“The way the processor industry is going, is to add more and more cores, but nobody knows how to program those things. I mean, two, yeah; four, not really; eight, forget it”

Steve Jobs, Apple CEO, 2008

Lecture B.1.:
Foundations of Parallel Computing

CS205: Computing Foundations for Computational Science
Dr. David Sondak
Spring Term 2020

HARVARD
School of Engineering
and Applied Sciences

IACS
INSTITUTE FOR APPLIED
COMPUTATIONAL SCIENCE
AT HARVARD UNIVERSITY

Lectures developed by Ignacio Illorente

Before We Start

Where We Are

Computing Foundations for Computational and Data Science

How to use modern computing platforms in solving scientific problems

Intro: Large-Scale Computational and Data Science

A. Parallel Processing Fundamentals

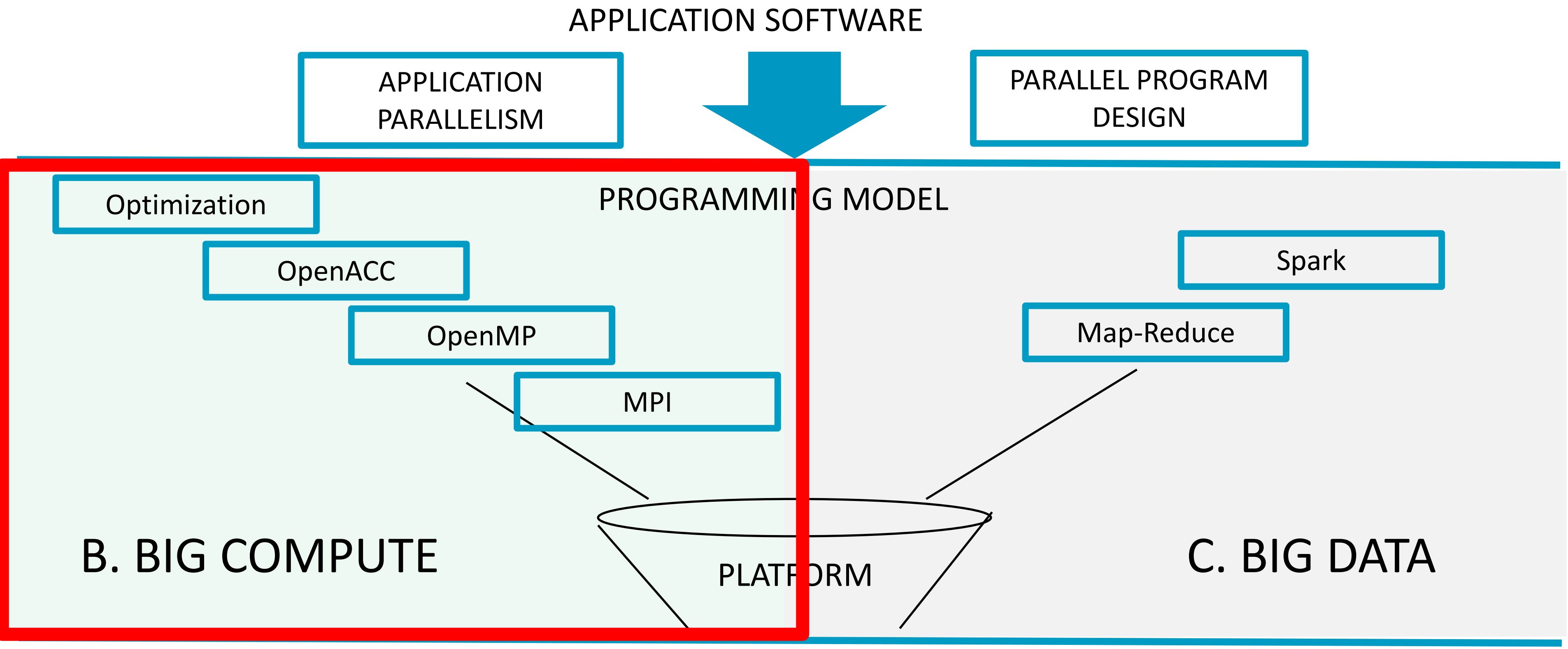
B. Parallel Computing

   B.1. Foundations of Parallel Computing
   B.2. Performance Optimization
   B.3. Accelerated Computing
   B.4. Shared-memory Parallel Processing
   B.5. Distributed-memory Parallel Processing

C. Parallel Data Processing

Wrap-Up: Advanced Topics

CS205: Contents



APPLICATION SOFTWARE

APPLICATION PARALLELISM

PARALLEL PROGRAM DESIGN

PROGRAMMING MODEL

Optimization

OpenACC

OpenMP

MPI

Spark

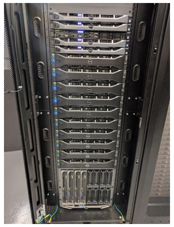
Map-Reduce

B. BIG COMPUTE

PLATFORM

C. BIG DATA

amazon web services



Open Nebula

FAS RC

ODYSSEY HARVARD FAS RESEARCH COMPUTING

FAS RC

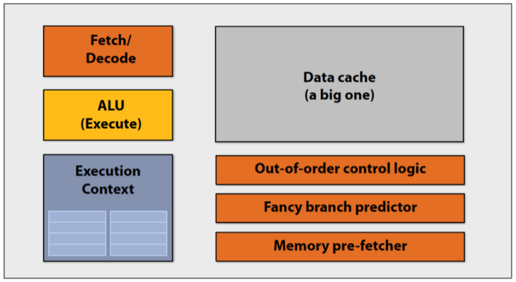
CLOUD COMPUTING

PARALLEL ARCHITECTURES

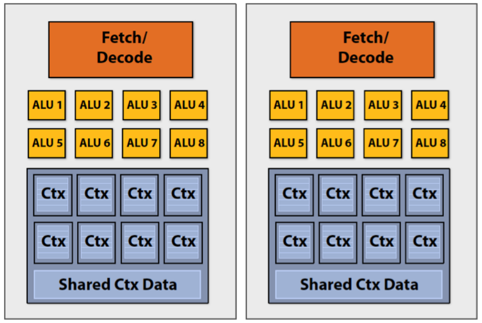
Context

Foundations of Parallel Computing

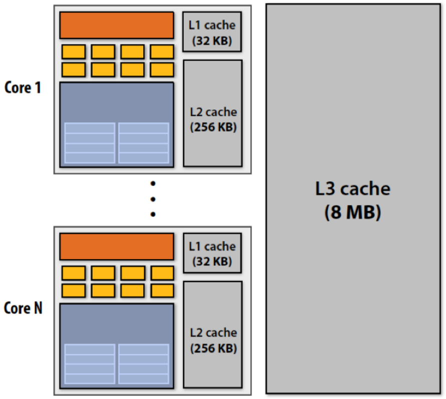
How to develop code that can make effective use of existing parallelism at different levels?



