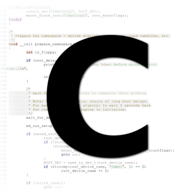
Lecture Notes

"Scientific Computing II"

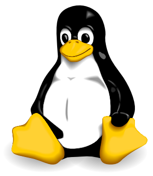
Summer Term 2021

Dr. Jens Saak
jens.saak
@mpi-magdeburg.mpg.de

Dipl.-Math. Martin Köhler
martin.koehler
@mpi-magdeburg.mpg.de



```
1 for $j := i, \dots, n$ do
2     for $k := 1, \dots, i-1$ do
3         $A_{i,j} -= A_{i,k} A_{k,j}$
```



June 28, 2021

Preface

Contents

CHAPTER 1

Introduction

Contents

1.1 Why Parallel Computing?

Classically the area of Scientific Computing, or more precisely that of High Performance Computing (HPC) has been the major source for the demand on parallel computing techniques and abilities. Nowadays the picture is changing drastically due to the emerging field of multi and many Core architectures. In only a few years of time one has to expect computer programs of all kinds to have to run on highly parallel desktop computers. Therefore, we can currently state three important reasons for the interest in parallel computing techniques:

1. Problem size exceeds desktop capabilities.
2. Problem is inherently parallel (e.g. Monte-Carlo simulations).
3. *Modern architectures require parallel programming skills to take best adavatage of their features.*

Here, the last one is clearly the modern one that has come up with the availability of MultiCore processors in basically all modern devices starting from desktop computers, possibly equipped with high performance graphics processing units (GPUs), but also reaching down to modern smart-phones with MultiCore mobile processors. The first two are more classical approaches requiring specialized HPC hardware or networks of many standard PCs (the so called *Clusters*). Although especially for the second MultiCore processors may give a huge impact, too.

1.2 Flynn's Taxonomy of Parallel Architectures

The basic definition of a parallel computer is very vague in order to cover a large class of systems. Important details that are not considered by the definition are:

- How many processing elements?
- How complex are they?
- How are they connected?
- How is their cooperation coordinated?
- What kind of problems can be solved?

The basic classification allowing answers to most of these questions is known as *Flynn's taxonomy*. It distinguishes four categories of parallel computers.

The four categories allowing basic answers to the questions on global process control, as well as the resulting data and control flow in the machine are

1. Single-Instruction, Single-Data (SISD)
2. Multiple-Instruction, Single-Data (MISD)
3. Single-Instruction, Multiple-Data (SIMD)
4. Multiple-Instruction, Multiple-Data (MIMD)

All four of these serve as *machine models* for different kinds of actual hardware implementations. The models with their properties subdivisions and example hardware are discussed in the next few sections.

1.2.1 Single-Instruction, Single-Data (SISD)

The most simple and classically most common architectural model is the single-instruction, single-data model. It represents what has been the standard for desktop computers until the introduction of MultiCore processors only a few years ago. The SISD model is characterized by

- a single processing element,
- executing a single instruction,
- on a single piece of data,
- in each step of the execution.

It is thus in fact the standard sequential computer implementing, e.g., the *von Neumann* model[1].

Examples

- desktop computers until Intel® Pentium® 4 era,
- early netBooks on Intel® Atom™ basis,
- pocket calculators,
- abacus,
- embedded circuits

A schematic presentation of the model is given in Figure 1.1.

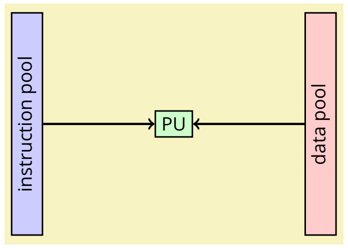


Figure 1.1: Single-Instruction, Single-Data (SISD) machine model

[1]see, e.g., http://en.wikipedia.org/wiki/Von_Neumann_architecture

1.2.2 Multiple-Instruction, Single-Data (MISD)

In contrast to the SISD model in the MISD architecture we have

- multiple processing elements,
- executing a separate instruction each,
- all on the same single piece of data,
- in each step of the execution.

The MISD model is usually not considered very useful in practice. Figure 1.2 shows the graphical representation of this model.

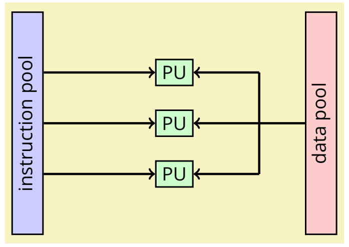


Figure 1.2: Multiple-Instruction, Single-Data (MISD) machine model

1.2.3 Single-Instruction, Multiple-Data (SIMD)

The single-instruction, multiple-data model is a much more useful approach compared to MISD. Here the characterization is

- multiple processing elements,
- execute the same instruction,
- on a multiple pieces of data,
- in each step of the execution.

This model is thus the ideal model for all kinds of vector operations

$$c = a + \alpha b.$$

Here for every triple of entries in a, b, and c we have to perform the same three instructions. First we scale the entry in b, then add it to the corresponding entry in a and store the result to the appropriate position in c.

Examples

- Graphics Processing Units,
- Vector Computers,
- SSE (Streaming SIMD Extension) registers of modern CPUs.

The attractiveness of the SIMD model for vector operations, i.e., linear algebra operations, comes at a cost.

Consider the simple conditional expression

```
  if (b==0) c=a; else c=a/b;
```

The SIMD model requires the execution of both cases sequentially. First all processes for which the condition is true execute their assignment, then the other do the second assignment. Therefore, conditionals need to be avoided on SIMD architectures to guarantee maximum performance. Figure 1.3 makes it even more obvious that SIMD architectures are ideal to utilize *data level parallelism*.

The SIMD model is characterized by the "run the same set of operations at the same time in parallel" paradigm. This is helpful in vector operations as seen above, but brings in the restrictions for conditionals we have also seen. The SPMD model described in the next section as one subclass of the MIMD model weakens this restriction.

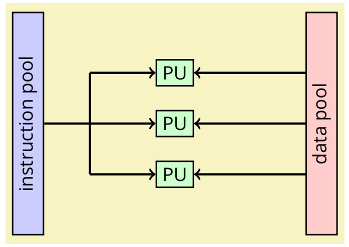


Figure 1.3: Single-Instruction, Multiple-Data (SIMD) machine model

1.2.4 Multiple-Instruction, Multiple-Data (MIMD)

The most flexible architectural approach is described by the multiple-instruction, multiple-data model. MIMD allows

- multiple processing elements,
- to execute a different instruction,
- on a separate piece of data,
- at each instance of time.

Examples

- multicore and multi-processor desktop PCs,
- cluster systems.

MIMD computer systems can be further divided into three class regarding their memory configuration:

distributed memory

Every processing element has a certain exclusive portion of the entire memory available in the system. Data needs to be exchanged via an interconnection network. These machines will be treated in more detail in Chapter 5.

shared memory

All processing units in the system can access all data in the main memory. Several different models describing the uniformity of the access like UMA, NUMA, and ccNUMA will be treated in Chapter 3.

hybrid

Certain groups of processing elements share a part of the entire data and instruction storage.

When working with MIMD systems users basically follow either of the following two programming models.

Single Program, Multiple Data (SPMD)

SPMD is a programming model for MIMD systems. “In SPMD multiple autonomous processors simultaneously execute the same program at independent points.”[2] This contrasts to the SIMD model where the execution points are not independent.

This is opposed to

Multiple Program, Multiple Data (MPMD)

A different programming model for MIMD systems, where multiple autonomous processing units execute different programs at the same time. Typically Master/Worker like management methods of parallel programs are associated with this class. There the Master is acting as a controller process executing a different programs than its Workers performing the actual computations.

[2]Wikipedia: `http://en.wikipedia.org/wiki/SPMD`

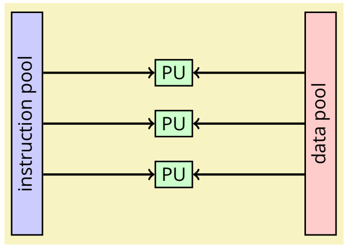


Figure 1.4: Multiple-Instruction, Multiple-Data (MIMD) machine model

1.3 Memory Hierarchies in Parallel Computers

1.3.1 Repetition Sequential Processor

Recall the basic ideas for the memory hierarchy on a sequential system we have discussed in Chapter 4 of the first part of this lecture. We have classified several types of storage into fast medium speed and slow storage/memory. Based on this classification we derived strategies for data management to optimize the run-time of a program executing a certain variant of an algorithm. For linear algebra operations this mainly lead to the reformulation of the algorithms in block matrix fashion where we could execute cubically many operations on quadratically many entries in terms of the matrix dimension. This turned out to be the optimal way of using the Caches on the processor.

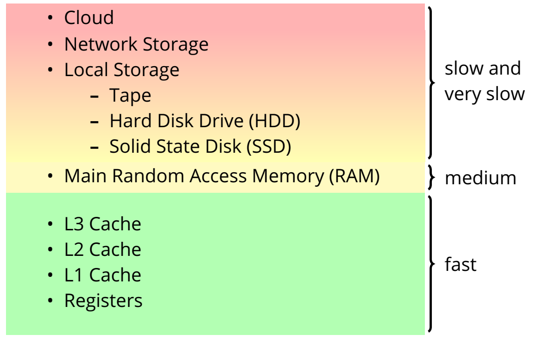


Figure 1.5: Basic memory hierarchy on a single processor system.

1.3.2 Shared Memory

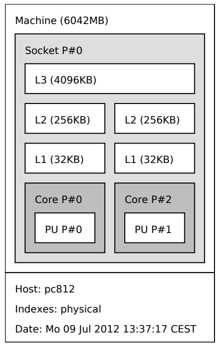


Figure 1.6: A sample dual core Xeon® setup

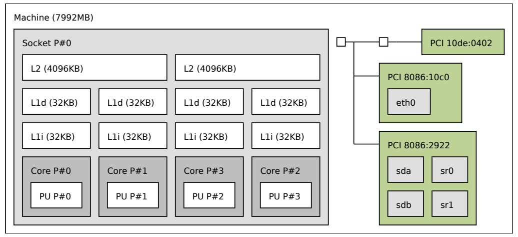


Figure 1.7: A sample Core™ 2 Quad setup

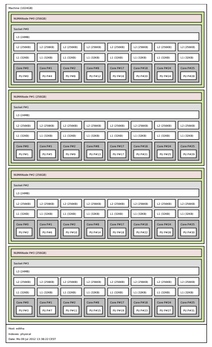


Figure 1.8: A four processor octa-core Xeon® system

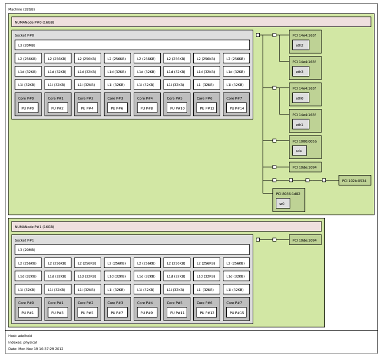


Figure 1.9: A dual processor octa-core Xeon® system

1.3.3 General Memory Setting

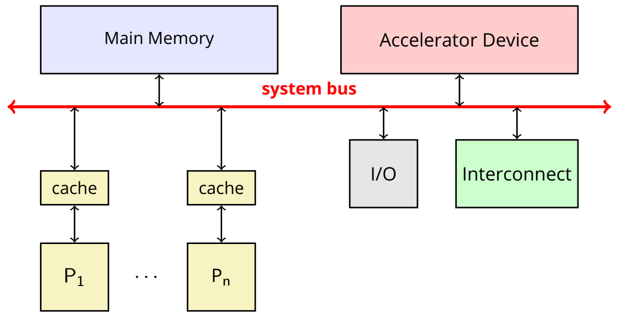


Figure 1.10: Schematic of a general parallel system

1.4 Communication Networks

The **Interconnect** in the last figure stands for any kind of Communication grid. This can be implemented either as

- local hardware interconnect,

or in the form of

- network interconnect.

In classical supercomputers the first was mainly used, whereas in today's cluster based systems often the network solution is used in the one form or the other.

1.4.1 Hardware

Network hardware is nowadays mainly one of the two types

MyriNet

- shipping since 2005
- transfer rates up to 10Gbit/s
- lower overhead compared to Ethernet
- less interference of the actual transfer and CPU

- thus higher throughput
- lost significance (2005 28.2% TOP500 down to 0.8% in 2011)

Infiniband

- transfer rates up to 300Gbit/s
- most relevant implementation driven by OpenFabrics Alliance[3], $\rightsquigarrow$ usable on Linux, BSD, Windows systems.
- features remote direct memory access (RDMA) $\rightsquigarrow$ reduced CPU overhead
- can also be used for TCP/IP communication

Omni-Path

- transfer rates up to 100Gbit/s
- introduced by Intel® in 2015/2016
- rising significance
- Intel®'s approach to address cluster of $> 10\,000$ nodes

1.4.2 Topologies

1. **Linear array:** nodes aligned on a string each being connected to at most two neighbors.
2. **Ring:** nodes are aligned in a ring each being connected to exactly two neighbors
3. **Complete graph:** every node is connected to all other nodes
4. **Mesh and Torus:** Every node is connected to a number of neighbors (2–4 in 2d mesh, 4 in 2d torus).
5. **k-dimensional cube / hypercube:** Recursive construction of a well connected network of 2^k nodes each connected to k neighbors. Line for two, square for four, cube fore eight.
6. **Tree:** Nodes are arranged in groups, groups of groups and so forth until only one large group is left, which represents the root of the tree.

[3] http://www.openfabrics.org/

CHAPTER 2

Performance Measures

Contents

2.1 Time Measurement and Operation Counts

2.1.1 The Single Processor Case

Definition 2.1:
In general we call the time elapsed between issuing a command and receiving its results the *runtime*, or *execution time* of the corresponding process. Some authors also call it elapsed time, or wall clock time.

In the purely sequential case it is closely related to the so called *CPU time* of the process. There the main contributions are:

- **user CPU time:** Time spent in execution of instructions of the process.
- **system CPU time:** Time spent in execution of operating system routines called by the process.
- **waiting time:** Time spent waiting for time slices, completion of I/O, memory fetches…

That means the time we have to wait for a response of the program includes the waiting times besides the CPU time. We have focused on minimizing the waiting times mainly caused by cache misses in part one of the lecture in winter term. Other waiting times caused by the overall load of the system are out of our reach anyway. Therefore, we will basically neglect this contribution for further considerations. The same holds for the system CPU time part. The time spent here strongly depends on the type and implementation of system calls in the underlying operating system, which we also can not influence. Only in multi-threading environments, where system calls (e.g. for storing intermediate results to disks) can be executed independent of the rest of the execution of the process, one should try and use threads (see Section 3.3.2) to minimize their influence.

In the following we have a closer look on the user CPU time contribution. A major part of this is due to the execution of different instructions issued by the program. The upcoming two sections will get into detail on the problems connected to varying instructions and the concepts of low and high level instructions.

2.1.2 Instructions: Timings and Counts

clock rate and cycle time

The *clock rate* of a processor tells us how often it can switch instructions per second. Closely related is the *(clock) cycle time*, i.e., the time elapsed between two subsequent clock ticks.

Example 2.2:
A CPU with a clock rate of 3.5 GHz $= 3.5 \cdot 10^9$ 1/s executes $3.5 \cdot 10^9$ clock ticks per second. The length of a clock cycle thus is

$$1/(3.5 \cdot 10^9)\,\text{s} = 1/3.5 \cdot 10^{-9} \cdot \text{s} \approx 0.29\,\text{ns}$$

Different instructions require different times to get executed. This is represented by the so called *cycles per instruction* (CPI) of the corresponding instruction. An average CPI is connected to a process A via $CPI(A)$.

This number determines the total user CPU time together with the number of instructions and cycle time via

$$T_{U\_CPU}(A) = n_{instr}(A) \cdot CPI(A) \cdot t_{cycle}$$

Clever choices of the instructions can influence the values of $n_{instr}(A)$ and $CPI(A)$. $\rightsquigarrow$ compiler optimization.

2.1.3 MIPS versus FLOPS

A common performance measure of CPU manufacturers is the *Million instructions per second (MIPS) rate*. It can be expressed as

$$MIPS(A) = \frac{n_{instr}(A)}{T_{U\_CPU}(A) \cdot 10^6} = \frac{r_{cycle}}{CPI(A) \cdot 10^6},$$

where r_{cycle} is the cycle rate of the CPU.

This measure can be misleading in high performance computing, since higher instruction throughput does not necessarily mean shorter execution time.

More common for the comparison in scientific computing is the rate of floating point operations (FLOPS) executed. The MFLOPS rate of a program A can be expressed as

$$MFLOPS(A) = \frac{n_{FLOPS}(A)}{T_{U\_CPU}(A) \cdot 10^6}\,[1/\text{s}],$$

with $n_{FLOPS}(A)$ the total number of FLOPS issued by the program A.

Note that not all FLOPS (see also Chapter 4 winter term) take the same time to execute. Usually divisions and square roots are much slower. The MFLOPS rate, however, does not take this into account.

2.1.4 CPU_Time versus Execution Time

On a parallel machine time measurement includes some notable side effects that are best explained by a simple example. The following is a simple script in MATLAB®.

Example 2.3 (A simple MATLAB test)**:**

Input:

```
ct0=0;
A=randn(1500);
```

```
tic
ct0=cputime;
pause(2)
toc
cputime-ct0

tic
ct0=cputime;
[Q,R]=qr(A);
toc
cputime-ct0
```

Output:

```
Elapsed time is 2.000208 seconds.

ans =

    0.0300

Elapsed time is 0.733860 seconds.

ans =

   21.6800
```

The results have been obtained in MATLABR2010b on a system with four octa-core Xeon® processors.

Obviously, in a parallel environment the CPU time can be much higher than the actual execution time elapsed between start and end of the process.

In any case, it can be much smaller, as well.

The first result is easily explained by the splitting of the execution time into user/system CPU time and waiting time. The process is mainly waiting for the `sleep` system call to return whilst basically accumulating no active CPU time.

The second result is due to the fact that the activity is distributed to several cores. Each activity accumulates its own CPU time and these are summed up to the total CPU time of the process.

2.2 Parallel Cost and Optimality

Definition 2.4 (Parallel cost and cost-optimality)**:**
The cost of a parallel program with data size n is defined as

$$C_p(n) = p * T_p(n).$$

Here $T_p(n)$ is the *parallel runtime* of the process, i.e., its execution time on p processors.

The parallel program is called *cost-optimal* if

$$C_p = T^*(n).$$

Here, $T^*(n)$ represents the execution time of the fastest sequential program solving the same problem.

In practice $T^*(n)$ is often approximated by $T_1(n)$. This is due to the fact that the optimal algorithm for a problem is often not available for comparison, or not available at all. The later basically means that a proof of existence for the algorithm exists, but is not constructive so that the actual algorithm fulfilling the optimal sequential cost is unknown.

2.3 Speedup

The *speedup* of a parallel program

$$S_p(n) = \frac{T^*(n)}{T_p(n)},$$

is a measure for the acceleration, in terms of execution time, we can expect from a parallel program. The speedup is strictly limited from above by p since otherwise the parallel program would motivate a faster sequential algorithm. A method to derive such a sequential algorithm is described in [18, Chapter 4].

In practice often the speedup is computed with respect to the sequential version of the code, i.e.,

$$S_p(n) \approx \frac{T_1(n)}{T_p(n)}.$$

2.4 Parallel Efficiency

Usually, the parallel execution of the work a program has to perform comes at the cost of certain management of subtasks. Their distribution, organization and interdependence leads to a fraction of the total execution, that has to be done extra.

Definition 2.5:
The fraction of work that has to be performed by a sequential algorithm as well is described by the *parallel efficiency* of a program. It is computed as

$$E_p(n) = \frac{T^*(n)}{C_p(n)} = \frac{S_p(n)}{p} = \frac{T^*}{p \cdot T_p(n)}.$$

The parallel efficiency obviously is limited from above by $E_p(n) = 1$ representing the perfect speedup of p. Since the values range from 0 to 1 the parallel efficiency is often also represented as a percent value, i.e., $E_p(n)$is identified with $100 * E_p(n)$.

2.5 Amdahl's Law

In many situations it is impossible to parallelize the entire program. Certain fractions remain that need to be performed sequentially. When a (constant) fraction f of the program needs to be executed sequentially, *Amdahl's law* describes the maximum attainable speedup.

The total parallel runtime $T_p(n)$ then consists of

- $f \cdot T^*(n)$ the time for the sequential fraction and
- $(1-f)/p \cdot T^*(n)$ the time for the fully parallel part.

The best attainable speedup can thus be expressed as

$$S_p(n) = \frac{T^*(n)}{f \cdot T^*(n) + \frac{1-f}{p} T^*(n)} = \frac{1}{f + \frac{1-f}{p}} \leqslant \frac{1}{f}.$$

2.6 Scalability of Parallel Programs

Question

Is the parallel efficiency of a parallel program independent of the number of processors p used?

The question is answered by the concept of *parallel scalability*. Scientific computing and HPC distinguish two forms of scalability:

- **strong scalability** captures the dependence of the parallel runtime on the number of processors for a fixed total problem size.
- **weak scalability** captures the dependence of the parallel runtime on the number of processors for a fixed problem size per processor.

CHAPTER 3

Multicore and Multiprocessor Systems

Contents

3.1 Symmetric Multiprocessing

Definition 3.1 (Symmetric Multiprocessing (SMP))**:**
The situation where two or more identical processing elements access a shared periphery (i.e., memory, I/O,. . .) is called *symmetric multiprocessing* or simply (SMP).

The most common examples are

- Multiprocessor systems, where the processing elements are the single processors,
- Multicore CPUs, where the processing elements are given by the single cores.

3.2 Memory Hierarchy

3.2.1 Basic Memory Layout

3.2.2 Uniform Memory Access (UMA)

UMA is a shared memory computer model, where

- one physical memory resource,
- is shared among all processing units,
- all having uniform access to it.

Especially, that means that all memory locations can be requested by all processors at the same time scale, independent of which processor performs the request and which chip in the memory holds the location.

Local caches one the single processing units are allowed. That means classical multicore chips are an example of a UMA system.

3.2.3 Non-Uniform Memory Access (NUMA)

Contrasting the UMA model in NUMA the system consists of

- one *logical* shared memory unit,
- gathered from two or more physical resources,
- each bound to (groups of) single processing units.

Due to the distributed nature of the memory, access times vary depending on whether the request goes to local or foreign memory.

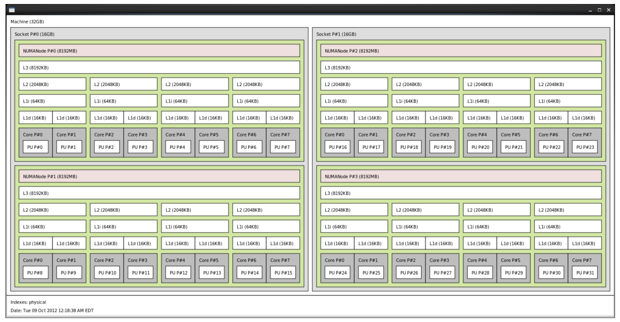


Figure 3.1: AMDs Bulldozer layout is a NUMA example.
By The Portable Hardware Locality (hwloc) Project (Raysonho@Open Grid Scheduler / Grid Engine) [see web page for license], via Wikimedia Commons

Examples are current multiprocessor systems with multicore processors per socket and a separate portion of the memory controlled by each socket, or "cluster on a chip" design processors like AMDs bulldozer series.

