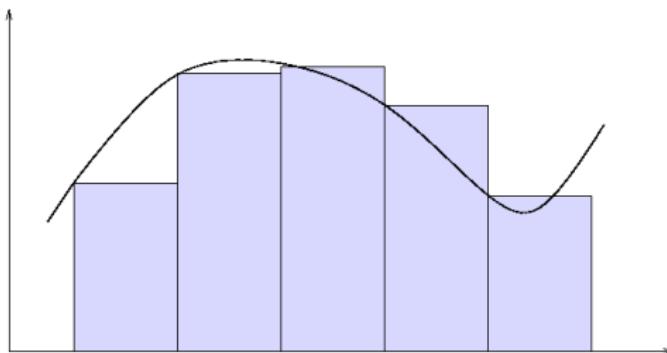
```
#pragma omp parallel for
    for (i=0; i<N; i++)
        a[i] = b[i] + c[i];
```

## More remarks

- The loop can not contain break, return, exit statements.
- The continue statement is allowed.
- The index update has to be an increment (or decrement) by a fixed amount.
- The loop index variable is automatically private, and changes to it inside the loop are not allowed.

# Numerical integration

How to numerically calculate  $\int_{x_0}^{x_1} f(x)dx$  ?



$$w = \frac{x_1 - x_0}{N}$$

$$\begin{aligned}\int_{x_0}^{x_1} f(x)dx &\approx w \left( f(x_0 + \frac{w}{2}) + f(x_0 + w + \frac{w}{2}) + \dots \right. \\ &\quad \left. \dots + f(x_0 + (N-1)w + \frac{w}{2}) \right)\end{aligned}$$

# Numerical integration for calculating $\pi$

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

Serial implementation:

```
int N, i;
double w = 1.0/N, x, approx_pi;
double sum = 0.;

for (i=1; i<=N; i++) {
    x = w*(i-0.5);
    sum = sum + 4.0/(1.0+x*x);
}

approx_pi = w*sum;
```

# A naive OpenMP implementation

```
int N, i;
double w = 1.0/N, x, approx_pi = 0.;
double sum = 0.;

#pragma omp parallel private(x) firstprivate(sum)
{

#pragma omp for
for (i=1; i<=N; i++) {
    x = w*(i-0.5);
    sum = sum + 4.0/(1.0+x*x);
}

#pragma omp critial
{
approx_pi = approx_pi + w*sum;
}

} // end of the parallel region
```

## OpenMP critical regions

- Concurrent write accesses to a shared variable must be avoided by all means to circumvent **race conditions**.
- An OpenMP critical code block is executed by **one thread at a time**. This is one way to avoid race conditions. (There are other ways.)
- The variable `approx_pi` in the above example is a shared variable, to which all the threads will write. Thus, “protection” is provided by a **critical** code block.
- Use of the **critical** directive will incur overhead.
- Improper use of the **critical** directive may lead to **deadlock**.

# Use of OpenMP's reduction clause

Actually, using `critical` to prevent concurrent writes to the variable `approx_pi` is an “over-kill”. The reduction clause of OpenMP is designed for this particular purpose:

```
int N, i;
double sum = 0.;
double w = 1.0/N, x, approx_pi;

#pragma omp parallel for private(x) reduction(+:sum)
for (i=1; i<=N; i++) {
    x = w*(i-0.5);
    sum = sum + 4.0/(1.0+x*x);
}

approx_pi = w*sum;
```

## Another example of using the reduction clause

$$s = a_0^2 + a_1^2 + \dots + a_{N-1}^2$$

```
#pragma omp parallel for reduction(+:s)
for (i=0; i<N; i++)
    s = s + a[i]*a[i];
```

# Loop scheduling

```
#pragma omp parallel for  
for (i=0; i<N; i++)  
    a[i] = b[i] + c[i];
```

How are the loop iterations exactly divided among the threads?

- Mapping of loop iterations to threads is configurable in OpenMP.

- The “secret” is the `schedule` clause:

```
#pragma omp parallel for schedule(static|dynamic|guided [,chunk])
```

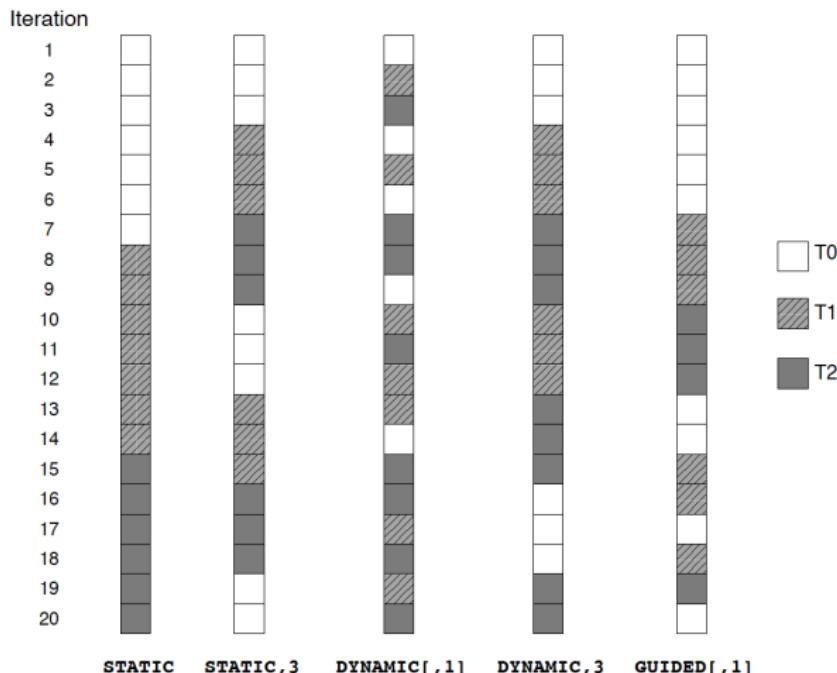
- Default scheduling is `static` (no need to specify), which divides the iterations into contiguous chunks of (roughly) equal size.

- Other alternatives of scheduling: `dynamic` and `guided`

- It is possible to prescribe a chunksize in the `schedule` clause.

```
#pragma omp parallel for schedule(dynamic,3)
```

# Examples of different schedulers



**Figure 6.2:** Loop schedules in OpenMP. The example loop has 20 iterations and is executed by three threads (T0, T1, T2). The default chunksize for DYNAMIC and GUIDED is one. If a chunksize is specified, the last chunk may be shorter. Note that only the STATIC schedules guarantee that the distribution of chunks among threads stays the same from run to run.

# Tasking

- A task can be defined by OpenMP's task directive, containing the code to be executed.
- When a thread encounters a task construct, it may execute it right away or set up the appropriate data environment and defer its execution. The task is then ready to be executed later by any thread of the team.

# An example of OpenMP tasks

```
#pragma omp parallel private(r, i)
{
    #pragma single
    {
        for (i=0; i<N; i++) {
            r = rand(); // a randomly generated number
            if (p[i] > r) {
                #pragma task
                {
                    do_some_work (p[i]);
                }
            } // enf of if-test
        } // end of for-loop
    } // end of the single directive
} // end of the parallel region
```

The actual number of calls to `do_some_work` is unknown, so tasking is a natural choice for work division.

## single and master

A “single” code block in OpenMP will be entered by one thread only, namely the thread that reaches the `single` directive first. All others skip the code and wait at the end of the `single` block due to an implicit barrier.

A “master” code block is only entered by the master thread, all the other threads skip over without waiting for the master thread to finish.

OpenMP has a separate “barrier” directive for explicit synchronization among the threads. (Use with care!!!)

## Yet another example

```
int myid, numthreads;

#pragma omp parallel private(myid)
{
    my_id = omp_get_thread_num();

#pragma omp single
{
    numthreads = omp_get_num_threads();
}

#pragma omp critical // not strictly necessary
{
    printf("This is thread No.%d out of %d threads\n",
           my_id, numthreads);
}

} // end of the parallel region
```

# IN3200/IN4200: Chapter 6

## Shared-memory parallel programming with OpenMP (Part 2)

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

- A quick recap of simple OpenMP programming
- Two examples of parallelizing computations on uniform grids

## Quick recap of OpenMP

- OpenMP: A parallel programming standard for shared-memory parallel computers
  - A set of compiler directives (and additional clauses)
  - A small number of library functions
  - A few environment variables
- The central execution entities in an OpenMP program are *threads*—lightweight processes.
- The execution model of OpenMP is “fork & join”—a team of threads created inside each *parallel region*, but only the master thread exists between parallel regions.

# OpenMP compiler directives

An OpenMP compiler directive is a “parallelization hint” intended for a compatible compiler to automatically create parallel code.

To write an OpenMP program “means” to insert OpenMP compiler directives (plus additional code) at suitable locations of an existing serial program.

Some of the most important OpenMP directives:

- `#pragma omp parallel`
- `#pragma omp for`
- `#pragma omp single`
- `#pragma omp master`
- `#pragma omp critical`
- `#pragma omp barrier`

# Data scoping

Any variables declared before a parallel region are by default *shared* between all threads in the parallel region.

It can be necessary for the threads to have some *private* variables.

- Each thread can either declare new local variables inside the parallel region, these variables are private “by birth”;
- Or, each thread can “privatize” some of the shared variables that already existed before a parallel region (using the `private` or `firstprivate` clause):

```
int blen, bstart, bend;
#pragma omp parallel private(blen, bstart, bend)
{
    // ...
}
```

- The private variables’ scope is until the end of the parallel region.

# Hello world in OpenMP (each thread writes a line of text)

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

int main (int nargs, char **args) {

    printf("I'm the master thread, I'm alone.\n");

#pragma omp parallel
{
    int num_threads, thread_id;
    num_threads = omp_get_num_threads();
    thread_id = omp_get_thread_num();
    printf("Hello world! I'm thread No.%d out of %d threads.\n",
           thread_id,num_threads);
}

    return 0;
}
```

Example of compilation: `gcc -fopenmp hello_world_omp.c`

Example of choosing number of threads:

`setenv OMP_NUM_THREADS 4` (or `export OMP_NUM_THREADS=4`)

Parallel execution: `./a.out`

# Parallelizing a for-loop is easy in OpenMP

- Parallelizing for-loops is OpenMP's main work-sharing mechanism.
- OpenMP has several built-in strategies for dividing the iterations among the threads (with help of the schedule clause).

```
#pragma omp parallel
{
#pragma omp for
    for (i=0; i<N; i++)
        a[i] = b[i] + c[i];
} // end of parallel region
```

or simply

```
#pragma omp parallel for
for (i=0; i<N; i++)
    a[i] = b[i] + c[i];
```

# OpenMP's reduction clause

Used to specify one or more thread-private variables that are subject to a reduction operation at the end of a parallel region

Often used reduction operations: +, \*, max, min

```
max_val = 0.;

#pragma omp parallel for reduction(max: max_val)
for (i=0; i<N; i++) {
    if (arr[i] > max_val)
    {
        max_val = arr[i];
    }
}
```

# Jacobi algorithm on a 2D uniform grid

Serial C implementation (slightly different from Listing 3.1 in Chapter 3):

```
double maxdelta = 1.0, eps = 1.0e-14;

while (maxdelta > eps) {
    maxdelta = 0.;

    for (k=1; k<kmax-1; k++)
        for (i=1; i<imax-1; i++) {
            phi_new[k][i] = (phi[k-1][i]+ph[k][i-1]
                               +phi[k][i+1]+phi[k+1][i])*0.25;
            maxdelta = max(maxdelta, abs(phi_new[k][i]-phi[k][i]));
        }

    /* pointer swapping */
    temp_ptr = phi_new;
    phi_new = phi;
    phi = temp_ptr;
}
```

# OpenMP-parallel Jacobi algorithm

OpenMP code (slightly different from Listing 6.5 in the textbook):

```
double maxdelta = 1.0, eps = 1.0e-14;

while (maxdelta > eps) {
    maxdelta = 0.;

    #pragma omp parallel for reduction(max: maxdelta) private(i)
    {
        for (k=1; k<kmax-1; k++)
            for (i=1; i<imax-1; i++) {
                phi_new[k][i] = (phi[k-1][i]+ph[k][i-1]
                                  +phi[k][i+1]+phi[k+1][i])*0.25;
                maxdelta = max(maxdelta, abs(phi_new[k][i]-phi[k][i]));
            }
    } // end of the parallel region

    /* pointer swapping */
    temp_ptr = phi_new;
    phi_new = phi;
    phi = temp_ptr;
}
```

# OpenMP-parallel Jacobi algorithm (version 2, just one parallel region)

```
double maxdelta = 1.0, eps = 1.0e-14;

#pragma omp parallel
{
    while (maxdelta > eps) {
        #pragma omp barrier // preventing maxdelta from being zeroed too earlier
        #pragma omp single
        {
            maxdelta = 0.;

        #pragma omp for reduction(max: maxdelta) private(i)
        {
            for (k=1; k<kmax-1; k++)
                for (i=1; i<imax-1; i++) {
                    phi_new[k][i] = (phi[k-1][i]+ph[k][i-1]
                                      +phi[k][i+1]+phi[k+1][i])*0.25;
                    maxdelta = max(maxdelta, abs(phi_new[k][i]-phi[k][i]));
                }
        }

        #pragma omp master
        {
            /* pointer swapping */
            temp_ptr = phi_new;
            phi_new = phi;
            phi = temp_ptr;
        }
    } // end of while loop

} // end of the entire parallel region
```

# Gauss-Seidel algorithm on a 3D uniform grid

What if the iterations of a triple loop nest are *not entirely independent*?

**Example:** Gauss-Seidel algorithm on a 3D grid (we only show the computational core: *stencil update* per inner grid point)

```
for (k=1; k<kmax-1; k++)
    for (j=1; j<jmax-1; j++)
        for (i=1; i<imax-1; i++)
            phi[k][j][i] = (phi[k-1][j][i] + phi[k][j-1][i]
                            +phi[k][j][i-1] + phi[k][j][i+1]
                            +phi[k][j+1][i] + phi[k+1][j][i])/6.0;
```

We cannot just add `#pragma omp parallel for` before the `k`-indexed loop. (There are loop-carried independences.)

**Note:** The upper limits of `k`, `j` and `i` are different from those given in Chapter 6 of the textbook (Listing 6.6).

## Wavefront parallelization

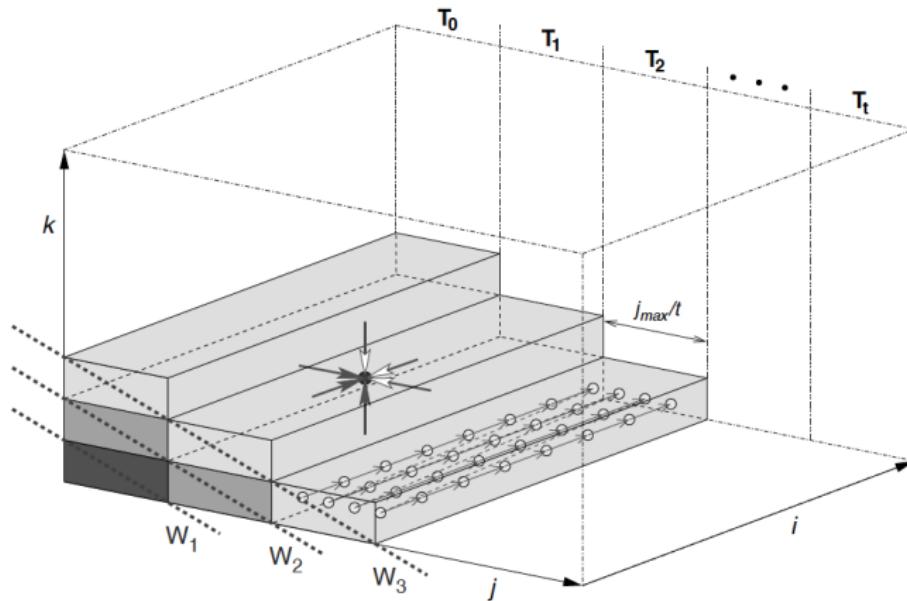
Although not as simple as the Jacobi algorithm, it is still possible to parallelize the Gauss-Seidel algorithm with OpenMP.

The key idea is to find a different way of traversing the 3D grid (with parallelism) while still fulfilling the dependency constraints imposed by the stencil update.

**Note:** As soon as an entire  $x$ -row associated with index  $(j, k)$  is computed, two entire  $x$ -rows become ready for *concurrent* computation: associated with index  $(j + 1, k)$  and index  $(j, k + 1)$ .

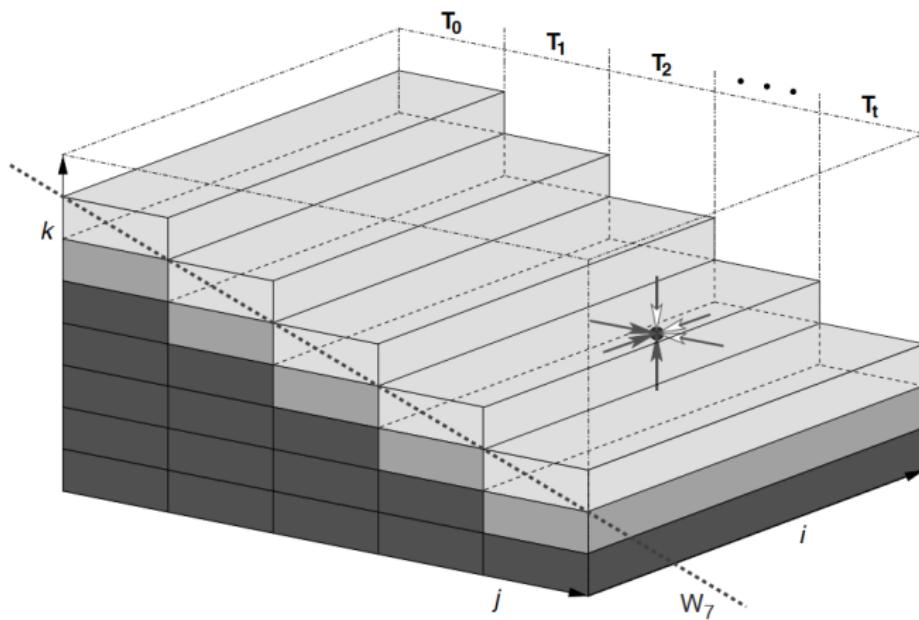
A *wavefront* travels in the  $k$  direction. The dimension along which to parallelize is  $j$ . Each of the threads,  $T_0, T_1, \dots, T_{t-1}$ , is assigned a consecutive chunk of the  $j$  indices.

## Wavefront parallelization (2)



**Figure 6.4:** Pipeline parallel processing (PPP), a.k.a. wavefront parallelization, for the Gauss–Seidel algorithm in 3D (wind-up phase). In order to fulfill the dependency constraints of each stencil update, successive wavefronts ( $W_1, W_2, \dots, W_n$ ) must be performed consecutively, but multiple threads can work in parallel on each individual wavefront. Up until the end of the wind-up phase, only a subset of all  $t$  threads can participate.

## Wavefront parallelization (3)



**Figure 6.5:** Wavefront parallelization for the Gauss–Seidel algorithm in 3D (full pipeline phase). All  $t$  threads participate. Wavefront  $W_7$  is shown as an example.

## Important observations

- The  $k$  index goes between 1 and  $k_{\max}-2$ .
- All the  $j$  indices  $1, 2, \dots, j_{\max}-2$  are divided evenly into consecutive chunks:  $J_0, J_1, \dots, J_{t-1}$  (one chunk per thread).
- Total number of wavefronts:  $(k_{\max}-2)+t - 1$ , for computing through the entire 3D lattice
  - Wavefront  $W_1$  has only one block ( $k=1, J_0$ )
  - Wavefront  $W_2$  has two concurrent blocks ( $k=1, J_1$ ) and ( $k=2, J_0$ )
  - Wavefront  $W_3$  has three concurrent blocks ( $k=1, J_2$ ), ( $k=2, J_1$ ) and ( $k=3, J_0$ )
  - ...
  - For wavefronts  $W_t, W_{t+1}, \dots, W_{k_{\max}-2}$ , each has  $t$  concurrent blocks
  - Wavefronts  $W_{k_{\max}-1}, \dots, W_{k_{\max}-2+t-1}$  have fewer and fewer concurrent blocks (the wind-down phase).

# OpenMP waveform parallelization