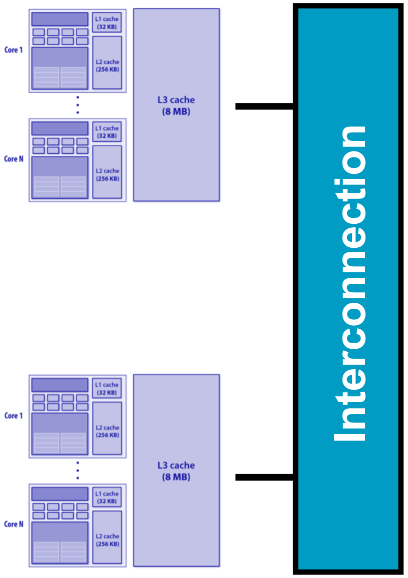
ILP/Data



Many-core



Multi-core



Multi-node

Roadmap

Foundations of Parallel Computing

Performance Optimization

Accelerated Computing

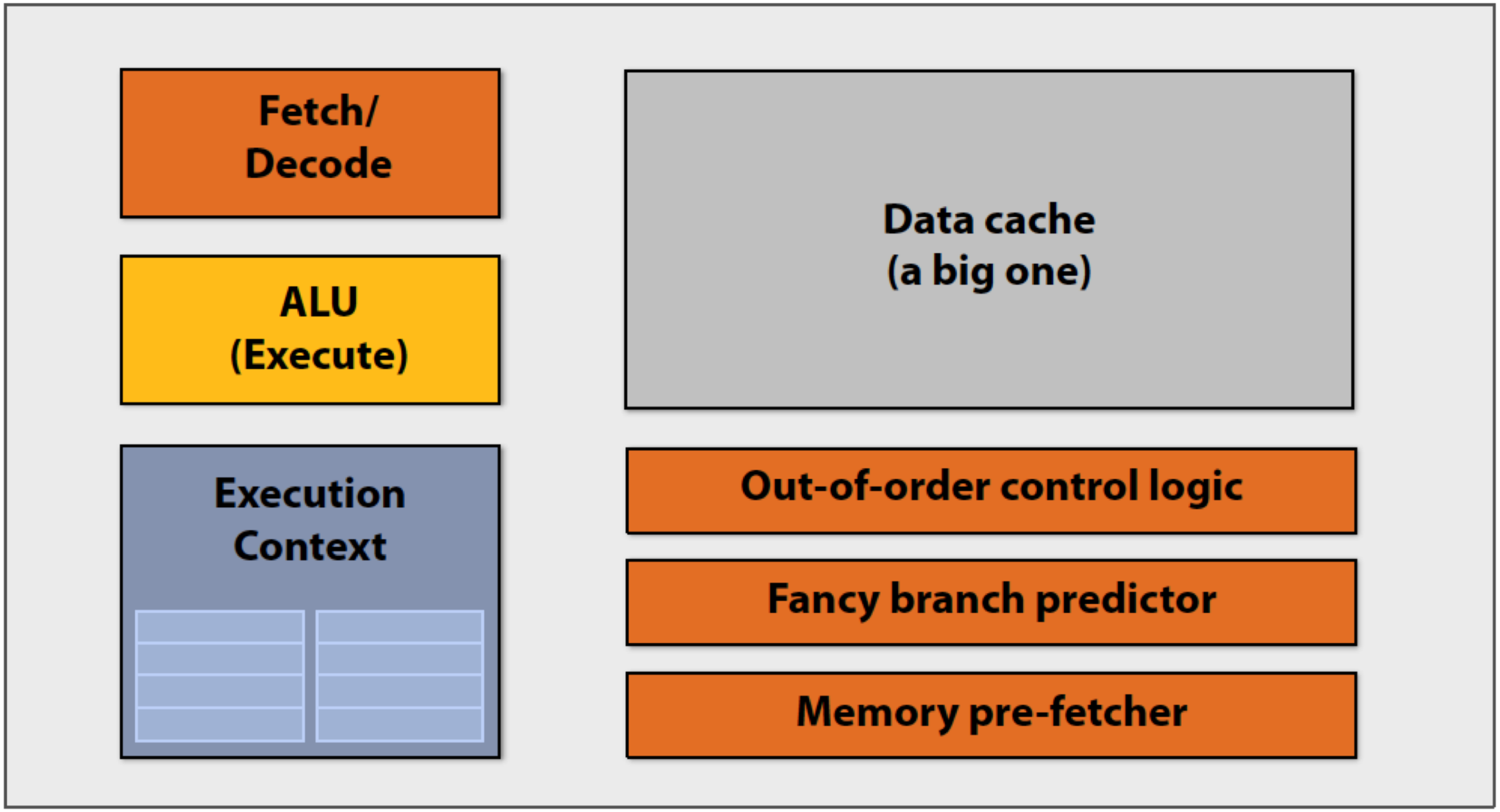
Shared-Memory Programming

Distributed-Memory Programming

Reproducibility and Replicability

Performance Optimization

Parallelism Level


Fetch/
Decode

ALU
(Execute)

Execution
Context

Data cache
(a big one)

Out-of-order control logic

Fancy branch predictor

Memory pre-fetcher

Performance Optimization

Execution Model

CPUs Use Two Main Techniques for Performance

- Instruction Level Parallelism (Superscalar and Pipelining)
  - ✓ Superscalar processors have multiple “functional units” that can run in parallel
  - ✓ Pipelining is a form of parallelism, like an assembly line in a factory

- Caches (Memory Hierarchy)
  - ✓ Small amount of fast memory where values are “cached” in hope of reusing recently used or nearby data

Performance Optimization

Different Libraries and Approaches

Programmer/Compiler can adapt the code to exploit these two techniques

Option	Level	Execution Time	Code Size	Memory Usage	Compile Time
-O0	optimization for compilation time (default)	+	+	-	-
-O1 or -O	optimization for code size and execution time	-	-	+	+
-O2	optimization more for code size and execution time	--		+	++
-O3	optimization more for code size and execution time	---		+	+++
-Os	optimization for code size		--		++
-Ofast	O3 with fast none accurate math calculations	---		+	+++

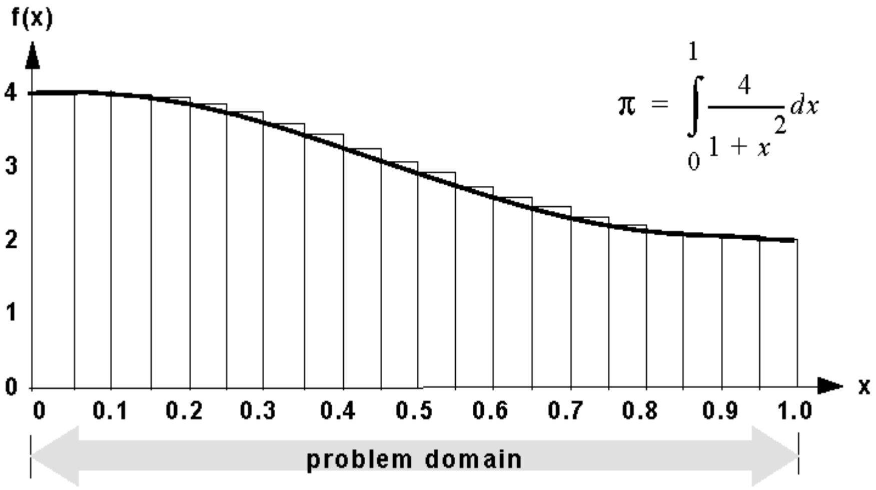
```
int x;
for (x = 0; x < 100; x += 5 ) {
  delete(x);
  delete(x + 1);
  delete(x + 2);
  delete(x + 3);
  delete(x + 4);
}
```

Performance Optimization

Anatomy of an Application

An Example: Pi with Loop Unrolling

```
#include <stdio.h>
#define N 2000000000
#define vl 1024
int main(void) {
  double pi = 0.0f;
  long long i;
  for (i=0; i<N; i++) {
    double t= (double)((i+0.5)/N);
    pi +=4.0/(1.0+t*t);
}
  printf("pi=%11.10f\n",pi/N);
  return 0;
}
```



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

Performance Optimization

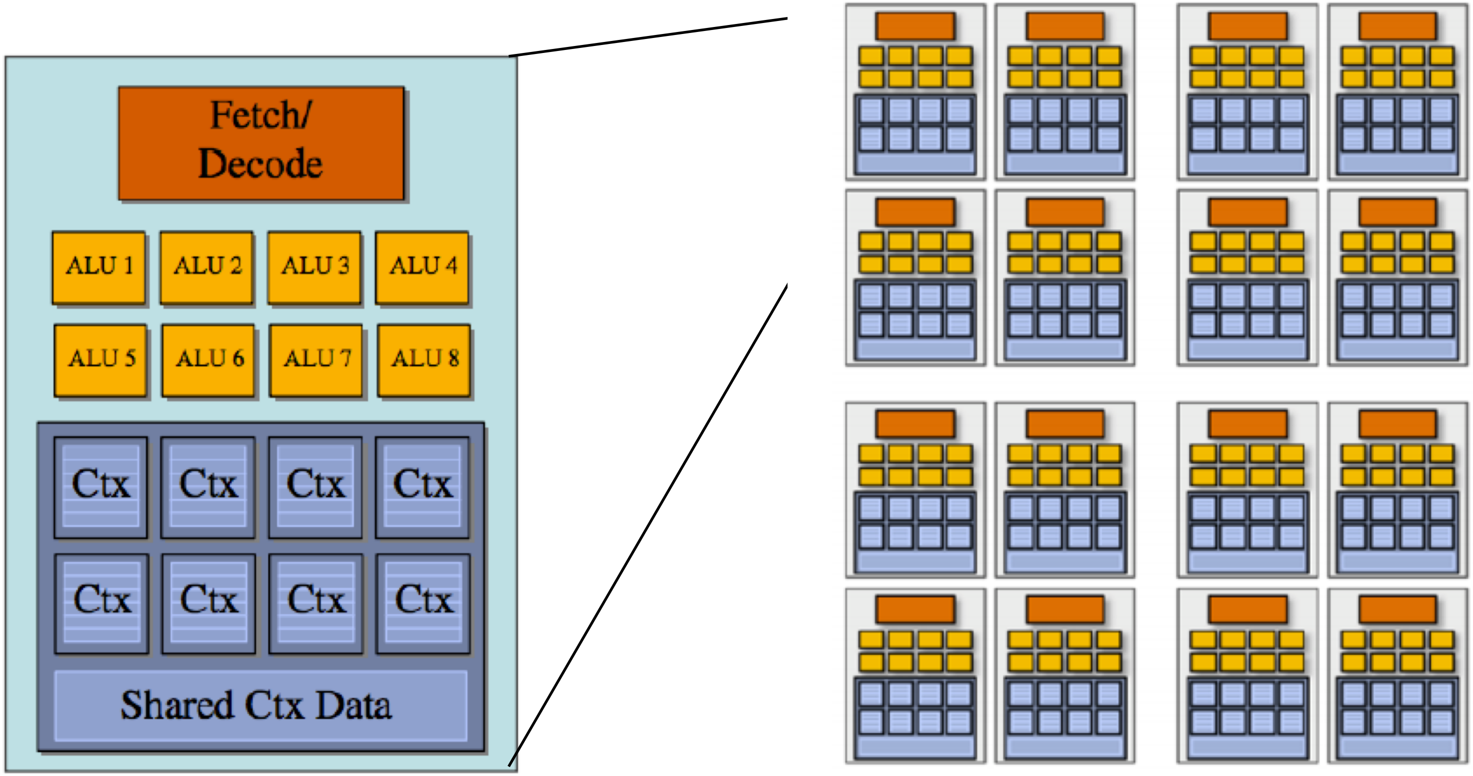
Anatomy of an Application

An Example: Pi with Loop Unrolling

```
#include <stdio.h>
#define N 2000000000
#define vl 1024
int main(void) {
  double pi = 0.0f;
  long long i;
  for (i=0; i<N; i+=2) {
    double t= (double)((i+0.5)/N);
    pi +=4.0/(1.0+t*t);
    double t= (double)((i+1+0.5)/N);
    pi +=4.0/(1.0+t*t);
  }
  printf("pi=%11.10f\n",pi/N);
  return 0;
}
```