3.2.4 Cache Coherence

Definition 3.2 (cache coherence)**:**
The problem of keeping multiple copies of a single piece of data in the local caches of the different processors, that hold it, consistent is called *cache coherence* problem.

Cache coherence protocols:

- guarantee a consistent view of the main memory at any time.
- Several protocols exist.
- Basic idea is to invalidate all other copies whenever one of them is updated.

3.3 Processes and Threads

3.3.1 Multiprocessing

Definition 3.3 (Process)**:**
A computer program in execution is called a *process*.

A process consists of:

- the programs machine code,
- the program data worked on,
- the current execution state, i.e., the context of the process, register and cache contents, . . .

Each process has a separate address space in the main memory.

Execution time slices are assigned to the active processes by the operating system's (OS's) scheduler. A switch of processes requires exchanging the process context, i.e., a short execution delay.

Multiple processes may be used for the parallel execution of compute tasks.

On Unix/Linux systems the `fork()` system call can be used to generate child processes. Each child process is generated as a copy of the calling parent process. It receives an exact copy of the address space of the parent and a new unique process ID (PID).

Communication between parent and child processes can be implemented via sockets or files, which usually leads to large overhead for data exchange.

3.3.2 Threading

Definition 3.4 (Thread)**:**
In the thread model, a process may consist of several execution sub-entities, i.e control flows, progressing at the same time. These are usually called *threads*, or *lightweight processes*.

All threads of a process share the same address space. Thus communication and thread generation is simple and fast. There is no need to exchange data since everything is shared anyway.

Two types of implementations exist:

- **user level threads:**
  - administration and scheduling in user space,
  - threading library maps the threads into the parent process,

   - quick task switches avoiding the OS.
- **kernel threads:**
   - administration and scheduling by OS kernel and scheduler,
   - different threads of the same process may run on different processors,
   - blocking of single threads does not block the entire process,
   - thread switches require OS context switches.

Here we concentrate on POSIX threads, or Pthreads. These are available on all major OSes. The actual implementations range from user space wrappers (`pthreads-w32` mapping pthreads to windows threads) to lightweight process type implementations (e.g. Solaris 2).

3.3.3 Mapping of user level threads to kernel threads or processes

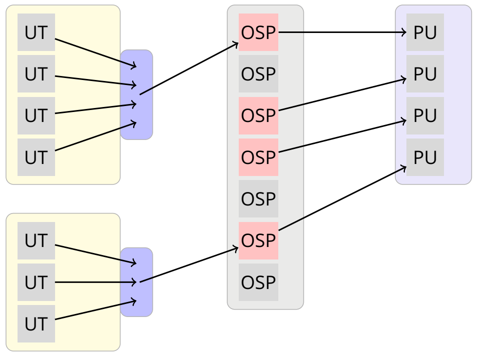


Figure 3.2: N:1 mapping for OS incapable of kernel threads

3.3.4 Properties and Problems

Parallel versus concurrent execution

1. Often the two notions *parallel* and *concurrent* execution are used as synonyms of each other. In fact concurrent is more general.
2. The parallel execution of a set of tasks requires parallel hardware on which they can be executed simultaneously.

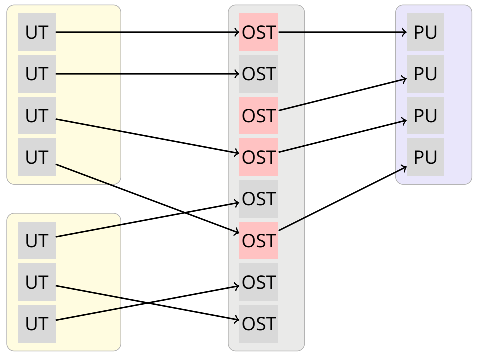


Figure 3.3: 1:1 mapping of user threads to kernel threads

3. The concurrent execution only requires a quasi parallel environment that allows all tasks to be in progress at the same time.
4. That means "parallel" execution defines a subset of "concurrent" execution.

Definition 3.5 (race condition)**:**
When several threads/processes of a parallel program have read and *write access* to a *common* piece of *data*, access needs to be *mutually exclusive*. Failure to ensure this, leads to a ***race condition***, where the *final value depends* on the sequence of *uncontrollable/random events*. Usually data corruption is then unavoidable.

Example 3.6:
Two possible execution orders due to random external events (e.g. in the operating system.)

Thread 1	Thread 2	value
		0
`read`		0
`increment`		0
`write`		1
	`read`	1
	`increment`	1
	`write`	2

Thread 1	Thread 2	value
		0
`read`		0
	`read`	0
`increment`		0
`write`		1
	`increment`	1
	`write`	1

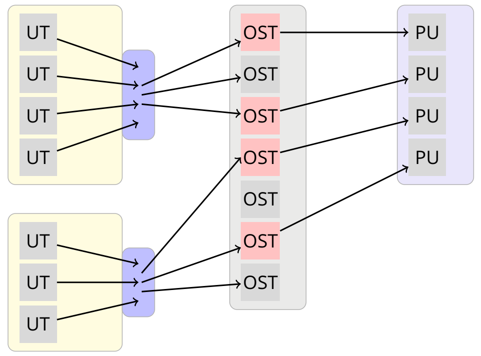


Figure 3.4: N:M mapping of user threads to kernel threads with library thread scheduler

3.3.5 Protection of critical regions

Definition 3.7 (semaphore)**:**
A *semaphore* is a simple flag (binary semaphore) or a counter (counting semaphore) indicating the availability of shared resources in a critical region.

Definition 3.8 (mutual exclusion variable (mutex))**:**
The *mutual exclusion variable*, or shortly *mutex* variable, implements a simple locking mechanism regarding the critical region. Each process/thread checks the lock upon entry to the region. If it is open the process/thread enters and locks it behind. Thus, all other processes/threads are prevented from entering and the process in the critical region has exclusive access to the shared data. When exiting the region the lock is opened.

Both the above definitions introduce the programming models. Actual implementations may be more or less complete. For example the `pthreads`-implementation lacks counting semaphores.

Definition 3.9 (deadlock)**:**
A ***deadlock*** describes the unfortunate situation, when semaphores, or mutexes have not, or have inappropriately been applied such that no process/thread is able to enter the critical region anymore and the parallel program is unable to proceed.

3.3.6 Dining Philosophers

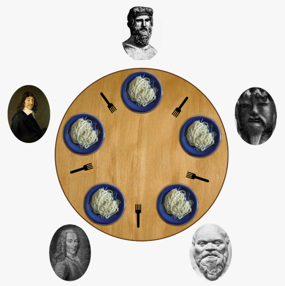


Figure 3.5: The dining philosophers problem

Example 3.10 (dining philosophers)**:**

- Each philosopher alternatingly eats or thinks,
- to eat the left and right forks are both required,
- every fork can only be used by one philosopher at a time,
- forks must be put back after eating.

simple solution attempt

- think until the left fork is available; when it is, pick it up;
- think until the right fork is available; when it is, pick it up;
- when both forks are held, eat for a fixed amount of time;
- then, put the right fork down;
- then, put the left fork down;
- repeat from the beginning.

All philosophers decide to eat at the same time ⇒ deadlock.

More sophisticated solutions avoiding the deadlocks have been found since [4]. Three of them are also available on Wikipedia[1].

3.4 POSIX Threads

3.4.1 Basics

Common to all the following commands:

Compiling and linking needs to be performed with `-pthread` (for GNU and Intel compilers).

The pthread functions and related data types are made available in a C program using:

```
#include <pthread.h>
```

3.4.2 Creation of threads

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

- `thread` unique identifier to distinguish from other threads,
- `attr` attributes for determining thread properties. `NULL` means default properties,
- `start_routine` pointer to the function to be started in the newly created thread,
- `arg` the argument of the above function.

Note that only a single argument can be passed to the threads start function.

The argument of the start function is a `void` pointer. We can thus define:

```
  struct point3d{ double x,y,z; };
  struct norm_args{
    struct point3d *P;
    double norm;
  };
  struct norm_args args;
```

[1] http://en.wikipedia.org/wiki/Dining_philosophers_problem

and upon thread creation pass

```
err=pthread_create(tid, NULL, norm, (void *) &args);
```

to a start function

```
void *norm(void *arg) {
  struct norm_args *args=(struct norm_args *)arg;
  struct point3d *P;
  P = args->P;
  args->norm = P->x * P->x + P->y * P->y + P->z * P->z;
  return NULL;
};
```

```
int main(int argc, char* argv[]){
  pthread_t tid1,tid2;

  struct point3d point;
  struct norm_args args;

  args.P = &point;

  point.x=10; point.y=10; point.z=0;
  pthread_create(&tid1, NULL, norm, &args);

  point.x=20; point.y=20; point.z=-50;
  pthread_create(&tid2, NULL, norm, &args);

  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
}
```

Depending on the execution of thread `tid1` the argument `point` may get overwritten before it has been fetched, the analogue holds for the `norm` argument inside the function.

3.4.3 Exiting threads and waiting for their termination

Pthreads can exit in different forms:

- they return from their start function,
- they call `pthread_exit()` to cleanly exit,
- they are aborted by a call to `pthread_cancel()`,
- the process they are associated to is terminated by an `exit()` call.

```
int pthread_exit(void *retval);
```

- `retval` return value of the exiting thread to the calling thread,

- threads exit implicitly when their start function is exited,
- the return value may be evaluated from another thread of the same process via the `pthread_join()` function,
- after the last thread in a process exits the process terminates calling `exit()` with a zero return value. Only then shared resources are released automatically.

```
int pthread_join(pthread_t thread, void **retval);
```

- Waits for a thread to terminate and fetches its return value.
- `thread` the identifier of the thread to wait for,
- `retval` destination to copy the return value (if not `NULL`) to.

3.5 Pthread coordination mechanisms

3.5.1 Mutex and condition variables

The Pthread standard supports four types of synchronization and coordination facilities:

1. `pthread_join()`; we have seen this function above
2. Mutex variable functions for handling mutexes as defined above
3. Condition variable functions treat a condition variable that can be used to indicate a certain event in which the threads are interested. Condition variables may be used to implement semaphore like structures and triggers for special more complex situation that require the threads to act in a certain way.
4. `pthread_once()` can be used to make sure that certain initializations are performed by one and only one thread when called by multiple ones.

3.5.2 Mutex variables

A mutex variable needs to be initialized before its first use. This can be done in two ways.

Dynamic initialization:

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict
                          attr);
```

Static/Macro initialization:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- `mutex` is the mutex variable to be initialized
- `attr` can be used to adapt the mutex properties, as for the pthreads `NULL` gives the default attributes,
- `restrict`[2] is a C99-standard keyword limiting the pointer aliasing features and guiding compilers and aiding in the caching optimization.
- initialization may fail if the system has insufficient memory (error code `ENOMEM`) or other resources (`EAGAIN`)

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- If `mutex` is unlocked the function returns with the mutex in locked state,
- If `mutex` is already locked the execution is blocked until the lock is released and it can proceed as above,
- Four types of mutexes are defined:
  - `PTHREAD_MUTEX_NORMAL`
  - `PTHREAD_MUTEX_ERRORCHECK`
  - `PTHREAD_MUTEX_RECURSIVE`
  - `PTHREAD_MUTEX_DEFAULT`

  All of them show different behavior when locked mutexes should again be locked by the same thread or a thread tries to unlock a previously unlocked mutex and similar unintended situations. This, especially, regards error handling and deadlock detection.

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- The function is equivalent to `pthread_mutex_lock()`, except that it returns immediately in any case.
- Success or failure are determined from the return value.
- If the mutex type is `PTHREAD_MUTEX_RECURSIVE` the lock count is increased by one and the function returns success.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- the function releases the lock
- what exactly “release” means, depends on the properties of the mutex variable

[2]For further information refer to the C99 standard [12] or the brief description at https://en.wikipedia.org/wiki/Restrict

- e.g., for type PTHREAD_MUTEX_RECURSIVE mutexes it means that the counter is decreased by one and they become available once it reaches zero
- if the mutex becomes available, i.e., unlocked by the function call and there are blocked threads waiting for it, the threading policy decides which thread acquires mutex next.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- destroys the mutex referenced by mutex
- the destroyed mutex then becomes uninitialized
- pthread_mutex_init() can be used to initialize the same mutex variable again
- if mutex is locked or referenced, pthread_mutex_destroy() fails with error code EBUSY

3.5.3 Avoiding mutex triggered deadlocks

Example 3.11 (A deadlock situation when locking multiple mutexes)**:**
Problem:

- Consider two mutex variables ma and mb, as well as two threads T1 and T2.
- T1 locks ma first and then mb,
- T2 locks mb first and then ma,
- If T1 is interrupted by the scheduler after locking ma, but before locking mb and in the meantime T2 succeeds in locking it, then the classical deadlock occurs.

Locking hierarchy solution: The basic idea, here, is that all threads need to lock the critical mutexes in the same order. This can easily be guaranteed by hierarchically ordering the mutexes.

Back off strategy solution: When we want to keep the differing locking orders, we may use pthread_mutex_trylock() with a back off strategy.

- Locking is tried in the desired order,
- when a trylock fails, the thread unlocks all previously locked mutexes (it backs off of the protected resources),
- after the back off it starts over from the first one.

3.5.4 Condition variables

Dynamic initialization:

```
int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict
                        attr);
```

Static/Macro initialization:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- cond the condition to be initialized
- attr can be used to adapt the condition properties, as for the pthreads NULL gives the default attributes,
- restrict: see pthread_mutex_init()
- every condition variable is associated to a mutex.

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

- destroys the condition variable referenced by cond
- the destroyed condition then becomes uninitialized
- pthread_cond_init() can reinitialize the same condition variable
- if cond is blocking threads when destroyed the standard does not specify the behavior of pthread_cond_destroy().

```
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
```

- assumes that mutex was locked before by the calling thread,
- results in the thread getting blocked and at the same time (atomically) releasing mutex
- another thread may evaluate this to wake up the now blocked thread (see pthread_cond_signal())
- upon waking up the thread automatically tries to gain access to mutex again,
- if it succeeds it should test the condition again to check whether another thread changed it in the meantime.

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- if no thread is blocked on the condition variable cond there is no effect,
- otherwise, *one* of the waiting threads is woken up and proceeds as described above.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- wakes up *all* threads blocking on `cond`,
- all of them try to acquire the associated mutex,
- only one of them can succeed,
- the others get blocked on the mutex now.

```
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict
                              abstime);
```

- equivalent to `pthread_cond_wait()` except that it only blocks for the period specified by `abstime`,
- if the thread did not get signaled or broadcast before `abstime` expires it returns with error code `ETIMEDOUT`.

3.5.5 A counting semaphore for Pthreads

Semaphores are not available in the POSIX Threads standard.

However, they can be created using the existing mechanisms of mutexes and conditions.

A counting semaphore should be a data type that acts like a counter with non-negative values and for which two operations are defined:

1. A signal operation increments the counter and wakes up a task blocked on the semaphore if one exists.
2. A wait operation simply decrements the counter if it is positive. If it was zero already the thread is blocking on the semaphore.

- data structure for the semaphore:

```
typedef struct _sema_t{
  int count;
  pthread_mutex_t m;
  pthread_cond_t c;
} sema_t;
```

- the initialization

```
void InitSema(sema_t *ps){
  pthread_mutex_init(&ps->m,NULL);
  pthread_cond_init(&ps->c,NULL);
}
```

- and the cleanup

```
void CleanupSema(void *arg){
  pthread_mutex_unlock((pthread_mutex_t *) arg);
}
```

```
void ReleaseSema(sema_t *ps){ // signal operation
  pthread_mutex_lock(&ps->m) ;
  pthread_cleanup_push(CleanupSema,&ps->m);
  {
    ps->count++;
    pthread_cond_signal(&ps->c) ;
  }
  pthread_cleanup_pop ( 1 ) ;
}


void AcquireSema(sema_t *ps){ // wait operation
  pthread_mutex_lock(&ps->mutex);
  pthread_cleanup_push(CleanupSema,&ps->m);
  {
    while(ps->count==0)
      pthread_cond_wait(&ps->c,&ps->m) ;
    ps->count--;
  }
  pthread_cleanup_pop(1);
}
```

3.5.6 A typical application example for semaphores

Example 3.12 (Producer/Consumer queue buffer protection)**:**
Basic setup:

- A buffer of fixed size n is shared by
- a producer thread generating entries and storing them in the buffer if it is not full,
- a consumer thread removing entries from the same buffer for further processing unless it is empty.

For the realization of the protected access two semaphores are required:

1. Number of entries occupied (initialized by 0),
2. Number of free entries (initialized by n).

The Mechanism works for an arbitrary number of producers and consumers.

3.5.7 Coordination models for the cooperation of threads

1. Master/Slave model:
    - A master thread is controlling the execution of the program,
    - the slave threads are executing the work.
2. Client/Server model:
    - Client threads produce requests,
    - Server threads execute the corresponding work.
3. Pipeline model:
    - Every thread (except for the first and last in line) produces output that serves as input for another thread,
    - after a startup phase (filling the pipeline) the parallel execution is achieved.
4. Worker model:
    - equally privileged workers organize their workload,
    - an important variant is the task pool treated as detailed example next.

3.6 Task Pools

3.6.1 Basic idea of the task pool

Idea:

Creation of a parallel threaded program that can dynamically schedule tasks on the available processors. This enables us to work on highly irregular problems like adaptive or hierarchical algorithms, as well as unbalanced problems like the sparse matrix vector product with strongly varying numbers of elements per row, much more efficiently.

Key ingredients in the approach are:

- usage of a fixed number of threads
- organization of the pending tasks in a task pool,
- threads fetch the tasks from the pool and execute them leading to a dynamic assignment of the work load.

Main advantages

- automatic dynamic load balancing among the threads
- comparably small overhead for the administration of threads

3.6.2 Implementation of a basic task pool

Data structures

- data strucutre for one task:

```
typedef struct _work_t{
    void (*routine) (void*); //worker function to call
    void* arg ;
    struct _work_t *next;
} work_t ;
```

- data structure for the task pool:

```
typedef struct _tpool_t{
    int num_threads ; // number of threads
    int max_size, curr_size; // max./cur. number of
       tasks in pool
    pthread_t *threads; //array of threads
    work_t *head , *tail; // start/end of the task
       queue
    pthread_mutex_t lock; //access control for the task
        pool
    pthread_cond_t not_empty ; // tasks are available
    pthread_cond_t not_full ; // tasks may be added
} tpool_t ;
```

Initialization

```
tpool_t *tpool_init(int num_threads , int max_size){
  int i;
  tpool_t *tpl;

  tpl=(tpool_t *) malloc (sizeof(tpool_t));
  tpl->num_threads=num_threads ;
  tpl->max_size=max_size ;
  tpl->cur_size=0;
  tpl->head=tpl->tail=NULL;

  pthread_mutex_init(&tpl->lock, NULL);
  pthread_cond_init(&tpl->not_empty, NULL);
  pthread_cond_init(&tpl->not_full, NULL);
  tpl->threads=(pthread_t *) malloc(num_threads *sizeof(
      pthread_t));
  for(i=0; i<num_threads; i++)
    pthread_create(tpl->threads+i, NULL, tpool_thread, (
        void *)tpl) ;
  return tpl;
}
```

Worker Threads

```
void *tpool_thread(void *vtpl){
  tpool_t *tpl=(tpool_t *) vtpl;
  work_t *wl ;

  for ( ; ; ) {
    pthread_mutex_lock(&tpl->lock);
    while(tpl->cur_size==0)
      pthread_cond_wait(&tpl->not_empty , &tpl->lock);
    wl=tpl->head; tpl->cur_size--;
    if(tpl->cur_size==0)
      tpl->head=tpl->tail=NULL;
    else tpl->head = wl->next;
    if (tpl->cur_size==tpl->max_size-1) // pool full
      pthread_cond_signal(&tpl->not_full);
    pthread_mutex_unlock(&tpl->lock);
    (*(wl->routine)) (wl->arg);
    free(wl);
  }
}
```

Task insertion

```
void tpool_insert(tpool_t *tpl, void(*f) (void*), void *arg
   ){
  work_t *wl ;

  pthread_mutex_lock(&tpl->lock);
  while(tpl->cur_size==tpl->max_size)
    pthread_cond_wait(&tpl->not_full, &tpl->lock);
  wl=(work_t *) malloc(sizeof(work_t));
  wl->routine=f; wl->arg=arg; wl->next=NULL ;
  if( tpl->cur_size==0){
    tpl->head=tpl->tail=wl;
    pthread_cond_signal(&tpl->not_empty);
```

```
  }
  else{
    tpl->tail->next=wl; tpl->tail=wl;
  }
  tpl->cur_size++;
  pthread_mutex_unlock(&tpl->lock);
}
```

3.7 Shared Memory Blocks

3.7.1 General shared memory blocks

In contrast to Threads, different processes do not share their address space. Therefore, different ways to communicate in multiprocessing applications are necessary.

One possible way are shared memory objects. Unix-like operating systems provide at least one of:

- **old:** System V Release 4 (SVR4) Shared Memory[3]
- **new:** POSIX Shared Memory[4].

Both techniques implement shared memory objects, like common memory, semaphores and message queues, which are accessible from different applications with different address spaces.

3.7.2 POSIX Shared Memory

Common Memory Locations

- They are used to share data between applications.
- They are managed by the kernel and not by the application.
- Each location is represented as a file in `/dev/shm/`.
- They are handled like normal files.
- They are created using `shm_open` and mapped to the memory using `mmap`.
- Exist as long as no application deletes them.
- Even when the creating program exits they stay available,

[3] *System V Interface Definition, AT&T Unix System Laboratories, 1991*
[4] *IEEE Std 1003.1-2001Portable Operating System Interface System Interfaces*

- See manpage: `man 7 shm_overview`.

POSIX Semaphores

- Counting semaphores are available form different address spaces.
- They correspond to `pthread_mutex_*` in threaded applications.
- They are represented as a file in `/dev/shm/sem.*`.
- See manpage: `man 7 sem_overview`.

Message Queues

- They represent a generalized Signal concept which can transfer a small payload (2 to 4 KiB).
- They correspond to `pthread_cond_*` in threaded applications.
- They can be represented as file in `/dev/mqueue`.
- See manpage: `man 7 mq_overview`.

3.8 Open Multi-Processing (OpenMP)

3.8.1 This is OpenMP

Mission

> *"The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms. Jointly defined by a group of major computer hardware and software vendors, OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer."* [a]

[a]The Mission statement from http://www.openmp.org/about/about-us/

The OpenMP Architecture Review Board (ARB)

The ARB is a non-profit enterprise owning the OpenMP brand and responsible for overseeing, producing and approving the OpenMP standards.

Members of the ARB include: **(status: April 22, 2021)**

- Hardware manufacturers: e.g. AMD, NVidia, IBM, NEC, HP
- Software companies: e.g. Oracle, SuSe
- Compute centers: e.g. Sandia NL, Lawrence Livermore NL, CSC, Leibniz Supercomputing Center, Barcelona Supercomuting Center
- Universities: e.g. University of Tennessee, RWTH Aachen, Uni Basel, Uni Bristol

Complete list available at: http://www.openmp.org/about/members/

History

Oct. 1997 OpenMP 1.0 for Fortran,

Oct. 1998 OpenMP 1.0 for C/C++,

Nov. 2000 OpenMP 2.0 for Fortran,

March 2002 OpenMP 2.0 for C/C++,

May 2005 OpenMP 2.5 (first joint Fortran/C/C++ version),

May 2008 OpenMP 3.0,

Sept. 2011 OpenMP 3.1,

July 2013 OpenMP 4.0,

Nov. 2015 OpenMP 4.5,

Nov. 2018 OpenMP 5.0,

Nov. 2020 OpenMP 5.1 (current standard)

3.8.2 What OpenMP can do for us

- Easy shared memory parallel adaption of existing sequential codes
- Easy preservation of sequential implementations
- Easy porting to different platforms and compilers
- Parallel implementation of only fragments
- No extra runtime environment
- Easy to learn and apply

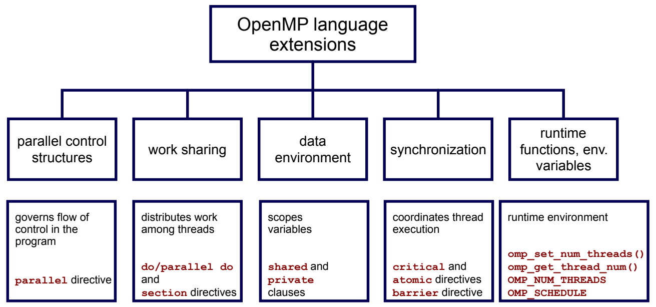

OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
governs flow of control in the program
parallel directive
distributes work among threads
do/parallel do and section directives
scopes variables
shared and private clauses
coordinates thread execution
critical and atomic directives barrier directive
runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

Figure 3.6: Classification of the OpenMP extensions by tasks of the elements (Image Source: https://commons.wikimedia.org/wiki/File:OpenMP_language_extensions.svg).

3.8.3 What OpenMP is NOT for!

- Distributed memory parallel systems (by itself)
- Most efficient use of shared memory systems
- Automatic checking for (data dependencies,) data conflicts, race conditions, or deadlocks
- Automatic synchronization of input and output

3.8.4 The Structure of the Standard

The standard divides the extensions into four classes:

1. **Directives:** Basic control structures that initialize/end the parallel environments
2. **Clauses:** Fine tuning parameters added to the directives.
3. **Environment Variables:** Variables in the calling shell used to control the parallel environment without recompilation.
4. **Runtime Library Routines:** Runtime usable functions for the determination and modification of parameters of the parallel environment.

3.8.5 OpenMP directives

The `#pragma` directive was introduced in C89 as the universal method for extending the space of directives. It was further standardized in C99, where especially the token `STDC` was reserved for standard C extensions.

In part 1 of the Scientific Computing lecture we have seen the floating point environment for, e.g., checking the exception flags in IEEE arithmetic:

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
/* starting here the compiler needs to assume we are
   accessing the
floating point status and mode registers*/
```

OpenMP is an extension in the sense of C89 and enabled by the

```
#pragma omp
```

preprocessor directive. It applies to the succeeding structural code block. That means, it applies to the next single line of code, a block encapsulated by a C control structure (e.g. a `for`-loop), or a portion of code enclosed by `{}`.