```
#pragma omp parallel private(k,j,i)
{
    int numthreads, threadID, jstart, jend, m;

    numthreads = omp_get_num_threads();
    threadID = omp_get_thread_num();
    jstart = ((jmax-2)*threadID)/numthreads + 1;
    jend = ((jmax-2)*(threadID+1))/numthreads;

    for (m=1; m<=kmax+numthreads-3; m++) { // the wavefronts
        k = m - threadID;
        if (k>=1 && k<=kmax-2) {
            for (j=jstart; j<=jend; j++)
                for (i=1; i<imax-1; i++)
                    phi[k][j][i] = (phi[k-1][j][i] + phi[k][j-1][i]
                                    +phi[k][j][i-1] + phi[k][j][i+1]
                                    +phi[k][j+1][i] + phi[k+1][j][i])/6.0;
        }
        #pragma omp barrier
    }
} // end of the parallel region
```

# IN3200/IN4200: Chapter 7

## Efficient OpenMP programming

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

# Outline

- OpenMP programming is user-friendly
  - requires mostly inserting compiler directives
  - allows incremental parallelization
  - suits well for parallelizing loops with independent iterations
- However, writing a truly scalable OpenMP program is not trivial.
- This chapter pinpoints some of the performance problems and discusses how they can be circumvented.

# Profiling OpenMP programs

- Good profiling tools can clearly show some of the performance problems
  - Example profilers: **SCORE-P** (<https://www.vi-hps.org/projects/score-p/>), **TAU** (<https://www.cs.uoregon.edu/research/tau/home.php>)

## Example of “imbalanced iterations”

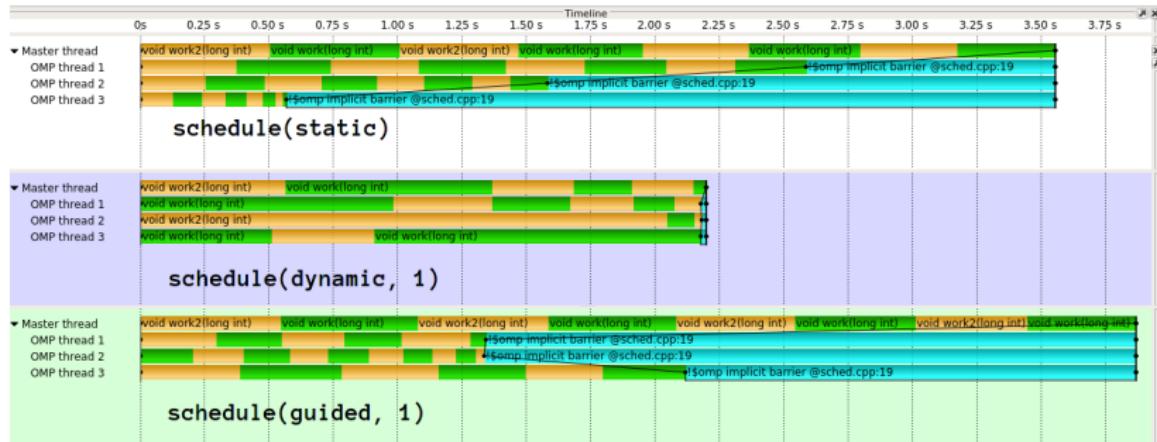
```
#include <omp.h>

void work(long ww) {
    volatile long sum = 0;
    for (long w = 0; w < ww; w++) sum += w;
}

int main() {
    const long max = 32, factor = 100000001;
    #pragma omp parallel for schedule(runtime)
    for (int i = 0; i < max; i++) {
        work((max - i) * factor);
    }
}
```

<https://stackoverflow.com/questions/42970700/openmp-dynamic-vs-guided-scheduling/43047074>

# Profiling results



Using GCC 6.3.1, Score-P / Vampir for visualization, two alternating work functions for coloring.

<https://stackoverflow.com/questions/42970700/openmp-dynamic-vs-guided-scheduling/43047074>

# Performance pitfalls

- OpenMP is prone to the “standard problems” of parallel programming: **serial fraction** (Amdahl’s law) and **load imbalance**.
- Communication (in terms of data transfer) on shared memory is usually much less costly than on distributed memory, but ccNUMA can potentially cause performance problem (will be discussed in Chapter 8).
- There are specific performance problems inherent with shared-memory programming.

## Non-negligible overhead

- Whenever a parallel region is started or stopped, or a parallel loop is initiated or ended, there is some overhead involved, because
  - threads must be spawned (or least woken up from an idle state)
  - the “loop work” for each thread must be determined
  - for dynamic and guided scheduling, each thread that becomes available must be supplied with a new task
  - the default **barrier** synchronizes the threads
- These costs can be counted as “communication overhead” in refined scalability models:

$$S_{\text{omp}}(N) = \frac{1}{s + (1 - s)/N + \kappa N + \lambda}$$

where  $N$  denotes the number of threads,  $\lambda$  denotes overhead that is independent of  $N$

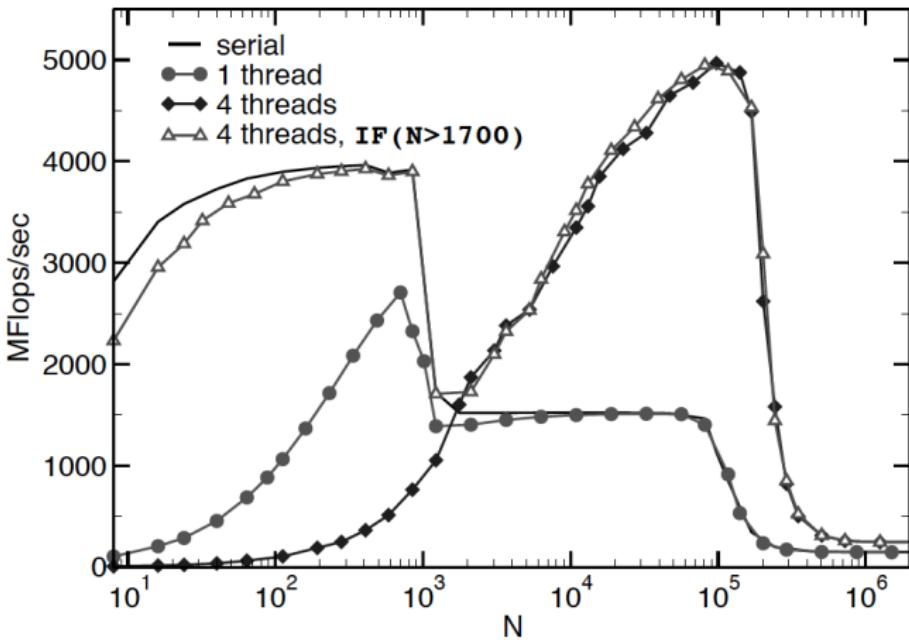
## Run serial code if parallelism does not pay off

If the worksharing construct does not contain enough “work” per thread because, e.g., each iteration of a short loop executes in a short time, OpenMP overhead will lead to very bad performance.

Use the “if” clause:

```
#pragma omp parallel for if (N>some_threadshold)
for (i=0; i<N; i++)
    A[i] = B[i] + C[i]*D[i];
```

# Performance example



**Figure 7.3:** OpenMP overhead and the benefits of the `IF(N>1700)` clause for the vector triad benchmark. (Dual-socket dual-core Intel Xeon 5160 3.0 GHz system like in Figure 7.2, Intel compiler 10.1).

## Another option

It is possible to reduce the number of threads for a particular part.

```
#pragma omp parallel for num_threads(2)
for (i=0; i<N; i++)
    A[i] = B[i] + C[i]*D[i];
```

“num\_threads” is a clause.

## Avoid implicit barriers

Be aware that most OpenMP worksharing constructs (including `#pragma omp for`) insert an automatic barrier at the end.

If you're **certain** that the default barrier (synchronization) is not necessary, then specify the "nowait" clause.

# Parallelizing loop nests

Try to parallelize loop nests on a level as far out as possible.

Parallelizing inner loop levels will lead to increased OpenMP overhead because a team of threads is spawned or woken up multiple times.

A bad example:

```
int i,j;
for (i=0; i<N; i++) {
    double R=0.;

#pragma omp parallel for reduction(+:R)
    for (j=0; j<M; j++)
        R += A[i][j]*B[j];

    C[i] = R;
}
```

How would you improve?

# Merging parallel regions

Before:

```
int i;
double S, R=0.;

#pragma omp parallel for reduction(+:R)
for (i=0; i<N; i++) {
    A[i] = do_work(B[i]);
    R += A[i];
}

S = sin(R);

#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] += S;
```

There are two parallel regions.

# Merging parallel regions

After:

```
int i;
double S, R=0.;

#pragma omp parallel private(S)
{
    #pragma omp for reduction(+:R)
    for (i=0; i<N; i++) {
        A[i] = do_work(B[i]);
        R += A[i];
    }
    S = sin(R);

    #pragma omp for nowait
    for (i=0; i<N; i++)
        A[i] += S;
}

} // end of the parallel region
```

Only one parallel region.

## Beware of “trivial” load imbalance

The parallel loop trip count should be large compared with the number of threads, otherwise (serious) load imbalance can easily arise.

```
double res=0;
#pragma omp parallel for reduction(+:res) private(j,k)
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            res += A[i][j][k];
```

High possibility of load imbalance if the value of N is small.

## “Collapsing” a loop nest

```
double res=0;  
#pragma omp parallel for reduction(+:res) private(j,k) collapse(3)  
for (i=0; i<N; i++)  
    for (j=0; j<N; j++)  
        for (k=0; k<N; k++)  
            res += A[i][j][k];
```

“collapse” is a clause, can help to improve the load balance.

## Be careful about dynamic/guided scheduling or tasking

The dynamic/guided loop scheduling options and tasking constructs require some amount of nontrivial computation or bookkeeping in order to figure out which thread is to compute the next chunk or task. This overhead can be significant if each task contains only a small amount of work.

## Be careful about “critical” regions

How to parallelize the following code?

```
int i,j, r_number;  
  
for (i=0; i<N; i++) {  
    r_number = row_calc(i);  
    for (j=0; j<10; j++)  
        M[r_number][j] = M[r_number][j] + some_function(i,j);  
}
```

## One “bad” solution of parallelization

```
int i,j, r_number;

#pragma omp parallel for private(j, r_number)
for (i=0; i<N; i++) {
    r_number = row_calc(i); // different threads may get same 'r_number'!

    #pragma critical
    {
        for (j=0; j<10; j++)
            M[r_number][j] = M[r_number][j] + some_function(i,j);
    }
}
```

This is a bad solution, because the work is serialized.

## A better solution

To improve the previous solution, we can use an array of “locks”.

```
int i,j, r_number;
omp_lock_t locks[N];
for (i=0; i<N; i++)
    omp_init_lock(&locks[i]);

#pragma omp parallel for private(j, r_number)
for (i=0; i<N; i++) {
    r_number = row_calc(i); // different threads may get same 'r_number'!

    omp_set_lock(&locks[r_number]);
    for (j=0; j<10; j++)
        M[r_number][j] = M[r_number][j] + some_function(i,j);
    omp_unset_lock(&locks[r_number]);
}

for (i=0; i<N; i++)
    omp_destroy_lock(&locks[i]);
```

This solution is better, because it allows concurrent execution among the threads (working on different `r_number`), while avoiding race conditions. (There is overhead with setting/unsetting the locks.)

# Let's look at another example

How to parallelize the following code?

```
int i, ind;  
int S[8];  
  
for (ind=0; ind<8; ind++)  
    S[ind] = 0;  
  
for (i=0; i<N; i++) {  
    ind = A[i]%8;  
    S[ind] += 1;  
}
```

# One solution that will suffer from “false sharing”

```
int i, ind;
int S[max_num_threads+1][8];
for (ind=0; ind<8; ind++)
    S[0][ind] = 0;

#pragma omp parallel private(ind)
{
    int thread_id = omp_get_thread_num()+1;

    for (ind=0; ind<8; ind++)
        S[thread_id][ind] = 0;

    #pragma omp for nowait
    for (i=0; i<N; i++) {
        ind = A[i]%8;
        S[thread_id][ind] += 1;
    }

    #pragma critical
    {
        for (ind=0; ind<8; ind++)
            S[0][ind] += S[thread_id][ind];
    }
}
```

# Avoiding false sharing

The problem with the previous solution is that each row of the 2D array “S” has only 8 integers (32 bytes). Therefore, multiple rows of “S” will occupy the same cache line → **false sharing**.

“**Padding**” the 2D array “S” will remove this potential bottleneck.  
For example:

```
int S[max_num_threads+1][64];
```

# IN3200/IN4200: Chapter 8

## Locality optimizations on ccNUMA architectures

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

# Objective

- Utilize “maximum” memory bandwidth on ccNUMA architectures
  - Data placement
  - Locality of access

## Recap of ccNUMA

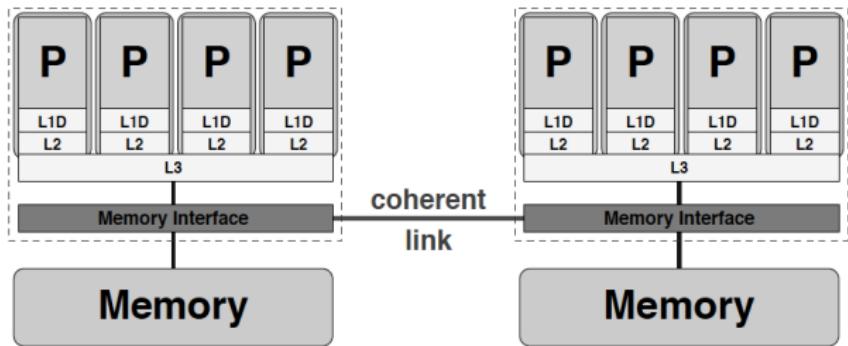
- *Cache-coherent Nonuniform Memory Access* (ccNUMA) systems have a physically distributed memory that is *logically shared*. The aggregated memory appears as one single address space. Memory access performance depends on which CPU (core) accesses which parts of memory ("local" vs. "remote" access).
- Although there is a single address space (shared memory), there are private caches, or partially shared caches, for the different CPU cores (also true on UMA systems).
- *Cache coherence* protocols guarantee *consistency* between cached data and data in the shared memory at all times.

## Locality domains

- A *locality domain* (LD) is a set of processor cores together with locally connected memory. This “local” memory can be accessed by the set of processor cores in the most efficient way, without resorting to a network of any kind.
- Each LD is a UMA building block.
- Multiple LDs are linked via a coherent interconnect, which can mediate direct, cache-coherent memory accesses. (This mechanism is transparent for the programmer.)

# Example of ccNUMA

**Figure 4.5:** A ccNUMA system with two locality domains (one per socket) and eight cores.



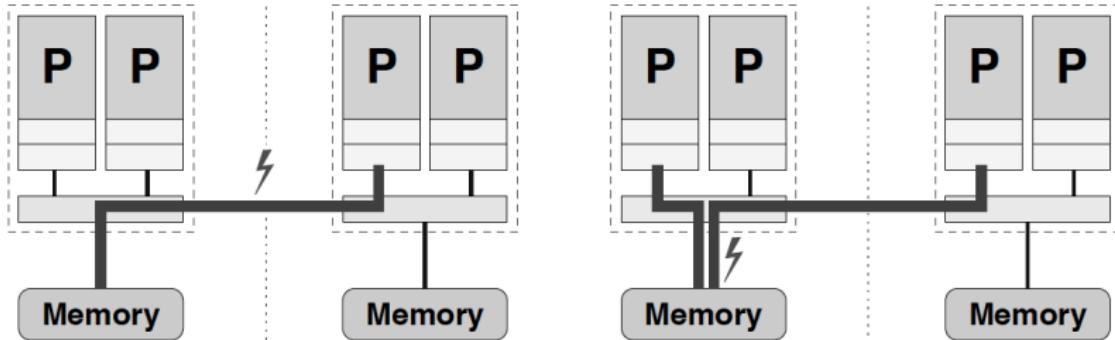
## Penalty for non-local transfers

The *locality problem*: Non-local memory transfers (between LDs) are more costly than local transfers (within a LD).

The *contention problem*: If two processors from different LDs access memory in the same LD, fighting for memory bandwidth.

Both problems can be “solved” (alleviated) by carefully observing the data access patterns of an application and restricting data access of each processor (mostly) to its own LD, through proper programming.

# Locality and contention problems

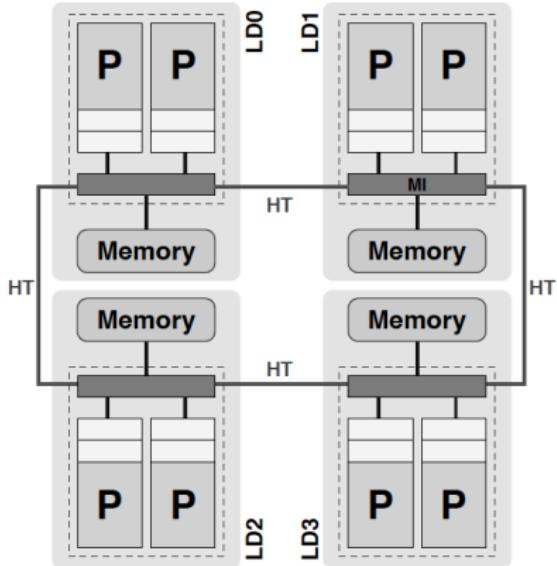


**Figure 8.1:** Locality problem on a cc-NUMA system. Memory pages got mapped into a locality domain that is not connected to the accessing processor, leading to NUMA traffic.

**Figure 8.2:** Contention problem on a cc-NUMA system. Even if the network is very fast, a single locality domain can usually not saturate the bandwidth demands from concurrent local and nonlocal accesses.

Both problems are due to “misplacement” of threads and data.

# Example of non-uniform memory access costs



**Figure 8.3:** A ccNUMA system (based on dual-core AMD Opteron processors) with four locality domains LD0 ... LD3 and two cores per LD, coupled via a HyperTransport network. There are three NUMA access levels (local domain, one hop, two hops).

# The “placement” problem

Proper page placement is essential.

- Make sure that memory gets mapped into the locality domains of processors that actually access them. This minimizes NUMA traffic across the network. (“Mapping” means that a page table entry is set up, which describes the association of a physical with a virtual memory page.)
- Threads or processes must be pinned to those CPUs which had originally mapped their memory regions in order not to lose locality of access. (Need to use appropriate thread affinity mechanisms.)

## Page placement by first touch

- Operating system and runtime libraries normally support a **first touch** policy for memory pages: A page gets mapped into the locality domain of the processor that first **writes** to it.
- Note:** Merely *allocating* memory (such as `malloc`) is not sufficient.

## Data initialization is key

- The data initialization part deserves close attention on ccNUMA.
  - Allocating arrays should use dynamic (heap) memory (`malloc`).
  - The `calloc` function will most probably be counterproductive.

## A bad example