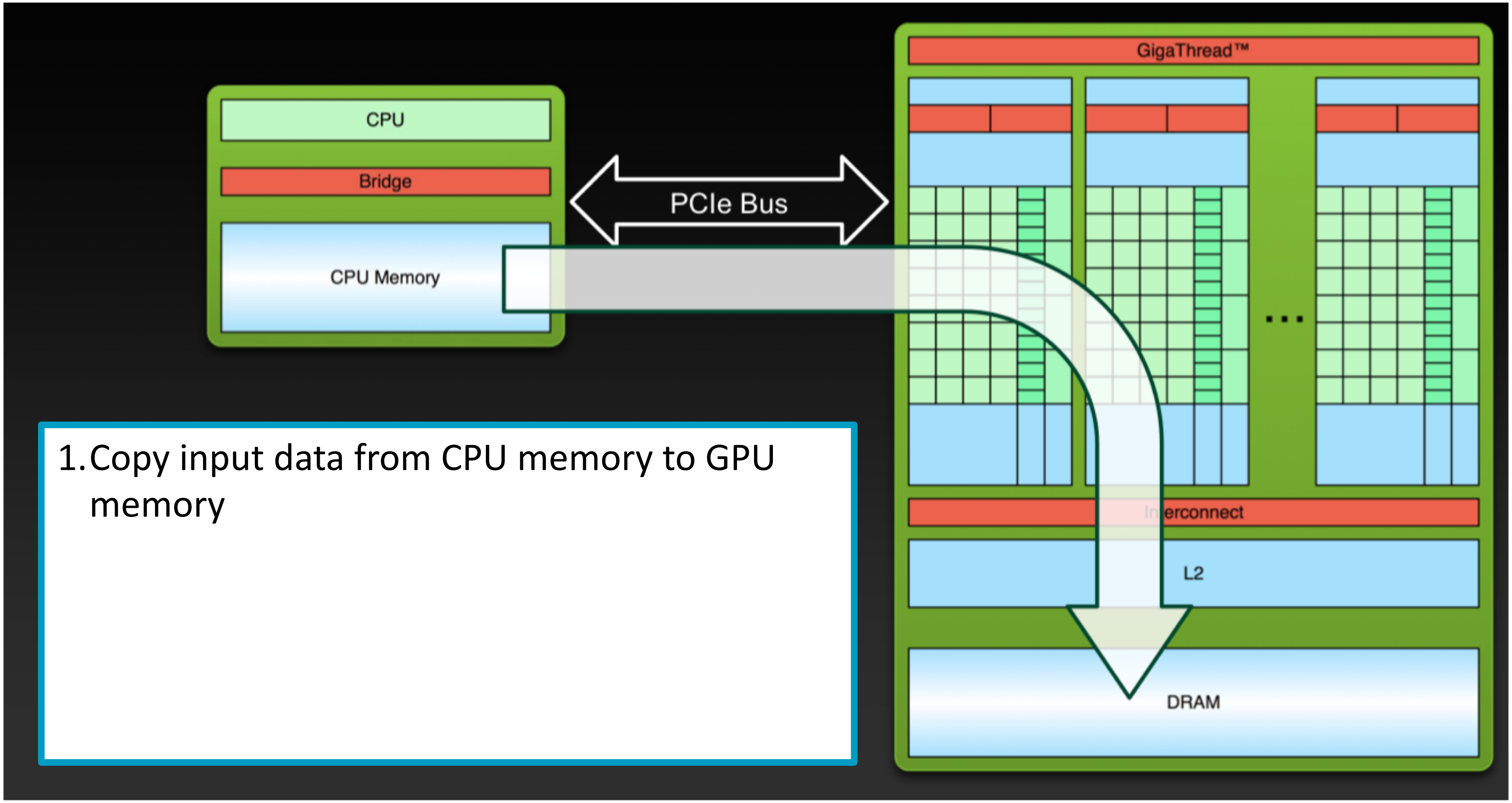
Accelerated Computing

Parallelism Level



Accelerated Computing

Execution Model

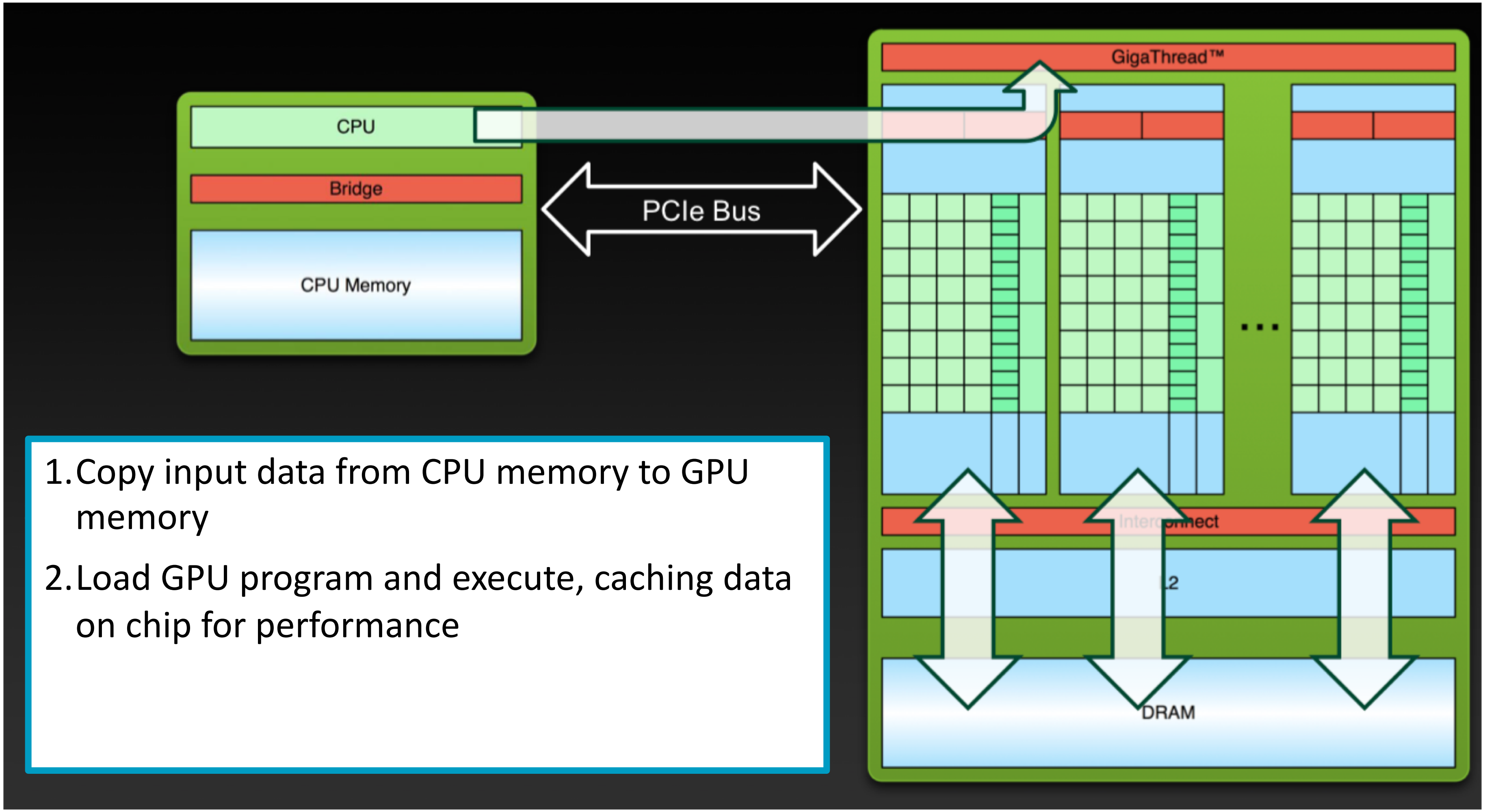


1. Copy input data from CPU memory to GPU memory

Source: NVIDIA

Accelerated Computing

Execution Model

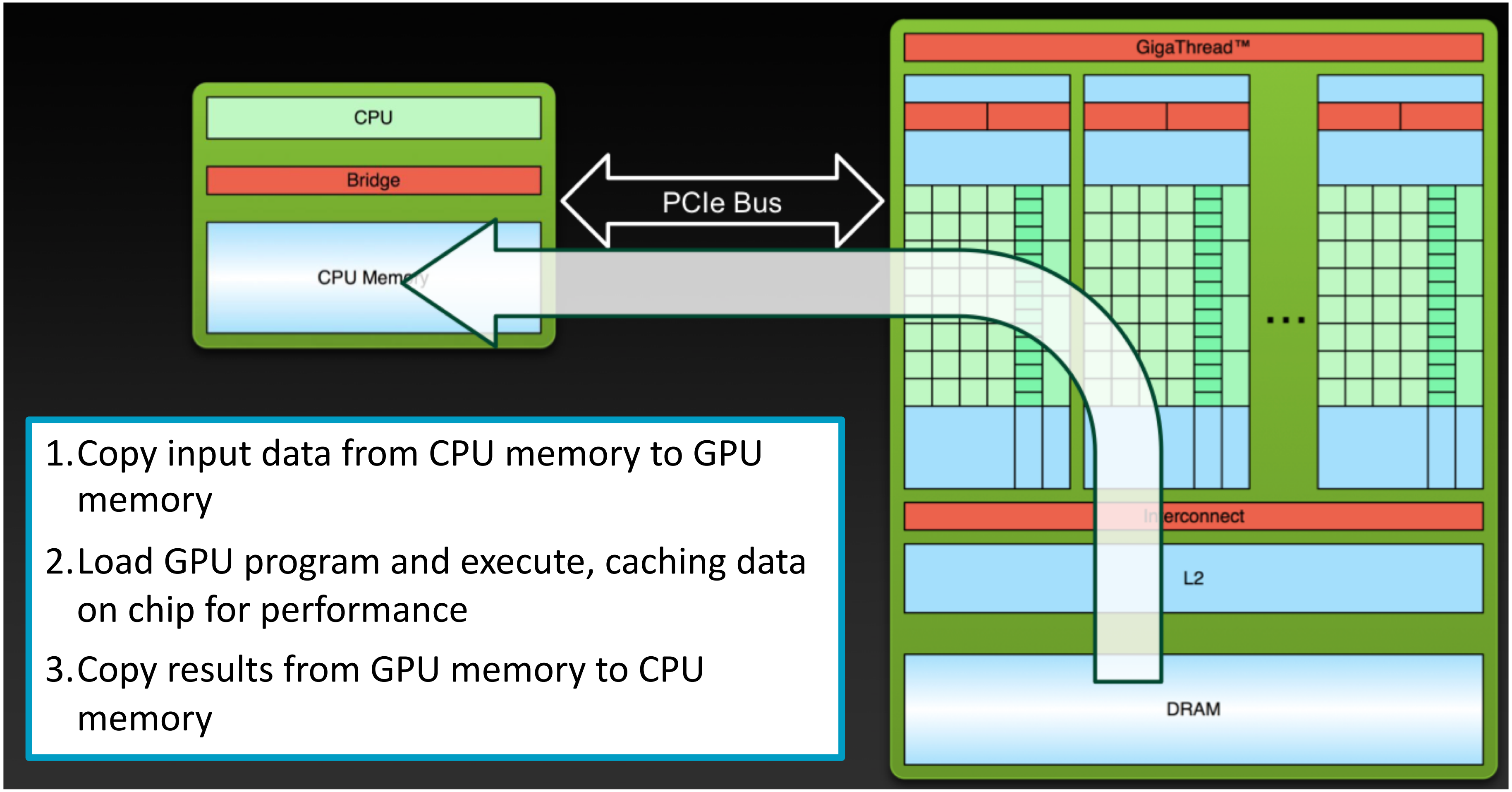


1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Source: NVIDIA

Accelerated Computing