Compilers that do not know the `omp` pragma simply ignore it. The following switches enable OpenMP support for your code compilation:

GNU GCC	-fopenmp
Intel ICC	-qopenmp
LLVM CLANG	-fopenmp
IBM XLC	-qsmp
PGI	-mp

Otherwise the `omp` pragmas are ignored and the sequential code version is compiled.

A list of compilers supporting OpenMP can be found at https://www.openmp.org/resources/openmp-compilers-tools/

The following versions of the OpenMP standard are currently supported by the GCC compiler (April 2017):

OpenMP 2.5 starting from GCC 4.2.0,

OpenMP 3.0 starting from GCC 4.4.0,

OpenMP 3.1 starting from GCC 4.7.0,

OpenMP 4.0 starting from GCC 4.9.1,

OpenMP 4.5 starting from GCC 6.1.0 (except of `gfortran` (planned for GCC 11)).

In the following we give a brief list of the most classic directives with their most basic clauses. The recent standards extend, both, the list of directives and the available clauses per directive. Unless explicitly specified differently, this list represents the status of OpenMP 3.0.

omp parallel directive. The *parallel* construct initializes a group of threads and starts parallel execution:

```
#pragma omp parallel [clause[[,]clause]...]
```

The clauses can be used to influence the behavior of the parallel execution. They will be explained in Section 3.8.6.

Available clauses for `parallel`:

- `if(scalar expression)`
- `num_threads(integer expression)`
- `default(shared| none)`
- `private(list)`
- `firstprivate(list)`
- `shared(list)`
- `copyin(list)`
- `reduction(operation:list)`

Example 3.14 (A minimal OpenMP parallel "hello world" program)**:**

```
#include <stdio.h>

int main(void)
{
#pragma omp parallel
    printf("Hello,␣world.\n");
  return 0;
}
```

Example 3.14 automatically lets OpenMP tune the number of threads used to the number of available processors. Afterward the parallel execution environment is started and all threads execute the `printf` statement.

omp loop directive. The `loop` construct specifies that the iterations of the loop should be distributed among the active threads.

```
#pragma omp for [clause[[,]clause]...]
  for loops
```

The `for`-loop construct needs to be used inside a structured code block of a `parallel` construct.

Available clauses for `for`:

- `private(list)`
- `firstprivate(list)`
- `lastprivate(list)`
- `reduction(operator:list)`
- `schedule(kind[,chunk_size])`
- `collapse(n)`
- `ordered`
- `nowait`

omp parallel loop directive. Since often the `parallel` environment is used to introduce a `for`-loop construction only, a shortcut `parallel for` exists for this special task

```
#pragma omp parallel for [clause[[,] clause]...]
```

With the exception of the `nowait` clause all clauses accepted by `parallel` and `for` can be used with `parallel for` with the identically same behaviors and restrictions.

Example 3.15 (OpenMP parallel vector triad)**:**

```
double triad(double *a, double *b, double *c, double *d,
    int length){
  int i,j;
  const int repeat=100;
  double start, end;

  get_walltime(&start);
  for (j=0; j<repeat; j++){
#pragma omp parallel for
    for (i=0 ; i<length; i++){
      a[i]=b[i] + c[i] * d[i];
    } /*end of parallel section*/
  }
  get_walltime(&end);
  return repeat*length*2.0 / ((end-start) * 1.0e6); /*
     return MFLOPS */
}
```

Note that loop counters are protected automatically.

omp sections directive. When different tasks are to be distributed among the encountering team of threads the `sections` construct can be used. Each section will be executed only once by one thread in the team.

```
#pragma omp sections [clause[[,] clause]...]
{
  [#pragma omp section]
    structured code block
  [#pragma omp section]
    structured code block
  ...
}
```

Available clauses for `sections`:

- `private(list)`
- `firstprivate(list)`
- `lastprivate(list)`
- `reduction(operator:list)`
- `nowait`

omp parallel sections directive. Analogous to the `loop` constructs, also `sections` can be used only inside a `parallel` construct. The `parallel sections` construct merges them for easier use

```
#pragma omp parallel sections [clause[[,] clause]...]
{
  [#pragma omp section]
    structured code block
  [#pragma omp section]
    structured code block
  ...
}
```

Available clauses are those available for `parallel` and `sections` with the exception of `nowait`, as in the case of `for`.

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

```
#define N     50

int main (int argc, char *argv[]) {
  int i, nthrd, tid;
  float a[N], b[N], c[N], d[N];

  /* Some initializations */
  for (i=0; i<N; i++) {
    a[i] = i * 1.5;
    b[i] = i + 42.0;
    c[i] = d[i] = 0.0;
  }
  /* Start 2 threads */
#pragma omp parallel shared(a,b,c,d,nthrd) private(i,tid)
   num_threads(2)
{
  tid = omp_get_thread_num();
  if (tid == 0) {
    nthrd = omp_get_num_threads();
    printf("Number␣of␣threads␣=␣%d\n", nthrd);
  }
  printf("Thread␣%d␣starting...\n",tid);
#pragma omp sections
    {
#pragma omp section
      {
        printf("Thread␣%d␣doing␣section␣1\n",tid);
        for (i=0; i<N; i++) {
          c[i] = a[i] + b[i];
        }
        sleep(tid+2);  /* Delay the thread for a few
           seconds */
      } /* End of first section */

#pragma omp section
      {
        printf("Thread␣%d␣doing␣section␣2\n",tid);
        for (i=0; i<N; i++) {
          d[i] = a[i] * b[i];
        }
        sleep(tid+2);   /* Delay the thread for a few
           seconds */
      } /* End of second section */
    }  /* end of sections */
    printf("Thread␣%d␣done.\n",tid);
  }  /* end of omp parallel */

/* Print the results */
  printf("c:␣␣");
  for (i=0; i<N; i++) {
    printf("%.2f␣", c[i]);
  }
  printf("\n\nd:␣␣");
  for (i=0; i<N; i++) {
    printf("%.2f␣", d[i]);
  }
  printf("\n");
  exit(0);
}
```

`omp single` directive. A construct that makes sure that a structured code block is executed by only one thread (not necessarily the master thread) in a team of threads is given by the `single` directive.

```
#pragma omp single [clause[[,] clause]...]
```

Available clauses for the `single` construct are:

- `private(list)`
- `firstprivate(list)`
- `lastprivate(list)`
- `nowait`

Example 3.16 (OpenMP 4.5 Example 1.11 — single1.c)**:**

```
#include <stdio.h>

void work1() {}
void work2() {}
void main()
{
#pragma omp parallel
  {
  #pragma omp single
      printf("Beginning␣work1.\n");
  work1();
  #pragma omp single
      printf("Finishing␣work1.\n");
  #pragma omp single nowait
      printf("Finished␣work1␣and␣beginning␣work2.\n");
  work2();
  }
}
```

`omp master` directive. The `master` construct specifies a structured block that is executed by the master thread of the team.

```
#pragma omp master
```

The following structured block is only executed by the master thread of the parallel team. There is no synchronization on entry or on exit with the other threads.

Example 3.17 (OpenMP 4.5 Example 1.13 — master1.c)**:**

```
void master_example( float* x, float* xold, int n, float
   tol ){
  int c = 0, i, toobig;   loat error, y;
  #pragma omp parallel
  {
    do{
      #pragma omp for private(i)
      for( i = 1; i < n-1; ++i ){ xold[i] = x[i]; }
      #pragma omp single
      toobig = 0;
      #pragma omp for private(i,y,error) reduction(+:toobig
         )
      for( i = 1; i < n-1; ++i ){
        y = x[i];
        x[i] = average( xold[i-1], x[i], xold[i+1] );
        error = y - x[i];
        if( error > tol || error < -tol ) ++toobig;
      }
      #pragma omp master
      { printf( "iteration␣%d,␣toobig=%d\n", ++c, toobig );
          }
    }while( toobig > 0 );
  }
}
```

`omp tasks` directive. The OpenMP task construct (introduced with OpenMP 3.0) allows to parallelize irregular algorithms. The task construct inserts a piece of work into a thread pool running in the background (see Section 3.6).

```
#pragma omp task [clause[[,] clause]...]
structured code block
```

Available clauses for `task` (not complete):

- `if(scalar-expression)`

- `final(scalar-expression)`
- `default(shared | none)`
- `private(list)`
- `firstprivate(list)`
- `shared(list)`
- `priority(priority-value)`

Using the `taskwait` directive:

```
#pragma omp taskwait
```

one can wait for the completion of all previously created tasks at any position inside a parallel region in order to synchronize the parallel execution.

Due to the fact that tasks are running in the background they are mostly emitted by a single thread or a sequential code block. Therefore, mostly the `single` and `master` directives are used.

Tasks have to be defined inside an OpenMP `parallel` region. The end of the parallel region, unless it is used with the `nowait` clause, is an implicit synchronization point and the program waits until all tasks created inside the parallel region are finished.

The support to describe data dependencies between tasks is one of the most beneficial features of the **OpenMP 4** standard. For scientific computing this means that algorithms relying on dependency-graphs can be parallelized without using other third-party code or libraries.

> *"Although we expect to see DAG-based models widely adopted, changes in other parts of the software ecosystem will inevitably affect the way that that model is implemented. The appearance of DAG scheduling constructs in the OpenMP 4.0 standard offers a particularly important example of this point. [...] However, the inclusion of DAG scheduling constructs in the OpenMP standard, along with the rapid implementation of support for them (with excellent multithreading performance) in the GNU compiler suite, throws open the doors to widespread adoption of this model in academic and commercial applications for shared memory."* [a]
>
> [a]Jack Dongarra et. al., Numerical Algorithms and Libraries at Exascale

The data dependencies are defined using the `depend` clause during the task creation:

```
#pragma omp task depend(direction:list) [depend(direction:
    list)] [clauses...]
structured code block
```

Each `depend` clause consists of a data-flow direction and a list of identifiers. Possible directions are:

- `in` — The identifiers are **input** dependencies.
- `out` — The identifiers are **output** dependencies.
- `inout` — The identifiers are **input** and **output** dependencies.

The list of identifiers is a comma separated list of variables from which a pointer can be created.

Tasks with a common `inout` or `output` dependencies are executed in the order as they are created.

Example 3.18 (OpenMP 4.5 Example 3.3.4 — task_dep4.c)**:**

```
#include <stdio.h>
int main() {
  int x = 1;
  #pragma omp parallel
    #pragma omp single
  {
    #pragma omp task shared(x) depend(out: x)
    x = 2;
    #pragma omp task shared(x) depend(in: x)
    printf("x␣+␣1␣=␣%d.␣", x+1);
    #pragma omp task shared(x) depend(in: x)
    printf("x␣+␣2␣=␣%d\n", x+2);
  }
  return 0;
}
```

Array elements, e.g. `y[i]` or `A[i+ldA*j]`, are also valid identifiers in the `depend` clause. Intervals on arrays, like `A[i:j]`, are also allowed. Refer to the "array sections" section in the standard for details on the latter.

Example 3.19 (OpenMP 4.5 Example 3.3.5 — task_dep5.c)**:**

```
// Assume BS divides N perfectly
```

```
void matmul_depend(int N, int BS, float A[N][N], float B[N
    ][N], float C[N][N] )
{
  int i, j, k, ii, jj, kk;
  for (i = 0; i < N; i+=BS) {
    for (j = 0; j < N; j+=BS) {
      for (k = 0; k < N; k+=BS) {
        #pragma omp task private(ii, jj, kk) firstprivate(i
            ,j,k) \
          depend ( in: A[i][k], B[k][j] ) \
          depend ( inout: C[i][j] )
        for (ii = i; ii < i+BS; ii++ )
          for (jj = j; jj < j+BS; jj++ )
            for (kk = k; kk < k+BS; kk++ )
              C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj
                  ];
      }
    }
  }
}
```

`omp barrier` directive. A synchronization construct that makes the threads wait until all threads in the team have reached this point and only then continues execution.

```
#pragma omp barrier
```

Note that all constructs that allow the `nowait` clause have an implicit barrier at their end. Still sometimes explicit synchronization is required.

3.8.6 OpenMP clauses

The OpenMP clauses we have seen above can be divided into two classes

1. attribute clauses related to data sharing
2. clauses controlling data copying

We have the following basic properties for clauses:

- clauses usually take a list of arguments
- lists are comma separated and enclosed by `()`.
- all list items must be visible to the clause

Data sharing

Data sharing attributes of a variable in a `parallel` or `task` construct can be one of

- **predetermined**, e.g., loop counters in `for` or `parallel for` constructs are always `private`, `const` qualified variables are `shared`, more can be found in the Section on the loop construct of the OpenMP standard
- **explicitly determined** are those attributes where variables are referenced in a clause setting the attributes
- **implicitly determined**, are the attributes of variables referenced in a given construct but are neither predetermined nor explicitly specified

The following is a list of the most frequently used data sharing clauses.

`default(shared|none)`

- determines the default attributes of variables in the context of a `task` or `parallel` construct.
- defaults to `shared` when not explicitly given in a `parallel` construct
- all other (except `task`) constructs inherit the default from the enclosing construct if no `default` clause is given explicitly.

`shared(list)`

Sets the data sharing attributes of all variables in `list` to be of shared type. That means the variable is considered to be in the shared memory of the team of threads.

`private(list)`

Each variable of the `list` is declared to be a private copy of the thread and not accessible from other threads in the team. It can not be applied to variables that are part of other variables*(elements in arrays or members of a structure)*.

`firstprivate(list)`

As above but additionally the value of the item in the list is initialized from the corresponding original item when the construct is encountered. The clause has a few more restrictions found in the standard.

`lastprivate(list)`

As `private` but causes the original item to be updated after the end of the region from the last iterate of the enclosed loop or the lexically last `section` in a `sections` region.

`reduction(operator:list)`

Accumulates all items of the list into a private copy according to the given `operator` and then combines it with the original instance.

+	(0)	|	(0)
*	(1)	ˆ	(0)
-	(0)	&&	(1)
&	(˜0)	||	(0)
max	(Least number in reduction list item type)		
min	(Largest number in reduction list item type)		

Table 3.1: Operators for `reduction` with initialization values in ()

Example 3.20 (OpenMP reduction minimal example)**:**

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

int main (int argc, char *argv[]) {
int   i, n;
float a[100], b[100], sum;

/* Some initializations */
n = 100;
for (i=0; i < n; i++)
  a[i] = b[i] = i * 1.0;
sum = 0.0;

#pragma omp parallel for reduction(+:sum)
  for (i=0; i < n; i++)
    sum = sum + (a[i] * b[i]);
printf("␣␣␣Sum␣=␣%f\n",sum);
}
```

The following are cited from OpenMP 3.1 API C/C++ Syntax Quick Reference Card:

“These clauses support the copying of data values from private or threadprivate

variables on one implicit task or thread to the corresponding variables on other implicit tasks or threads in the team.”

`copyin(list)`

“Copies the value of the master thread’s threadprivate variable to the threadprivate variable of each other member of the team executing the `parallel` region.”

`copyprivate(list)`

“Broadcasts a value from the data environment of one implicit task to the data environments of the other implicit tasks belonging to the `parallel` region.”

3.8.7 OpenMP Environment Variables

Environment variables can be used to influence the behavior of an OpenMP process, without recompiling the binary, at runtime.

`OMP_SCHEDULE`

Specifies the runtime schedule type. Available values are `static`, `dynamic`, `guided`, or `auto` together with an optional `chunk` size.

`OMP_NUM_THREADS`

Must be set to a list of positive integers determining the numbers of threads at the corresponding nested level.

`OMP_PROC_BIND`

The value of this variable must be `true` or `false`. It determines whether threads may be moved between processors at runtime.

More environment variables can be found in Section 6 of the OpenMP 5.1 standard.

3.8.8 OpenMP runtime library functions

We only treat thread and processor number related functions

```
void omp_set_num_threads(int num_threads)
```

Determines the number of threads in subsequent parallel regions that do not specify a `num_threads` clause.

```
int omp_get_num_threads(void)
```

Returns the number of threads in the current team.

```
int omp_get_max_threads(void)
```

Provides the maximum number of threads that could be used in a subsequent `parallel` construct.

```
int omp_get_thread_num(void)
```

Returns the thread ID of the current thread. IDs are integers from zero (the master thread) to the number of threads in the team minus one.

```
int omp_get_num_procs(void)
```

returns the number of processors available to the program.

More runtime library functions and detailed descriptions can be found in Section 3 of the OpenMP standard.

Example 3.21 (Hello World revisited)**:**

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

int main (int argc, char *argv[]) {
  int th_id, nthreads;
#pragma omp parallel private(th_id)
  {
     th_id = omp_get_thread_num();

     printf("Hello␣World␣from␣thread␣%d\n", th_id);
     #pragma omp barrier
     if ( th_id == 0 ) {
       nthreads = omp_get_num_threads();
       printf("There␣are␣%d␣threads\n",nthreads);
     }
  }
  return EXIT_SUCCESS;
}
```

Two important rules of thumb:

In case of nested loops it is usually best to apply the parallelization to the outermost possible loop.

It is in general a good idea to first optimize the sequential code and only then

add parallelism to further increase the speed of execution.

3.9 Tree Reduction

3.9.1 The OpenMP reduction minimal example revisited

Recall Example 3.20, where we have investigated the usage of the OpenMP `reduction` clause. This section is dedicated to the detailed introduction of one possible implementation of such a reduction operation. Note that still in the OpenMP standard version 3.1 we do have the reduction clause only for arrays of basic scalar data types. Thus we might be interested to implement similar strategies for complex data types using, e.g., PThreads ourselves.

The main properties of the reduction are

- accumulation of data via a binary operator (here $+$)
- intrinsically sequential operation causing a race condition in multi-thread based implementations (since every iteration step depends on the result of its predecessor.)

3.9.2 Basic idea of tree reduction

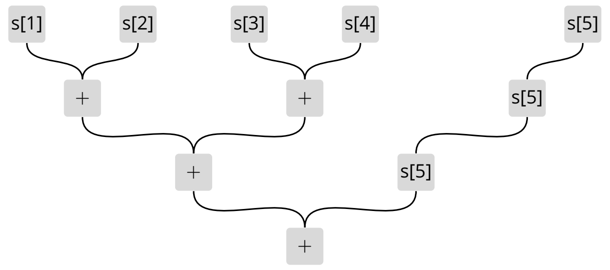


Figure 3.7: Tree reduction basic idea.

- ideally the number of elements is a power of 2
- best splitting of the actual data depends on the hardware used

3.9.3 Practical tree reduction on multiple cores

Consider the setting as before $a, b \in \mathbb{R}^{100}$. Further we have four equal cores. How do we compute the accumulation in parallel? Basically 2 choices

Example 3.22 (Another approach for the `dot` example)**:Task pool approach:** define a task pool and feed it with $n/2 = 50$ work packages accumulating 2 elements into 1. When these are done, schedule the next 25 and so on by further binary accumulation of 2 intermediate results per work package.

2. **#Processors=#Threads approach:** Divide the work by the number of threads, i.e. on our 4 cores each gets 25 subsequent indices to sum up. The reduction is then performed on the results of the threads.

Note that in the first approach we will need to make sure that the next level will only be started as soon as the data is ready, e.g. by condition variables, or barriers. The second can accumulated the partial sums in a vector of length 4 and do the final sum after the parallel region sequentially.

3.10 Dense Linear Systems of Equations

3.10.1 Repetition blocked algorithms

We have seen in the Chapter 6 of part I of the lecture that blocked operations are crucial for optimal exploitation of the memory hierarchy and especially the caches.

Algorithm 3.1: Gaussian elimination — row-by-row-version

```
Input: $A \in \mathbb{R}^{n\times n}$ allowing LU decomposition
Output: $A$ overwritten by $L, U$
1 for $k = 1 : n-1$ do
2     $k_0 = \operatorname{argmax}_{i=k:n} |A(i,k)|$;
3     Swap rows $k$ and $k_0$;
4     $A(k+1:n, k) = A(k+1:n, b)/A(k,k)$;
5     for $i = k+1 : n$ do
6         for $j = k+1 : n$ do
7             $A(i,j) = A(i,j) - A(i,k)A(k,j)$;
```

Observation:

- Innermost loop performs rank-1 update on the $A(k+1:n, k+1:n)$ submatrix in the lower right,

- i.e. a BLAS level 2 operation.