```
int *A, *B, *C, *D, i;

A = (int*)malloc(N*sizeof(int));
B = (int*)malloc(N*sizeof(int));
C = (int*)malloc(N*sizeof(int));
D = (int*)malloc(N*sizeof(int));

for (i=0; i<N; i++) {
    B[i] = i;
    C[i] = i%5;
    D[i] = i%10;
}

#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i]*D[i];
```

## Why is the previous example bad?

- If the code is run across several locality domains, it will not scale beyond the maximum performance achievable on a single LD (if the working set does not fit into cache).
- This is because the initialization loop is executed by a single thread, writing to B, C, and D for the first time.
- Hence, all memory pages belonging to those arrays will be mapped into a single LD.
- Problem: many “remote” accesses and severe contention

# Fixing the initialization problem

```
int *A, *B, *C, *D, i;

A = (int*)malloc(N*sizeof(int));
B = (int*)malloc(N*sizeof(int));
C = (int*)malloc(N*sizeof(int));
D = (int*)malloc(N*sizeof(int));

#pragma omp parallel for
for (i=0; i<N; i++) {
    B[i] = i;
    C[i] = i%5;
    D[i] = i%10;
}
```

## First touch by “fake” initialization

- Sometimes initializing an array with useful values can only be done by one thread.
- Then, a parallelized “fake” (unnecessary) initiation beforehand can secure the desirable first touch.

## Some requirements

- The OpenMP loop schedules of initialization and work loops must be identical and reproducible.
  - Must use static scheduler and identical chunksize
- For successive parallel loops with the same number of iterations and the same number of parallel threads, each thread should get the same part of the iteration space in both loops.
  - Also must use static scheduler and identical chunksize
- Beware of cache-coherence traffic, which can destroy the achievable aggregate memory bandwidth.
- Beware of global data (which is initialized before the `main()` function). Properly mapped local copies of global data may be a possible solution.

## NUMA-unfriendly OpenMP scheduling

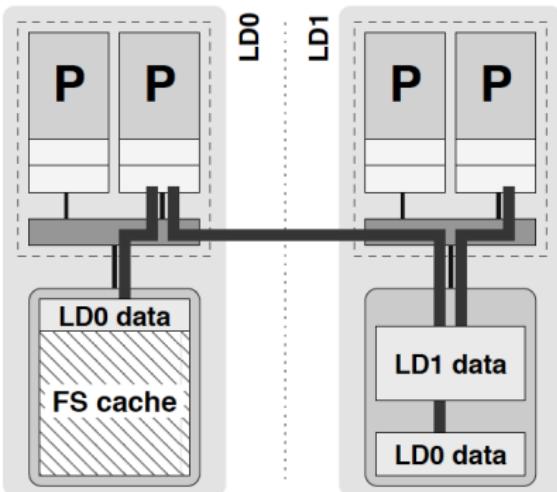
- Dynamic/guided loop scheduling and OpenMP task constructs could be preferable over static work distribution in poorly load-balanced situations.
- On the other hand, any sort of dynamic scheduling (including tasking) will necessarily lead to scalability problems if the thread team is spread across several locality domains.
- This is because the assignment of tasks to threads is unpredictable and even changes from run to run, which rules out an “optimal” page placement strategy.
- In such cases, it may be best to distribute the working set’s memory pages round-robin (choosing a suitable chunksize) across the locality domains.

## Another potential problem: File system cache

- Disk I/O operations cause operating systems to set up buffer caches which store recently read or written file data for efficient re-use.
- The size and placement of such caches can be unfortunate with respect to ccNUMA locality.

## An example of bad impact due to file system cache

**Figure 8.8:** File system buffer cache can prevent locally touched pages to be placed in the local domain, leading to nonlocal access and contention. This is shown here for locality domain 0, where FS cache uses the major part of local memory. Some of the pages allocated and initialized by a core in LD0 get mapped into LD1.



# IN3200/IN4200: Chapter 9

## Distributed-memory parallel programming with MPI

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

# Objective

- The concept of explicit “message passing”
- Introduction to MPI (the Message Passing Interface)

# Message passing

- Shared-memory programming (such as OpenMP) does not work for distributed-memory parallel computers
  - There is no way for one processor to directly access the address space of another process
- Explicit “**message passing**” is required on distributed-memory systems
  - This can also be a programming model for shared-memory or hybrid systems

# Basic features of message-passing programming

- The same program runs on all processes (Single Program Multiple Data, or **SPMD**)—no difference from OpenMP programming in this regard
- The work of each process is implementation in a sequential language (such as C)
  - Data exchange (sending and receiving messages) is done via calls to an appropriate library
- All variables in a process are **local** to this process (nothing is shared)

“Messages” carry data to be exchanged between processes

Information needed about a message:

- Which process is sending the message?
- Where is the data on the sending process?
- What kind of data is being sent?
- How much data constitutes the message?
- Which process is going to receive the message?
- Where should the data be placed on the receiving process?
- What amount of data is the receiving process prepared to accept?

# MPI (Message Passing Interface)

MPI is a *library standard* for programming distributed memory

- MPI implementation(s) available on almost every major parallel platform (also on shared-memory machines)
- Portability, good performance & functionality
- Collaborative computing by a group of individual processes
- Each process has its own local memory
- Explicit message passing enables information exchange and collaboration between processes

# MPI C language binding

```
#include <mpi.h>  
  
rv = MPI_Xxxxx(parameter, ... )
```

Beware of the **case-sensitive** naming pattern.

The return value (rv) transports information about the success of the MPI operation. (**MPI\_SUCCESS** is returned if the MPI routine completed successfully.)

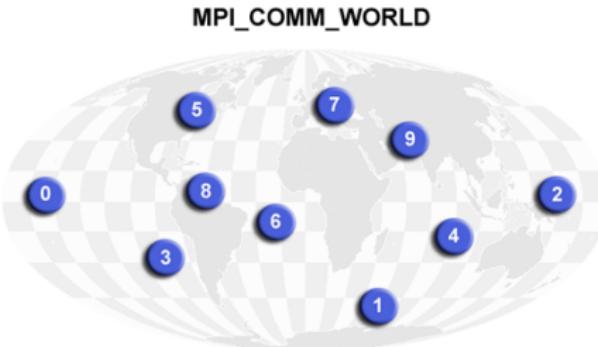
## MPI\_Init

The MPI\_Init function initializes the parallel environment.

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

The MPI\_Init function takes pointers to the main() function's input arguments so that the library can evaluate and remove any additional command line arguments that may have been added by the MPI startup process.

# MPI communicator



<https://computing.llnl.gov/tutorials/mpi/>

- An MPI communicator is a "communication universe" for a group of processes
- MPI\_COMM\_WORLD – name of the default MPI communicator, i.e., the collection of all processes (started by an execution)
- Each process in a communicator is identified by its unique rank
- Almost every MPI command needs to provide a communicator as input argument

## MPI process rank

- Each process has a unique rank, i.e. an integer identifier, within a communicator
- The rank value is between 0 and #procs-1
- The rank value is used to distinguish one process from another
- Commands MPI\_Comm\_size & MPI\_Comm\_rank are very useful

```
int size, my_rank;  
MPI_Comm_size (MPI_COMM_WORLD, &size);  
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
```

## MPI\_Finalize

An MPI parallel program is shut down by a call to `MPI_Finalize()`.

Note that no MPI process except rank 0 is guaranteed to execute any code beyond `MPI_Finalize()`.

# "Hello world" in MPI

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

int main (int nargs, char** args)
{
    int size, my_rank;
    MPI_Init (&nargs, &args);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
    printf("Hello world, I've rank %d out of %d procs.\n",
           my_rank,size);
    MPI_Finalize ();
    return 0;
}
```

## Example running result of “Hello world”

- Example of compilation: `mpicc hello_world_mpi.c`
- Example of parallel execution: `mpirun -np 4 ./a.out`
- Example running result (note: no deterministic order of the output):

Hello world, I've rank 2 out of 4 procs.

Hello world, I've rank 1 out of 4 procs.

Hello world, I've rank 3 out of 4 procs.

Hello world, I've rank 0 out of 4 procs.

**Note:** Compilation and execution can vary from system to system.

# An abstract “picture” of parallel execution

Process 0

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

int main (int argc, char** argv)
{
    int size, my_rank;
    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
    printf("Hello world, I'm rank %d out of %d proc.\n",
           my_rank, size);
    MPI_Finalize ();
    return 0;
}
```

Process 1

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

int main (int argc, char** argv)
{
    int size, my_rank;
    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
    printf("Hello world, I'm rank %d out of %d proc.\n",
           my_rank, size);
    MPI_Finalize ();
    return 0;
}
```

...

Process  $P-1$

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

int main (int argc, char** argv)
{
    int size, my_rank;
    MPI_Init (&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
    printf("Hello world, I'm rank %d out of %d proc.\n",
           my_rank, size);
    MPI_Finalize ();
    return 0;
}
```

# MPI message

An MPI message is simply an array of elements of a particular MPI data type.

Data types can either be standard types (pre-defined) or *derived types*, which must be defined by appropriate MPI calls.

| MPI type   | C type      |
|------------|-------------|
| MPI_CHAR   | signed char |
| MPI_INT    | signed int  |
| MPI_LONG   | signed long |
| MPI_FLOAT  | float       |
| MPI_DOUBLE | double      |
| MPI_BYTE   | N/A         |

**Table 9.2:** A selection of the standard MPI data types for C. Unsigned variants exist where applicable.

## Point-to-point communication

- One “sender” and one “receiver”: point-to-point communication
- Both the sender and receiver are identified by their ranks (within an MPI communicator)
- Each point-to-point message can carry an additional integer label, the so-called **tag**

## The simplest MPI send command

```
int MPI_Send(void *buffer, int count,  
            MPI_Datatype datatype,  
            int dest, int tag,  
            MPI_Comm comm);
```

This *blocking* send function returns when the data has been delivered to the system and **the message buffer can be safely modified**. The message may not have been received by the destination process.

## The simplest MPI receive command

```
int MPI_Recv(void *buffer, int count  
            MPI_Datatype datatype,  
            int source, int tag,  
            MPI_Comm comm,  
            MPI_Status *status);
```

- This *blocking* receive function waits until a matching message is received from the system so that the buffer contains the incoming message.
- Match of data type, source process (or MPI\_ANY\_SOURCE), message tag (or MPI\_ANY\_TAG).
- Receiving fewer datatype elements than count is ok, but receiving more elements is an error.

## MPI\_Status

The MPI\_Recv function has an additional output argument: the status object, which can be used to determine “unknown” parameters (if any).

For example, the source or tag of a received message may not be known if wildcard values were used in the receive function.

To query the information stored in a status object:

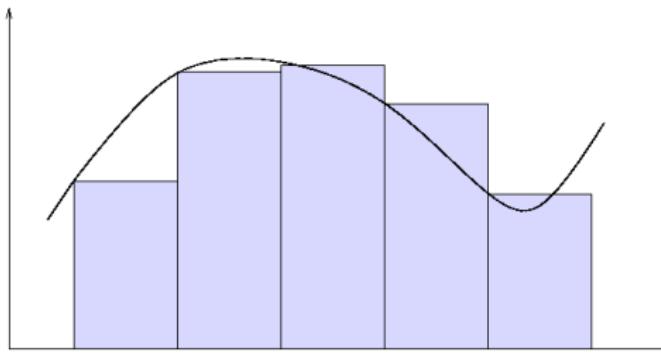
```
status.MPI_SOURCE
```

```
status.MPI_TAG
```

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

# Example of MPI parallelization; numerical integration

A quick recap of numerical integration (already discussed in Chapter 6):



[http://spiff.rit.edu/classes/phys317/lectures/num\\_integ2/num\\_integ2.html](http://spiff.rit.edu/classes/phys317/lectures/num_integ2/num_integ2.html)

The basic idea:

- Divide the integral domain into many small intervals
- Evaluate the function inside each small interval
- Sum up all the evaluated results (multiplied with the interval width)

## MPI parallelization; overall idea

- Each MPI process is assigned a “subdomain’ to work on
- Each MPI process then computes its subdomain result
- Thereafter, all the subdomain results need to be summed up
- One possible strategy is to let each process (except rank 0) send its subdomain result to process rank 0, which sums up all the subdomain results

# MPI implementation

```
int rank, size;
double a=0.0, b=1.0, mya, myb, psum;
MPI_Status status;

MPI_Init (&nargs, &args);
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

maya = a + rank*(b-a)/size;
myb = mya + (b-a)/size;
psum = integrate(mya,myb); // Each integrates over its "subdomain"

if (rank==0) {
    double res = psum;
    for (i=1; i<size; i++) {
        MPI_Recv(&psum,1,MPI_DOUBLE,i,0,MPI_COMM_WORLD,&status);
        res += psum;
    }
    printf("Result: %g\n", res);
}
else {
    MPI_Send(&psum,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
}
MPI_Finalize();
```

## Additional remarks

The preceding MPI example can be improved in several ways:

- Rank 0 receives in total `size-1` messages, one from each of the other processes. The order of receiving the messages is fixed, probably not the same as the incoming order of the messages. Using `MPI_ANY_SOURCE` (wildcard) instead of a prescribed sender rank is more appropriate.
- Rank 0 calls `MPI_Recv` after its own calculation (`integrate(mya,myb)`). If some of the other processes finish their computation earlier, communication cannot proceed, and it cannot be overlapped with computation on rank 0. (Non-blocking point-to-point communication will be more appropriate, see Section 9.2.4.)
- Rank 0 can easily become a “bottleneck”, because it is responsible for receiving all the partial results. (Use of collective communication, such as that specifically designed for “reduction”, will be more appropriate, see Section 9.2.3.)

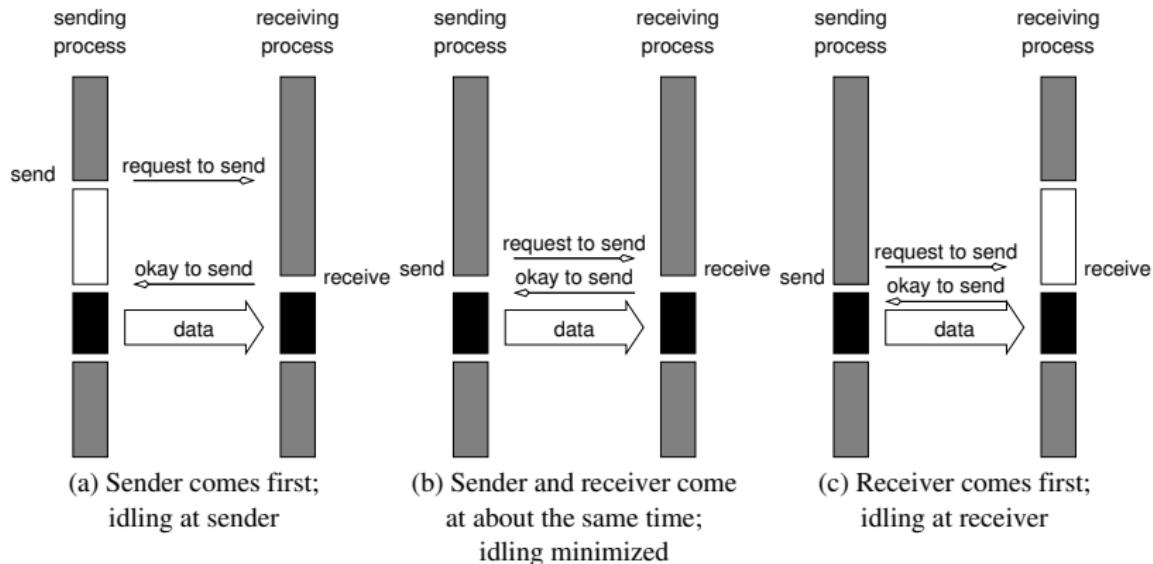
## “Uncertainties/risks” with MPI\_Send

While MPI\_Send is easy to use, one should be aware that the MPI standard allows for a considerable amount of freedom in its actual implementation.

- Internally it may work completely **synchronously**, meaning that the call can not return until a message transfer has at least started after a “handshake” with the receiver.
- However, it may also copy the message to an intermediate system buffer and return right away, leaving the “handshake” and data transmission to another mechanism, like a background thread.
- It may even change its behavior depending on any explicit or hidden parameters.

# Scenarios of “handshake”

## Non-System-Buffered Blocking Message Passing Operations



Source: [A. Grama, A. Gupta, G. Karypis, and V. Kumar. Introduction to Parallel Computing](#). Addison Wesley, 2003

# One scenario for deadlock

**Deadlocks** may occur if the possible synchronousness of MPI\_Send is not taken into account. A typical communication pattern where this may become crucial is a “ring shift”:

```
int rank, size, left, right,in_buf[N], out_buf[N];
MPI_Status status;

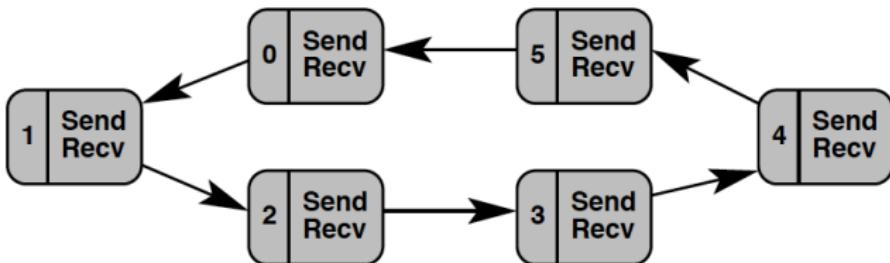
// .....

MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

right = rank==size-1 ? 0 : rank+1;
left = rank==0 ? size-1 : rank-1;

MPI_Send(out_buf,N,MPI_INT,right,0,MPI_COMM_WORLD);
MPI_Recv(in_buf,N,MPI_INT,left,0,MPI_COMM_WORLD,&status);
```

# Depiction of the “ring shift” pattern



**Figure 9.1:** A ring shift communication pattern. If sends and receives are performed in the order shown, a deadlock can occur because `MPI_Send()` may be synchronous.

- If `MPI_Send` is **synchronous** (and there is no system buffering), all processes call it first and then wait forever for a matching receive to be posted—deadlock.
- However, the ring shift **may** run without problems if the messages are sufficiently short. In fact, most MPI implementations provide a (small) internal buffer for short messages and switch to synchronous mode when the buffer is full or too small.

# One simple solution

```
int rank, size, left, right,in_buf[N], out_buf[N];
MPI_Status status;

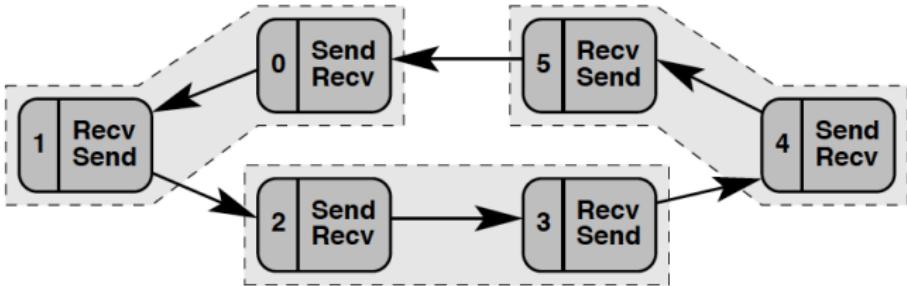
// ......

MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

right = rank==size-1 ? 0 : rank+1;
left = rank==0 ? size-1 : rank-1;

if (rank%2) {
    MPI_Recv(in_buf,N,MPI_INT,left,0,MPI_COMM_WORLD,&status);
    MPI_Send(out_buf,N,MPI_INT,right,0,MPI_COMM_WORLD);
}
else {
    MPI_Send(out_buf,N,MPI_INT,right,0,MPI_COMM_WORLD);
    MPI_Recv(in_buf,N,MPI_INT,left,0,MPI_COMM_WORLD,&status);
}
```

# Depiction of the simple solution



**Figure 9.2:** A possible solution for the deadlock problem with the ring shift: By changing the order of MPI\_Send() and MPI\_Recv() on all odd-numbered ranks, pairs of processes can communicate without deadlocks because there is now a matching receive for every send operation (dashed boxes).

## Special MPI functions for “ring shifts”

```
int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 int dest, int sendtag,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int source, int recvtag,
                 MPI_Comm comm, MPI_Status *status)
```

```
int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype,
                        int dest, int sendtag, int source, int recvtag,
                        MPI_Comm comm, MPI_Status *status)
```

Both routines use blocking communication, are guaranteed to not be subject to the deadlock effects that may occur with separate send and receive.

# Collective communication in MPI

Recall the numerical integration example: summing up the partial sums is a *reduction* operation.

MPI has mechanisms that make reductions much simpler and in most cases more efficient than looping over all ranks and collecting results.

Since a reduction is a procedure involves all ranks in a communicator, it belongs to the so-called **collective** communication operations in MPI.

**A collective MPI routine must be called by all ranks in a communicator.**

## MPI\_Barrier

MPI\_Barrier is the simplest collective in MPI, it does not actually perform any real data transfer.

```
int MPI_Barrier(MPI_Comm comm)
```

The barrier *synchronizes* the members of the communicator, i.e., all processes must call it before they are allowed to return to the user code.

**Don't over-use MPI\_Barrier!** There are other MPI routines that allow for implicit or explicit synchronization with finer control.

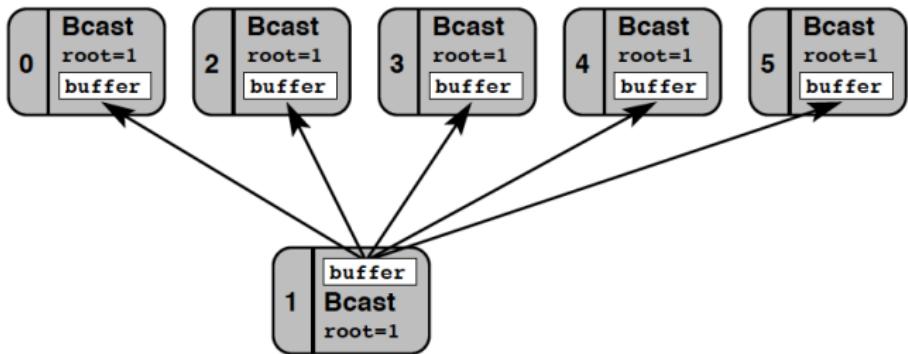
## MPI\_Bcast

```
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root,  
               MPI_Comm comm )
```

**MPI\_Bcast** sends a message from one process (the “root”) to all others in the communicator.

The **buffer** argument to **MPI\_Bcast()** is a send buffer on the root and a receive buffer on any other process.

# Depiction of MPI\_Bcast



**Figure 9.3:** An MPI broadcast: The “root” process (rank 1 in this example) sends the same message to all others. Every rank in the communicator must call `MPI_Bcast()` with the same `root` argument.

## MPI\_Gather & MPI\_Scatter

Examples of more advanced MPI collective calls that are concerned with global data distribution:

- `MPI_Gather()` collects the send buffer contents of all processes and concatenates them in rank order into the receive buffer of the root process
- `MPI_Scatter()` does the reverse, distributing equal-sized chunks of the root's send buffer
- `MPI_Gatherv()` and `MPI_Scatterv()` support arbitrary per-rank chunk sizes

## MPI\_Gather & MPI\_Scatter syntax

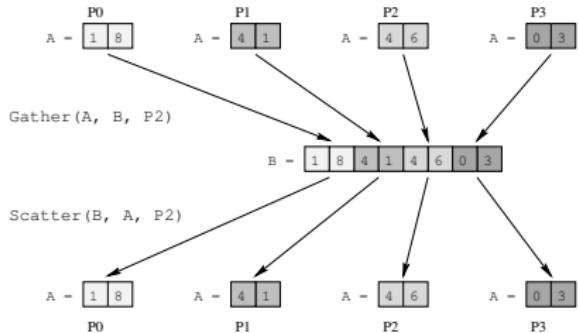
```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)

int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
                MPI_Comm comm)
```

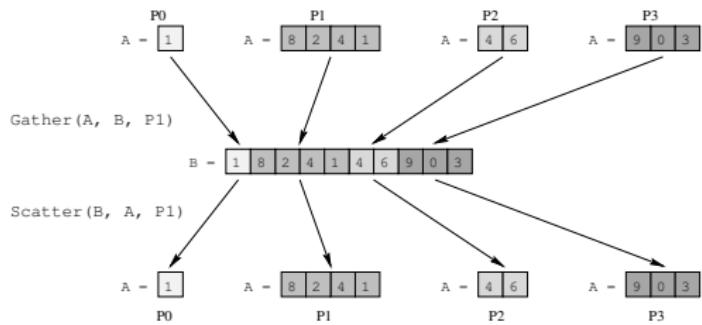
The syntax of MPI\_Gatherv() and MPI\_Scatterv() is more complex, not listed here.

# Depiction of MPI\_Gather & MPI\_Scatter

(a) Equal-Size Gather and Scatter Operations



(b) Unequal-Size Gather and Scatter Operations



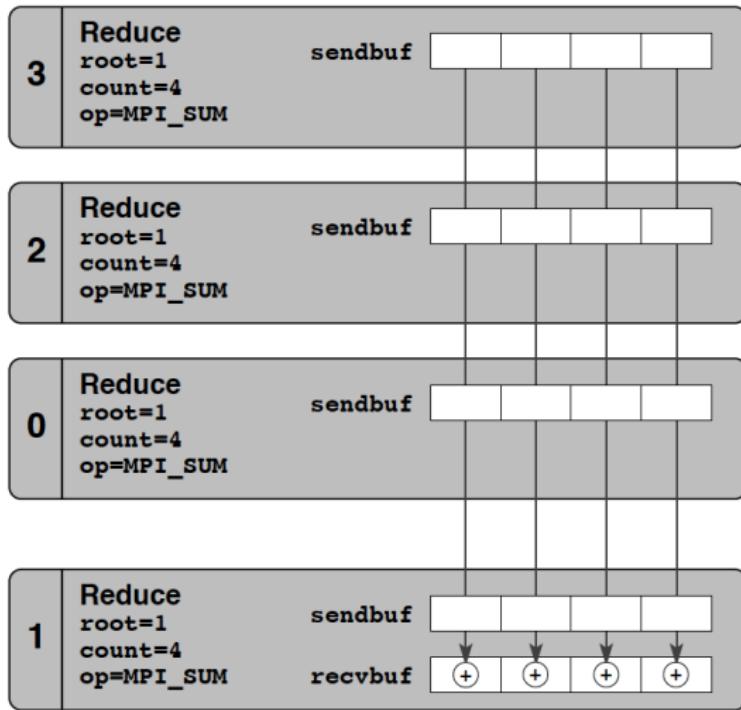
## MPI\_Reduce

```
int MPI_Reduce(const void *sendbuf, void *recvbuf,  
               int count, MPI_Datatype datatype,  
               MPI_Op op, int root, MPI_Comm comm)
```

MPI\_Reduce() combines the contents of the sendbuf array on all processes, element-wise, using an operator encoded by the op argument, and stores the result in recvbuf on root

Examples of predefined operators: MPI\_MAX, MPI\_MIN, MPI\_SUM and MPI\_PROD

# Depiction of MPI\_Reduce



**Figure 9.4:** A reduction on an array of length `count` (a sum in this example) is performed by `MPI_Reduce()`. Every process must provide a send buffer. The receive buffer argument is only used on the root process. The local copy on root can be prevented by specifying `MPI_IN_PLACE` instead of a send buffer address.

## Rewrite of the “numerical integration” example

```
int rank, size;
double a=0.0, b=1.0, mya, myb, psum, res=0.;
MPI_Status status;

MPI_Init (&nargs, &args);
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

mya = a + rank*(b-a)/size;
myb = mya + (b-a)/size;
psum = integrate(mya,myb); // Each integrates over its "subdomain"

MPI_Reduce(&psum,&res,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
if (rank==0)
    printf("Result: %g\n", res);

MPI_Finalize();
```

## MPI\_Allreduce

```
int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,  
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

The result of the reduction operation (`recvbuf`) is available on all MPI ranks.

## Nonblocking point-to-point communication

Point-to-point communication can be performed with nonblocking semantics.

- A nonblocking point-to-point call merely **initiates** a message transmission and returns very quickly to the user code.
- In an efficient implementation, waiting for data to arrive and the actual data transfer occur in the background, leaving resources free for computation.
- In other words, nonblocking MPI is a way in which communication may be overlapped with computation if implemented efficiently.
- The message buffer **must not be used** as long as the user program has not been notified that it is safe to do so (which can be checked by suitable MPI calls).
- Nonblocking and blocking MPI calls are mutually compatible, i.e., a message sent via a blocking send can be matched by a nonblocking receive.

## MPI\_Isend

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

- As opposed to the blocking send (`MPI_Send()`), `MPI_Isend()` has an additional output argument, the *request* handle (of type `struct MPI_Request`).
- It serves as an identifier by which the program can later refer to the “pending” communication request.

## MPI\_Irecv

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

- The status object known from MPI\_Recv() is **not** needed for MPI\_Irecv.

## MPI\_Test & MPI\_Wait

Checking a pending communication for completion can be done via the MPI\_Test() and MPI\_Wait() functions. The former only tests for completion and returns a flag, while the latter blocks until the buffer can be used.

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

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

The status object contains useful information only if the pending communication is a completed receive (i.e., in the case of MPI\_Test() the value of flag must be true).

## MPI\_Waitall

In the case of multiple non-blocking communication operations (multiple requests pending), it is more convenient to use the MPI\_Waitall function:

```
int MPI_Waitall(int count, MPI_Request array_of_requests[],
                MPI_Status array_of_statuses[])
```

## The “numerical integration” example, yet again

```
// ...
double *tmp;
MPI_Request *request_array;
MPI_Status *status_array;

if (rank==0) {
    // allocate arrays of tmp, request_array & status_array...
    for (i=1; i<size; i++)
        MPI_Irecv(&tmp[i-1],1,MPI_DOUBLE,i,0,MPI_COMM_WORLD,&request_array[i-1]);
}

mya = a + rank*(b-a)/size;
myb = mya + (b-a)/size;
psum = integrate(mya,myb); // Each integrates over its "subdomain"

if (rank==0) {
    double res = psum;
    MPI_Waitall (size-1,request_array,status_array);
    for (i=1; i<size; i++)
        res += tmp[i-1];
    printf("Result: %g\n", res);
}
else {
    MPI_Send(&psum,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
}
```

## Benefits of nonblocking communication

- Nonblocking communication provides an obvious way to overlap communication, i.e., overhead, with useful work.
- The possible performance advantage, however, depends on many factors, and may even be nonexistent.
- But even if there is no real overlap, multiple outstanding nonblocking requests may improve performance because the MPI library can decide which of them gets serviced first.
- Nonblocking communication helps to avoid deadlock.

# A short summary

|             | Point-to-point                                                                                                                                                                                                                                                                                       | Collective                                                                                                                                                                                                            |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Blocking    | <code>MPI_Send()</code><br><code>MPI_Ssend()</code><br><code>MPI_Bsend()</code><br><code>MPI_Recv()</code>                                                                                                                                                                                           | <code>MPI_Barrier()</code><br><code>MPI_Bcast()</code><br><code>MPI_Scatter() /</code><br><code>MPI_Gather()</code><br><code>MPI_Reduce()</code><br><code>MPI_Reduce_scatter()</code><br><code>MPI_Allreduce()</code> |
| Nonblocking | <code>MPI_Isend()</code><br><code>MPI_Irecv()</code><br><code>MPI_Wait() / MPI_Test()</code><br><code>MPI_Waitany() /</code><br><code>    MPI_Testany()</code><br><code>MPI_Waitsome() /</code><br><code>    MPI_Testsome()</code><br><code>MPI_Waitall() /</code><br><code>    MPI_Testall()</code> | N/A                                                                                                                                                                                                                   |

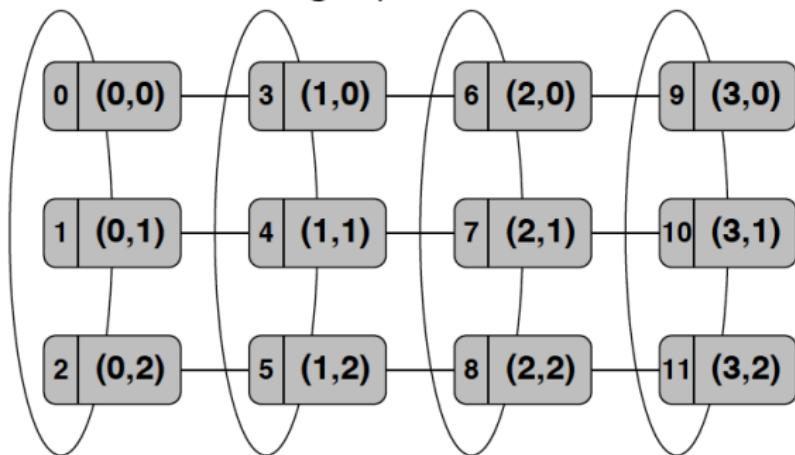
**Table 9.3:** MPI's communication modes and a nonexhaustive overview of the corresponding subroutines.

## Virtual topologies

- MPI suits very well for implementing domain decomposition (Section 5.2.1) on distributed-memory parallel computers.
- However, setting up the process grid and keeping track of which ranks have to exchange halo data is nontrivial.
- MPI contains some functionality to support this recurring task in the form of *virtual topologies*.
- These provide a convenient process naming scheme, which fits the required communication pattern.

## Cartesian topologies

Example: a 2D global Cartesian mesh of size  $3000 \times 4000$ . Suppose we want to use  $3 \times 4 = 12$  MPI processes to divide the global mesh, each holding a piece of  $1000 \times 1000$ .



**Figure 9.6:** Two-dimensional Cartesian topology: 12 processes form a  $3 \times 4$  grid, which is periodic in the second dimension but not in the first. The mapping between MPI ranks and Cartesian coordinates is shown.

## Cartesian topologies (2)

- As shown in the preceding figure, each process can either be identified by its rank or its Cartesian coordinates.
- Each process has a number of neighbors, which depends on the grid's dimensionality. (In our example, the number of dimensions is two, which leads to at most four neighbors per process.)
- MPI can help with establishing the mapping between ranks and Cartesian coordinates in the process grid.

## MPI\_Cart\_create

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],  
                    const int periods[], int reorder, MPI_Comm * comm_cart)
```

- A new, “Cartesian” communicator `comm_cart` is generated, which can be used later to refer to the topology.
- The `periods` array specifies which Cartesian directions are periodic, and the `reorder` parameter allows, if true, for rank reordering so that the rank of a process in communicators `comm_old` and `comm_cart` may differ.
- Here, MPI merely keeps track of the topology information.

## MPI\_Cart\_coords & MPI\_Cart\_rank

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
```

```
int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)
```

- These are two “service” functions responsible for the translation between Cartesian process coordinates and an MPI rank.
- `MPI_Cart_coords()` calculates the Cartesian coordinates for a given rank.
- The reverse mapping, i.e., from Cartesian coordinates to an MPI rank, is performed by `MPI_Cart_rank()`.