Execution Model

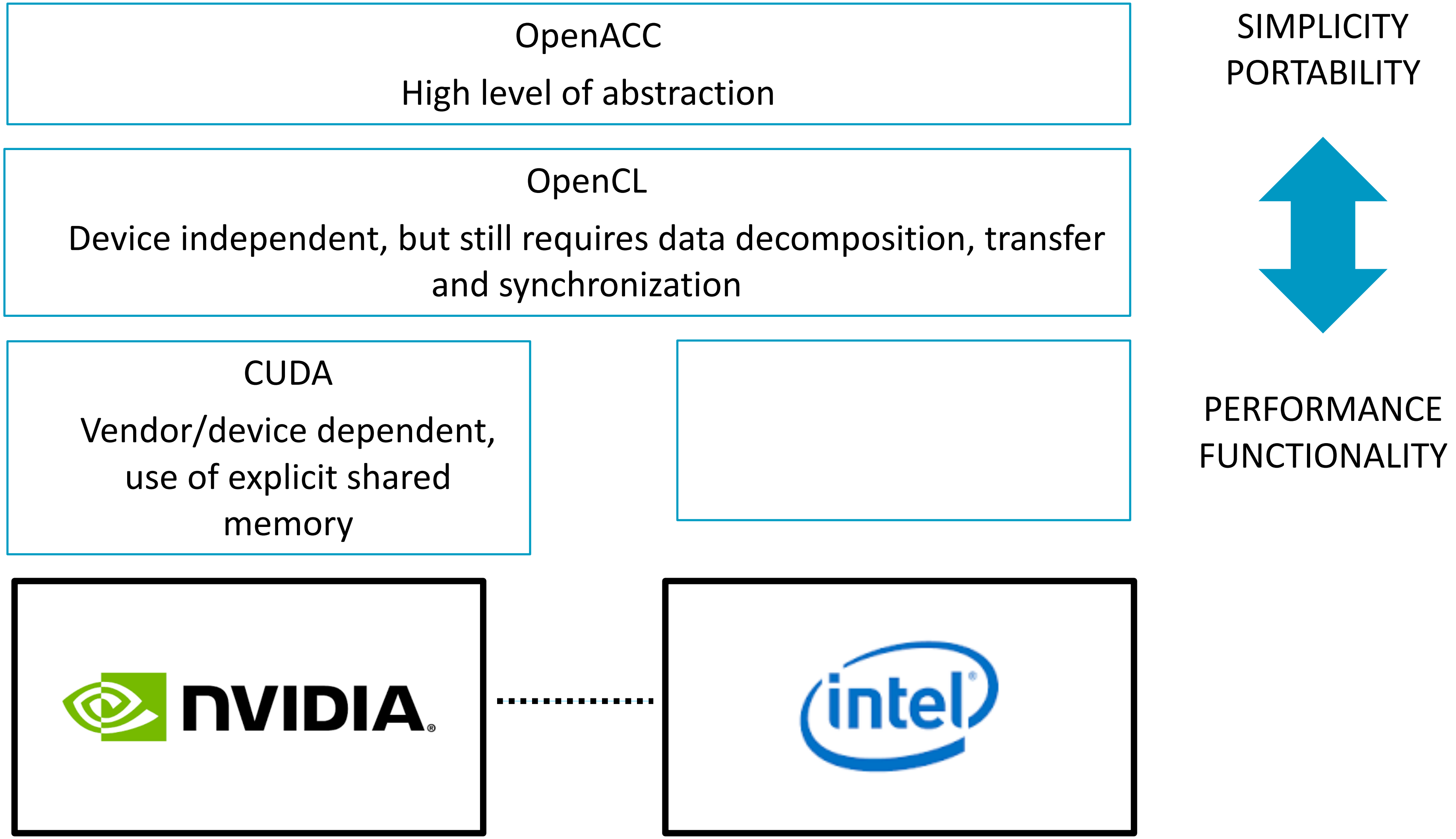


1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Source: NVIDIA

Accelerated Computing

Different Libraries and Approaches



OpenACC

High level of abstraction

OpenCL

Device independent, but still requires data decomposition, transfer and synchronization

CUDA

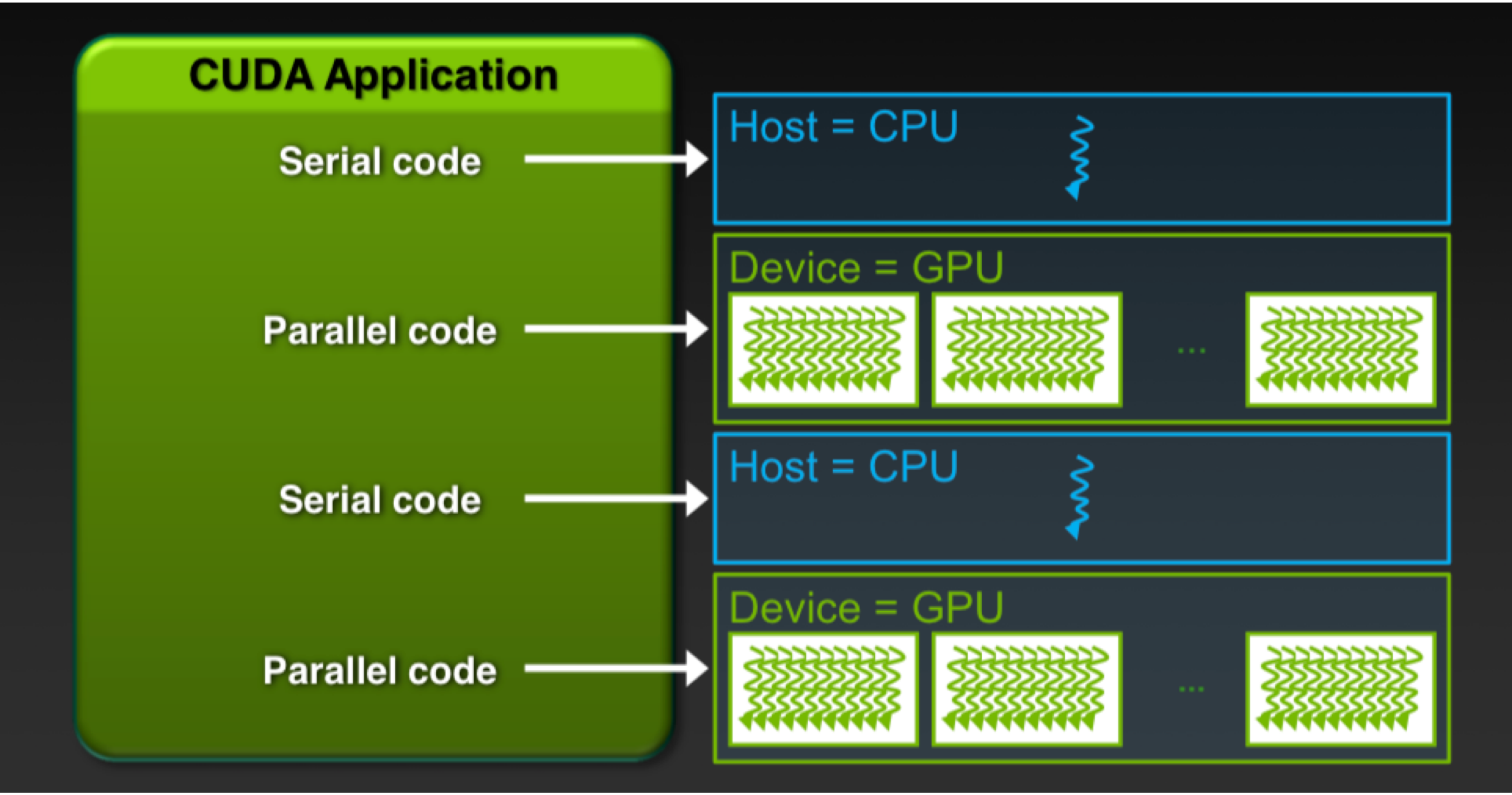
Vendor/device dependent, use of explicit shared memory