Algorithm 3.2: Gaussian elimination — Outer product formulation

```
Input: $A \in \mathbb{R}^{n\times n}$ allowing LU decomposition
Output: $L, U \in \mathbb{R}^{n\times n}$ such that $A = LU$ stored in $A$ stored in $A$
1 for $k = 1 : n-1$ do
2     rows= $k+1 : n$;
3     $A($rows$, k) = A($rows$, k)/A(k,k)$;
4     $A($rows,rows$) = A($rows,rows$) - A($rows$, k)A(k,$rows$)$;
```

Idea of the blocked version

- Replace the rank-1 update (BLAS level 2) by a rank-r update (BLAS level 3),
- Thus replace the $\mathcal{O}(n^2)$ / $\mathcal{O}(n^2)$ operation per data ratio the more desirable $\mathcal{O}(n^3)$ / $\mathcal{O}(n^2)$ ratio,
- Therefore exploit the fast local caches of modern CPUs more optimally.

Algorithm 3.3: Gaussian elimination — Block outer product formulation

```
Input: $A \in \mathbb{R}^{n\times n}$ allowing LU decomposition, $r$ prescribed block size
Output: $A = LU$ with $L, U$ stored in $A$
1 $k = 1$;
2 while $k \leqslant n$ do
3     $\ell = \min(n, k+r-1)$;
4     Compute $A(k:\ell, k:\ell) = \tilde{L}\tilde{U}$ via Algorithm 3.1;
5     Solve $\tilde{L}Z = A(k:\ell, \ell+1:n)$ and store $Z$ in $A$;
6     Solve $W\tilde{U} = A(\ell+1:n, k:\ell)$ and store $W$ in $A$;
7     Perform the rank-r update:
       $A(\ell+1:n, \ell+1:n) = A(\ell+1:n, \ell+1:n) - WZ$;
8     $k = \ell + 1$;
```

Algorithm 3.3 now replaces the rank-1 update by rank-r BLAS level 3 updates. Due to the fact that BLAS level 3 uses $\mathcal{O}(n^3)$ operations on $\mathcal{O}(n^2)$ data this is ensures more optimal exploitation of the local cache hierarchy. The block size r can be further exploited in the computation of W and Z and the rank-r update. It is used to optimize the data portions for the cache.

3.10.2 Fork-Join parallel implementation for multicore machines

We have basically two ways to implement naive parallel versions of the block outer product elimination in Algorithm 3.3.

Threaded BLAS available

Assuming we have a BLAS implementation that uses threading to speed up especially the level 3 operations, we can get a naive parallel version of Algorithm 3.3 following:

- Compute line 4 with the sequential version of the LU
- Exploite the threaded BLAS for the block operations in lines 5–7

Netlib BLAS

If we have only the strictly sequential reference implementation of BLAS from Netlib available we can still do the following:

- Compute line 4 with the sequential version of the LU
- Employ OpenMP/PThreads to perform the BLAS calls for the block operations in lines 5–7 in parallel.

Both these approaches fall into the class of parallel codes described by the following paradigm.

Definition 3.23 (Fork-Join Parallelism)**:**
An algorithm that performs certain parts sequentially between others that are executed in parallel is *called fork-join-parallel*.

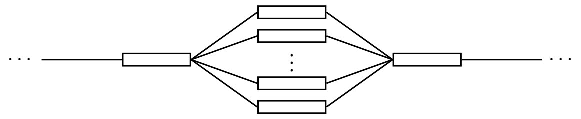


Figure 3.8: A sketch of the fork-join execution model.

Advantages

- Easy to achieve.
- Many threaded BLAS implementations available.
- Basically usable from any user code that requires linear system solves.

Disadvantages

- Very naive implementation.
- Sequential fraction limits the speedup (Amdahl's law).
- Therefore, only useful for small numbers of cores.

3.10.3 DAG scheduling of block operations aiming at manycore systems

Definition 3.24 (Directed Acyclic Graph (DAG))**:**
A *directed acyclic graph* is a graph where

- all edges have one distinct direction,
- directions are such that no cycles are possible for any path in the graph.

Where is the connection to parallel mathematical algorithms?

- Consider every node in the graph a task in the computation.
- Every task requires a certain number of previous tasks to have finished.
- Also none of the previous tasks depend on the later ones.
- Thus, the dependencies give us the directions and cycles can not appear by construction.

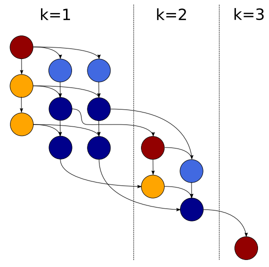


Figure 3.9: Dependency graph of Algorithm 3.3 for a 3×3 block subdivision.

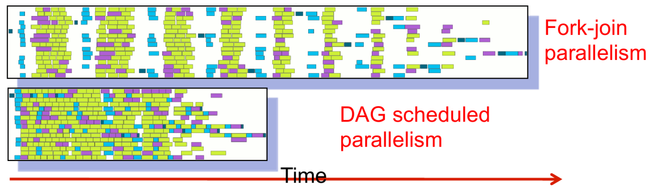


Figure 3.10: The superiority of DAG scheduling of tasks over fork-join parallelism.

Algorithm 3.4: Conjugate Gradient Method

```
Input: $A \in \mathbb{R}^{n\times n}, b \in \mathbb{R}^n, x_0 \in \mathbb{R}^n$
Output: $x = A^{-1}b$
1  $p_0 = r_0 = b - Ax_0,\ \alpha_0 = \|r_0\|_2^2$;
2  for $m = 0, \dots, n-1$ do
3      if $\alpha_m \neq 0$ then
4          $v_m = Ap_m$;
5          $\lambda_m = \frac{\alpha_m}{(v_m, p_m)}$;
6          $x_{m+1} = x_m + \lambda_m p_m$;
7          $r_{m+1} = r_m - \lambda_m v_m$;
8          $\alpha_{m+1} = \|r_{m+1}\|_2^2$;
9          $p_{m+1} = r_{m+1} + \frac{\alpha_{m+1}}{\alpha_m} p_m$;
10     else
11         STOP;
```

3.11 Sparse Linear Systems of Equations

3.11.1 The Conjugate Gradient (CG) Method (a prototype iterative solver)

CG (Algorithm 3.4) uses

- one matrix vector product (performing the main work),
- one `dot`,
- two `axpy`,
- one `nrm2`,
- and a nonstandard `axpy` operation with result in `x`.

The last operation can not be performed in a single BLAS call. A combination

of `scal` and `axpy` is required at least. That especially means that at least one argument vector is likely to be pulled through the cache hierarchy twice.

3.11.2 Sparse Matrix Vector Products

The key ingredient in the CG method, as in all Krylov-subspace based iterative solvers and many other linear algebra based algorithms, is the sparse matrix vector product (SpMVP).

We learned in part 1 of the lecture that sparse matrix operations are *bandwidth limited*, i.e., the crucial point is always the data transfer for matrix pattern and entries to the processing units. We have further observed that bandwidth limitations on the matrix provide certain benefits for the caching of the vector, since only local portions are used and these do not change much from row to row.

On the other hand, the SpMVP is trivially parallel due to data parallelism. On multicore architectures the obvious questions are:

- What is the optimal number of threads to use?
- How should the data be distributed among the threads?

The first of the two questions will be treated in a little more detail in the exercises. The second questions is investigated a lot in the literature. We will only sketch a small selection of approaches considering $x = Ab$ for $x, b \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n\times n}$ sparse with properties specified separately in the method descriptions. add references!

Naive row blocking. (e.g., using OpenMP `parallel for`)

If the matrix A is banded with moderate bandwidth and the number of entries per row is almost the same for all rows, simply grouping the rows in blocks of rows (the chunks in OpenMP) will likely do a good job.

The bandwidth limitations guarantee data locality on b and good cache performance due to little changes of the local portion of b required by the members of the group.

Furthermore, the similar lengths of the sparse rows will automatically provide a proper load balancing.

This provides the easiest form of 1d-partitioning. Alternatively column grouping is used. This guarantees locality for the access on b but requires mutual exclusion when writing the results into x.

Hypergraph Partitioning.

The simplest form of 2d-partitioning of the matrix A uses (blocks of) columns and (blocks of) rows at the same time. It is usually referred to as hypergraph partitioning since the choice fits the following definition.

Definition 3.25 (Hypergraph)**:**
A *hypergraph* is an ordered pair $(\mathcal{V}, \mathcal{E})$ of sets. It is a generalization of a graph that consists of vertices (in the set $\mathcal{V}$) and *hyperedges* in the set $\mathcal{E}$. In contrast to an edge in a graph a hyperedge can be an arbitrary subset of $\mathcal{V}$ and not just a pair.

Example 3.26:
Schematic representation of a hypergraph with seven vertices and four hyperedges.

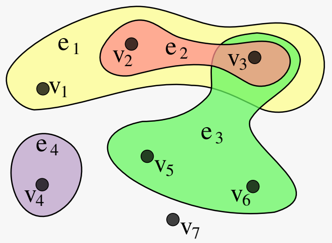


Image source: https://commons.wikimedia.org/wiki/File:Hypergraph-wikipedia.svg

The idea of hypergraph partitioning is to use the hyperedges to find the optimal partitioning of the vertices into k equal sets for optimal balancing of the workload and data communication. The problem of finding the optimal partition is however np-hard. Therefore cheap heuristics are employed to approximate the optimal partition.

An interesting variant, especially for symmetric patterns, is the corner symmetric partitioning in Figure 3.11.

Graph Model Partitioning to Nested Dissection

The central node 8 is called *vertex separator*. The identification of such a preferably small (group of) node(s) is the central question in the graph model based partitioning. Successive application of this idea leads to the nested dissection scheme.

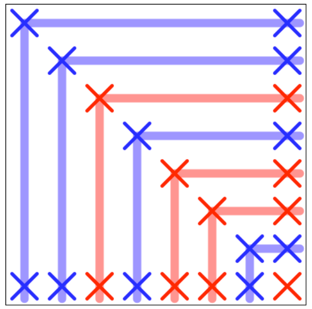


Figure 3.11: Corner symmetric partitioning of the arrowhead matrix with 2 partitions.

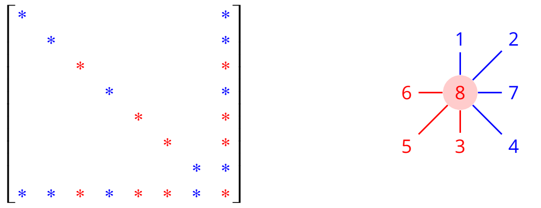


Figure 3.12: arrowhead matrix pattern and connectivity graph.

3.11.3 Preconditioning

Recall:

A *preconditioner* is an invertible linear operator P that approximates the action of A^{-1} for a linear system $Ax = b$.

- Invertibility required to ensure proper preservation of solution,
- preconditioner need not be formed as a matrix, as long as its action on a vector can be provided as a function,
- main purpose of the preconditioner is the grouping of eigenvalues, ideally in a single cluster at $+1$.

Algorithm 3.5: Preconditioned Conjugate Gradient Method

```
Input: $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$, $x_0 \in \mathbb{R}^n$, $A^{-1} \approx P \in \mathbb{R}^{n\times n}$
Output: $x = A^{-1}b$
1  $r_0 = b - Ax_0$, $p_0 = z_0 = Pr_0$,$\alpha_0 = (r_0, p_0)$;
2  for $m = 0 : n-1$ do
3      if $\alpha_m \neq 0$ then
4          $v_m = Ap_m$;
5          $\lambda_m = \frac{\alpha_m}{(v_m,p_m)_2}$;
6          $x_{m+1} = x_m + \lambda_m p_m$;
7          $r_{m+1} = r_m - \lambda_m v_m$;
8          $z_{m+1} = Pr_{m+1}$;
9          $\alpha_{m+1} = (r_{m+1}, z_{m+1})_2$;
10         $p_{m+1} = z_{m+1} + \frac{\alpha_{m+1}}{\alpha_m} p_m$;
11     else
12         STOP;
```

3.11.4 Preconditioned CG

3.11.5 Diagonal/Jacobi Preconditioner

Let $D \in \mathbb{R}^{n\times n}$ be a diagonal matrix containing the diagonal of A. Then $P = D^{-1}$ is called Jacobi or diagonal preconditioner.

Properties

+ embarrassingly parallel in computation and application,
+ storage requirement n `double` numbers,
− only useful for diagonally dominant systems.

3.11.6 Sparse Approximate Inverse (SPAI) Preconditioning

The basic idea of SPAI is to find the best matrix P approximating A^{-1}, while maintaining the sparsity pattern of A.

$$\min_{\mathcal{P}(P)=\mathcal{P}(A)} \|AP - I\|_F^2 = \min_{\mathcal{P}(P)=\mathcal{P}(A)} \underbrace{\sum_{j=1}^{n} \|Ap_j - e_j\|_F^2}_{n \text{ independent least squares problems}}$$

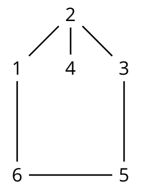
+ only SpMVP needed for the application,

+ n independent least squares problems allow two multicore approaches:
  - rely on threaded BLAS when solving the least squares problems sequentially via `dgeqrs()` from LAPACK,
  - use sequential BLAS with OpenMP for parallel solution of the least squares problems.

− efficient preconditioning requires additional fill-in, which leads to extra storage demands and increased computational complexity.

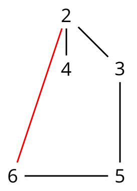
3.11.7 Issues of Sparse Direct Solvers

The following two figures demonstrate the fill in problem as a result of the elimination during the computation of the Cholesky decomposition of a simple 6×6 symmetric example matrix.


(a) initial graph $\mathcal{G}_0$

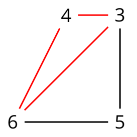
$$H_0 = \begin{bmatrix} 1 & * & & & & * \\ * & 2 & * & * & & \\ & * & 3 & & * & \\ & * & & 4 & & \\ & & * & & 5 & * \\ * & & & & * & 6 \end{bmatrix}$$

(b) corresponding submatrix 0


(c) elimination graph $\mathcal{G}_1$

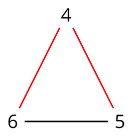
$$H_1 = \begin{bmatrix} 2 & * & * & & {\color{red}*} \\ * & 3 & & * & \\ * & & 4 & & \\ & * & & 5 & * \\ {\color{red}*} & & & * & 6 \end{bmatrix}$$

(d) corresponding submatrix 1


(e) elimination graph $\mathcal{G}_2$

$$H_2 = \begin{bmatrix} 3 & {\color{red}*} & * & {\color{red}*} \\ {\color{red}*} & 4 & & {\color{red}*} \\ * & & 5 & * \\ {\color{red}*} & {\color{red}*} & * & 6 \end{bmatrix}$$

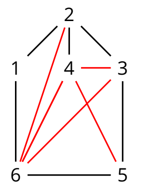
(f) corresponding submatrix 2


(g) elimination graph $\mathcal{G}_3$

$$H_3 = \begin{bmatrix} 4 & {\color{red}*} & {\color{red}*} \\ {\color{red}*} & 5 & * \\ {\color{red}*} & * & 6 \end{bmatrix}$$

(h) corresponding submatrix 3

Figure 3.13: Basic graph elimination procedure for a symmetric matrix and the Cholesky decomposition


(a) The filled graph $\mathcal{G}^+(A) = \mathcal{G}(F)$

$$F = \begin{bmatrix} 1 & * & & & & * \\ * & 2 & * & * & & {\color{red}*} \\ & * & 3 & {\color{red}*} & * & {\color{red}*} \\ & * & {\color{red}*} & 4 & {\color{red}*} & {\color{red}*} \\ & & * & {\color{red}*} & 5 & * \\ * & {\color{red}*} & {\color{red}*} & {\color{red}*} & * & 6 \end{bmatrix}$$

(b) The final matrix $F = L + L^T$ with fill.

Figure 3.14: The filled graph and matrix of a Cholesky decomposition example.

Now

$$L = \begin{bmatrix} 1 & & & & & \\ * & 2 & & & & \\ & * & 3 & & & \\ & * & {\color{red}*} & 4 & & \\ & & * & {\color{red}*} & 5 & \\ * & {\color{red}*} & {\color{red}*} & {\color{red}*} & * & 6 \end{bmatrix}$$

and thus, the forward elimination is purely sequential. Are we lost?

Consider the Cholesky factor:

$$L = \begin{bmatrix} 1 & & & & & & & & & \\ * & 2 & & & & & & & & \\ & & 3 & & & & & & & \\ & & * & 4 & & & & & & \\ & & * & * & 5 & & & & & \\ & & & & & 6 & & & & \\ & & * & * & * & & 7 & & & \\ & * & * & * & * & & * & 8 & & \\ & & & & & * & & & 9 & \\ & * & * & * & * & & * & * & * & 10 \end{bmatrix}$$

We have a very specific sparsity pattern in some column blocks below a certain diagonal block in columns 3–5 and 7–8. This special structure motivates the following definitions.

Definition 3.27 (column pattern)**:**
The j-th *column pattern* $\mathcal{P}_{*j}$ is the set of row indices of all non-diagonal nonzero entries in the j-th column.

Definition 3.28 (Supernode)**:**
A *supernode* is a set of contiguous column indices

$$\mathcal{I}(p) = \{p, p+1, \dots, p+q-1\},$$

such that for all columns $i \in \mathcal{I}(p)$ we have

$$\mathcal{P}_{*i} = \mathcal{P}_{*(p+q-1)} \cup \{i+1, \dots, p+q-1\}$$

- Supernodes, thus are special dense diagonal blocks that have the identically same pattern in each column below the diagonal block.
- Column modifications in forward substitution can be expressed in terms of supernodes rather than single diagonal entries.
- Inside the supernode block operations we can exploit parallelism.

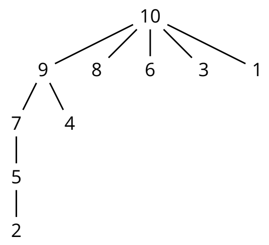
The elimination tree for the Cholesky factor of a matrix is determined by a simple method, when the factor is known:

- For each column find the index of the first entry below the diagonal.
- This index determines the parent node in the tree.
- If the connectivity graph is connected, then connecting each node with its parent determines the elimination tree. If it is not connected the method creates a forest of smaller trees.

3.11.8 A Task Pool Approach to Parallel Triangular Solves

Consider the Cholesky factor and corresponding elimination tree

$$L = \begin{bmatrix} 1 &&&&&&&&& \\ & 2 &&&&&&&& \\ && 3 &&&&&&& \\ &&& 4 &&&&&& \\ & * &&& 5 &&&&& \\ &&&&& 6 &&&& \\ &&&& * && 7 &&& \\ &&&&&&& 8 && \\ & * && * & * && * && 9 & \\ * & * & * & * & * & * & * & * & * & 10 \end{bmatrix}$$



+ many elimination steps can be executed independently

+ a simple task pool scheduling the independent tasks enables parallel execution and load balancing

− elimination tree must be computed to enable proper scheduling and identification of independent tasks

Remark

Note that elimination trees can be computed without computing the filled graph or the Cholesky factor first.

3.12 Relevant Software and Libraries

The following is a (very likely incomplete) list of software packages relevant to scientific computing on Unix platforms. All these packages make use of threading techniques to exploit multicore processors, implement the PThreads standard, or are intended to control the scheduling of threads and processes and placement of the process memory in such environments.

Dense Linear Algebra

1. **OpenBLAS** based on the earlier GotoBLAS project OpenBLAS implements a complete set of optimized BLAS routines. On a machine with a single socket it is likely the fastest BLAS implementation one can get.[5]
2. **Intel® Math Kernel Library (MKL)** is Intel®s optimized implementation of BLAS and LAPACK. It is the strongest opponent of OpenBLAS on single socket systems. On a system with several sockets no other BLAS library outperforms MKL.[6]

[5] http://xianyi.github.io/OpenBLAS/
[6] http://software.intel.com/en-us/intel-mkl

3. **PLASMA** The **P**arallel **L**inear **A**lgebra **S**ubroutines for **M**ulticore **A**rchitectures employs DAG scheduling to increase performance of the linear algebra subsystem on multicore architectures.[7]

Sparse Linear Algebra

1. UMFPACK comes as part of the SuiteSparse package of software libraries for sparse linear systems of equations. Uses thread parallel multifrontal techniques to solve linear systems of equations.[8]
2. **Boost uBLAS** *"is a C++ template class library that provides BLAS level 1, 2, 3 functionality for dense, packed and sparse matrices. The design and implementation unify mathematical notation via operator overloading and efficient code generation via expression templates."*[9]
3. **MTL** the **M**atrix **T**emplate **L**ibrary provides an easy to use template based C++ interface to linear algebra operations. It relies on Boost for fast and efficient codes.[10]
4. **Eigen***"is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms."* Eigen is self-contained and only relies on a C++98 compliant compiler.[11]
5. **SuperLU_MT** Supernode based multithreaded LU decomposition.[12]

PThreads and Scheduling/Memory Control

1. **nptl** is the **N**ative **P**OSIX Linux **T**hread **l**ibrary that currently provides PThread support on most Linux platforms.[13]
2. **likwid** (**L**ike **I** **K**new **W**hat **I** **D**o) is a light weight library that supports software developers to design high performance scientific computing programs with little overhead.[14]
3. **numactl** referred to as `libnuma` by several Linux distributions, numactl is a small program/library that can be used to control placement of process memory in NUMA environments. The library version seems to be preferred by the Linux kernel policies.[15]

[7] http://icl.cs.utk.edu/plasma/software/
[8] http://faculty.cse.tamu.edu/davis/suitesparse.html
[9] http://www.boost.org/doc/libs/1_70_0/libs/numeric/ublas/doc/index.htm
[10] http://www.simunova.com/en/mtl4
[11] https://eigen.tuxfamily.org/index.php?title=Main_Page
[12] http://crd-legacy.lbl.gov/~xiaoye/SuperLU/
[13] http://en.wikipedia.org/wiki/Native_POSIX_Thread_Library
[14] http://code.google.com/p/likwid/
[15] http://oss.sgi.com/projects/libnuma/

CHAPTER 4

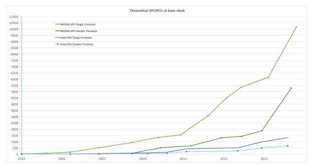
GPU Computing and Accelerators

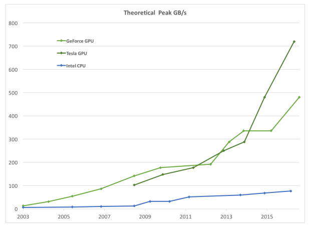
Contents

4.1 Why use accelerators?

We have already discussed in Flynn's taxonomy that GPUs are SIMD devices. In the course of this chapter we will see, how this is realized by the so called streaming multiprocessors on NVIDIA®'s CUDA devices. Other manufacturers use similar techniwques and thus this is a common theme for general purpose GPUs. Generally speaking that makes them ideal candidates for linear algebra, i.e.vector based operations.Other accelerators have different strengths, but we will focus on GPUs and their programming via CUDA.

The fact that these GPUs in general come with their own memory installed makes using them slightly more complicated, as we are responsible for making sure that data is always passed to the required locations before use. However, as we can see from Figure 4.1, these devices feature drastically higher throughput in both floating point operations and memory, such that the additional transfer times can often be mitigated, easily.