# Example of 2D Cartesian grid

```
int rank, size;
MPI_Comm comm;
int dim[2], period[2], reorder;
int coord[2], id;
// .....

dim[0]=4; dim[1]=3;
period[0]=0; period[1]=1;
reorder=1;

MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, reorder, &comm);

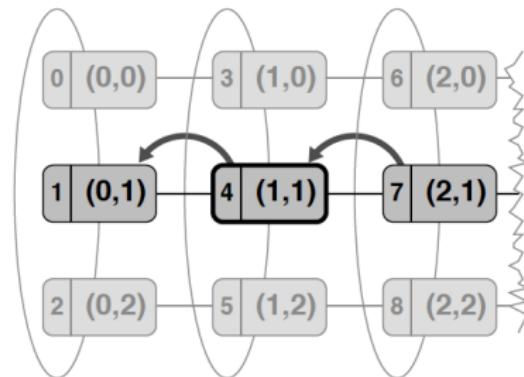
if (rank == 5) {
    MPI_Cart_coords(comm, rank, 2, coord);
    printf("Rank %d coordinates are %d %d\n", rank, coord[0], coord[1]);
}

if(rank==0) {
    coord[0]=3; coord[1]=1;
    MPI_Cart_rank(comm, coord, &id);
    printf("The processor at position (%d, %d) has rank %d\n", coord[0], coord[1], id);
}
```

# MPI\_Cart\_shift

A regular task with domain decomposition is to find out who the next neighbors of a certain process are along a certain Cartesian dimension.

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int *rank_source,  
                   int *rank_dest)
```



**Figure 9.7:** Example for the result of `MPI_Cart_shift()` on a part of the Cartesian topology from Figure 9.6. Executed by rank 4 with `direction=0` and `disp=-1`, the function returns `rank_source=7` and `rank_dest=1`.

# IN3200/IN4200: Chapter 10

## Efficient MPI programming

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

# Objectives

- Understand some of the performance characteristics of MPI communication
- Learn some guidelines for efficient MPI programming

# How costly is a point-to-point message?

We learned in Chapter 4:

$$T = T_\ell + \frac{M}{B}$$

- $M$ : size of the message [bytes]
- $T_\ell$ : latency
- $B$ : maximum network point-to-point bandwidth [bytes/sec]
- For simplicity,  $T_\ell$  and  $B$  were considered as constants in Chapter 4, but they may not be in reality....

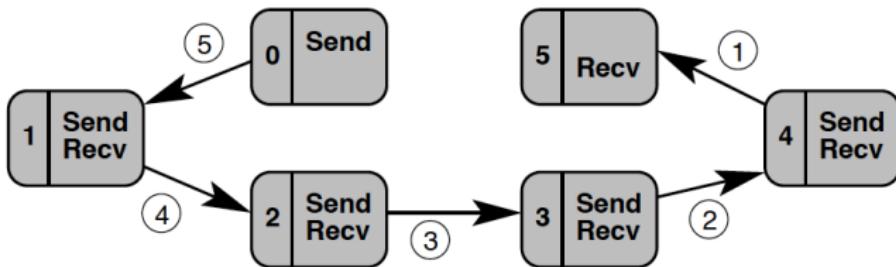
## Two different protocols

Most MPI implementations switch between different variants, depending on the message size and other factors:

- **Eager protocol:** For short messages, the entire message may be sent and stored at the receiver side in some preallocated buffer space, without the receiver's cooperation.
  - Advantage: synchronization overhead (due to handshaking) is reduced.
  - Disadvantage: need a large amount of preallocated buffer space, too many eager messages may overflow those buffers and lead to contention or program crashes.
- **Rendezvous protocol:** For large messages, the message "envelope" is immediately delivered, but the actual message transfer blocks until the user's receive buffer is available.
  - Advantage: Extra data copies could be avoided, improving effective bandwidth.
  - Disadvantage: sender and receiver must synchronize.

**Performance problems in form of  
synchronization, serialization, contention**

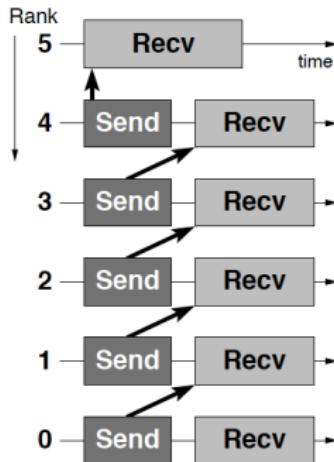
## Case study: linear shift communication pattern



**Figure 10.4:** A linear shift communication pattern. Even with synchronous point-to-point communication, a deadlock will not occur, but all message transfers will be serialized in the order shown.

- Slightly different from the “ring shift” communication pattern
- “Linear shift” will not suffer from deadlock, but the performance depends on several factors

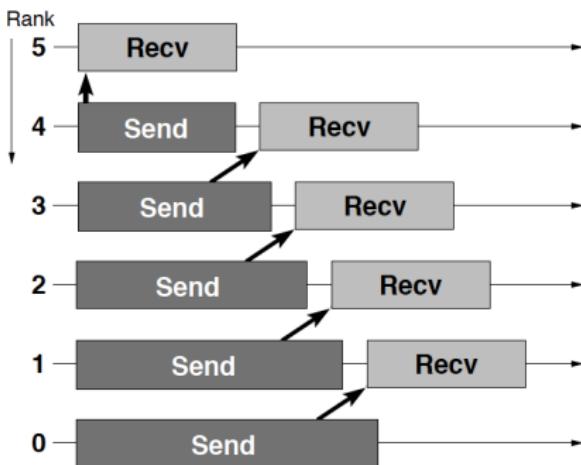
# Linear shift: scenario 1



**Figure 10.5:** Timeline view of the linear shift (see Figure 10.4) with blocking (but not synchronous) sends and blocking receives, using eager delivery. Message transmissions can overlap, making use of a nonblocking network. Eager delivery allows a send to end before the corresponding receive is posted.

- Assuming MPI\_Send() is not synchronous and “eager delivery”;
- Message transfers can overlap if the network is nonblocking;
- All send operations terminate early, most of the time is spent receiving data.

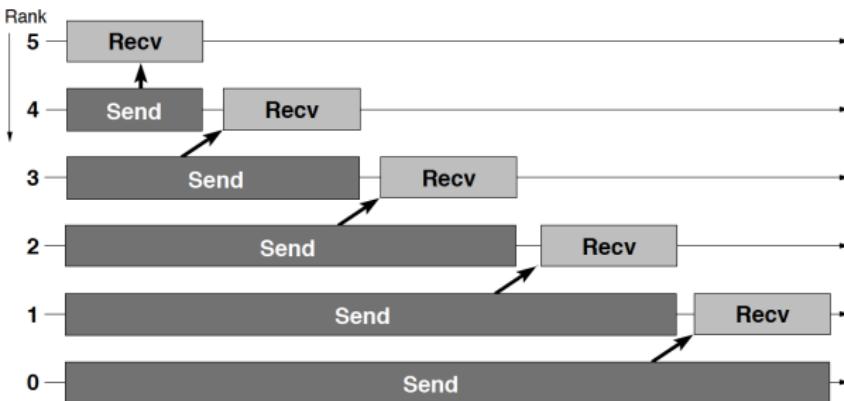
## Linear shift: scenario 2



**Figure 10.6:** Timeline view of the linear shift (see Figure 10.4) with blocking synchronous sends and blocking receives, using eager delivery. The message transfers (arrows) might overlap perfectly, but a send can only finish just after its matching receive is posted.

- The length of the message is large such that `MPI_Send()` is actually executed as `MPI_Ssend()`;
- `MPI_Ssend()` does not return to the user code before a matching receive is posted;
- Message delivery still follows the eager protocol;
- Some performance penalty compared with scenario 1.

## Linear shift: scenario 3



**Figure 10.7:** Timeline view of the linear shift (see Figure 10.4) with blocking sends and blocking receives, using the rendezvous protocol. The messages (arrows) are transmitted in serial because buffering is ruled out.

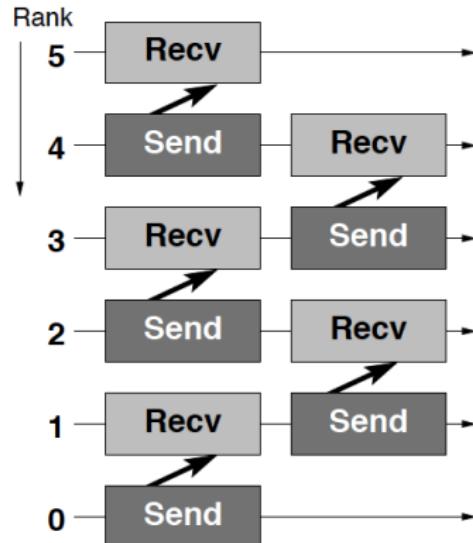
- Messages are transmitted according to the rendezvous protocol;
- Sender and receiver must synchronize;
- For the linear shift pattern, messages will be transmitted in serial, one after the other → **implicit serialization**

## Avoiding implicit serialization

Implicit serialization should be avoided because it is not only a source of additional communication overhead but can also lead to load imbalance.

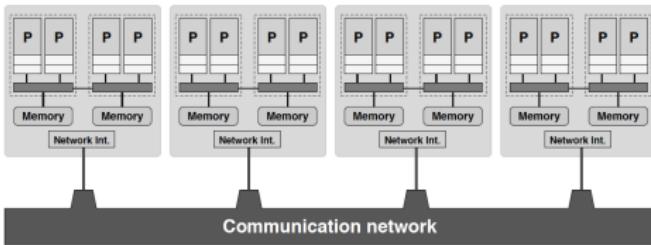
- Change the order of sends and receives on, e.g., all odd-numbered processes;
- Use nonblocking functions, which have the additional benefit that multiple outstanding communications can be handled by the MPI library in a (hopefully) optimal order. Moreover they provide an opportunity for *asynchronous* communication, where auxiliary threads and/or hardware mechanisms transfer data even while a process is executing user code.
- Use blocking point-to-point functions that are guaranteed not to deadlock, such as `MPI_Sendrecv()` or `MPI_Sendrecv_replace()`.

## Example of changing the order of sends/receives



**Figure 10.8:** Even if sends are synchronous and the rendezvous protocol is used, exchanging the order of sends and receives on all odd-numbered ranks exposes some parallelism in communication.

# Network contention



**Figure 4.8:** Typical hybrid system with shared-memory nodes (ccNUMA type). Two-socket building blocks represent the price vs. performance “sweet spot” and are thus found in many commodity clusters.

On a typical hybrid (hierarchical) parallel computer:

- Multiple threads or processes on a node may issue communication requests to other nodes. The available bandwidth per connection will typically go down;
- The network topology may not be fully nonblocking, i.e., the bisection bandwidth may be lower than the product of the number of nodes and the single-connection bandwidth;
- Even if bisection bandwidth is optimal, static routing can lead to contention for certain communication patterns.

## More on network contention

In general, network contention of some kind is hard to avoid.

- Some communication patterns are especially prone to causing contention, like all-to-all message transmission where every process sends to every other process; MPI's `MPI_Alltoall()` function is a special form of this.
- Any optimization that reduces communication overhead and message transfer volume will most probably also reduce contention.
- Even if there is no way to lessen the amount of message data, it may be possible to rearrange communication calls so that contention is minimized.

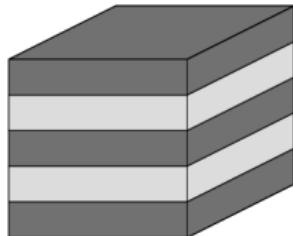
## **Reducing communication overhead**

# Domain decomposition

Domain decomposition is one of the most important implementations of data parallelism:

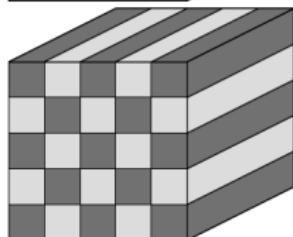
- We want to understand the impact of domain decomposition on communication overhead.
- We want to avoid “wrong” decompositions.

## Example: decomposing a 3D domain



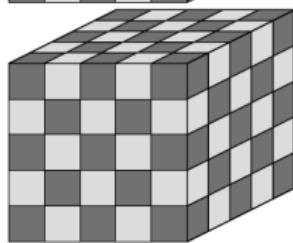
“Slabs”

$$c_{1d}(L, N) = L \cdot L \cdot w \cdot 2 \\ = 2wL^2$$



“Poles”

$$c_{2d}(L, N) = L \cdot \frac{L}{\sqrt{N}} \cdot w \cdot (2 + 2) \\ = 4wL^2N^{-1/2}$$



“Cubes”

$$c_{3d}(L, N) = \frac{L}{\sqrt[3]{N}} \cdot \frac{L}{\sqrt[3]{N}} \cdot w \cdot (2 + 2 + 2) \\ = 6wL^2N^{-2/3}$$

**Figure 10.9:** 3D domain decomposition of a cubic domain of size  $L^3$  (strong scaling) and periodic boundary conditions: Per-process communication volume  $c(L, N)$  for a single-site data volume  $w$  (in bytes) on  $N$  processes when cutting in one (top), two (middle), or all three (bottom) dimensions.

## The amount of communication depends on decomposition

The cubic domain is of size  $L^3$ , which has periodic boundary conditions, is decomposed into  $N$  subdomains:

- The domain cuts can be performed in one, two, or all three dimensions;
- The number of elements that must be communicated by one process with its neighbors,  $c(L, N)$ , can be found as simple formulas;

## Decomposing as “slabs”

- 1D cuts

$$c_{1d}(L, N) = L \cdot L \cdot w \cdot 2 = 2wL^2$$

- Should not be used in general because they incur a much larger and, more importantly,  $N$ -independent overhead (doesn't decrease when  $N$  grows).
- A constant cost of communication per subdomain will greatly harm strong scalability, because performance saturates at a lower level, determined by the message size (i.e., the slab area) instead of the latency.

## Decomposing as “poles”

- 2D cuts

$$c_{2d}(L, N) = L \cdot \frac{L}{\sqrt{N}} \cdot w \cdot (2 + 2) = 4w \frac{L^2}{\sqrt{N}}$$

- The volume of communication per process decreases with increasing  $N$ ;
- However, the surface-to-volume ratio will grow with  $N$ .

## Decomposing as “cubes”

- 3D cuts

$$c_{3d}(L, N) = \frac{L}{\sqrt[3]{N}} \cdot \frac{L}{\sqrt[3]{N}} \cdot w \cdot (2 + 2 + 2) = 6w \frac{L^2}{\left(\sqrt[3]{N}\right)^2}$$

- Smaller communication volume per process than 2D decomposition (as long as  $\sqrt[6]{N} > \frac{3}{2}$ );
- Still has the same problems as with 2D decomposition;
- If latency dominates the overhead, “optimal” 3D decomposition may even be counterproductive because of the six neighbors for each domain.
- **Another factor for consideration:** cost of message packing & unpacking

## Mapping issues

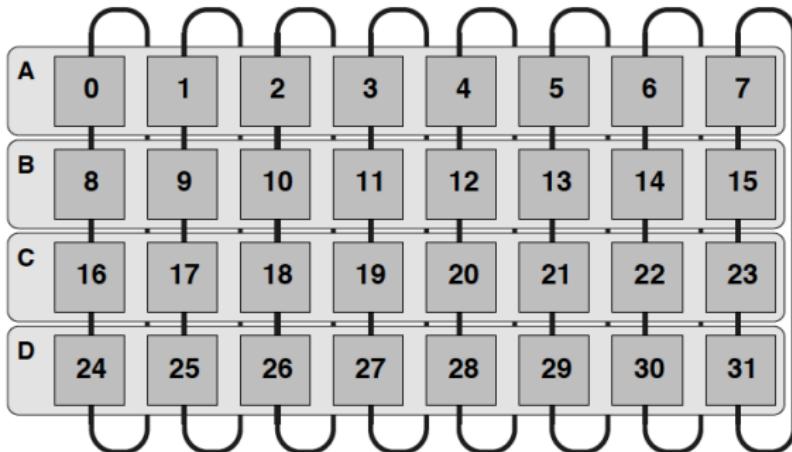
- Modern parallel computers are inevitably of the hierarchical type. They all consist of shared-memory multiprocessor “nodes” coupled via some network;
- The simplest way to use this kind of hardware is to run one MPI process per core;
- Assuming that any point-to-point MPI communication between two cores located on the same node is much faster than between cores on different nodes, the **mapping of computational subdomains to cores** has a large impact on communication overhead;
- `MPI_Cart_create()` with rank reordering may help, but the interconnect topology may **not** be available for `MPI_Cart_create()`.

# An example of 2D decomposition for a 2D domain

Specific case:  $N = 32$  MPI processes that form a virtual  $4 \times 8$  Cartesian grid.

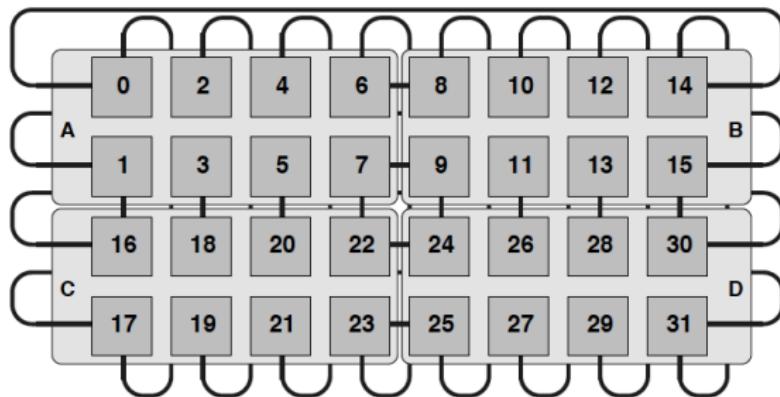
Hardware: 4 nodes each with 8 CPU cores.

**Figure 10.10:** A typical default mapping of MPI ranks (numbers) to sub-domains (squares) and cluster nodes (letters) for a two-dimensional  $4 \times 8$  periodic domain decomposition. Each node has 16 connections to other nodes. Intranode connections are omitted.



- A straightforward mapping will lead to 16 inter-node “connections” per node.

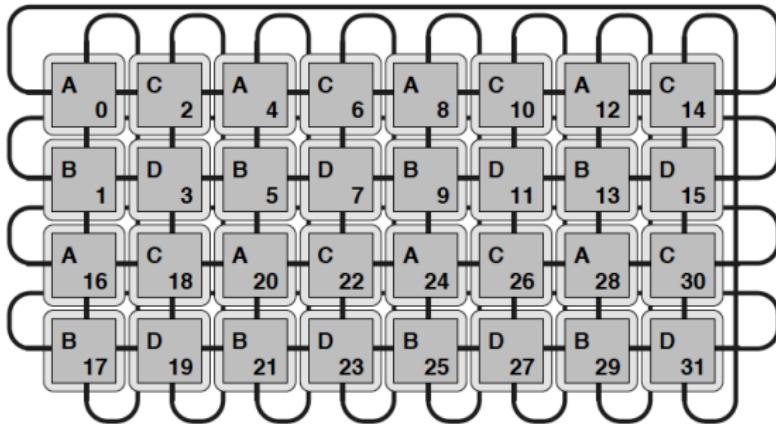
If a different numbering of the subdomains is chosen



**Figure 10.11:** A “perfect” mapping of MPI ranks and subdomains to nodes. Each node has 12 connections to other nodes.

- Now, the number of inter-node “connections” is reduced to 12 per node;
- A 25% reduction in inter-node communication volume.

## A bad mapping example



**Figure 10.12:** The same rank-to-subdomain mapping as in Figure 10.11, but with ranks assigned to nodes in a “round-robin” way, i.e., successive ranks run on different nodes. This leads to 32 internode connections per node.

- In the previous two cases, we have assumed that successive MPI ranks are mapped to the same node.
- In case the MPI ranks are mapped to the nodes in a *round-robin* distribution, then the inter-node connections per node can be very bad: **32**

## Aggregating messages

- If a parallel algorithm requires transmission of a lot of small messages between processes, communication becomes latency-bound because each message incurs latency.
- Hence, small messages should be aggregated into contiguous buffers and sent in larger chunks so that the “latency penalty” must only be paid once, and effective communication bandwidth is good.
- This advantage pertains to point-to-point and collective communication alike.

## Things to consider...

Aggregation will only pay off if the additional time for copying the messages to a contiguous buffer does not outweigh the “latency penalty” for separate sends:

$$(m - 1)T_\ell > \frac{mL}{B_c}$$

- $m$  is the number of messages
- $L$  is the message length
- $B_c$  is the bandwidth for memory-to-memory copies
- For simplicity we assume that all messages have the same length, and that latency for memory copies is negligible.

# The actual advantage of message aggregation

The actual advantage depends on the raw network bandwidth  $B_n$  as well.

- $T_s$ : “serialized” communication time

$$T_s = m \left( T_\ell + \frac{L}{B_n} \right)$$

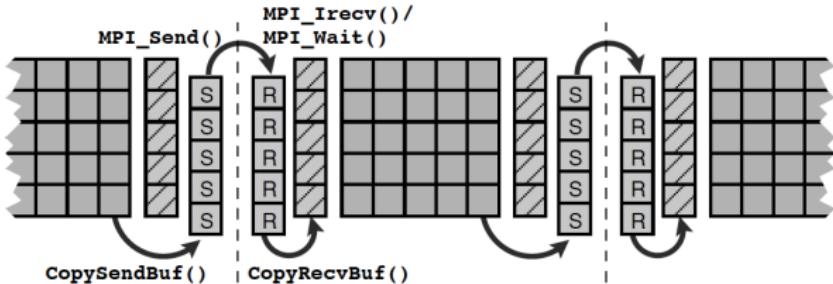
- $T_a$ : “aggregated” communicatin time

$$T_a = \frac{mL}{B_c} + T_\ell + \frac{mL}{B_n}$$

- On a slow network, where  $B_n$  is very low, the  $\frac{T_s}{T_a}$  ratio will be close one, that is, aggregation will not be beneficial.

# Need for message aggregation

- A typical case for message aggregation comes up when separate, i.e., noncontiguous data items must be transferred between processes.
- Example:  $x$ -direction communication related to 2D domain decomposition—need to send and receive “columns”.



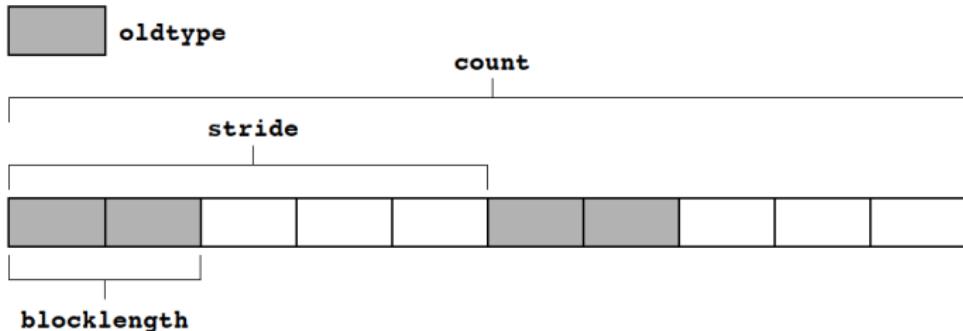
**Figure 9.9:** Halo communication for the Jacobi solver (illustrated in two dimensions here) along one of the coordinate directions. Hatched cells are ghost layers, and cells labeled “R” (“S”) belong to the intermediate receive (send) buffer. The latter is being reused for all other directions. Note that halos are always provided for the grid that gets read (not written) in the upcoming sweep. Fixed boundary cells are omitted for clarity.

## MPI's derived data types

- MPI's derived data types can help to simplify message packing/unpacking.
- The programmer can introduce new datatypes beyond the built-in ones, and use them in communication calls.
- There is a variety of choices for defining new types.
- The new type must first be defined using `MPI_Type_XXXXX()`, where “XXXXX” designates one of the variants.

## MPI\_Type\_vector