SIMPLICITY
PORTABILITY

PERFORMANCE
FUNCTIONALITY

Accelerated Computing

Anatomy of an Application



Source: NVIDIA

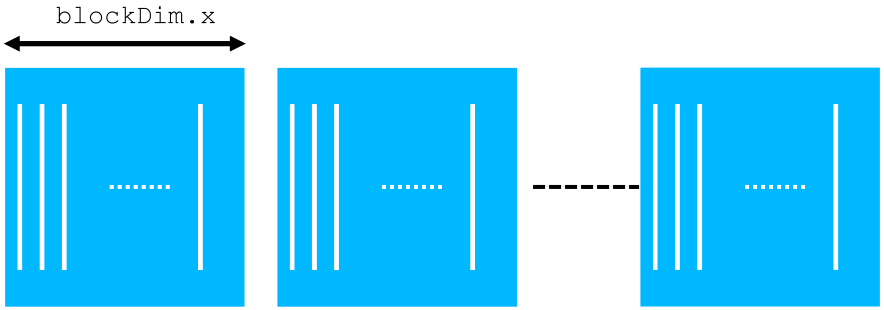
Accelerated Computing

Anatomy of an Application

CUDA

An Example: SAXPY with CUDA

```
#include <stdio.h>

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

int main(void)
{
  int N = 1<<20;
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

  // Perform SAXPY on 1M elements
  saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);

  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
 …
}
```



Accelerated Computing

Anatomy of an Application

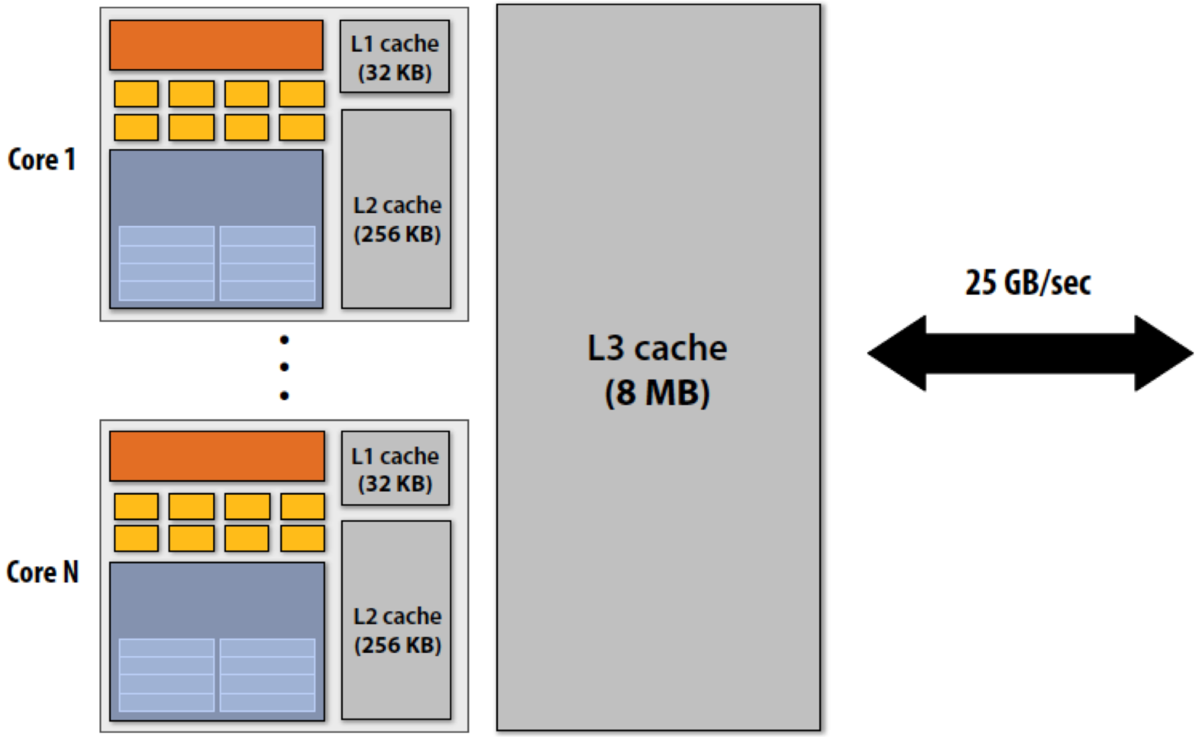
OpenACC

An Example: Pi with OpenACC

```
#include <stdio.h>
#define N 2000000000
#define vl 1024
int main(void) {
  double pi = 0.0f;
  long long i;
  #pragma acc parallel vector_length(vl)
  #pragma acc loop reduction(+:pi)
  for (i=0; i<N; i++) {
    double t= (double)((i+0.5)/N);
    pi +=4.0/(1.0+t*t);
  }
  printf("pi=%11.10f\n",pi/N);
  return 0;
}
```

Shared-Memory Programming

Parallelism Level



Core 1

L1 cache (32 KB)

L2 cache (256 KB)

Core N

L1 cache (32 KB)

L2 cache (256 KB)

L3 cache (8 MB)

25 GB/sec

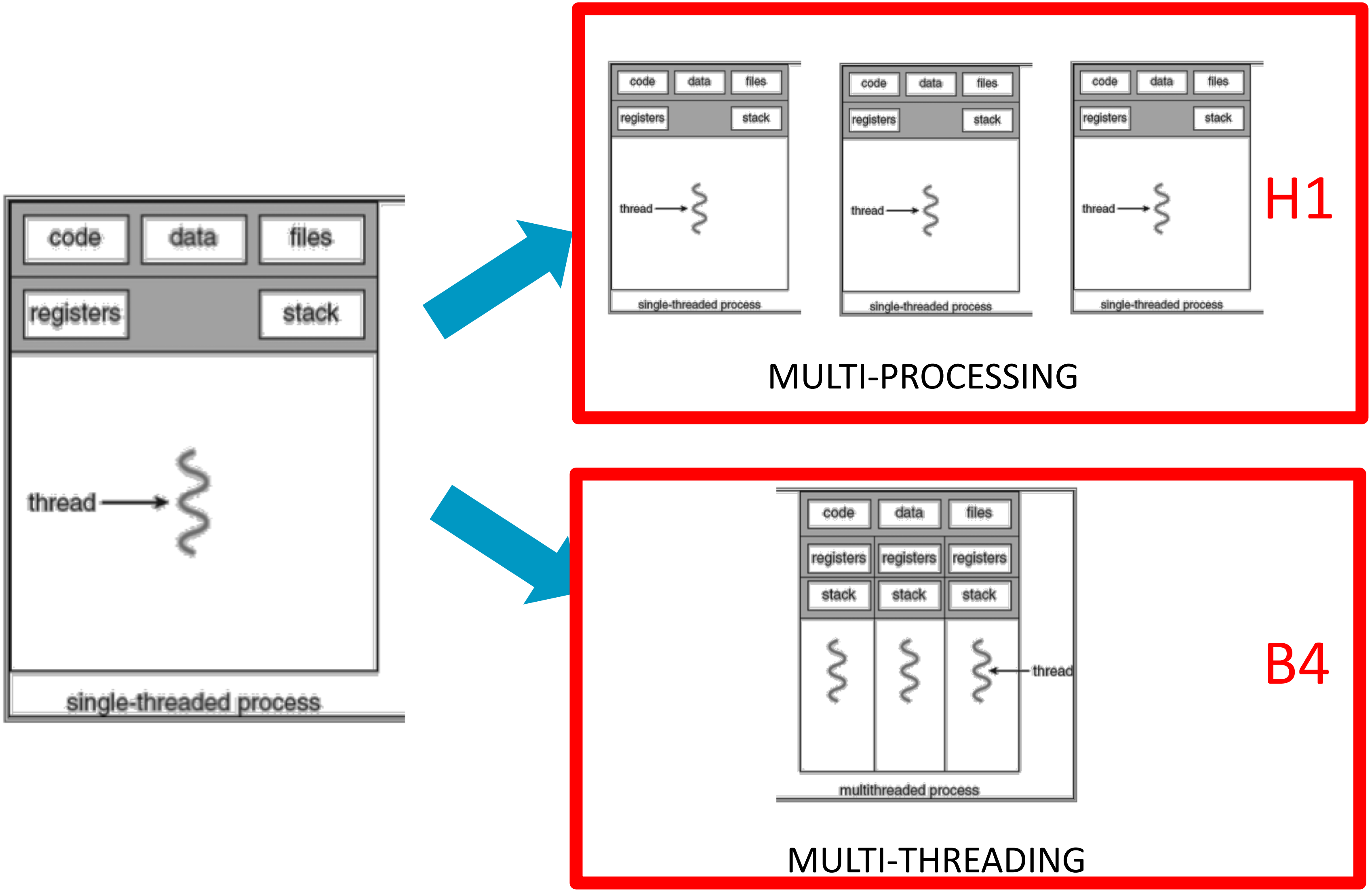
Memory

DDR3 DRAM

(Gigabytes)