(a) Floating point operations



(b) Memory bandwidth

Figure 4.1: Throughput comparison of Multicore CPUs and CUDA enabled GPUs (taken from CUDA C Programming Guide)

A second strong point of GPUs, once the data is sufficiently large, is that due to their vector processing oriented setup, their are often far more energy efficient on these operations compared to general purpose CPUs. Table 4.1 shows this

also in comparison to further accelerators. We can see that GPUs may not be the best choice for the energy, but they have far easier programming toolkits and are thus much easier to apply than, e.g. the digital signal processors (DSPs), which win this comparison. Cell processors, on the other hand, have lost significance soon after the comparison was made.

Architecture	GFLOPS	GFLOPS/Watt	Utilization
Core i7-960	96	1.14	**95%**
NVIDIA® GTX280	410	2.6	66%
Cell	200	5.0	88%
NVIDIA® GTX480	**940**	5.4	70%
TI C66x DSP	74	**7.4**	57%

Table 4.1: Power efficieny comparison of Multicore CPUs and accelerator chips (taken from Conference Poster by F. Igual and M. Ali)

Recent years have also seen the rise and fall of Intel® Xeon® Phi coprocessor boards, which, again due to their more complicated programming model, could not gain a significant market share.

4.2 Memory Hierarchy with Accelerators

4.2.1 Common Features

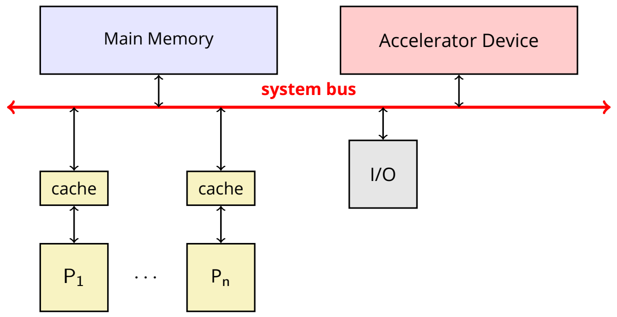


Figure 4.2: Schematic of an accelerator based parallel system

4.2.2 Graphics Processing Units (GPUs)

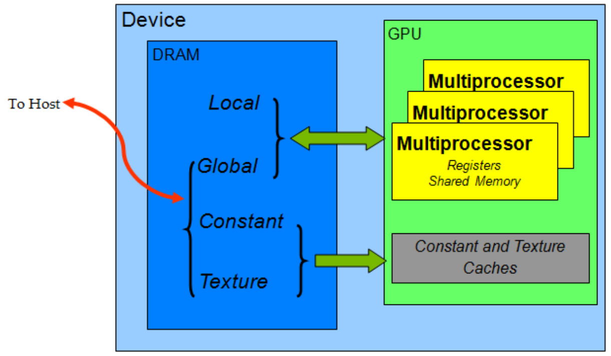


Figure 4.3: Memory configuration of a CUDA Device (taken from CUDA C Programming Guide)

4.2.3 Field Programmable Gate Arrays (FPGAs)

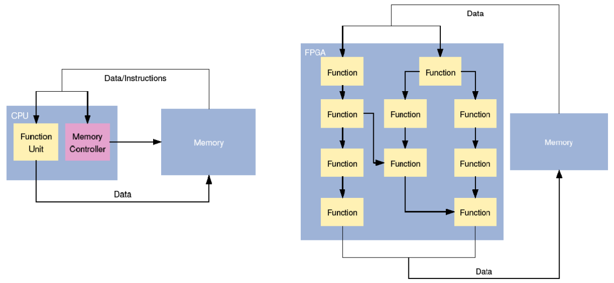


Figure 4.4: Comparison of CPUs and FPGA execution models.

4.3 Compute Unified Device Architecture (CUDA)

4.3.1 What is CUDA?

CUDA is two things at the same time:

1. platform model for the hardware implementation of general purpose graphics processing units made by the NVIDIA® Corporation.
2. programming model realizing the software implementation and scheduling of tasks of the parallel programs on the above hardware.

4.3.2 Basic Definitions

Definition 4.1 (thread)**:**
A *thread*, or more precisely *GPU-thread* is the smallest unit of data and instructions to be executed in a parallel CUDA program.

In contrast to CPU-threads a task switch between GPU-threads is usually almost for free, or at least a lot less expensive than on Intel-architecture CPUs, due to the special CUDA architecture.

Definition 4.2 (warp)**:**
The CUDA hardware consists of streaming multi-processors that are executing several threads simultaneously. The GPU-threads are therefore grouped in so called *warps* of threads per multi-processor.

The number of threads in a warp may depend on the hardware. One finds mostly 32 threads per warp which in turn is the smallest number of tasks executed in SIMD style. We will see later in this section how the number of threads per warp can be determined at runtime. In fact NVIDIA® calls their execution model SIMT for single instruction multiple threads rather than SIMD.

To increase the abstraction and reduce granularity in the programming, but also to get more flexibility for the scheduling, CUDA arranges the tasks to be executed in grids of blocks.

Definition 4.3 (block)**:**
A *block* is a larger group of threads that can contain 64–512 threads on older devices and up to 1 024 on devices starting from the Kepler generation.

Ideally, it contains a multiple of 32 threads so it can be split optimally into warps, by the CUDA environment, for scheduling.

> **Definition 4.4** (grid)**:**
> The actual work to be performed by a program or algorithm is distributed to a one, two (maximum for pre-Kepler devices), or three dimensional *grid* of blocks.

The grid represents the largest freedom in design that the developer has.

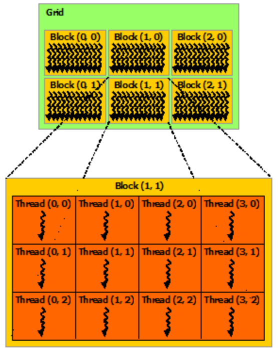


Figure 4.5: Grids of Thread Blocks (taken from CUDA C programming guide)

The central notions to understand data management in a CUDA program are those of *host* and *device*. Here, *host* refers to the computer that hosts the GPU. Especially the CPU and memory of the host are relevant. The *device* then is the GPU installed on the host system.

In case multiple GPUs are installed on a single host system with multiple CPUs, each GPU is connected to a single CPU representing a single NUMA node of the host system.

The host CPU controls the execution of the program. However, host and device may execute their tasks asynchronously. When not specified differently data transfers between them serve as implicit synchronization points. Data transfers can be made asynchronous as well. Then explicit synchronization is necessary.

> **Definition 4.5** (kernel)**:**
> The *kernel* is the core element of a CUDA parallel program. It represents

> the function that specifies the work a certain thread in a block on a grid has to execute.

We will see in the course of this Chapter how the thread executing the kernel knows which part of the global problem it has to perform.

4.3.3 Most Basic Syntax of the CUDA C Extension

We will next introduce the most basic elements of the CUDA C language extension. These consist of two important things.

1. **qualifiers** that apply to functions and specify where (i.e., on host or device) the function should be executed,
2. **launch size specifiers** that control the grid and block sizes that are used to run a kernel.

An extensive API, defining C-style functions and data types to be used in CUDA programs, together with a handful of libraries for several kinds of tasks (e.g., a BLAS implementation) complete the picture. See also Figure 4.5.

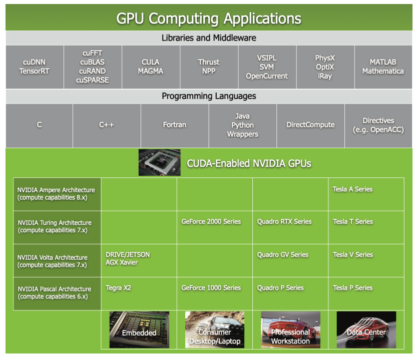


Figure 4.6: The CUDA GPU computing applications framework (taken from CUDA C programming guide 11.3.1)

- `__global__` This qualifier applies to a function and is used to indicate that it in fact represents a kernel.
- `__device__` The qualifier that specifies functions that should be run on the device, but are not kernels. It can be useful for subtasks called in a kernel. It also applies to variables determining them to reside on the device.

- __host__ Being basically redundant this qualifier can be used to explicitly state that a function is to be executed on the host. It is therefore optional.
- __shared__applies to a variable declaring that it should reside in the shared memory of a streaming multiprocessor
- __constant__applies to a variable specifying the residence in the constant memory.

Note that __global__ functions are not allowed to be recursive. On old gardware (before Fermi generation) the same is true for __device__ functions.

The most basic launch size specification for a kernel takes the form

```
<<< grid , block size >>>
```

where `grid` specifies the block distribution and `block size` indicates the number of threads per block in the grid.

Example 4.6:
Simple 1 one dimensional distributions:

- `<<<1,1>>>` launches 1 block with 1 thread
- `<<<N,1>>>` launches N blocks with 1 thread each
- `<<<1,N>>>` launches 1 block with N threads
- `<<<N,M>>>` launches a 1d grid of N blocks running M threads each

Both the arguments can also be two, or even three dimensional distributions. CUDA defines special tuple hiding types for these declarations. Using

```
   dim3 grid(3,2)
   dim3 threads(16,16)
```

one defines a 3×2 grid of blocks for running 256 threads arranged in a 16×16 local grid. These are then used in the launch specification as

```
   <<< grid, threads>>>
```

Launch size specifications are simply appended to the kernel function name upon calling it.

4.3.4 Introductory Examples

The following examples are taken from the “CUDA by Example” book [19].

Example 4.7:

```
#include "../common/book.h"

__global__ void kernel( void ) { }

int main( void ) {
    kernel<<<1,1>>>();
    printf( "Hello,␣World!\n" );
    return 0;
}
```

Example 4.8:

```
#include "../common/book.h"

__global__ void add( int a, int b, int *c ) {
    *c = a + b;
}

int main( void ) {
    int c;
    int *dev_c;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, sizeof(int) )
        );

    add<<<1,1>>>( 2, 7, dev_c );

    HANDLE_ERROR( cudaMemcpy( &c, dev_c, sizeof(int),
                              cudaMemcpyDeviceToHost ) );
    printf( "2␣+␣7␣=␣%d\n", c );
    HANDLE_ERROR( cudaFree( dev_c ) );

    return 0;
}
```

Example 4.9:

```
#include "../common/book.h"

__device__ int addem( int a, int b ) {
    return a + b;
}

__global__ void add( int a, int b, int *c ) {
    *c = addem( a, b );
}

int main( void ) {
    int c;
    int *dev_c;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, sizeof(int) )
        );

    add<<<1,1>>>( 2, 7, dev_c );

    HANDLE_ERROR( cudaMemcpy( &c, dev_c, sizeof(int),
                              cudaMemcpyDeviceToHost ) );
    printf( "2␣+␣7␣=␣%d\n", c );
    HANDLE_ERROR( cudaFree( dev_c ) );

    return 0;
}
```

4.3.5 Compiling CUDA Programs

In order to be able to compile the previous examples, one needs to check a few prerequisites:

- NVIDIA® device drivers and hardware,
- NVIDIA® CUDA toolkit installation,
- compiler for the host code.

Basic information on CUDA in general, the actual toolkit, and all the information on the included accelerated libraries and developer tools can be found at `https://developer.nvidia.com/cuda-toolkit`.

Regarding the hardware, basically every NVIDIA® GPU released after the appearance of the GeForce 8800 GTX in 2006 is CUDA enabled. However, one needs to make sure that the OS version, the device driver and CUDA Toolkit version are fitting. Working combinations should be available in the toolkits documentation.

Regarding the compilers NVIDIA® recommends the following

- **Microsoft Windows:** Visual Studio

- **Linux:** Gnu Compiler Collection (GCC) or CLANG
- **MacOS:** GCC as well via Apple's Xcode

On Linux in an `x86_64` hardware environment also Intel® and PGI compilers are supported. On `arm` platforms also PGI and Arm C/C++ or XLC can work, while on IBM's POWER9 series PGI and XLC are supported. One should carefully check the versions in the Toolkit installation guide, though.

We will in the following restrict ourselves to the Linux world again.

Consider our basic "Hello World!" example is stored in a text file called `hello_world.cu`. Using the `nvcc` compiler provided in the CUDA Toolkit we can compile it by

```
nvcc hello_world.cu
```

Since on Linux `nvcc` uses `gcc` to compile the host code this will also generate a binary called `a.out`. As for `gcc` we can specify the output filename, i.e. name of the resulting executable via

```
nvcc hello_world.cu -o hello_world
```

The file extension `.cu` is used to indicate that we have a C file with CUDA C extensions.

Among the further compiler options we meet many old friends:

`-c`	for generating object files of single `.c` or `.cu` files
`-g`	for generating debug information in the host code
`-pg`	the same for profiling information
`-O`	for specifying the optimization level for the host code
`-m`	specify 32 vs 64bit host architecture

And we have a few more for the device code, e.g.

`-G`	generates debug information for the device code
`-arch`	specifies the GPU architecture to be assumed, i.e. the compute capabilities of the device (e.g. `-arch=sm_20`)

4.3.6 Compute Capabilities

The *compute capabilities* of a device describe the basic set of features and instructions supported by this specific hardware. They are given as a *major.minor* style version number defining a general set of capabilities via the major number and incremental changes encoded in the minor number. (this coincides with the `sm_majorminor` representation for compute capabilities in the `-arch` flag of `nvcc` above)

The compute capabilities are a feature of the specific hardware and unrelated to the toolkit version number describing the software installation. Each toolkit is supporting a range of compute capabilities. this is due to the fact that older hardware platforms do not get supported with newer toolkit versions any longer. The supported compute capabilities of the latest toolkit version can be found in the CUDA C programming guide.

We have seen a few thing in this regard already in the definitions of warps and blocks. Other features are IEEE 754 support, rounding modes etc, as discussed in the next section. A recent prominent further example is the support for so called tensor processing units (TPUs), or simply tensor cores, taking the vector units on the CPUs to the matrix level for low precision floating point numbers, as a consequence of their common application in many machine learning applications.

4.3.7 CUDA and IEEE 754 Floating Point Computations

Compute capabilities 1.3

Double precision floating point numbers have been added in Version 1.3 of the CUDA compute capabilities. It additionally provides a fused multiply add operation merging multiplication and addition to be faster and more accurate, but non IEEE 754 compliant.

Compute Capabilities 2.0 and above

Compute capabilities 2.0 introduces IEEE 754 compliance for most parts of the standard as the default. The compiler switches

- `-ftz=false|true`,
- `-prec-div=true|false`,
- `-prec-sqrt= true|false`

influence IEEE compliance of the computation. If the second option is used everywhere one switches to fast mode. The first options are the default though.

IEEE 754 Rounding Modes

IEEE 754 defines four rounding modes

- round to nearest,
- round towards zero,
- round towards $+\infty$,
- round towards $-\infty$,

all of which are supported by CUDA. However in contrast to x86 CPUs where they can be dynamically switched, CUDA uses them statically.

Compiler intrinsics can be used to change the mode for individual operations, though.

Main Differences to x86 CPUs

- no dynamical control of rounding modes, instead rounding modes can be hard-coded per instruction
- floating point exceptions not handled (especially all `NaN`s are silent)
- no status flags indicating the exceptions exist

4.3.8 Data Communication Issues

Local versus Remote memory

Viewing from the host perspective, the device memory is remote memory that can only be accessed via the comparably slow system bus.

Looking at things from the device perspective the same holds true for the hosts memory. Going even further, already the device memory may be considered slow from the view of the streaming multiprocessors. The local memory of the multiprocessors should be used to implement a user controlled cache.

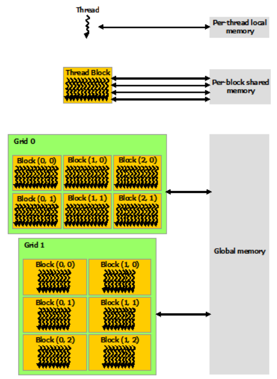
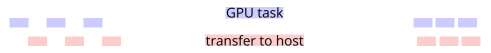


Figure 4.7: The CUDA memory hierarchy (taken from CUDA C programming guide)



(a) bad pattern causing waiting times due to communication.

(b) good pattern.

Figure 4.8: Execution patterns for CUDA programs

Consequences for CUDA Programs

- Keep data movements between device and host as little as possible
- If they are necessary, try to overlap communication and computations. See also Figure 4.8
- Make use of multiprocessors local shared memory to cache buffer kernel operations and avoid frequent access to global device memory

Example 4.10:

```
/*
 * Copyright 1993-2010 NVIDIA Corporation.  All rights
   reserved.
 *
 * NVIDIA Corporation and its licensors retain all
   intellectual property and
 * proprietary rights in and to this software and related
   documentation.
 * Any use, reproduction, disclosure, or distribution of
   this software
 * and related documentation without an express license
   agreement from
 * NVIDIA Corporation is strictly prohibited.
 *
 * Please refer to the applicable NVIDIA end user license
   agreement (EULA)
 * associated with this source code for terms and
   conditions that govern
 * your use of this NVIDIA software.
 *
 */


#include "../common/book.h"

#define imin(a,b) (a<b?a:b)
```

```
const int N = 33 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid =
            imin( 32, (N+threadsPerBlock-1) /
                threadsPerBlock );


__global__ void dot( float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    float   temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }

    // set the cache values
    cache[cacheIndex] = temp;

    // synchronize threads in this block
    __syncthreads();

    // for reductions, threadsPerBlock must be a power of 2
    // because of the following code
    int i = blockDim.x/2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }

    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}


int main( void ) {
    float   *a, *b, c, *partial_c;
    float   *dev_a, *dev_b, *dev_partial_c;

    // allocate memory on the cpu side
    a = (float*)malloc( N*sizeof(float) );
    b = (float*)malloc( N*sizeof(float) );
    partial_c = (float*)malloc( blocksPerGrid*sizeof(float)
        );
    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                              N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
                              N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,
                              blocksPerGrid*sizeof(float) )
                                  );

    // fill in the host memory with data
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i*2;
    }

    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N*sizeof(float),
                              cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N*sizeof(float),
                              cudaMemcpyHostToDevice ) );

    dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b,
                                            dev_partial_c )
                                               ;

    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,
                              blocksPerGrid*sizeof(float),
                              cudaMemcpyDeviceToHost ) );

    // finish up on the CPU side
    c = 0;
    for (int i=0; i<blocksPerGrid; i++) {
        c += partial_c[i];
    }

    #define sum_squares(x)  (x*(x+1)*(2*x+1)/6)
    printf( "Does␣GPU␣value␣%.6g␣=␣%.6g?\n", c,
             2 * sum_squares( (float)(N - 1) ) );

    // free memory on the gpu side
    HANDLE_ERROR( cudaFree( dev_a ) );
    HANDLE_ERROR( cudaFree( dev_b ) );
    HANDLE_ERROR( cudaFree( dev_partial_c ) );

    // free memory on the cpu side
    free( a );
```

```
    free( b );
    free( partial_c );
}
```

Note:

- automatic scaling of `blocksPerGrid`
- usage of local shared buffer `cache`
- synchronization in reduction block

4.3.9 The CUDA Application Programmers Interface

We have seen some elements of the CUDA API in the examples before:

- qualifiers:
  - `__global__`,
  - `__device__`,
  - `__host__`,
  - `__shared__`,
  - `__constant__`
- launch size specifiers `<<<grid, block size>>>`
- type `dim3`
- predefined variables:
  - `threadIdx.x`,
  - `blockIdx.x`,
  - `blockDim.x`,
  - `gridDim.x`
- memory functions: `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
- thread synchronization mechanism: `__syncthreads();`

Some have been introduced earlier. For the others and a few more we will go into some more detail now.

Important Memory Operations

-
  ```
  cudaError_t cudaFree ( void* devPtr )
  ```

Frees the memory on the device that is refered to by `devPtr`.

-
  ```
  cudaError_t cudaMalloc ( void** devPtr, size_t size )
  ```

  Allocate an amount corresponding to `size` of memory on the device and associate it to `devPtr`.

-
  ```
  cudaError_t cudaMemcpy ( void* dst, const void* src,
      size_t count, cudaMemcpyKind kind )
  ```

  Copy data between host and device. `src` and `dst` represent the source and destination memory locations. The direction of operation is specified by `kind` and can be either `cudaMemcpyHostToDevice`, or `cudaMemcpyDeviceToHost`. The `count` argument is used to specify the amount of data to be copied.

Device Management Basics

-
  ```
  cudaError_t cudaGetDeviceCount ( int* count )
  ```

  Returns the number of compute-capable devices available in the system.

-
  ```
  cudaError_t cudaChooseDevice ( int* device, const
      cudaDeviceProp* prop )
  ```

  Select compute-device which best matches criteria specified in `prop`. These can, e.g., be `int major`, `int minor` version numbers of the compute capabilities, or whether the chip is `int integrated` in the chipset or a plugged in device, but also simply the `char name[256]` of the device, and many more.

-
  ```
  cudaError_t cudaGetDevice ( int* device )
  ```

  Returns which device is currently used by the program.

-
  ```
  cudaError_t cudaSetDevice ( int  device )
  ```

  Set device to be used for GPU executions

-
  ```
  cudaError_t cudaDeviceSynchronize ( void )
  ```

  Wait for compute device to finish. If for the current device the synchronization flag `cudaDeviceScheduleBlockingSync` was set, the host thread will block until the device has finished its work.

Error Handling

-
  ```
  const __cudart_builtin__ char* cudaGetErrorString (
      cudaError_t error )
  ```

  Returns the message string for the error code given in `error`.

-
  ```
  cudaError_t cudaGetLastError ( void )
  ```

  Returns the last error that has been produced by any of the runtime calls in the same host thread and resets it to `cudaSuccess`.

-
  ```
  cudaError_t cudaPeekAtLastError ( void )
  ```

  As above but does not reset the error code.

Event Handling (Measuring Performance)

-
  ```
  cudaError_t cudaEventCreate ( cudaEvent_t* event )
  ```

  Creates, i.e., initializes the event object `event`.

-
  ```
  cudaError_t cudaEventRecord ( cudaEvent_t event,
      cudaStream_t stream = 0 )
  ```

  Record `event`. The record may take some time so before evaluation it is recommended to use `cudaEventSynchronize()` to make sure it has terminated.

-
  ```
  cudaError_t cudaEventSynchronize ( cudaEvent_t event
      )
  ```

  Wait until `event` has completed operations.

-
  ```
  cudaError_t cudaEventElapsedTime ( float* ms,
      cudaEvent_t start, cudaEvent_t end )
  ```

  Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds).

Example 4.11:
A minimal performance measurement configuration:

```
cudaEvent_t start, stop;
cudaEventCreate(start);
cudaEventCreate(stop);
cudaEventRecord(start, 0);

// complete some tasks

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

float etime;
cudaEventElapsedTime( &etime, start, stop);
```

4.3.10 Streams

In Figure 4.8, we have already mentioned, that it is crucial to try and overlap computation and communication in a CUDA program transferring data between host and device. Here, we want to introduce additional details on how to do this, especially when we want to split the execution of our program into several portions of work. Thinking, e.g., of large matrix-vector operations where the entire data would not fit into the device memory, it may be helpful to split the work into several execution lines, where some are performing useful work, whilst the others are transferring the required data to the device or the host.

Definition 4.12 (Stream)**:**
Streams are a mechanism that introduces an additional level of parallelism into the CUDA framework. While the basic setup, we have seen until here, is SIMD or more precisely SIMT, using streams one can have the GPU do different things at the same time. Streams are not as flexible and "general purpose" as tasks on the host CPU, though.

We have ignored/neglected the concept already earlier, when we were talking about events. The function `cudaEventRecord()` (see Section 4.3.9) takes, as the last argument, the stream in which the event is to be recorded.