```
int MPI_Type_vector(  
    int count,  
    int blocklength,  
    int stride,  
    MPI_Datatype old_type,  
    MPI_Datatype *newtype_p  
) ;
```



**Figure 10.13:** Required parameters for `MPI_Type_vector`. Here `blocklength=2`, `stride=5`, and `count=2`.

## Example of using MPI\_Type\_vector

x-direction communication for 2D domain decomposition, each process with arrays of dimension  $\text{YMAX} \times \text{XMAX}$  (including the ghost layer)

```
MPI_Datatype new_type;
MPI_Type_vector(YMAX-2,1,XMAX,MPI_DOUBLE,&new_type);
MPI_Type_commit(&new_type);

....  

MPI_Send(&(array[1][1]),1,new_type,left_dest,tag0,MPI_COMM_WORLD);
MPI_Send(&(array[1][XMAX-2]),1,new_type,right_dest,tag1,MPI_COMM_WORLD);

....  

MPI_Type_free(&new_type);
```

# Asynchronous communication

- A further chance for increasing efficiency of parallel programs is overlapping communication and computation.
- Nonblocking point-to-point communication is the straightforward way to achieve this.
- Example of 2D domain decomposition for Jacobi computation:
  - ① Compute only the rows `phi_new[1] [...], phi_new[YMAX-2] [...]` and columns `phi_new[...] [1], phi_new[...] [XMAX-2]`
  - ② Initiate sending/receiving of all the messages by `MPI_Isend` and `MPI_Irecv`
  - ③ Compute the “interior” of the interior points (hopefully MPI messages are being transferred at the same time)
  - ④ Finish all MPI calls by `MPI_Wait`

## Difference between nonblocking and asynchronous communication

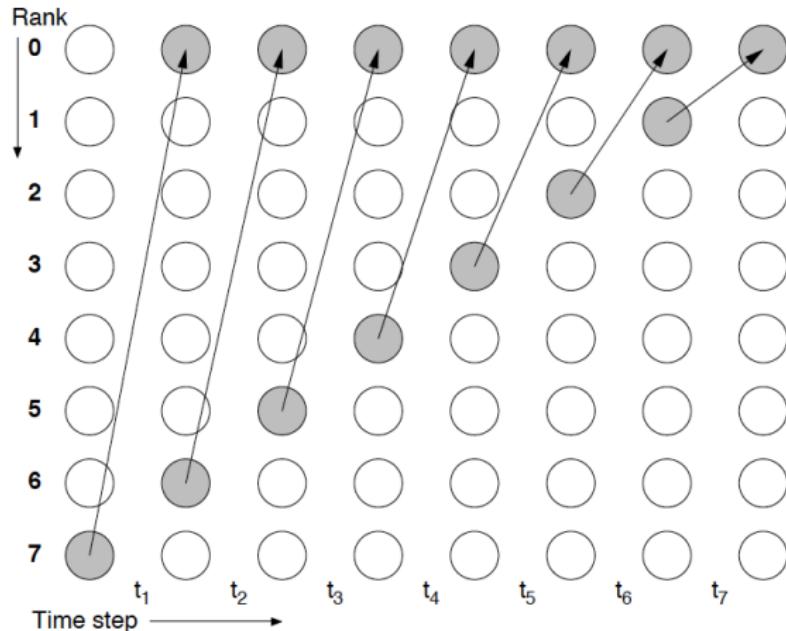
**Very important:** One must strictly differentiate between non-blocking and truly asynchronous communication. Nonblocking semantics, according to the MPI standard, merely implies that the message buffer cannot be used after the call has returned from the MPI library; **In other words**, it is entirely up to the implementation whether data transfer, i.e., MPI progress, takes place while user code is being executed outside MPI.

# Revisit of a naive implementation of numerical integration

Listing 9.3: Program fragment for parallel integration in MPI.

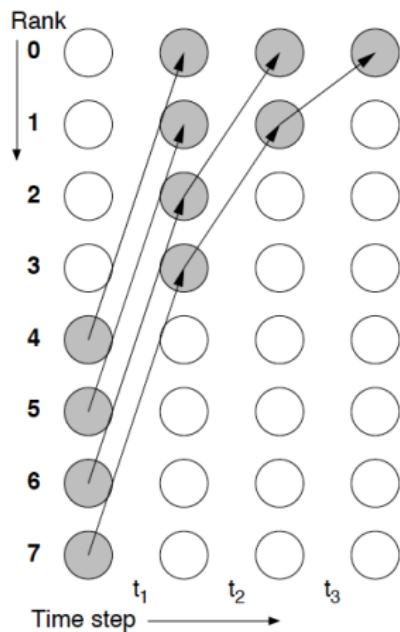
```
1 integer, dimension(MPI_STATUS_SIZE) :: status
2 call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
3 call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
4
5 ! integration limits
6 a=0.d0 ; b=2.d0 ; res=0.d0
7
8 ! limits for "me"
9 mya=a+rank*(b-a)/size
10 myb=mya+(b-a)/size
11
12 ! integrate f(x) over my own chunk - actual work
13 psum = integrate(mya,myb)
14
15 ! rank 0 collects partial results
16 if(rank.eq.0) then
17     res=psum
18     do i=1,size-1
19         call MPI_Recv(tmp, &      ! receive buffer
20                      1,       &      ! array length
21                      ! data type
22                      MPI_DOUBLE_PRECISION,&
23                      i,       &      ! rank of source
24                      0,       &      ! tag (unused here)
25                      MPI_COMM_WORLD,& ! communicator
26                      status,& ! status array (msg info)
27                      ierror)
28         res=res+tmp
29     enddo
30     write(*,*) 'Result: ',res
31 ! ranks != 0 send their results to rank 0
32 else
33     call MPI_Send(psum,      &    ! send buffer
34                   1,       &    ! message length
35                   MPI_DOUBLE_PRECISION,&
36                   0,       &    ! rank of destination
37                   0,       &    ! tag (unused here)
38                   MPI_COMM_WORLD,ierror)
39 endif
```

## Beware of single-process becoming the bottleneck



**Figure 10.15:** A global reduction with communication overhead being linear in the number of processes, as implemented in the integration example (Listing 9.3). Each arrow represents a message to the receiver process. Processes that communicate during a time step are shaded.

# Collective communication bears optimization potential



**Figure 10.16:** With a tree-like, hierarchical reduction pattern, communication overhead is logarithmic in the number of processes because communication during each time step is concurrent.

# IN3200/IN4200: Chapter 11: Hybrid parallelization with MPI and OpenMP

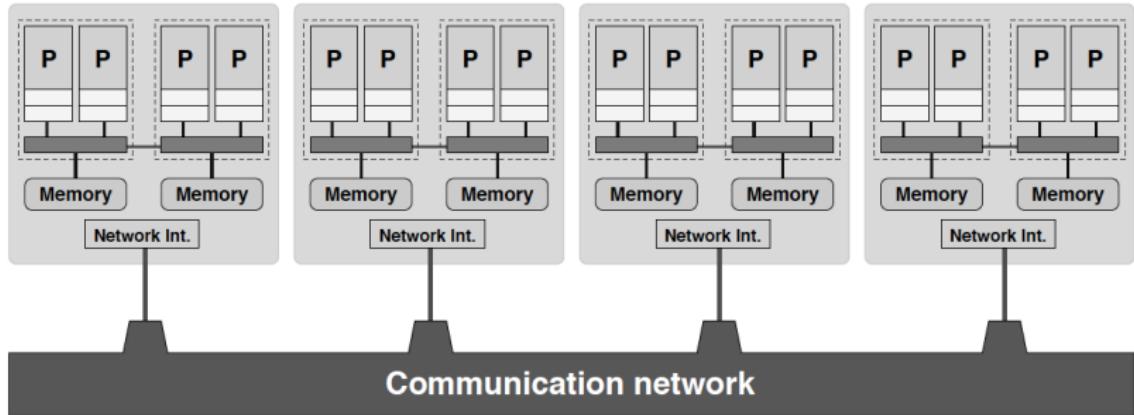
Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

# Objectives

- Basic guidelines for writing and running hybrid MPI/OpenMP code
  - Expected weaknesses and advantages of hybrid MPI/OpenMP programming
  - Performance of hybrid codes depends crucially on the association of threads and processes to core
- How OpenMP threads and MPI processes can interact within a shared-memory node

# Hardware motivation for hybrid MPI/OpenMP programming

A typical hybrid (hierarchical) parallel computer



**Figure 4.8:** Typical hybrid system with shared-memory nodes (ccNUMA type). Two-socket building blocks represent the price vs. performance “sweet spot” and are thus found in many commodity clusters.

# Software motivation for hybrid MPI/OpenMP programming

MPI and OpenMP have their advantages and disadvantages:

- MPI must be used on distributed memory clusters.
- When MPI was designed, however, clusters had nodes with only one or two cores. Today's CPUs can have 20+ cores, but the amount of memory per CPU core is rather limited.
- Using one MPI process per CPU core can be a waste of resources because each process needs its communication buffers, OS resources, etc.
- Managing hundreds of thousands (or millions) of MPI processes can be very hard.
- OpenMP is limited to shared memory parallelism, typically within a node of a cluster.
- Many pure OpenMP programs don't scale past some tens of threads due to threads overhead and runtime overhead.

## Applications that can potentially benefit from hybrid MPI/OpenMP programming

- Codes having limited MPI scalability (due to, e.g., MPI\_Alltoall)
- Codes limited by memory size while having a large amount of replicated data in each MPI process
- Codes suffering from inefficient local MPI implementation for intra-node communications
- Codes limited by the scalability of their algorithms (number of MPI processes)

# Basic MPI/OpenMP programming models

**The basic idea:** to allow any MPI process to spawn a team of OpenMP threads.

One or very few MPI processes per node; OpenMP is used to further exploit the parallelism within the node.

- Compute intensive loop constructs are the primary targets for OpenMP parallelization.
- Two basic hybrid programming approaches: *vector mode* & *task mode*.
  - These differ in the degree of interaction between MPI calls and OpenMP directives.

## Vector mode

**Vector mode:** all MPI functions are called outside OpenMP parallel regions.

- Major advantage: ease of programming and a “safe” approach (MPI is unaffected by multi-threading)
  - An existing pure MPI code can be turned hybrid just by adding OpenMP worksharing directives in front of the time-consuming loops
- Disadvantage: a rather “rigid” approach; limited possibility for hiding MPI overhead

# First example of hybrid MPI/OpenMP programming

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

int main(int argc, char *argv[]) {
    int numprocs, rank, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int iam = 0, np = 1;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(processor_name, &namelen);

    #pragma omp parallel default(shared) private(iam, np)
    {
        np = omp_get_num_threads();
        iam = omp_get_thread_num();
        printf("Hello from thread %d out of %d from process %d out of %d on %s\n",
               iam, np, rank, numprocs, processor_name);
    }

    MPI_Finalize();
}
```

# Hybrid MPI/OpenMP implementation of 3D Jacobi

## Pseudocode in Fortran:

**Listing 11.1:** Pseudocode for a vector mode hybrid implementation of a 3D Jacobi solver.

```
1   do iteration=1,MAXITER
2     ...
3   !$OMP PARALLEL DO PRIVATE(..)
4     do k = 1,N
5     ! Standard 3D Jacobi iteration here
6     ! updating all cells
7     ...
8   enddo
9   !$OMP END PARALLEL DO
10
11 ! halo exchange
12   ...
13   do dir=i,j,k
14
15     call MPI_Irecv( halo data from neighbor in -dir direction )
16     call MPI_Isend( data to neighbor in +dir direction )
17
18     call MPI_Irecv( halo data from neighbor in +dir direction )
19     call MPI_Isend( data to neighbor in -dir direction )
20   enddo
21   call MPI_Waitall( )
22 enddo
```

## Task mode

**Task mode** allows handling MPI communication within OpenMP parallel regions

- Advantage: a high level of flexibility.
- Functional task decomposition and decoupling of communication and computation are two areas where task mode can be useful.
- Disadvantage: larger code size and increased coding complexity
  - The convenient OpenMP worksharing parallelization directives can no longer be used. Workload has to be distributed “manually” among the threads.