Shared-Memory Programming

Execution Model



H1

MULTI-PROCESSING

B4

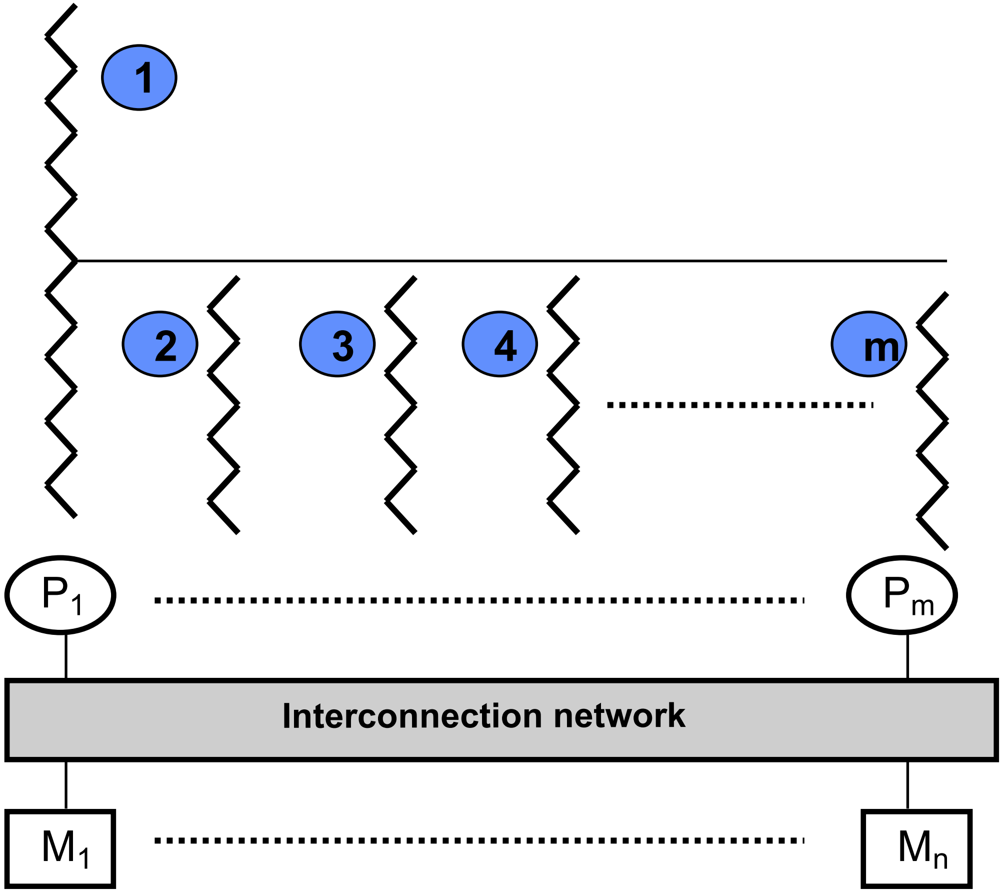
MULTI-THREADING

Shared-Memory Programming

Execution Model

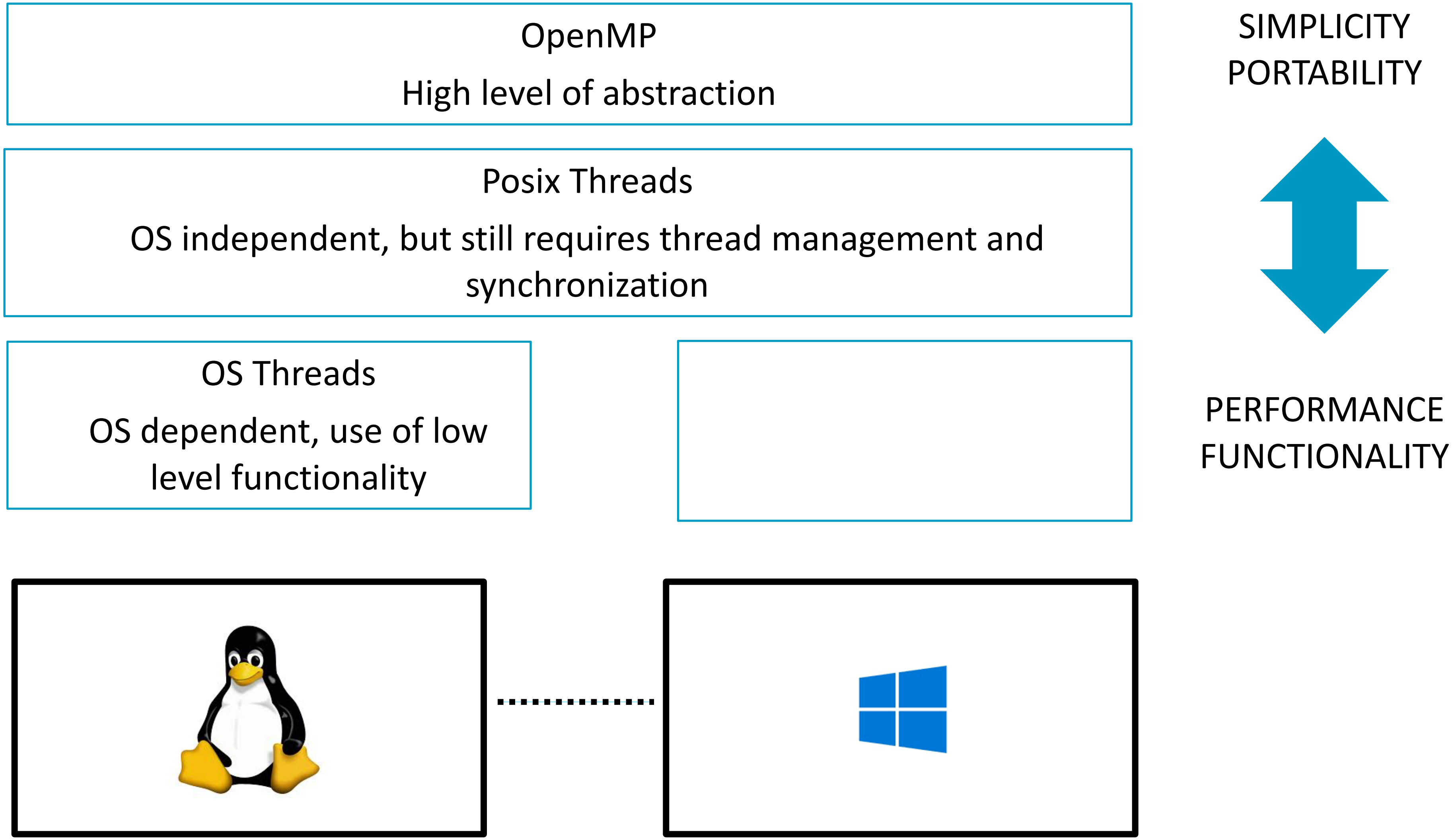
Description

- Shared memory communication
- Thread management routines
    - Create
    - Wait
    - Synchronization
    - …



Shared-Memory Programming

Different Libraries and Approaches



OpenMP

High level of abstraction

Posix Threads

OS independent, but still requires thread management and synchronization

OS Threads

OS dependent, use of low level functionality

SIMPLICITY
PORTABILITY

PERFORMANCE
FUNCTIONALITY

Shared-Memory Programming

Anatomy of an Application

Posix Threads

An Example: Hello World

```
void *print_message_function( void *ptr );
pthread_mutex_t mutex;
main()
{
     pthread_t thread1, thread2;
     pthread_attr_t pthread_attr_default;
     pthread_mutexattr_t pthread_mutexattr_defa
     struct timespec delay;
     char *message1 = "Hello";
     char *message2 = "World\n";

     delay.tv_sec = 10;
     delay.tv_nsec = 0;

     pthread_attr_init(&pthread_attr_default);
     pthread_mutexattr_init(&pthread_mutexattr_default);

     pthread_mutex_init(&mutex, &pthread_mutexattr_default);
     pthread_mutex_lock(&mutex);

     pthread_create( &thread1, &pthread_attr_default,
                    (void *) print_message_function, (void *) message1);
     pthread_mutex_lock(&mutex);
     pthread_create(&thread2, &pthread_attr_default,
                    (void *) print_message_function, (void *) message2);
     pthread_mutex_lock(&mutex);
     exit(0);
}
```

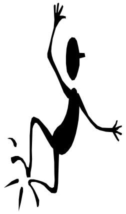
```
void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s ", message);
     pthread_mutex_unlock(&mutex);
     pthread_exit(0);
}
```

Shared-Memory Programming

Different Libraries and Approaches

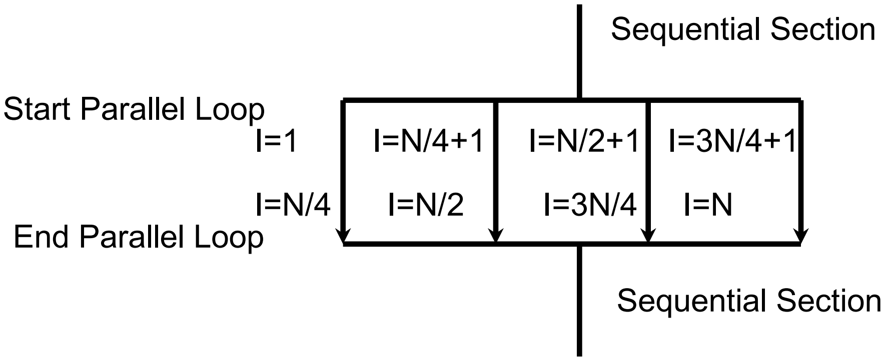
Easy Muti-threading Programming with Directives