The basic power of streams is to have memory transfers and computational operations overlap in an *asynchronous* way. Starting from the Kepler architecture all devices support the overlapping operations. For pre-Kepler devices, however, the support can be incomplete and missing different features depending on the chip.

4.3.11 Page-Locked Memory on the Host

Asynchronous data transfers in CUDA are not only performed without synchronization to the actual computation, they are also intended to interact with the computation as little as possible. Especially, they should not interrupt the CPU from performing useful work in the program. They are therefore set up to use direct memory access (DMA) circumventing CPU interaction.

However, in order to do this, we need to use a special portion of host memory, that is guaranteed to stay in place during the operation. The default portion of host memory that we allocate using `malloc()` is paged memory. It can be anywhere in the virtual memory of the host and is allowed to move around, e.g., to get swapped to disk when more space is required.

Definition 4.13 (page-locked memory)**:**
Page-locked memory is a portion of memory that is guaranteed to keep its position in the virtual memory. It is not available for any kind of paging operations, such as swapping. Therefore, it is sometimes also called *pinned* memory.

Advantages of pinned memory:

- can be used for DMA safely
- transfer speeds can be up to 2× faster than to/from pageable memory

Disadvantages:

- memory fragmentation increases and thus the usability deteriorates.

4.3.12 Streams and Compute Capabilities

Over the years, NVIDIA® has changed the way things are implemented. This is not only regarding the API in the CUDA toolkit, but also the underlying device hardware. The very first CUDA enabled devices could not overlap transfers and executions at all. Then, some devices used separate engines for copy and kernel executions. Modern hardware usually has even two engines for performing transfers in direction to the host and to the device separately. Basically, we can classify the devices as follows:

Comp. Capab.	Properties
1.0	No overlap
1.1-	1 copy engine and 1 kernel execution engine
2.x-	1 kernel execution engine, 1 copy to host engine and 1 copy to device engine
3.5-	eliminates the differences in asynchronous execution

Table 4.2: classification of CUDA enabled devices with respect to the ability of overlapping memory transfers and computations.

How can I know what my device can do?

The `cudaDeviceProp` structure can be used to find out whether a device supports overlapped operation and how many execution engines are available. The important members are

- `int deviceOverlap` indicating the availability of overlapped operations

- `int asyncEngineCount` storing the number of asynchronous execution engines available.

The important information, which type of asynchronous execution model is implemented in the hardware can thus be fetched with the function `cudaGetDeviceProperties()`.

4.3.13 An Introductory Asynchronous Transfer Example

We will see in the following example from the NVIDIA® developer's blog, why it is so crucial to know the exact model.

What we want to do is

- copy data to the device
- perform some task (kernel) on it
- get the result back to the host

Example 4.14:
The the critical portion of the code would look like

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a)
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

according to what we have learned until now. This is regarding the default execution stream and uses synchronous communication.

Non Default Streams Before getting into the details on how to arrange code regarding asynchronous execution, we will now have a look on the generation of streams and assignment of tasks to the streams.

Example 4.15 (Creation and Destruction of Streams)**:**
Consider we have the two variables

```
 cudaStream_t stream1;
 cudaError_t result;
```

Then we can create a new stream using

```
 result = cudaStreamCreate(&stream1);
```

and later get rid of it via

```
 result = cudaStreamDestroy(stream1);
```

Example 4.16 (Memory transfers)**:**
Once we have acquired a new stream we have to tell the asynchronous copy routines to use it. The basic command `cudaMemcpyAsync()` takes the same arguments as `cudaMemcpy`. Only, it has an additional argument specifying the stream to use:

```
result = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice
    , stream1);
```

Example 4.17 (Kernel Execution)**:**
We need to use the extended launch size specification here:

```
<<< block distr., thread distr., dyn. mem. per block,
    associated stream >>>
```

The third argument can be used to allocate additional dynamic shared memory per block. We will use 0 here.

```
kernel<<<1,N,0,stream1>>>(d_a);
```

The influence of the number of engines (especially for copying data) is best displayed in a simple example.

Consider we have a group of streams cooperating on `kernel()`. Think of a situation where splitting the problem data into chunks is necessary to fit the data into the device memory. We basically have two ways to implement the cooperation,

1. loop over the entire copy-work-copy block
2. loop over the work and copies separately

Note that the asynchronous copy acts different on the control flow than the `cudaMemcpy()`. While in the default stream, using `cudaMemcpy()`, we can rely on the fact that as soon as the command returns, all data has been transferred, in the case of `cudaMemcpyAsync()` it does not even guarantee that the copy operation has started at all. It will only have scheduled the operation in a first in first out (FIFO) list of pending operations on the corresponding asynchronous execution engine.

Example 4.18 (Asynchronous Execution Version 1)**:**
Looping over the entire block of copy-work-copy operations is described by the

following code fragment

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes,
                  cudaMemcpyHostToDevice, stream[i]);
  kernel<<<>>>(d_a, offset);
  cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes,
                  cudaMemcpyDeviceToHost, stream[i]);
}
```

Example 4.19 (Asynchronous Execution Version 2)**:**
Looping over the single tasks in contrast looks like

```
for (int i = 0; i < nStreams; ++i) {
 int offset = i * streamSize;
 cudaMemcpyAsync(&d_a[offset], &a[offset],
                 streamBytes, cudaMemcpyHostToDevice,
                    stream[i]);
}

for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  kernel<<<>>>(d_a, offset);
}

for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&a[offset], &d_a[offset],
                  streamBytes, cudaMemcpyDeviceToHost,
                     stream[i]);
}
```

The main difficulty in the second variant stems from a problematic signaling on the C2050 series chips.

Kepler generation chips and later

These chips feature compute capabilities 3.5 and higher. Here, the time line looks as in “asynchronous version 1” in Figure 4.10 in both cases.

For completeness we list the entire test program that can be found at github[1], as well.

[1]https://github.com/parallel-forall/code-samples/blob/master/series/cuda-cpp/overlap-data-transfers/async.cu

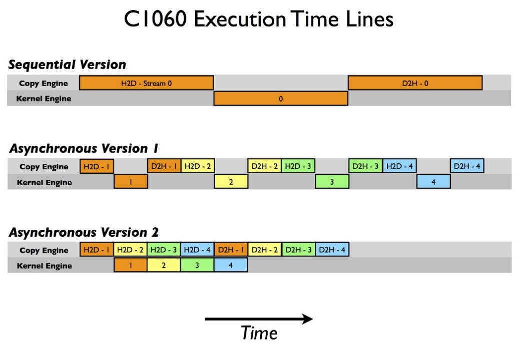


Figure 4.9: Execution time line on a device with a single copy engine.

```
  // Copyright 2012 NVIDIA Corporation

// Licensed under the Apache License, Version 2.0 (the "
   License");
// you may not use this file except in compliance with the
   License.
// You may obtain a copy of the License at

// http://www.apache.org/licenses/LICENSE-2.0

// Unless required by applicable law or agreed to in
   writing, software
// distributed under the License is distributed on an "AS
   IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
   express or implied.
// See the License for the specific language governing
   permissions and
// limitations under the License.

#include <stdio.h>

// Convenience function for checking CUDA runtime API
   results
// can be wrapped around any runtime API call. No-op in
   release builds.
inline
cudaError_t checkCuda(cudaError_t result)
{
#if defined(DEBUG) || defined(_DEBUG)
```

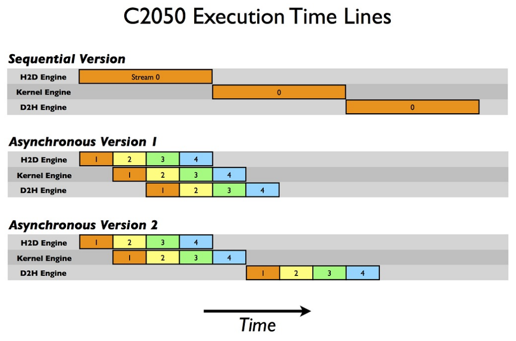


Figure 4.10: Execution time line on a device with separate copy engines for device to host (D2H) and host to device (H2D) operations.

```
  if (result != cudaSuccess) {
    fprintf(stderr, "CUDA␣Runtime␣Error:␣%s\n",
        cudaGetErrorString(result));
    assert(result == cudaSuccess);
  }
#endif
  return result;
}

__global__ void kernel(float *a, int offset)
{
  int i = offset + threadIdx.x + blockIdx.x*blockDim.x;
  float x = (float)i;
  float s = sinf(x);
  float c = cosf(x);
  a[i] = a[i] + sqrtf(s*s+c*c);
}

float maxError(float *a, int n)
{
  float maxE = 0;
  for (int i = 0; i < n; i++) {
    float error = fabs(a[i]-1.0f);
    if (error > maxE) maxE = error;
  }
  return maxE;
}

int main(int argc, char **argv)
{
```

```
  const int blockSize = 256, nStreams = 4;
  const int n = 4 * 1024 * blockSize * nStreams;
  const int streamSize = n / nStreams;
  const int streamBytes = streamSize * sizeof(float);
  const int bytes = n * sizeof(float);

  int devId = 0;
  if (argc > 1) devId = atoi(argv[1]);

  cudaDeviceProp prop;
  checkCuda( cudaGetDeviceProperties(&prop, devId));
  printf("Device␣:␣%s\n", prop.name);
  checkCuda( cudaSetDevice(devId) );

  // allocate pinned host memory and device memory
  float *a, *d_a;
  checkCuda( cudaMallocHost((void**)&a, bytes) ); // host
     pinned
  checkCuda( cudaMalloc((void**)&d_a, bytes) ); // device

  float ms; // elapsed time in milliseconds

  // create events and streams
  cudaEvent_t startEvent, stopEvent, dummyEvent;
  cudaStream_t stream[nStreams];
  checkCuda( cudaEventCreate(&startEvent) );
  checkCuda( cudaEventCreate(&stopEvent) );
  checkCuda( cudaEventCreate(&dummyEvent) );
  for (int i = 0; i < nStreams; ++i)
    checkCuda( cudaStreamCreate(&stream[i]) );

  // baseline case - sequential transfer and execute
  memset(a, 0, bytes);
  checkCuda( cudaEventRecord(startEvent,0) );
  checkCuda( cudaMemcpy(d_a, a, bytes,
     cudaMemcpyHostToDevice) );
  kernel<<<n/blockSize, blockSize>>>(d_a, 0);
  checkCuda( cudaMemcpy(a, d_a, bytes,
     cudaMemcpyDeviceToHost) );
  checkCuda( cudaEventRecord(stopEvent, 0) );
  checkCuda( cudaEventSynchronize(stopEvent) );
  checkCuda( cudaEventElapsedTime(&ms, startEvent,
     stopEvent) );
  printf("Time␣for␣sequential␣transfer␣and␣execute␣(ms):␣%f
     \n", ms);
  printf("␣␣max␣error:␣%e\n", maxError(a, n));

  // asynchronous version 1: loop over {copy, kernel, copy}
  memset(a, 0, bytes);
  checkCuda( cudaEventRecord(startEvent,0) );
  for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    checkCuda( cudaMemcpyAsync(&d_a[offset], &a[offset],
                               streamBytes,
                                   cudaMemcpyHostToDevice,
                               stream[i]) );
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i
        ]>>>(d_a, offset);
    checkCuda( cudaMemcpyAsync(&a[offset], &d_a[offset],
                               streamBytes,
                                   cudaMemcpyDeviceToHost,
                               stream[i]) );
  }
  checkCuda( cudaEventRecord(stopEvent, 0) );
  checkCuda( cudaEventSynchronize(stopEvent) );
  checkCuda( cudaEventElapsedTime(&ms, startEvent,
      stopEvent) );
  printf("Time␣for␣asynchronous␣V1␣transfer␣and␣execute␣(ms
      ):␣%f\n", ms);
  printf("␣max␣error:␣%e\n", maxError(a, n));

  // asynchronous version 2:
  // loop over copy, loop over kernel, loop over copy
  memset(a, 0, bytes);
  checkCuda( cudaEventRecord(startEvent,0) );
  for (int i = 0; i < nStreams; ++i)
  {
    int offset = i * streamSize;
    checkCuda( cudaMemcpyAsync(&d_a[offset], &a[offset],
                               streamBytes,
                                   cudaMemcpyHostToDevice,
                               stream[i]) );
  }
  for (int i = 0; i < nStreams; ++i)
  {
    int offset = i * streamSize;
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i
        ]>>>(d_a, offset);
  }
  for (int i = 0; i < nStreams; ++i)
  {
    int offset = i * streamSize;
    checkCuda( cudaMemcpyAsync(&a[offset], &d_a[offset],
                               streamBytes,
                                   cudaMemcpyDeviceToHost,
                               stream[i]) );
  }
  checkCuda( cudaEventRecord(stopEvent, 0) );
```

```
  checkCuda( cudaEventSynchronize(stopEvent) );
  checkCuda( cudaEventElapsedTime(&ms, startEvent,
      stopEvent) );
  printf("Time␣for␣asynchronous␣V2␣transfer␣and␣execute␣(ms
      ):␣%f\n", ms);
  printf("␣max␣error:␣%e\n", maxError(a, n));

  // cleanup
  checkCuda( cudaEventDestroy(startEvent) );
  checkCuda( cudaEventDestroy(stopEvent) );
  checkCuda( cudaEventDestroy(dummyEvent) );
  for (int i = 0; i < nStreams; ++i)
    checkCuda( cudaStreamDestroy(stream[i]) );
  cudaFree(d_a);
  cudaFreeHost(a);