## Another implementation of 3D Jacobi

- Purpose: **To overlap communication with computation**
- On each MPI process, the master thread is responsible for updating the “boundary” cells (that are immediately adjacent to the ghost cells)
  - The master thread is also responsible for the MPI communication
- On each MPI process, at the same time, the other threads are responsible for the remaining computation (inner cells)

# Pseudocode of 3D Jacobi in Fortran (part 1)

**Listing 11.2:** Pseudocode for a task mode hybrid implementation of a 3D Jacobi solver.

```
1 !$OMP PARALLEL PRIVATE(iteration,threadID,k,j,i,...)
2     threadID = omp_get_thread_num()
3     do iteration=1,MAXITER
4     ...
5     if(threadID .eq. 0) then
6     ...
7     ! Standard 3D Jacobi iteration
8     ! updating BOUNDARY cells
9     ...
10    ! After updating BOUNDARY cells
11    ! do halo exchange
12
13    do dir=i,j,k
14        call MPI_Irecv( halo data from neighbor in -dir direction )
15        call MPI_Send( data to neighbor in +dir direction )
16        call MPI_Irecv( halo data from neighbor in +dir direction )
17        call MPI_Send( data to neighbor in -dir direction )
18    enddo
19
20    call MPI_Waitall( )
21
22 else ! not thread ID 0
```

## Pseudocode of 3D Jacobi in Fortran (part 2)

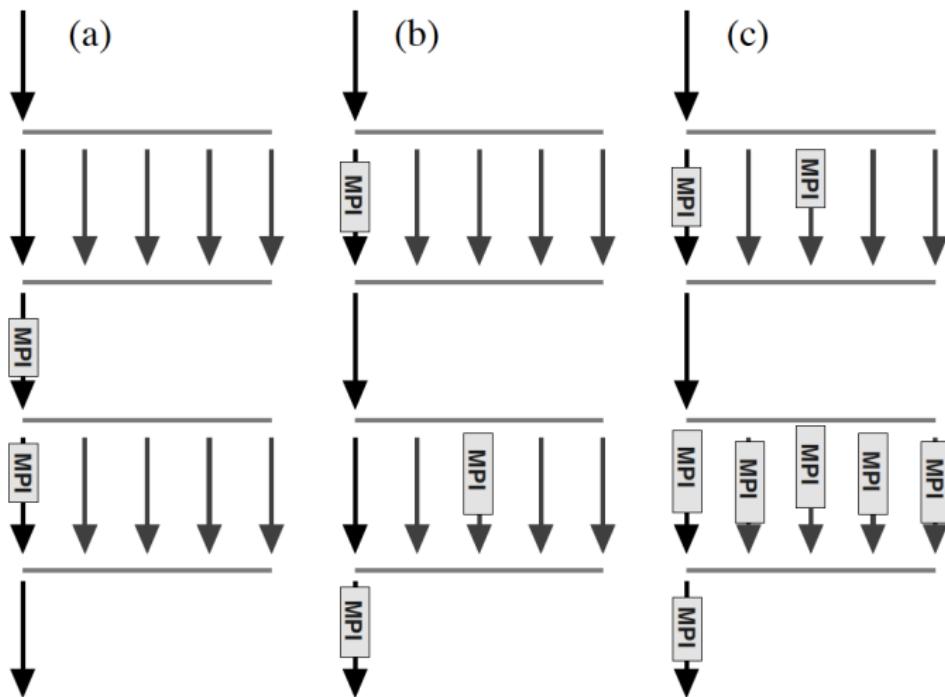
```
22     else ! not thread ID 0
23
24 ! Remaining threads perform
25 ! update of INNER cells 2,...,N-1
26 ! Distribute outer loop iterations manually:
27
28     chunksize = (N-2) / (omp_get_num_threads()-1) + 1
29     my_k_start = 2 + (threadID-1)*chunksize
30     my_k_end = 2 + (threadID-1+1)*chunksize-1
31     my_k_end = min(my_k_end, (N-2))
32
33 ! INNER cell updates
34     do k = my_k_start , my_k_end
35         do j = 2 , (N-1)
36             do i = 2 , (N-1)
37                 ...
38             enddo
39         enddo
40     enddo
41 endif ! thread ID
42 !$OMP BARRIER
43 enddo
44 !$OMP END PARALLEL
```

---

# MPI taxonomy of thread interoperability

- A fully “thread safe” implementation of an MPI library is a difficult undertaking.
  - Example of challenges: Providing shared coherent message queues or coherent internal message buffers.
- Since MPI may be implemented in environments with poor or no thread support, the MPI standard distinguishes four different levels of thread interoperability:
  - ① `MPI_THREAD_SINGLE`: Only one thread will execute (used in the vector mode)
  - ② `MPI_THREAD_FUNNELED`: The process may be multithreaded, but only the main thread will make MPI calls.
  - ③ `MPI_THREAD_SERIALIZED`: The process may be multithreaded, and multiple threads may make MPI calls, but only one at a time; MPI calls are not made concurrently from two distinct threads.
  - ④ `MPI_THREAD_MULTIPLE`: Multiple threads may call MPI anytime, with no restrictions.

# The three levels of thread interoperability for task mode



**Figure 11.2:** Threading levels supported by MPI: (a) `MPI_THREAD_FUNNELED`, (b) `MPI_THREAD_SERIALIZED`, and (c) `MPI_THREAD_MULTIPLE`. The plain MPI mode `MPI_THREAD_SINGLE` is omitted. Typical communication patterns as permitted by the respective level of thread support are depicted for a single multithreaded MPI process with a number of OpenMP threads.

## MPI\_Init\_thread

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

This function initializes MPI in the same way that a call to MPI\_Init would. In addition, it initializes the thread environment.

- **required** (input) — level of desired thread support
- **provided** (output) — level of provided thread support

## Example of using MPI\_Init\_thread

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

int main( int argc, char *argv[] )
{
    int errs = 0, provided, claimed;

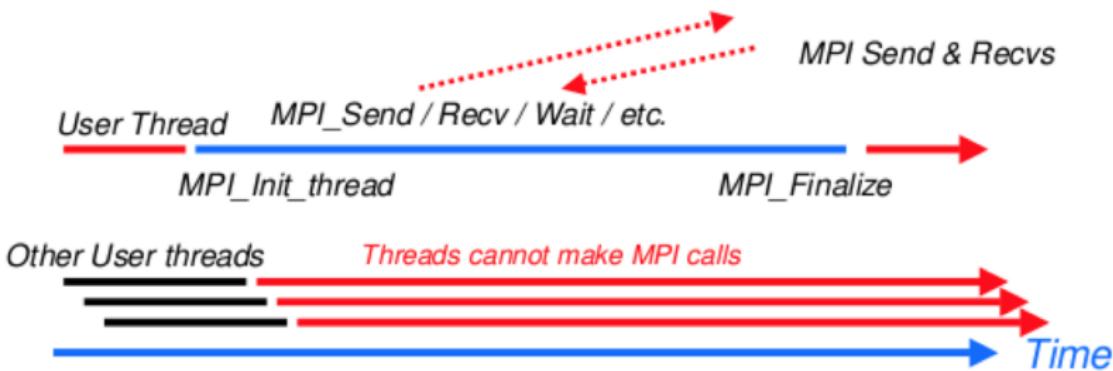
    MPI_Init_thread( &argc, &argv, MPI_THREAD_MULTIPLE, &provided );

    MPI_Query_thread( &claimed );
    if (claimed != provided) {
        errs++;
        printf("Query thread gave thread level %d but Init_thread gave %d\n",
               claimed, provided );fflush(stdout);
    }

    MPI_Finalize();
    return errs;
}
```

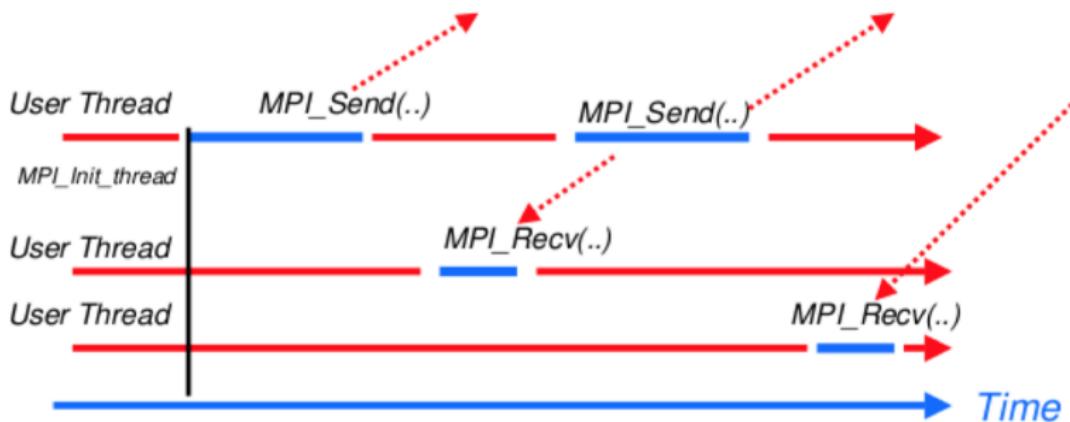
## More about MPI\_THREAD\_FUNNELED

- MPI calls can be outside the parallel region
- Inside the parallel region, MPI calls must be within “omp master”
- If other threads also take part in packing/unpacking messages, must use “omp barrier” to synchronise



## More about MPI\_THREAD\_SERIALIZED

- MPI calls can be outside the parallel region
- Inside the parallel region, MPI calls must be within “omp single” or “omp master”

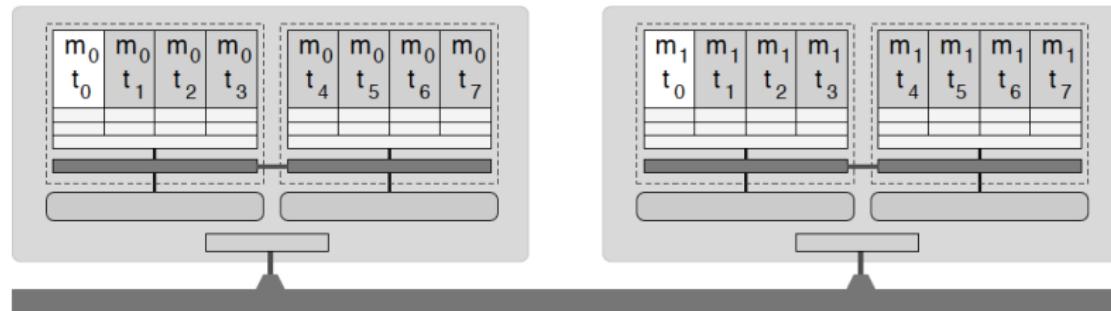


# Hybrid decomposition and mapping

Two important decisions need to be made before launching a hybrid MPI/OpenMP application:

- First one needs to select the number of OpenMP threads per MPI process and the number of MPI processes per node.
  - The total number of threads per node should not exceed the number of cores in the compute node.
  - In some rare cases it might be advantageous to either run a single thread per “virtual core” if the processor efficiently supports simultaneous multithreading or to even use fewer threads than available cores, e.g., if memory bandwidth or cache size per thread is a bottleneck.
- The mapping between MPI processes and OpenMP threads to sockets and cores within a compute node is another important decision.

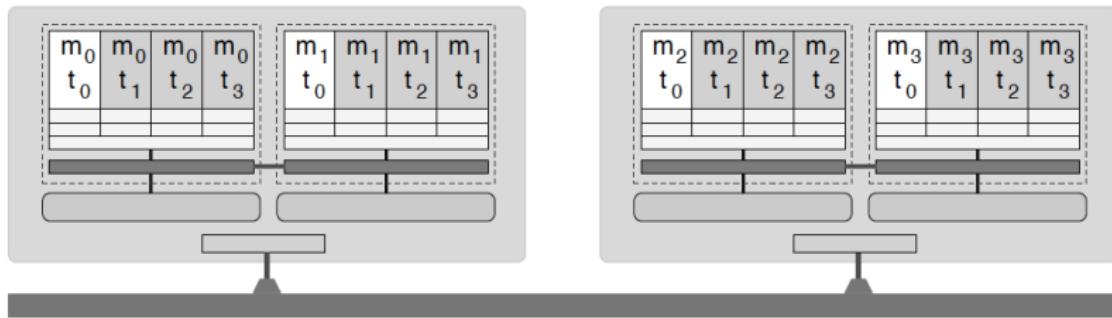
# One MPI process per node (that has multiple sockets)



**Figure 11.3:** Mapping a single MPI process with eight threads to each node.

- There is a clear asymmetry between the hardware design and the hybrid decomposition, which may show up in several performance relevant issues.
- Synchronization of all threads is costly since it involves off-chip data exchange and may become a major bottleneck.
- NUMA optimizations need to be considered;
- A single MPI process per node could also be insufficient to make full use of the latest interconnect technologies.

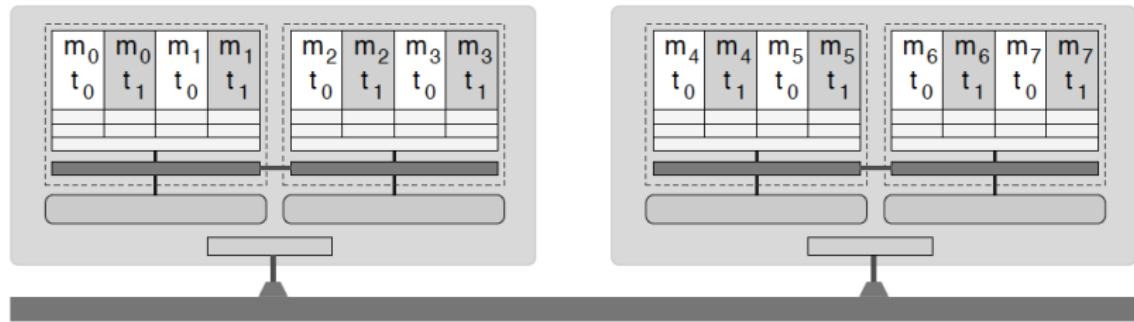
## One MPI process per socket



**Figure 11.4:** Mapping a single MPI process with four threads to each socket (L3 group or locality domain).

- Assigning one multithreaded MPI process to each socket matches the node topology perfectly.
- This mapping avoids ccNUMA data locality problems.
- Accessibility of a single shared cache for all threads of each MPI process allows for fast thread synchronization and increases the probability of cache re-use between the threads

## Multiple MPI processes per socket



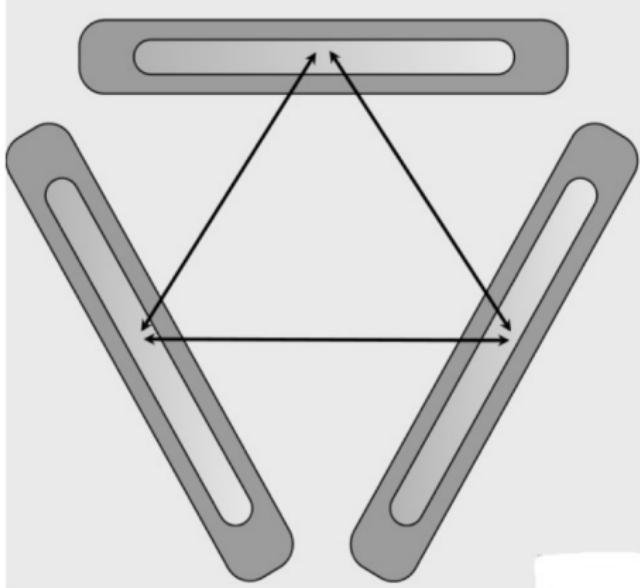
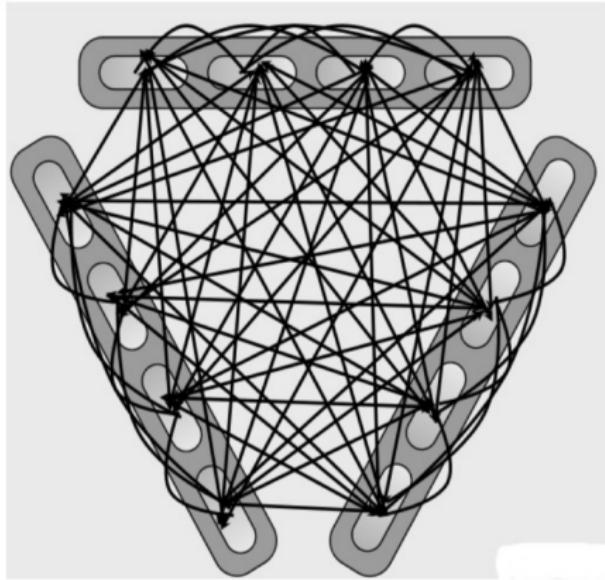
**Figure 11.6:** Mapping two MPI processes with two threads each to a single socket.

- MPI communication may show up on all potential levels: intrasocket, intersocket and internode.
- This decomposition approach can also be beneficial for memory-intensive codes with limited MPI scalability.
  - Often half of the threads are already able to saturate the main memory bandwidth of a single socket and the use of a multithreaded MPI process can add a bit to the performance gain.

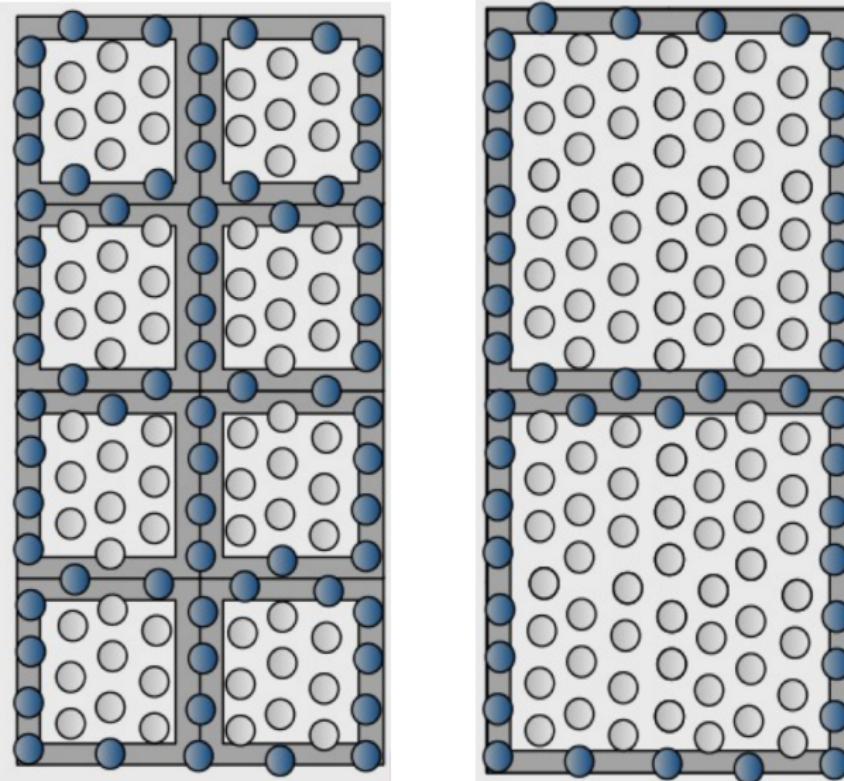
# Potential benefits and drawbacks of hybrid programming

- **Improved rate of convergence**
  - Some numerical algorithms may converge slower with increasing MPI processes
- **Re-use of data in shared caches**
- **Exploiting additional levels of parallelism**
- **Overlapping MPI communication and computation**
- **Reducing MPI overhead**
- **Multiple levels of overhead**
- **Bulk-synchronous communication in vector mode**

## Example: reduced collective communication cost



## Example: improved surface-to-volume ratio



# IN3200/IN4200: MPI programming by the example of Jacobi computations

Parts of the content are from the textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

# Goal

- Demonstrate MPI programming through parallelizing Jacobi computations
  - 1D, 2D, 3D
  - Domain decomposition
  - Use of basic MPI commands

## Let's start with 1D serial Jacobi computation

phi and phi\_new: 1D arrays of length imax