Thread programming problems
- Management of too many threads
- Data Races, Deadlocks, and Live Locks



Parallelization Directives
- Execute loop on multiple cores

```
C$PAR DOALL
         DO I=1, N
           A(I) = B(I)
         END DO
```



Sequential Section

Start Parallel Loop

I=1	I=N/4+1	I=N/2+1	I=3N/4+1
I=N/4	I=N/2	I=3N/4	I=N

End Parallel Loop

Sequential Section

Higher-level Abstraction
- Faster development and more portable
- But not so flexible and efficient

Shared-Memory Programming

Anatomy of an Application

OpenMP

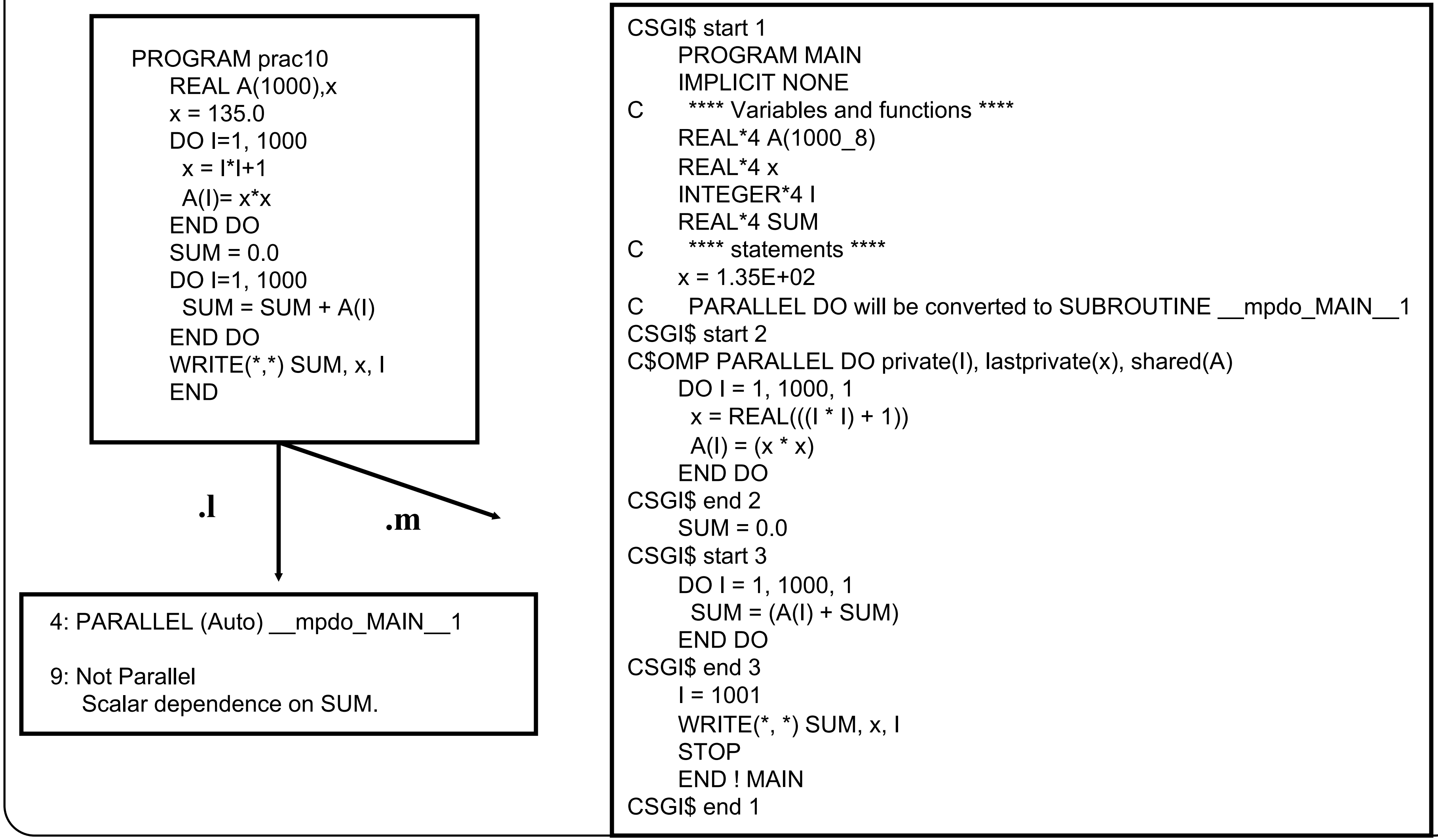
An Example: Hello World

```
#include <stdio.h>
#define N 2000000000
#define vl 1024
int main(void) {
  double pi = 0.0f;
  long long i;
  #pragma omp parallel for reduction(+:pi) private(i,t)
  for (i=0; i<N; i++) {
    double t= (double)((i+0.5)/N);
    pi +=4.0/(1.0+t*t);
  }
  printf("pi=%11.10f\n",pi/N);
  return 0;
}
```

Shared-Memory Programming

Automatic Parallelization

Automatic Parallelization



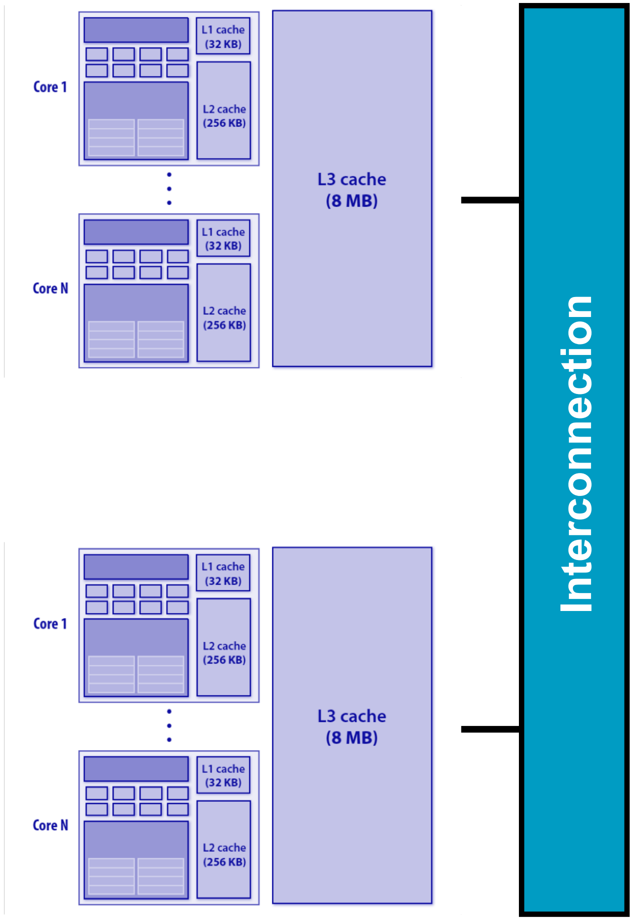
```
PROGRAM prac10
   REAL A(1000),x
   x = 135.0
   DO I=1, 1000
     x = I*I+1
     A(I)= x*x
   END DO
   SUM = 0.0
   DO I=1, 1000
     SUM = SUM + A(I)
   END DO
   WRITE(*,*) SUM, x, I
   END
```

```
4: PARALLEL (Auto) __mpdo_MAIN__1

9: Not Parallel
     Scalar dependence on SUM.
```

```
CSGI$ start 1
      PROGRAM MAIN
      IMPLICIT NONE
C       **** Variables and functions ****
      REAL*4 A(1000_8)
      REAL*4 x
      INTEGER*4 I
      REAL*4 SUM
C       **** statements ****
      x = 1.35E+02
C       PARALLEL DO will be converted to SUBROUTINE __mpdo_MAIN__1
CSGI$ start 2
C$OMP PARALLEL DO private(I), lastprivate(x), shared(A)
      DO I = 1, 1000, 1
        x = REAL(((I * I) + 1))
        A(I) = (x * x)
      END DO
CSGI$ end 2
      SUM = 0.0
CSGI$ start 3
      DO I = 1, 1000, 1
        SUM = (A(I) + SUM)
      END DO
CSGI$ end 3
      I = 1001
      WRITE(*, *) SUM, x, I
      STOP
      END ! MAIN
CSGI$ end 1
```

Distributed-Memory Programming

Parallelism Level

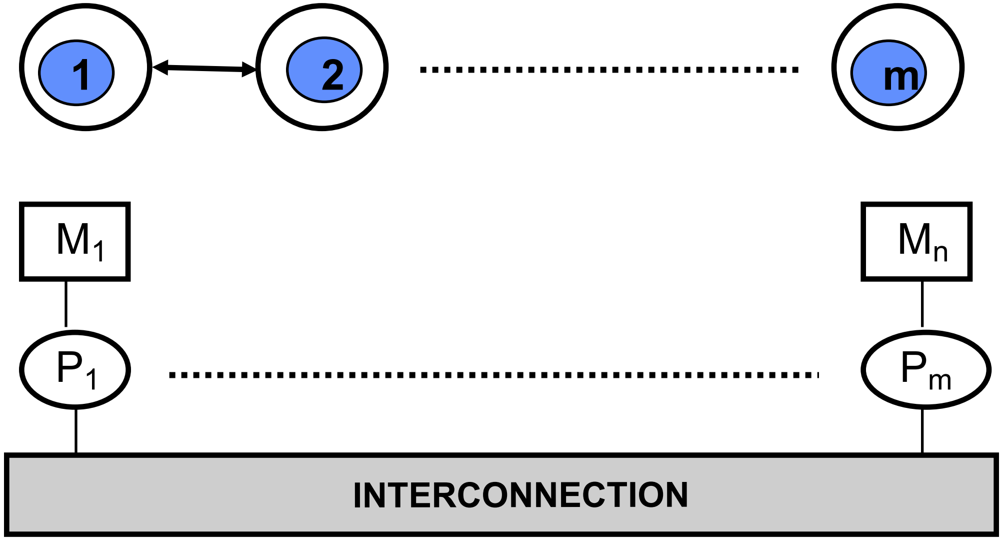


Distributed-Memory Programming

Execution Model

Description

- Message passing communication
- Communication management routines
  - Send
  - Receive
  - Synchronization
  - …