  return 0;
}
```

4.3.14 Other Topics

Interoperability with Graphics

Using the same GPU for computations and graphical display of results is possible. See, e.g. [19, Chapter 8], or [3].

Usage of Multiple GPUs

Usage of multiple GPUs in a single program requires the concepts of *zero-copy host memory*, and *portable pinned memory*. An introduction can be found in [19, Chapter 11].

4.4 Open Computing Language (OpenCL)

Main Message

The abstraction for the programming and hardware models are very similar to the CUDA concepts. Mainly OpenCL delivers slightly more flexible implementations due to vendor independence and uses slightly different vocabulary for the single ingredients of the concept.

CUDA	OpenCL
thread	(Work) item
block	(Work) group
streaming multiprocessor	compute unit
(CUDA) processor	processing unit

Table 4.3: A short CUDA to OpenCL dictionary

4.5 Hybrid CPU-GPU Linear System Solvers

One of the most prominent uses of GPUs is in the actual sense of an accelerator. The host is responsible for the execution flow anyway. It will keep performing large portions of the computational task. However, whenever the GPU is much better suited for certain operations, these will be offloaded to the device. Thus a hybrid implementation results. We will demonstrate this using the block outer product LU formulation.

4.5.1 The block outer product LU decomposition revisited

Our main question to be treated in this section is how Algorithm 3.1 can be exploited to set up hybrid solvers using both CPU and GPU as good as possible. We have seen in Section 3.10 that it can be very beneficial to split the computations of W and Z into computations with smaller blocks that can then be scheduled more flexibly. The keyword there was DAG scheduling to optimize execution times.

The central question for the hybrid CPU/GPU version of the algorithm now is where to execute the single steps of the algorithm compared to the DAG scheduled version.

Requirements

We want to take into account the following necessities and assumptions when developing the hybrid version.

- Keep data transfers between host and device limited
- optimize usage of both host and device features
- assume that the entire matrix fits into the device memory.

The assumption on the matrix size may be loosened but will then lead to a completely different algorithm.

In each outer iteration step perform the leading $r \times r$ blocks LU decomposition

4.5.2 Iterative Linear System Solvers

Consider the CG Algorithm 3.4 as one prototype iterative solver. It contains basically all important computational steps present in any Krylov subspace method for solving a linear system of equations.

There are mainly two observations we can draw from the algorithm.

1. The single steps need to be executed mainly sequentially
2. basically all operations are vector operations.

There is not much to distribute between host and device. To exploit the devices vector features all operations should be executed on the device. In case the matrix can not be stored in device memory completely it may be beneficial to use streams to split the operation into chunks that can be stored and operate on those streams in a round robin fashion.

4.5.3 Sparse Iterative Eigenvalue Approximation

Basic Idea

- Very similar to iterative linear solvers based on Krylov subspaces.
- Main ingredient is to use the basis of the subspace to project the eigenvalue problem to a much smaller space and solve it with dense methods there, i.e. $A \in \mathbb{R}^{n \times n}$ large and sparse $U \in \mathbb{R}^{m \times n}$, $m \ll n$ orthogonal, then
$$\underbrace{UAU^T}_{m \times m} x = \lambda x$$
is an m-dimensional dense eigenproblem.

Here one can offload the solution of the small eigenvalue problem to the host, while the device keeps extending the basis further. The host can then decide whether the approximation is good enough, or the extension is required and the computation needs to continue.

4.6 Relevant Software and Libraries

4.6.1 The CUDA Related Libraries

- **CUDA Math** provides basically all math functions in `math.h` as device functions.

- **CUBLAS** the CUDA device based implementation of BLAS
- **CUFFT** CUDA based Fast Fourier Transforms, i.e., divide and conquer based computation of Fourier transforms of complex and real valued data sets.
- **CURAND** The CURAND library provides facilities that focus on the simple and efficient generation of high-quality pseudorandom and quasirandom numbers.
- **CUSPARSE** Vector-vector and matrix-vector operations where at least one participant is sparse.
- **Thrust** A C++ template library based on the Standard Template library (STL) for minimal effort implementation of parallel programs.
- **CUSOLVER** Solvers for $Ax = b$, or $x = \operatorname{argmin}_z \|Az - b\|$ and sequences thereof. (both sparse and dense)

4.6.2 Derived Libraries

Matrix Algebra on GPU and Multicore Architectures (MAGMA)[2]

"The MAGMA project aims to develop a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures, starting with current "Multicore+GPU" systems.

The MAGMA research is based on the idea that, to address the complex challenges of the emerging hybrid environments, optimal software solutions will themselves have to hybridize, combining the strengths of different algorithms within a single framework. Building on this idea, we aim to design linear algebra algorithms and frameworks for hybrid manycore and GPU systems that can enable applications to fully exploit the power that each of the hybrid components offers."

Formal Linear Algebra Methodology Environment (FLAME)[3]

"The objective of the FLAME project is to transform the development of dense linear algebra libraries from an art reserved for experts to a science that can be understood by novice and expert alike. Rather than being only a library, the project encompasses a new notation for expressing algorithms, a methodology for systematic derivation of algorithms, Application Program Interfaces (APIs) for representing the algorithms in code, and tools for mechanical derivation, implementation and analysis of algorithms and implementations."

CUSP[4]

[2] http://icl.cs.utk.edu/magma/index.html
[3] http://www.cs.utexas.edu/~flame/web/
[4] https://github.com/cusplibrary

"Cusp is a library for sparse linear algebra and graph computations on CUDA. Cusp provides a flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems. Get Started with Cusp today!"

Matrix formats:

- Coordinate (COO)
- Compressed Sparse Row (CSR)
- Diagonal (DIA)
- ELL (ELL)
- Hybrid (HYB)

More Features:

- Format conversion
- Dense Arrays
- File I/O (Matrix Market format)

Supported Iterative Solvers:

- Conjugate-Gradient (CG)
- Biconjugate Gradient (BiCG)
- Biconjugate Gradient Stabilized (BiCGstab)
- Generalized Minimum Residual (GMRES)
- Multi-mass Conjugate-Gradient (CG-M)
- Multi-mass Biconjugate Gradient stabilized (BiCGstab-M)

Preconditioners:

- Algebraic Multigrid (AMG) based on Smoothed Aggregation
- Approximate Inverse (AINV)
- Diagonal

CULA tools[5]

"CULA is a set of GPU-accelerated linear algebra libraries utilizing the NVIDIA CUDA parallel computing architecture to dramatically improve the computation speed of sophisticated mathematics."

[5] http://www.culatools.com

They have separate packages for sparse and dense operation. The libraries are however commercial.

Besides those, there are many scientific computing packages that support GPU operations in one way or the other. Also python has packages for both CUDA (pyCUDA) and OpenCL (pyOpenCL) and MATLAB supports (basically dense only) operation on CUDA devices.

Ginkgo[6]

"Ginkgo is a high-performance linear algebra library for manycore systems, with a focus on sparse solution of linear systems. It is implemented using modern C++ (you will need at least C++11 compliant compiler to build it), with GPU kernels implemented in CUDA."

Matrix formats:

- Coordinate (COO)
- Compressed Sparse Row (CSR)
- hybrid
- dense
- ELL-P
- SELL-P

More Features:

- Format conversion
- Dense Arrays
- File I/O (Matrix Market format)
- UFL Collection

Supported Iterative Solvers:

- Conjugate-Gradient (CG)
- Biconjugate Gradient (BiCG)
- Biconjugate Gradient Stabilized (BiCGstab)
- Conjugate-Gradient squared (CGS)
- flexible CG (FCG)

[6] https://github.com/ginkgo-project/ginkgo

- Generalized Minimum Residual (GMRES)

Preconditioners:

- Jacobi type

CHAPTER 5

Distributed Memory Systems

Contents

5.1 Distributed Memory Hierarchy

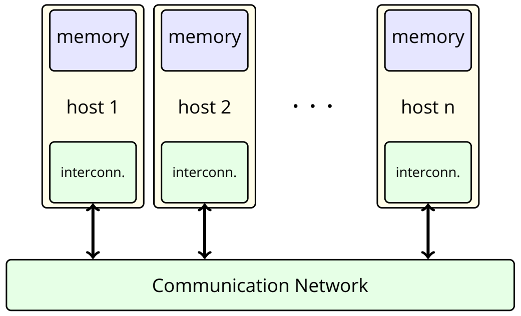


Figure 5.1: Distributed memory computer schematic

5.2 Comparison of Distributed Memory Systems

5.2.1 Rankings

Several approaches have been followed to compare the large HPC Cluster systems to each other leading to different kinds of rankings. The classical approach is following the idea, that linear algebra operations are the core component of most scientific computing codes. Sophisticated benchmarks in that area have been developed and serve as comparison criterion for the performance of those systems. Over the years it became more and more obvious, and today it is the most pressing difficulty for large computing centers, that the power consumption of the systems is becoming an issue for their operation. The largest systems

today consume the power provided by a moderate power plant. For example the Japanese K computer would use the energy providable by 8–10 medium size (66m diameter rotor) wind turbines. This development gave rise to the idea of including the energy consumption for operation and cooling of the devices in the comparison. More recent developments like social networks have made operations on graphs more attractive and necessary than before. The centrality of certain nodes in such networks is a question often asked. This usually requires specialized algorithms and possibly hardware. The three most important rankings in view of this are described in the following.

1. **TOP500[1]:** List of the 500 fastest HPC machines in the world sorted by their maximal LINPACK[2] performance (in TFlops) achieved.
2. **Green500[3]:** Taking into account the energy consumption the Green500 is basically a resorting of the TOP500 according to TFlops/Watt as the ranking measure.
3. **(Green) Graph500[4]:**Designed for data intensive computations it uses a graph algorithm based benchmark to rank the supercomputers with respect to GTEPS (10^9 Traversed edges per second). As for the TOP500 a resorting of the systems by an energy measure is provided, as the Green Graph 500 list[5].

5.2.2 Architectural Streams Currently Pursued

The ten leading systems in the TOP500 list are currently (list of November 2018) of three different types representing the main streams pursued in increasing the performance of distributed HPC systems.

Mainly all HPC systems today consist of single hosts of one of the following three types. The performance boost is achieved by connecting ever increasing numbers of those hosts in large clusters.

1. **Hybrid accelerator/CPU hosts**, `Summit` — IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM at DOE/SC/Oak Ridge National Laboratory United States `Piz Daint` — Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect, NVIDIA Tesla P100, Cray Inc. at Swiss National Supercomputing Centre (CSCS) Switzerland
2. **Manycore and embedded hosts** `Sunway TaihuLight` — Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC `Sequoia` — BlueGene/Q, Power BQC 16C 1.60 GHz at DOE/NNSA/LLNL United States

[1] http://www.top500.org/
[2] http://www.netlib.org/benchmark/hpl/
[3] http://www.green500.org/
[4] http://www.graph500.org/
[5] http://green.graph500.org/

3. **Multicore CPU powered hosts**, `SuperMUC-NG` — ThinkSystem SD530, Xeon Platinum 8174 24C 3.1GHz, Intel Omni-Path, Lenovo at Leibniz Rechenzentrum Germany

5.2.3 Hybrid Accelerator/CPU Hosts

We have elaborately studied these hosts in the previous chapter.

Compared to a standard desktop (as treated there) in the cluster version the interconnect plays a more important role. Especially, Multi-GPU features may use GPUs on remote hosts (as compared to remote NUMA nodes) more efficiently due to the high speed interconnect.

Compared to CPU-only hosts, these systems usually benefit from the large number of cores generating high flop-rates at comparably low energy costs.

5.2.4 Manycore and Embedded Hosts

Manycore and embedded systems are designed to use low power processors to get a good flop per Watt ratio. They make up for the lower per core flop counts by using enormous numbers of cores.

BlueGene/Q

- Base chip IBM PowerPC 64Bit based, 16(+2) cores, 1.6GHz
- each core has a SIMD Quad-vector double precision FPU
- 16 user cores, 1 system assist core, 1 spare core
- cores connected to 32MB eDRAM L2Cache (half core speed) via crossbar switch
- crates of 512 chips arranged in 5d torus ($4 \times 4 \times 4 \times 4 \times 2$)
- chip-to-chip communication at 2Gbit/s using on-chip logic
- 2 crates per rack $\rightsquigarrow$ 1024 compute nodes $=$ 16,384 user cores
- interconnect added in 2 drawers with 8 PCIe slots (e.g. for Infiniband, or 10Gig Ethernet.)

5.2.5 Multicore CPU Hosts

Basically these clusters are a collection of standard processors. The actual multicore processors, however, are not necessarily of x86 or amd64 type, e.g. many employ IBM Power 9 processors.

Standard x86 or amd64 provide the obvious advantage of easy usability, since software developed for standard desktops can be ported easily. The SPARC and POWER processors overcome some of the x86 disadvantages (e.g. expensive task switches) and thus often provide increased performance due to reduced latency.

5.2.6 The 2020 vision: Exascale Computing

difference	name (symbol)	meaning
	Kilobyte (kB)	10^3 Byte = 1 000 Byte
2,40%	Kibibyte (KiB)	2^{10} Byte = 1 024 Byte
	Megabyte (MB)	10^6 Byte = 1 000 000 Byte
4,86%	Mebibyte (MiB)	2^{20} Byte = 1 048 576 Byte
	Gigabyte (GB)	10^9 Byte = 1 000 000 000 Byte
7,37%	Gibibyte (GiB)	2^{30} Byte = 1 073 741 824 Byte
	Terabyte (TB)	10^{12} Byte = 1 000 000 000 000 Byte
9,95%	Tebibyte (TiB)	2^{40} Byte = 1 099 511 627 776 Byte
	Petabyte (PB)	10^{15} Byte = 1 000 000 000 000 000 Byte
12,6%	Pebibyte (PiB)	2^{50} Byte = 1 125 899 906 842 624 Byte
	Exabyte (EB)	10^{18} Byte = 1 000 000 000 000 000 000 Byte
15,3%	Exbibyte (EiB)	2^{60} Byte = 1 152 921 504 606 846 976 Byte
	Zettabyte (ZB)	10^{21} Byte = 1 000 000 000 000 000 000 000 Byte
18,1%	Zebibyte (ZiB)	2^{70} Byte = 1 180 591 620 717 411 303 424 Byte
	Yottabyte (YB)	10^{24} Byte = 1 000 000 000 000 000 000 000 000 Byte
20,9%	Yobibyte (YiB)	2^{80} Byte = 1 208 925 819 614 629 174 706 176 Byte

Table 5.1: decimal and binary prefixes

The two standard prefixes in decimal and binary representations of memory sizes are given in Table 5.1. The decimal prefixes are also used for displaying numbers of floating point operations per second (flops) executed by a certain machine.

name (location)	cores	LINPACK perfomance [TFlop/s]
Summit (USA)	2 397 824	143 500.0
Sunway TaihuLight(China)	10 649 600	93 014.6
Sequoia (USA)	1 572 864	16 324.8
Tianhe-2A (China)	4 981 760	61 444.5

Table 5.2: Petascale systems available

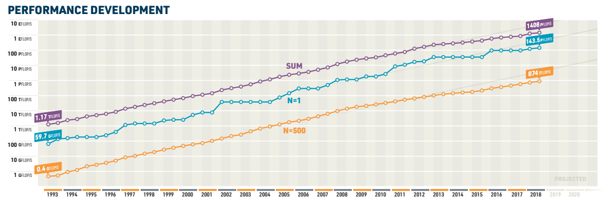


Figure 5.2: Performance development of TOP500 HPC machines taken from TOP500 poster November 2014

5.2.7 State of the art (statistics)

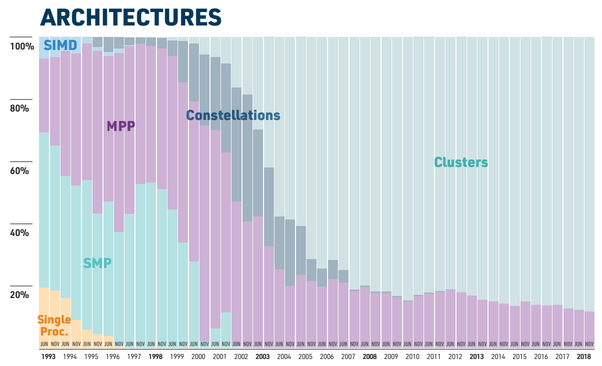


Figure 5.3: TOP500 architectures taken from TOP500 poster November 2018

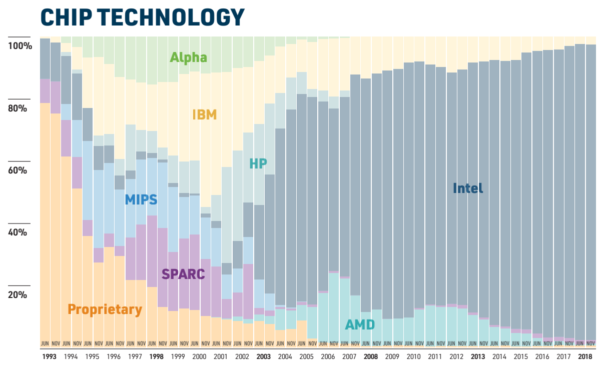


Figure 5.4: Chip technologies of TOP500 HPC machines taken from TOP500 poster November 2018

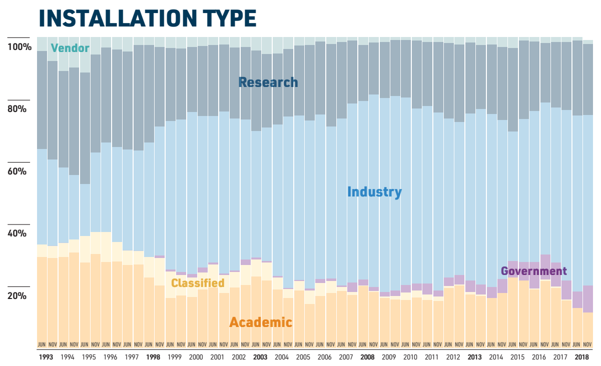


Figure 5.5: Installation types of TOP500 HPC machines taken from TOP500 poster November 2018

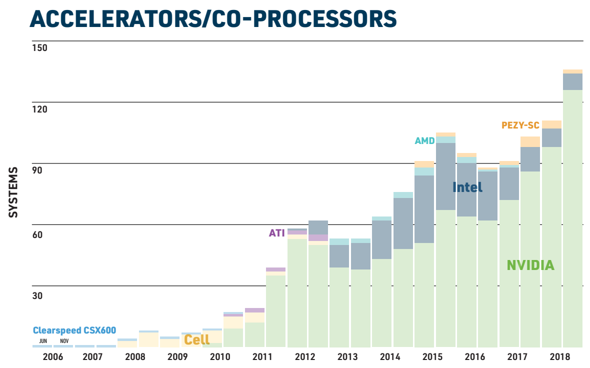


Figure 5.6: Accelerators and Co-Processors employed in TOP500 HPC machines taken from TOP500 poster November 2018

5.3 Communication of Data

5.3.1 Communication via Message Passing

Message passing

is the programming model commonly used for distributed memory systems, where each node has its own exclusive memory and we have an overall distributed address space. Exchange of data between the local memories of separate hosts is realized by sending messages between the hosts.

Usually, the communication is (network) socket based, although the basic principles can also be applied to multicore machines, e.g., by using shared memory blocks to implement the communication.

Blocking vs. Non-blocking Communication operations in the Message Passing Interface (MPI) are belonging to two global classes categorized by their local (process on host) behavior.

Definition 5.1 (blocking operation)**:**
A communication operation is called *blocking* if the return of the process control to the calling process means that the operation has completed the entire transfer.

Definition 5.2 (non-blocking operation)**:**
In a *non-blocking* operation the process control is returned to the calling process as soon as the communication has been initiated. The communication may be ongoing while the calling process continues its program.

Synchronous vs. Asynchronous Looking at the same operations from a global perspective, i.e., not looking at the local message but the global communication, they determine the two classes of

Definition 5.3 (synchronous communication)**:**
The *synchronous* communication between a sending and a receiving process is implemented such that sending operations do not complete (i.e. return control to the calling process) before the receiving counterpart has at least started the execution.

Definition 5.4 (asynchronous communication)**:**
In *asynchronous* communication the sending and receiving processes are not coordinated, i.e., the sender can execute its operation without the receiving counterpart waiting in its operation.

Example 5.5:
We know counterparts of those types of communication in our daily life:

- oral or telephone chats are synchronous communications, since all partners are engaged in the communication simultaneously.
- classic mail or electronic mail are asynchronous communication, where the sender never knows if, or when the message was actually received.

Communication Types Communication between MPI processes can not only be classified via their influence on global or local process flow, but also with respect to the number of partners involved. MPI is distinguishing between

- **point-to-point communication**, where both ends are occupied by a single process, and
- **collective communication** where a single process sends out messages to multiple receiving processes, or collects messages from several sending processes.

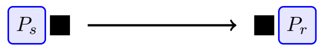


Figure 5.7: Point-to-Point Communication

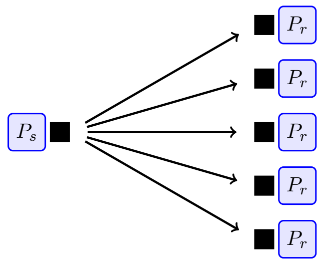


Figure 5.8: Broadcast Operation

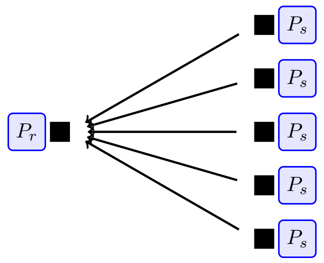


Figure 5.9: Reduction Operation

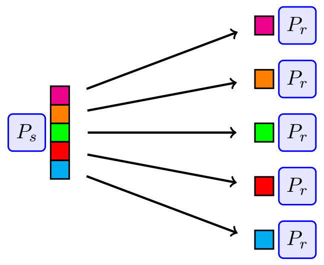


Figure 5.10: Scatter Operation

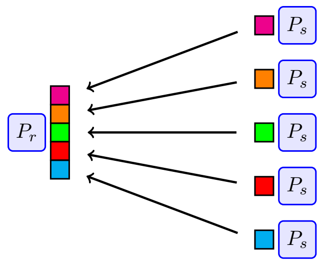


Figure 5.11: Gather Operation

Additionally we have two global, i.e. all-to-all-type, communication types.

Multi-broadcast

This is the situation when all nodes issue a broadcast at the same time, i.e. each node sends the exact same message to all other nodes.

Total exchange

This is the situation when all nodes issue a scatter at the same time, i.e. each node sends a specific message to to every other node.

5.4 Communication Networks (revisited)

5.4.1 Asymptotic Message runtimes in some Standard Network Topologies

Assumptions

- All network links are bidirectional
- All-Port-Communication: each node can send out messages on all outgoing links simultaneously
- The same holds for receiving messages
- A messages consists of several bytes sent uninterruptedly
- The time for transmission of a message of m bytes size is

$$T(m) = t_s + mt_b,$$

where t_s is a startup time for initialization of the communication and t_b is the time for sending a single byte.

- The communication is such that the length of the path from source node to destination node in the corresponding network graph determines the number of time steps required.

Landau Θ-notation

The $\Theta(g(x))$ notation describes a class of functions f for which roughly speaking we have that *"f is growing essentially as fast as g."*

More precisely we have,

$$\begin{aligned}\Theta(g(x)) = \{f(x) \mid \exists c_1, c_2 > 0 \text{ and } x_0,\ &\text{such that}\\ &\forall x \geqslant x_0\ \ c_1|g(x)| \leqslant f(x) \leqslant c_2|g(x)|\}\end{aligned}$$

This basically means $f \in \Theta(g)$ when $f \in \mathcal{O}(g)$ and $g \in \mathcal{O}(f)$.

Critical operations are the collective communication operations, since they produce a notable load on the entire range of links in the network. We will investigate the following in more detail:

1. broadcast
2. scatter
3. multi-broadcast (each node broadcasts)
4. total exchange (each node scatters)

In the following p specifies the number of nodes in the network.

Complete Graph

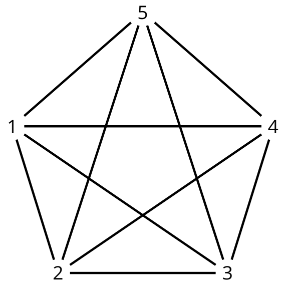


Figure 5.12: A complete graph network broadcast example

- all nodes connected, i.e., path length is one,
- by the assumptions all messages in all types of point to point and collective communication operations can be sent simultaneously,
- the operations can be performed in $\Theta(1)$.

Linear Array

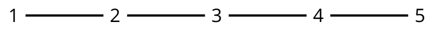


Figure 5.13: A linear array network example

Single Broadcast

- The root node sends messages to its left and right neighbors starting with the most distant recipients,
- in all other steps each node forwards the message received from one neighbor in the previous step to its other neighbor.
- The minimal runtime is $\lfloor \frac{p}{2} \rfloor$ (root is the center node)
- The maximal runtime is $p - 1$ (root is an end node)
- Thus the runtime class is $\Theta(p)$.

Multi Broadcast

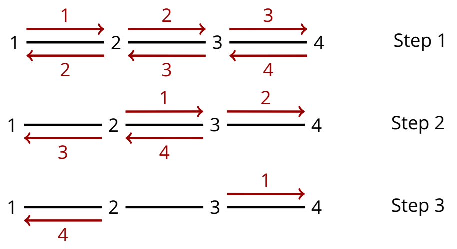


Figure 5.14: A linear array network multi broadcast example

Here the arrows represent messages with sending direction and the numbers indicate the original source node. The runtime is the worst case runtime of the single broadcast and thus the complexity is $\Theta(p)$.

Scatter

The basic idea is that of the single broadcast, only the contents of the messages need to be treated more carefully. Therefore, the complexity is $\Theta(p)$ as well.

Total exchange

An upper bound to the runtime is given by p scatter operations, resulting in basically p^2 communication steps. In [18, Section 4.3.1.3] the authors present an algorithm that can do it in $\frac{p^2}{4}$. Anyway the complexity is $\Theta(p^2)$.

Ring

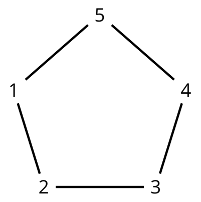


Figure 5.15: A ring network example

The ring is a prototype for the linear array where the root node is always in the center. Thus, we get the same complexities as in the best case for the linear array.

Note, however, that we need to cut the transmission at half way around the ring.

Mesh

We consider the d-dimensional mesh with $\sqrt[d]{p}$ nodes per direction, such that we have p nodes in total as before. The diameter then is $d(\sqrt[d]{p}-1)$.

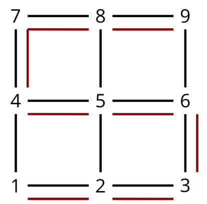
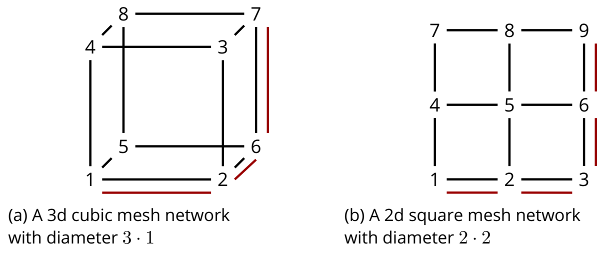


Figure 5.17: The linear array embedded in a 2d mesh.



(a) A 3d cubic mesh network with diameter $3 \cdot 1$

(b) A 2d square mesh network with diameter $2 \cdot 2$

Figure 5.16: Two mesh network examples with diameter indications.

Single Broadcast

The single broadcast time is obviously proportional to the diameter of the network. This itself is proportional to the number of nodes in each direction. Therefore, the complexity class is $\Theta(\sqrt[d]{p})$.

Scatter

- Figure 5.17 shows clearly that the communication time is limited by that for the linear array from above.
- On the other hand, each node has d to $2d$ outgoing connections and $p-1$ messages need to be sent, i.e., $\lfloor\frac{p-1}{d}\rfloor$ is a lower limit.
- a scatter is possible in $\Theta(p)$.

Multi Broadcast

The multi broadcast time is observed similarly to be part of the complexity class $\Theta(\sqrt[d]{p})$.

Total Exchange

The authors in [18, Section 4.3.1.5] show a method that provides $\Theta(p^{\frac{d+1}{d}})$.

5.4.2 Some Remarks on the Hypercube

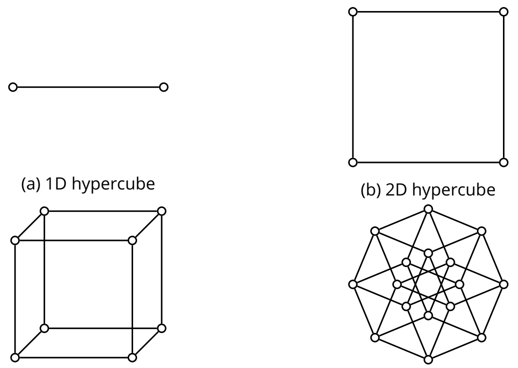

(a) 1D hypercube (b) 2D hypercube
(c) 3D hypercube (d) 4D hypercube

Figure 5.18: The first four hypercubes

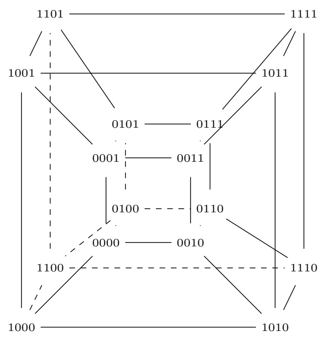


Figure 5.19: The hypercube network in 4d

We denote the nodes in the d-dimensional hypercube by d-tuples of bits, i.e., we use $n_1, \ldots, n_p \in \{0,1\}^d$. Let $a, b, c \in \{0,1\}^d$ and a_i, b_i, c_i the i-th bit positions.

We denote by $\oplus$ the bitwise exclusive `or` operation, i.e.,

$$a_1 \ldots a_d \oplus b_1 \ldots b_d = c_1 \ldots c_d$$

with

$$c_i = \begin{cases} 1 & \text{where } a_i \neq b_i, \\ 0 & \text{otherwise} \end{cases} \quad \text{for } 1 \leqslant i \leqslant d.$$

Note that $\forall z \in \{0,1\}^d$ we have

$$00\ldots0 \oplus z = z,$$

and if $v, w \in \{0,1\}^d$ differ in only a single bit, so do $v \oplus z$ and $w \oplus z$.

Properties of the Hypercube graph

- nodes are bit d-tuples,
- each node has d links to other nodes
- neighbors differ in a single bit position
- the diameter of the graph (i.e., the length of the longest path between two nodes) is $d = \log(p)$.

5.4.3 Communication Routing on the Hypercube

Construction of Spanning Trees for Single Broadcasts

Definition 5.6 (Spanning tree)**:**
A *spanning tree* of a graph is a tree that

- picks one node of the graph as its root,
- contains all other nodes as nodes or leaves exactly once,
- has only edges that represent valid links in the graph.

Construction Rules for root $00\ldots0$

1. all root connections coincide with the links in the graph.
2. children are generated by inverting a single bit right of the rightmost 1.

The rules above imply

- that all leave nodes end on a 1 bit,
- the depth of the tree is $d+1$ since d bits are inverted on the path to the deepest leave $11\ldots1$.

Root nodes other than $00\ldots0$

Spanning trees for other root nodes v are derived by replacing all nodes w by $w \oplus v$ in the entire tree for root $00\ldots0$.

Why is this the case? We noted above the properties of $\oplus$ that

- $00\ldots0$ is the neutral element, and
- v, w differ in only a single bit $\Rightarrow$ $v \oplus z$, $w \oplus z$ do so as well.

Thus, if (v,w) is a hypercube link, then $(v \oplus z, w \oplus z)$ is one as well.

Single Broadcast

The single broadcast can be implemented in $\Theta(\log p) = \Theta(d)$ successively descending through the spanning tree. It can also not be better than that since the diameter of the hypercube is d.

Scatter

A scatter operation needs to send out $p-1$ different messages along the d links of the root node. It can thus not be faster than $\lceil\frac{p-1}{d}\rceil$ time steps.

We will see in the following that this is the time also needed for a multi-broadcast. Since a single scatter can not be slower than that we immediately have that a scatter is $\Theta(\frac{p-1}{\log(p)}) = \Theta(\frac{p-1}{d})$.

Collision Avoiding Spanning Trees for Multi-Broadcast Operations

Problem

The single broadcast spanning trees for the 2^d nodes in the d-dimensional hypercube are not disjoint in the sense that each link is only used by a single operation in each time step if the multi-broadcast is treated as 2^d isolated single broadcasts.

It is mandatory to construct spanning trees such that all sets of edges used in a single time step by the different single broadcasts are disjoint.

Definition 5.7:
- The spanning tree for root node $t \in \{0,1\}^d$ is called T_t, and simply T_0 for $t = 00\ldots0$.
- The set of edges active in time step i for T_t is called $A_i(t)$

Construction

The sets of active edges for root node $t \in \{0,1\}^d$ may be constructed such that for any two edges (x,y) and (x',y') in $A_i(0)$ x,y and x',y' do not differ in the same bit position and the sets for the other root nodes are derived as

$$A_i(t) = \{(x \oplus t, y \oplus t) \mid (x,y) \in A_i\} \qquad \forall 1 \leqslant i \leqslant m,$$

where m is the total number of time steps required.

The basic idea behind this is to assume the opposite. Then for two edges differing in the same bit position, we could find a node t such that the one edge can be transformed into the other using the $\oplus t$ operation. Thus the construction of the $A_i(t)$ would immediately lead to sets of edges that are not pairwise disjoint.

The set A_i of active edges in the i-th step can have at most d entries, since we only have d bit positions available in the node labels.

Construct the sets A_i such that $|A_i| = d$ for $1 \leqslant i < m$ and $|A_m| \leqslant d$.

What is m?
Since each of the $p = 2^d$ nodes in the tree has an incoming link, except the root, we have $2^d - 1$ edges in total that are distributed among the A_i, i.e.,

$$\left|\bigcup_{i=1}^{m} A_i\right| = 2^d - 1.$$

This immediately provides a first estimate for m:

$$m = \left\lceil \frac{2^d - 1}{d} \right\rceil$$

Note that we can also not get better than that, since each node in the hypercube has to receive $2^d - 1$ messages from the other nodes across its d incoming links.

Definition 5.8:
We collect some further notation:

- $N_k := \{t \in \{0,1\}^d \mid t$ has k unit bits and $d-k$ zero bits.$\}$
- These sets have
$$n_k := |N_k| = \begin{pmatrix} d \\ k \end{pmatrix} = \frac{d!}{k!(d-k)!}$$
elements.
- The N_k are further subdivided into m_k equivalence classes $R_{k1}, \dots, R_{km_k}$ with respect to left rotation. That means all elements in a set R_{ki} can be formed from each other by successive cyclic permutation to the left of the bit positions. They are ordered by rightmost concentration of the unit bits, i.e., R_{k1} is the class containing $(0^{d-k}1^k)$.
- The elements in the equivalence classes can be ordered by rightmost concentration of unit bits as well.
- $n(t)$ is the global number of node t in this order.
- $m(t) = 1 + [n(t) - 1 \mod d]$ is t's local number of inside the equivalence class. Note that $m(t)$ cycles across the boundaries of equivalence classes.

Note that the global ordering via $n(t)$ does not take over the local numbering. If $|R_{k-1m_{k-1}}| = \ell < d$ then in the global order the element with $m(t) = \ell + 1$ is the first one in R_{k1} in the global order. Let us denote the sets of destination nodes in A_i by E_i. Then we set:

$$\begin{aligned} E_0 &= \{00\dots0\} \\ E_i &= \{t \in \{0,1\}^d \mid (i-1)d + 1 \leqslant n(t) \leqslant id\} && 1 \leqslant i < m \\ E_m &= \{t \in \{0,1\}^d \mid (m-1)d + 1 \leqslant n(t) \leqslant 2^d - 1\} \end{aligned}$$

The set of active edges are then constructed by the rules:

1. connect $t \in E_i$ to start node t' with the $m(t)$th bit inverted,
2. if $t = 11\dots1$ and $m(t) = d$ connect to $t' = 101\dots1$ instead.

By construction in each step the tree uses d edges and all sets $A_i(t)$ for the different t are disjoint. Thus, all 2^d single broadcasts can be performed simultaneously and the multi-broadcast can be done in $\Theta(\frac{p-1}{d})$.

Note that although the d-hypercube has only $\frac{d}{2} \cdot 2^d$ edges we can use $d \cdot 2^d$ links in the graph due to the assumption of bidirectional communication.

5.5 Message Passing Interface API

The Message Passing Interface is a standard for creation of parallel programs using the message passing programming model. It describes

- functionality,
- behavior,
- API syntax

of the required routines. It does, however, not prescribe any implementation details. It is, e.g., completely open by what means a message is transferred.

The MPI uses a specialized execution environment that spawns and administrates the instances of a process. Relevant functions for

- setup and destruction of the working environments context
- grouping processes
- actual message transmission
- ...

are collected in the `mpi.h` header file. We will see in Section 5.6, for the case of the Open MPI[6] implementation of the standard, how we can compile and run a program using the MPI features.

5.5.1 MPI Context Initialization and Finalization

The most basic components of the MPI program are

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

to make the standard available. Then, before we can use any message passing routines, we need to initialize the execution context via