```
// ...
/* initialization (example) */
for (i=1; i<imax-1; i++) {
    x = i*dx;                  // coordinate of the grid point
    phi[i] = sin(M_PI*x);     // suppose we use the sin function
}

maxdelta = 1.0; eps = 1.0e-14;

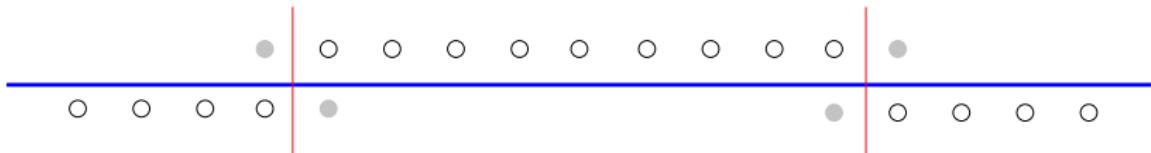
while (maxdelta > eps) {
    maxdelta = 0.;

    for (i=1; i<imax-1; i++) {
        phi_new[i] = (phi[i-1]+phi[i+1])*0.5;
        maxdelta = max(maxdelta, abs(phi_new[i]-phi[i]));
    }

    /* pointer swapping */
    temp_ptr = phi_new; phi_new = phi; phi = temp_ptr;
}
```

# Domain decomposition as the foundation of parallelization

- Note: total number of grid points is  $\text{imax}$ , but the left and right boundary points do not need computation
- Divide the  $\text{imax}-2$  interior points evenly among  $P$  processes
- For programming convenience, each process gets in addition two *ghost points* (also called *halo points*)



## Decomposition and array allocation on each process

P: total number of processes

my\_rank: unique rank of a process

```
my_start = my_rank*(imax-2)/P;
my_stop = (my_rank+1)*(imax-2)/P;
my_imax = my_stop-my_start+2; // including the two ghost points

my_phi_new = (double*)malloc(my_imax*sizeof(double));
my_phi = (double*)malloc(my_imax*sizeof(double));
```

The  $P$  processes are logically lined up from “left” to “right”.

The above code should work for any value of  $1 \leq P \leq imax-2$ , and the work division is as even as possible. (Can you verify?)

## Initializing my\_phi

Each process initializes its my\_phi array:

```
for (i=1; i<my_imax-1; i++) {  
    x = (my_start+i)*dx;      // coordinate of the grid point  
    my_phi[i] = sin(M_PI*x); // same formula as in the serial computation  
}
```

Note: each process initializes for all its subdomain interior points

## Computational work of each process per iteration

```
my_maxdelta = 0.;  
  
for (i=1; i<my_imax-1; i++) {  
    my_phi_new[i] = (my_phi[i-1]+my_phi[i+1])*0.5;  
    my_maxdelta = max(my_maxdelta, abs(my_phi_new[i]-my_phi[i]));  
}  
  
my_phi = my_phi_new;
```

The **same** as the serial computation, except that the computation is now **within** the subdomain of each process

## What else should each process do per iteration?

- Before computation: Must get the ghost point values (by communication)
  - Receive one value from the left neighbor, into `my_phi[0]`
  - Send `my_phi[1]` to the left neighbor
  - Receive one value from the right neighbor, into `my_phi[my_imax-1]`
  - Send `my_phi[my_imax-2]` to the right neighbor
- After computation: Find out the global maximum among all the subdomain values of `my_maxdelta`

## Entire work per process

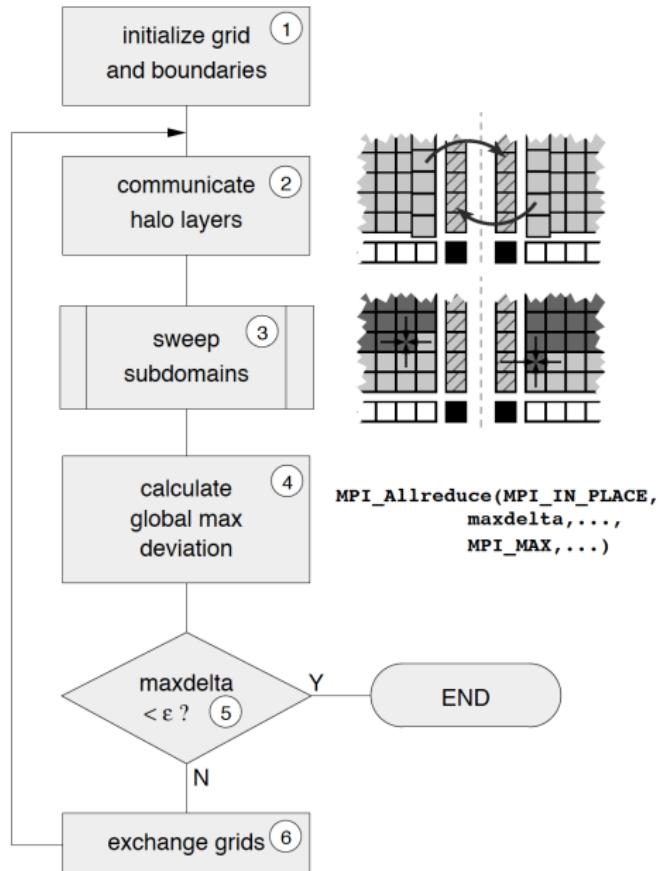
```
while (maxdelta > eps) {
    if (my_rank>0)
        MPI_Irecv(&my_phi[0],1,MPI_DOUBLE,myrank-1,0,MPI_COMM_WORLD,&req_left);
    if (my_rank<P-1)
        MPI_Irecv(&my_phi[my_imax-1],1,MPI_DOUBLE,myrank+1,0,MPI_COMM_WORLD
                  &req_right);
    if (my_rank>0)
        MPI_Send(&my_phi[1],1,MPI_DOUBLE,myrank-1,0,MPI_COMM_WORLD);
    if (my_rank<P-1)
        MPI_Send(&my_phi[my_imax-2],1,MPI_DOUBLE,myrank+1,0,MPI_COMM_WORLD);
    if (my_rank>0)
        MPI_Wait(&req_left,&status_left);
    if (my_rank<P-1)
        MPI_Wait(&req_right,&status_right);

    my_maxdelta = 0.;
    for (i=1; i<my_imax-1; i++) {
        my_phi_new[i] = (my_phi[i-1]+my_phi[i+1])*0.5;
        my_maxdelta = max(my_maxdelta, abs(my_phi_new[i]-my_phi[i]));
    }

    MPI_Allreduce(&my_maxdelta,&maxdelta,1,MPI_DOUBLE,MPI_MAX,MPI_COMM_WORLD);

    /* pointer swapping */
    temp_ptr = my_phi_new; my_phi_new = my_phi; my_phi = temp_ptr;
}
```

# Flow chart of parallel Jacobi computation



## Serial 2D Jacobi computation

Now, phi and phi\_new: 2D arrays of dimension jmax×imax

```
/* initialization (example) */
for (j=1; j<jmax-1; j++) {
    y = j*dy; // y coordinate
    for (i=1; i<imax-1; i++) {
        x = i*dx; // x coordinate
        phi[j][i] = sin(M_PI*y)*sin(M_PI*x);
    }
}

maxdelta = 1.0; eps = 1.0e-14;

while (maxdelta > eps) {
    maxdelta = 0.;

    for (j=1; j<jmax-1; j++)
        for (i=1; i<imax-1; i++) {
            phi_new[j][i] = (phi[j-1][i]+phi[j][i-1]+phi[j][i+1]+phi[j+1][i])*0.25;
            maxdelta = max(maxdelta, abs(phi_new[j][i]-phi[j][i]));
        }

    /* pointer swapping */
    temp_ptr = phi_new; phi_new = phi; phi = temp_ptr;
}
```

## Parallelizing 2D Jacobi computation

- 2D domain decomposition
- The  $P$  processes are organized as a 2D  $N \times M$  process topology (assume  $P = N \times M$ )
- The interior points are evenly divided among the processes
- Instead of two ghost points per process (in 1D), we need now a layer of ghost points around each 2D subdomain
- Each process has up to 4 neighbors, the messages to be sent/received will be arrays of values

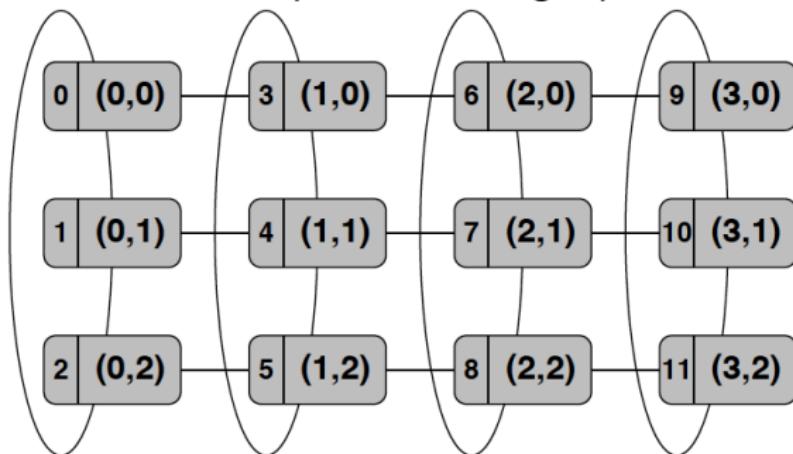
**Some “bookkeeping” is needed to find out which MPI processes are neighbors in 2D!**

## Virtual topologies

- MPI suits very well for implementing domain decomposition (Section 9.2.5) on distributed-memory parallel computers.
- However, setting up the “logical” process grid and keeping track of which ranks have to exchange halo data is nontrivial.
- MPI contains some functionality to support this recurring task in the form of *virtual topologies*.
- To provide a convenient process naming scheme, which fits the required communication pattern.

# Cartesian topologies

Example: a 2D global Cartesian mesh of size  $3000 \times 4000$ . Suppose we want to use  $3 \times 4 = 12$  MPI processes to divide the global mesh, with each process holding a piece of  $1000 \times 1000$  submesh.



**Figure 9.6:** Two-dimensional Cartesian topology: 12 processes form a  $3 \times 4$  grid, which is periodic in the second dimension but not in the first. The mapping between MPI ranks and Cartesian coordinates is shown.

## Cartesian topologies (2)

- As shown in the preceding figure, each process can either be identified by its rank or its Cartesian coordinates.
- Each process has a number of neighbors, which depends on the grid's dimensionality. (In our example, the number of dimensions is two, which leads to at most four neighbors per process.)
- MPI can help with establishing the **mapping** between MPI ranks and Cartesian coordinates in the process grid.

## MPI\_Cart\_create

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],  
                    const int periods[], int reorder, MPI_Comm * comm_cart)
```

- A new, “Cartesian” communicator `comm_cart` is generated, which can be used later to refer to the topology.
- The `periods` array specifies which Cartesian directions are periodic, and the `reorder` parameter allows, if true, for rank reordering so that the rank of a process in communicators `comm_old` and `comm_cart` may differ.
- Here, MPI merely keeps track of the topology information.

## Functions of MPI\_Cart\_coords & MPI\_Cart\_rank

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
```

```
int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)
```

- These are two “service” functions responsible for the translation between Cartesian process coordinates and an MPI rank.
- `MPI_Cart_coords()` calculates the Cartesian coordinates for a given MPI rank.
- The reverse mapping, i.e., from Cartesian coordinates to an MPI rank, is performed by `MPI_Cart_rank()`.

## 2D $N \times M$ domain decomposition

```
MPI_Comm comm_cart;
int dim[2], period[2], reorder, my_coord[2];

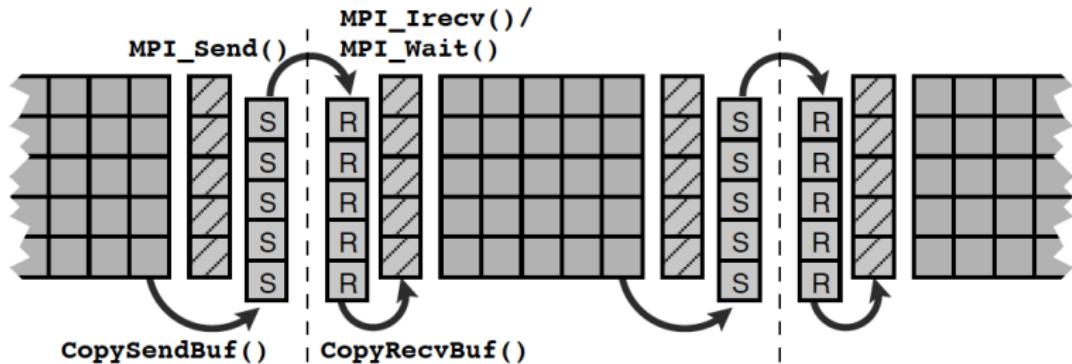
dim[0]=M; period[0]=0; // x direction
dim[1]=N; period[1]=0; // y direction
reorder=1;

MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, reorder, &comm_cart);
MPI_Comm_rank(comm_cart, &my_rank);
MPI_Cart_coords(comm_cart, my_rank, 2, my_coord)

my_xstart = my_coord[0]*(imax-2)/M;
my_xstop = (my_coord[0]+1)*(imax-2)/M;
my_imax = my_xstop-my_xstart+2;

my_ystart = my_coord[1]*(jmax-2)/N;
my_ystop = (my_coord[1]+1)*(jmax-2)/N;
my_jmax = my_ystop-my_ystart+2;
```

## 2D Jacobi: communication in x-direction



**Figure 9.9:** Halo communication for the Jacobi solver (illustrated in two dimensions here) along one of the coordinate directions. Hatched cells are ghost layers, and cells labeled “R” (“S”) belong to the intermediate receive (send) buffer. The latter is being reused for all other directions. Note that halos are always provided for the grid that gets read (not written) in the upcoming sweep. Fixed boundary cells are omitted for clarity.

## 2D Jacobi: communication in x-direction (2)

- Need two send-buffers (downward & upward), both of length `my_jmax-2`
  - Need to pack outgoing messages before send
- Need also two receive-buffers, of length `my_jmax-2`
  - Need to unpack incoming messages after completed receive

```
void copySendBuf0 (double **array, double *sbuf0, int my_imax, int my_jmax)
{
    for (int j=1; j<my_jmax-1; j++)
        sbuf0[j-1] = array[j][1];
}

void copyRecvBuf0 (double **array, double *rbuf0, int my_imax, int my_jmax)
{
    for (int j=1; j<my_jmax-1; j++)
        array[j][0] = rbuf0[j-1];
}
```

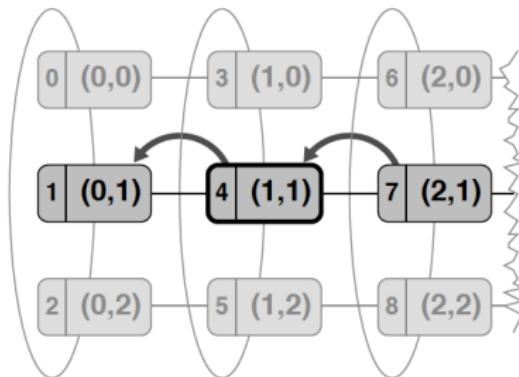
## 2D Jacobi: communication in $y$ -direction

- No need for send-buffers and receive-buffers
- The data points that constitute an outgoing  $y$ -direction message already reside contiguously in memory
- Similarly, an incoming  $y$ -direction message can be stored directly to the target data array

# MPI\_Cart\_shift

A regular task with domain decomposition is to find out, for each process, their neighbors (more specifically, **the neighbors' MPI ranks**) along a certain Cartesian dimension.

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source,  
                   int *rank_dest)
```



**Figure 9.7:** Example for the result of `MPI_Cart_shift()` on a part of the Cartesian topology from Figure 9.6. Executed by rank 4 with `direction=0` and `disp=-1`, the function returns `rank_source=7` and `rank_dest=1`.

# Implementing x-direction communication

```
MPI_Cart_shift(comm_cart, 0, -1, &rank1, &rank0);

if (rank1!=MPI_PROC_NULL)
    MPI_Irecv(rbuf1,my_jmax-2,MPI_DOUBLE,rank1,0,comm_cart,&req_x1);
if (rank0==MPI_PROC_NULL)
    MPI_Irecv(rbuf0,my_jmax-2,MPI_DOUBLE,rank0,0,comm_cart,&req_x0);

if (rank0!=MPI_PROC_NULL) {
    copySendBuf0(phi,sbuf0,my_imax,my_jmax);
    MPI_Send(sbuf0,my_jmax-2,MPI_DOUBLE,rank0,0,comm_cart);
}
if (rank1!=MPI_PROC_NULL)  {
    copySendBuf1(phi,sbuf1,my_imax,my_jmax);
    MPI_Send(sbuf1,my_jmax-2,MPI_DOUBLE,rank1,0,comm_cart);
}

if (rank1!=MPI_PROC_NULL) {
    MPI_Wait(&req_x1,&status_x1);
    copyRecvBuf1(phi,rbuf1,my_imax,my_jmax);
}
if (rank0!=MPI_PROC_NULL) {
    MPI_Wait(&req_x0,&status_x0);
    copyRecvBuf0(phi,rbuf0,my_imax,my_jmax);
}
```

# Implementing y-direction communication

```
MPI_Cart_shift(comm_cart, 1, -1, &rank1, &rank0);

if (rank1!=MPI_PROC_NULL)
    MPI_Irecv(&(phi[my_jmax-1][1]),my_imax-2,MPI_DOUBLE,
              rank1,0,comm_cart,&req_y1);

if (rank0!==MPI_PROC_NULL)
    MPI_Irecv(&(phi[0][1]),my_imax-2,MPI_DOUBLE,
              rank0,0,comm_cart,&req_y0);

if (rank0!=MPI_PROC_NULL)
    MPI_Send(&(phi[1][1]),my_imax-2,MPI_DOUBLE,rank0,0,comm_cart);

if (rank1!=MPI_PROC_NULL)
    MPI_Send(&(phi[my_jmax-2][1]),my_imax-2,MPI_DOUBLE,rank1,0,comm_cart);

if (rank1!=MPI_PROC_NULL)
    MPI_Wait(&req_y1,&status_y1);

if (rank0!=MPI_PROC_NULL)
    MPI_Wait(&req_y0,&status_y0);
```

## How to parallelize 3D Jacobi?

- Need to construct 3D Cartesian topology
- 3D decomposition of all the interior points
- Each process has up to 6 neighbors
- The entire ghost layer consists of up to 6 sides
- Communication in  $z$ -direction is the simplest, no need to pack/unpack messages
  - actually communicates a little bit more than strictly necessary
- For  $x$  and  $y$ -directions, need to pack outgoing messages before send, and unpack incoming messages after completed receive