Distributed-Memory Programming

Anatomy of an Application

MPI (*Message Passing Interface*)

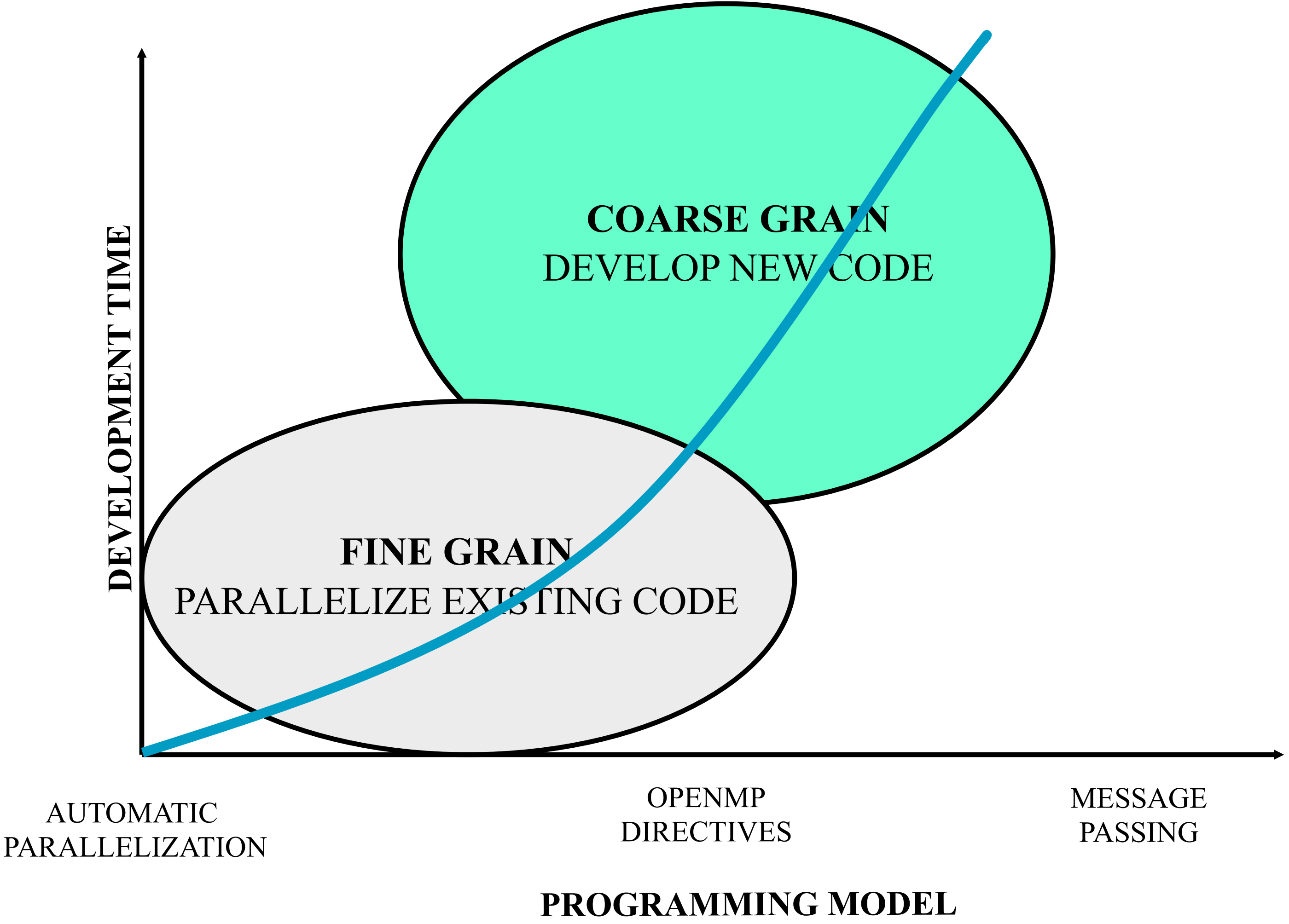
Basic Program Structure (Only 6 routines)

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

main(int argc, char **argv)
  {
    int rank, size, tag=50, destination=0, source;
    char message[100];
    MPI_Status state;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank !=0) {
      sprintf(message,"¡Greetings from process %d!", rank);
      MPI_Send(message, strlen(message)+1, MPI_CHAR, destination, tag, MPI_COMM_WORLD);
    } else {
      for (source = 1; source < size; source++) {
        MPI_Recv(message, strlen(message)+1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &state);
        printf("%s\n", message);
      }
    }
    MPI_Finalize();
  }
```

Distributed-Memory Programming

Shared vs Distributed Memory

The Right Model Depends on (Time, Cost, Performance)



Reproducibility and Replicability

The Four Rs

Reusability

- Reusability refers to the possibility to reuse the software or parts of it for different purposes, in different environments, and by researchers other than the original authors.

Rewriteability

- Rewriteability refers to the possibility to modify and extend the software or parts of it.

Reproducibility

- Reproducibility of a computational experiment means that it can be repeated by a different researcher in a different computing infrastructure but with the same execution environment and to come to the same numerical results.

Replicability

- The attribute Replicability describes the ability to repeat a computational experiment on the same computing infrastructure and to come to the same numerical results and computing performance.

Reproducibility and Replicability

The Four Rs

AERODYNAMICS

NAVIER-STOKES

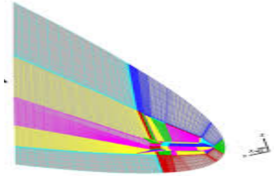
$$\frac{\partial u}{\partial t}+\frac{1}{r^{\tau}}\frac{\partial(r^{\tau}uu)}{\partial r}+\frac{\partial(vu)}{\partial z}=-\frac{\partial p}{\partial r}+\frac{1}{Re}\frac{\partial}{\partial z}\left(\frac{\partial u}{\partial z}-\frac{\partial v}{\partial r}\right)+\frac{1}{Fr^{2}}g_{r},$$

$$\frac{\partial v}{\partial t}+\frac{1}{r^{\tau}}\frac{\partial(r^{\tau}uv)}{\partial r}+\frac{\partial(vv)}{\partial z}=-\frac{\partial p}{\partial z}+\frac{1}{Re\,r^{\tau}}\frac{\partial}{\partial r}\left(r^{\tau}\left(\frac{\partial u}{\partial z}-\frac{\partial v}{\partial r}\right)\right)+\frac{1}{Fr^{2}}g_{z},$$

$$\frac{1}{r^{\tau}}\frac{\partial(r^{\tau}u)}{\partial r}+\frac{\partial v}{\partial z}=0,$$

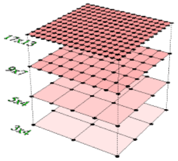
PHYSICS

FINITE DIFFERENCE



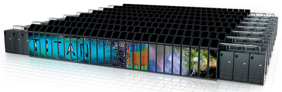
ACCURACY

MULTIGRID



COMPLEXITY

PARALLEL



SPEED-UP

EXECUTION ENVIRONMENT

- Algorithm, application version and dependencies

- VIRTUAL MACHINES
- SOFTWARE CONTAINERS

SYSTEM CAPACITY

- Execution time, performance…

- CLOUD PROVIDERS

Reproducibility and Replicability

Different Results for the Same Program?

NON-REPRODUCIBLE BEHAVIORS

- A bug in the software or a fault in the hardware
- Floating-point numbers cannot precisely represent all real numbers bringing rounding and overflow errors
- Code and compiler options may change operation ordering, sometimes x=a+b+… differs from x=b+a… (non-commutativity)
- Parallel processing changes operation ordering, sometimes, x=a+b+c+d does not give the same results when computed (on two units) either as x=(a+b)+(c+d) or x=(a+c)+(b+d) (non-associativity).

Reproducibility and Replicability

Round-off Error Propagation

NUMERICAL ACCURACY OF REDUCTION OPERATIONS

Floating point representation of real numbers may bring two types of error:

- **Overflow:** Representation range limited by the exponent
- **Round-off:** Finite number of bits in mantissa

IEEE 754: 53 bits mantissa / 11 bits exponent

2**1023 = 8.9 10^307; 2**1024 = Inf

1.0 +2**(-53) = 1

- **Error Accumulation**
- **Catastrophic Cancellation**

(-100.0+100.0+1.0e-15)*1.0e+32 = 1.0e+17

(-100.0+ 1.0e-15+100.0)*1.0e+32 = 0.0

Reading Assignments / Open Discussion

Application Containers in HPC

Gregory M. Kurtzer, Vanessa Sochat, Michael W. Bauer,*“Singularity: Scientific containers for mobility of compute”* PLoS One. 2017; 12(5): e0177459

What is reproducibility?

What is mobility of compute?

Can you describe a use case in science?

Next Steps

- **Lab session** tomorrow:
  - I2. OpenNebula Private Cloud Sandbox on AWS
  - I3. Docker on AWS

- Get ready for first **hands-on:**
  - H1. Python Multiprocessing

Questions

Foundations of Parallel Computing