```
int MPI_Init(int *argc, char ***argv)
```

passing on the usual arguments of the `main()` function of our C program. After we have finished our MPI related work, the execution context is destroyed using

```
int MPI_Finalize()
```

[6] http://www.open-mpi.org/

Processes may continue performing local work after the finalization, but with a very few exceptions none of the MPI functions work anymore. It is mandatory to make sure that all MPI operations have finished before calling `MPI_Finalize()`.

5.5.2 Process Groups and Communicators

Process Groups

Definition 5.9 (Process group)**:**
Processes in MPI may be clustered in so called *process groups*. These are ordered sets of instances of the program numbered from 0 to $n-1$. The local numbers of the processes are called `rank`.

From the programmers view an MPI group is an object of type `MPI_Group`, which can be accessed via a handle. There exists one predefined group constant `MPI_GROUP_EMPTY`, denoting the empty group.

Process Group Functions

```
int MPI_Group_union(MPI_Group group1,
                    MPI_Group group2,
                    MPI_Group *newgroup)
```

Generates the union of two existing groups by including all elements of the first group, followed by all elements of second group that are not in the first group.

- `group1`, `group2` groups to include
- `*newgroup` handle of the group to create. This may be equal to the empty group `MPI_GROUP_EMPTY`.

The operation is not commutative but associative.

```
int MPI_Group_intersection(MPI_Group group1,
                           MPI_Group group2,
                           MPI_Group *newgroup)
```

Produces a group at the intersection of two existing groups by including all elements of the first group that are also in the second group, ordered as in the first group.

- `group1`, `group2` groups to intersect,
- `*newgroup` handle of the group to create. This may be equal to the empty group `MPI_GROUP_EMPTY`.

The operation is not commutative but associative.

```
int MPI_Group_difference(MPI_Group group1,
                         MPI_Group group2,
                         MPI_Group *newgroup)
```

Generates the new group from the difference of the existing groups by including all elements of the first group that are not in the second group, ordered as in the first group.

- `group1`, `group2` groups to determine the difference from
- `*newgroup` handle of the group to create. This may be equal to the empty group `MPI_GROUP_EMPTY`.

```
int MPI_Group_incl(MPI_Group group,
                   int n,
                   int *ranks,
                   MPI_Group *newgroup)
```

Create a new group from an existing group by including a possibly reordered subset of the processes.

- `group` the existing group
- `n` number of ranks used in the new group
- `ranks` ordered list of members for the new group
- `*newgroup` handle of the group to create.

```
int MPI_Group_excl(MPI_Group group,
                   int n,
                   int *ranks,
                   MPI_Group *newgroup)
```

Create a new group from an existing group by excluding a possibly reordered subset of the processes.

- `group` the existing group
- `n` number of ranks used in the new group
- `ranks` ordered list of members to exclude from the new group
- `*newgroup` handle of the group to create.

```
int MPI_Group_rank(MPI_Group group, int *rank)
```

Find the `rank` (local number) of the current process in `group`.

```
int MPI_Group_size(MPI_Group group, int *size)
```

Determines the number of members of a group, returned in `size`.

```
int MPI_Group_compare(MPI_Group group1,
                      MPI_Group group2,
                      int *result)
```

Find out how different group1 and group2 are. The result is MPI_IDENT if they are the same, MPI_SIMILAR in case they only differ in the order of the processes and MPI_UNEQUAL otherwise.

Unused groups can be released by calling

```
int MPI_Group_free(MPI_Group *group)
```

On successful return group is set to MPI_GROUP_NULL

Communicators

Definition 5.10 (Communicators)**:**
The participants in a communication operation in MPI are usually determined via so called *communicators*. MPI distinguishes two types of communicators

- *intra-communicators* for the collective communication inside a process group
- *inter-communicators* for the point-to-point like communication between two process groups.

If we are following the SPMD programming model and do not want to have task-parallelism in our code, we are usually fine with the predefined default communicator MPI_COMM_WORLD. When people simply speak of a communicator they usually refer to an intra-communicator. Communicators are objects of type MPI_Comm

Communicator Functions

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group,
   MPI_Comm *newcomm)
```

Create a new communicator for a subset of the processes.

- comm base communicator
- group process group the new communicator will be associated with. Must be a subgroup of the group associated to comm.
- *newcomm handle to the newly created communicator.

```
int MPI_Comm_size (MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *
   result)
```

are the communicator equivalents of the equally called group functions. For comm equal to MPI_COMM_WORLD the total number of processes and the global ranks are returned. Otherwise those of the associated group are given.

For the MPI_Comm_compare function the value MPI_IDENT here means that the underlying groups are in fact the same. MPI_CONGRUENT is returned if the groups are equal (including the order of the ranks) but not the same one group. If only the order differs the result is MPI_SIMILAR again and MPI_UNEQUAL otherwise.

5.5.3 Point-to-Point Communication

```
int MPI_Send(void *buf,
             int count,
             MPI_Datatype datatype,
             int dest,
             int tag,
             MPI_Comm comm)
```

Perform a blocking send operation.

- buf address of the sendbuffer
- count number of elements to send
- datatype type of send buffer elements
- dest the rank of the destination process inside comm
- tag a message identifier
- comm the communicator to use for the transmission

```
int MPI_Recv(void *buf,
             int count,
             MPI_Datatype datatype,
             int source,
             int tag,
             MPI_Comm comm,
             MPI_Status *status)
```

Performs a standard-mode blocking receive.

- buf address of the send buffer
- count number of elements to send

- datatype type of send buffer elements
- source the rank of the sending process inside comm
- tag a message identifier
- comm the communicator to use for the transmission
- status a status object containing information about the sender, the message tag, and possible errors. Also the length of the message received can be retrieved from it using the MPI_Get_count function. This can be set to the constant MPI_STATUS_IGNORE to save resources if not needed by the application.

Variants of these functions performing the send and receive in a single call or that are non-blocking, exist, for the details see the standard and the man pages of MPI_Sendrcv(),MPI_Isend(),MPI_Irecv().

For the non-blocking communication operations the function MPI_Test() can be used to check whether a certain message has been transferred.

5.5.4 Single-Collective Communication

All collective communication functions need to be called on all ranks in the group. The specific argument root will determine how the operation is performed by the current instance of the process.

```
int MPI_Barrier(MPI_Comm comm)
```

Actually not performing a real communication this function makes sure that process flow stops until all processes in the group associated to comm have reached this point.

- comm the communicator to use the barrier for

```
int MPI_Bcast(void *buffer,
              int count,
              MPI_Datatype datatype,
              int root,
              MPI_Comm comm)
```

Broadcasts a message from one process to all other processes of the communicator.

- *buffer address of the send/receive buffer
- count number of elements to send
- datatype type of send buffer elements
- root the rank of the sending process

- comm the communicator to be use

```
int MPI_Reduce(void *sendbuf,
               void *recvbuf,
               int count,
               MPI_Datatype datatype,
               MPI_Op op,
               int root,
               MPI_Comm comm)
```

Reduces values on all processes within a group associated to a communicator

- *sendbuf address of the send buffer
- *recvbuf address of the receive buffer (only relevant on root)
- count number of elements to send
- datatype type of buffer elements
- op the arithmetic operation to use in the reduce
- root the rank of the root/receiving process
- comm the communicator to be use

```
int MPI_Scatter(void *sendbuf,
                int sendcount,
                MPI_Datatype sendtype,
                void *recvbuf,
                int recvcount,
                MPI_Datatype recvtype,
                int root,
                MPI_Comm comm)
```

Distributes data from one process among all processes in the communicator

- *sendbuf address of the send buffer
- sendcount number of elements to send
- sendtype type of the send buffer elements
- *recvbuf address of the receive buffer
- recvcount number of elements to receive
- recvtype type of the receive buffer elements
- root the rank of the root/sending process
- comm the communicator to be use

```
int MPI_Gather(void *sendbuf,
               int sendcount,
                  MPI_Datatype sendtype,
                  void *recvbuf,
                  int recvcount,
                  MPI_Datatype recvtype,
                  int root,
                  MPI_Comm comm)
```

Collects data from all processes on a single process.

- `*sendbuf` address of the send buffer
- `sendcount` number of elements to send
- `sendtype` type of the send buffer elements
- `*recvbuf` address of the receive buffer
- `recvcount` number of elements to receive
- `recvtype` type of the receive buffer elements
- `root` the `rank` of the root/receiving process
- `comm` the communicator to be use

5.5.5 Multi-Collective Communication

```
int MPI_Allgather(void *sendbuf,
                  int  sendcount,
                  MPI_Datatype sendtype,
                  void *recvbuf,
                  int recvcount,
                  MPI_Datatype recvtype,
                  MPI_Comm comm)
```

Collects and redistributes data from all processes to all processes.

- `*sendbuf` address of the send buffer
- `sendcount` number of elements to send
- `sendtype` type of the send buffer elements
- `*recvbuf` address of the receive buffer
- `recvcount` number of elements to receive
- `recvtype` type of the receive buffer elements
- `comm` the communicator to be use

```
int MPI_Allreduce(void *sendbuf,
                  void *recvbuf,
                  int count,
```

```
                  MPI_Datatype datatype,
                  MPI_Op op,
                  MPI_Comm comm)
```

Similar to the `MPI_Reduce()` function it combines values from all processes, but in addition it distributes the result back to all processes.

- `*sendbuf` address of the send buffer
- `*recvbuf` address of the receive buffer
- `count` number of elements to send
- `datatype` type of buffer elements
- `op` the arithmetic operation to use in the reduce
- `comm` the communicator to be use

```
int MPI_Alltoall(void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 MPI_Comm comm)
```

The total exchange operation, i.e., every process sends to all other processes.

- `*sendbuf` address of the send buffer
- `sendcount` number of elements to send
- `sendtype` type of the send buffer elements
- `*recvbuf` address of the receive buffer
- `recvcount` number of elements to receive
- `recvtype` type of the receive buffer elements
- `comm` the communicator to be use

5.6 Message Passing using Open MPI

Hello World

The obligatory “hello world!” program does no more than initializing the MPI context, printing the obligatory text from all instances and destroying the context again:

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

```
int main (int argc, char** argv){

  /* start MPI context*/
  MPI_Init(&argc, &argv);

  /*Do something*/
  printf("Hello␣world\n");

  /* Stop MPI context*/
  MPI_Finalize();
  return 0;
}
```

Compilation of Code

In Open MPI[7] a C wrapper compiler called `mpicc` is provided. Its sole purpose is to transparently

- add relevant compiler and linker flags to the user's compiler command line
- and then call the underlying compiler to perform the actual compilation.

Especially, we do not need to care where exactly the necessary MPI libraries are located and which additional flags are required. If we have specified additional parameters (e.g. for code optimization, or debugging), `mpicc` passes them on to the underlying compiler.

Example 5.11:
Thus, to compile the "hello world" code, we simply use:

```
mpicc hello_world.c -o hello_world -O2
```

Running a Parallel Program

The drawback of the MPI framework is that processes need to be started within a special runtime environment. In the case of Open MPI this is invoked using the `mpirun` tool:

```
mpirun [ options ] <program> [ <args> ]
```

The tool takes a couple of options that allow to steer the number of processes spawned, including where they are spawned, control their working environment

[7] http://www.open-mpi.org/

(path, working directory, environment variables, ...) and the redirection of standard input and output and many details more.

The most important options of `mpirun` for beginners are:

`-n <\#>` run this many copies, if unset Open MPI spawns one copy per processor (aliases are `-c`, `--n`, `-np`).

`-H` List of hosts (comma separate) to spawn the processes on (aliases `-host`, `--host`)

`-hostfile` Provide a hostfile to use instead of the list above. (aliases and synonyms `--hostfile`, `-machinefile`, `--machinefile`)

Example 5.12:
To run 1 copy of `hello_world` (from the local directory) each on the two hosts `alpha`, `beta` we may use

```
mpirun -np 2 -H alpha,beta ./hello_world
```

5.7 Data Distribution Schemes in Distributed LU

For a 2d data field (like a matrix) there are basically 3 types of data distribution patterns:

- row/column blocks,
- row/column cyclic,
- checkerboard.

All of them have their advantages and disadvantages in different algorithms. We will treat them all in the case of the LU decomposition in the following.

Before diving into the details of data distribution we recall that after 4 steps of the row-by-row LU decomposition (Algorithm 3.1) for a matrix $A \in \mathbb{R}^{10\times 10}$ we have the following:

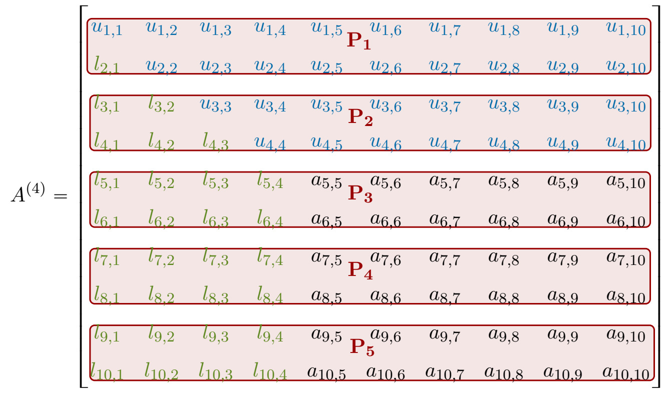
$$A^{(4)} = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & u_{1,4} & u_{1,5} & u_{1,6} & u_{1,7} & u_{1,8} & u_{1,9} & u_{1,10} \\ l_{2,1} & u_{2,2} & u_{2,3} & u_{2,4} & u_{2,5} & u_{2,6} & u_{2,7} & u_{2,8} & u_{2,9} & u_{2,10} \\ l_{3,1} & l_{3,2} & u_{3,3} & u_{3,4} & u_{3,5} & u_{3,6} & u_{3,7} & u_{3,8} & u_{3,9} & u_{3,10} \\ l_{4,1} & l_{4,2} & l_{4,3} & u_{4,4} & u_{4,5} & u_{4,6} & u_{4,7} & u_{4,8} & u_{4,9} & u_{4,10} \\ l_{5,1} & l_{5,2} & l_{5,3} & l_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} & a_{5,8} & a_{5,9} & a_{5,10} \\ l_{6,1} & l_{6,2} & l_{6,3} & l_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} & a_{6,8} & a_{6,9} & a_{6,10} \\ l_{7,1} & l_{7,2} & l_{7,3} & l_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} & a_{7,8} & a_{7,9} & a_{7,10} \\ l_{8,1} & l_{8,2} & l_{8,3} & l_{8,4} & a_{8,5} & a_{8,6} & a_{8,7} & a_{8,8} & a_{8,9} & a_{8,10} \\ l_{9,1} & l_{9,2} & l_{9,3} & l_{9,4} & a_{9,5} & a_{9,6} & a_{9,7} & a_{9,8} & a_{9,9} & a_{9,10} \\ l_{10,1} & l_{10,2} & l_{10,3} & l_{10,4} & a_{10,5} & a_{10,6} & a_{10,7} & a_{10,8} & a_{10,9} & a_{10,10} \end{bmatrix}$$

Furthermore the blue and green parts will no longer be touched and the algorithm proceeds on the smaller lower right part $A(5:10, 5:10)$ only.

5.7.1 Row-/Column Block Distribution

Basic Idea:

Group the rows/columns in blocks of $\lceil \frac{n}{p} \rceil$. Each processor then works on one of those blocks, performing all necessary operations that treat any rows/columns in the scope.



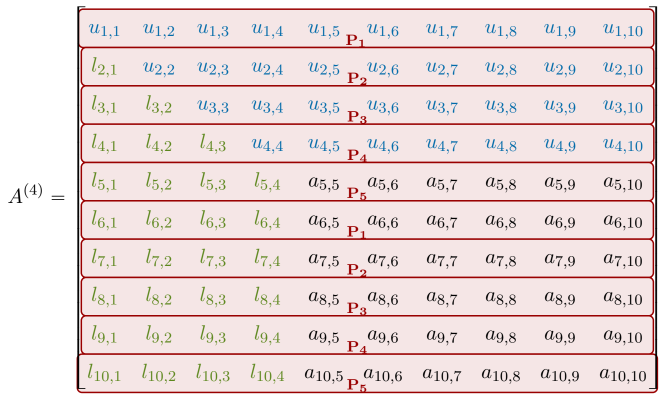
Processors $\mathbf{P_1}$ and $\mathbf{P_2}$ have no more work do do after step 4. This obviously leads to a very bad load balancing among the processors.

As a consequence we should not use the block distribution in cases when not the entire matrix is involved in all computations to make sure that all processors are equally well loaded. That means for parallel matrix-vector or matrix-matrix products it may serve well, but for the LU we need to find a data distribution that has a better distribution of the workload.

5.7.2 Cyclic-row/-column Distribution

Basic Idea:

Instead of distributing blocks of rows/columns assign a single row/column to a process until all got one and then start over until all rows/columns are distributed. For ease of presentation we will restrict to the row-wise distribution in the following.



Obviously now the processors only start to become idle after $n-p$ steps of the outermost loop, i.e. in $A^{(n-p)}$, which is reasonable for $p \ll n$. Still basically every processor is responsible for $\lceil \frac{n}{p} \rceil$ rows.

Pivoting

Since pivoting adds a considerable amount of extra communication effort, we do not neglect it here in contrast to earlier appearances. However, we restrict ourselves to the case of column pivoting. That means as the first step of the outer `for` loop we add the pivot selection and row swapping in Algorithm 3.1.

Differences to the sequential case

1. **Determination of the pivot element.** The column below the diagonal is owned by several processors in the distributed parallel case. That means each processor finds its local pivot element and afterward they are compared among all processors.
2. **Usage of the pivot element.** If we are lucky enough that the pivot row is owned by the same processor that owns the row containing the critical diagonal element, we are fine. We can perform a local row swap as in the sequential case. Otherwise the pivot row is exchanged with the process owning the "diagonal row".
3. **Distribution of the pivot row.** The pivot row is the key ingredient to the computation in the step. It is needed by all processors and thus needs to be broadcast to all active processors.
4. **Computation of the matrix element updates.** The update step can now be performed as in the sequential case. Only, each processor just works through the local rows it owns.

5.7.3 Checkerboard Distribution

Basic Idea:

Distribution of the d-dimensional data array to a d-dimensional processor grid. Note that we can follow the blocked or cyclic variants just as in the case above.

$a_{1,1}$ $a_{1,2}$ $\mathbf{P_{11}}$ $a_{2,1}$ $a_{2,2}$	$a_{1,3}$ $a_{1,4}$ $\mathbf{P_{12}}$ $a_{2,3}$ $a_{2,4}$	$a_{1,5}$ $a_{1,6}$ $\mathbf{P_{13}}$ $a_{2,5}$ $a_{2,6}$	$a_{1,7}$ $a_{1,8}$ $\mathbf{P_{14}}$ $a_{2,7}$ $a_{2,8}$
$a_{3,1}$ $a_{3,2}$ $\mathbf{P_{21}}$ $a_{4,1}$ $a_{4,2}$	$a_{3,3}$ $a_{3,4}$ $\mathbf{P_{22}}$ $a_{4,3}$ $a_{4,4}$	$a_{3,5}$ $a_{3,6}$ $\mathbf{P_{23}}$ $a_{4,5}$ $a_{4,6}$	$a_{3,7}$ $a_{3,8}$ $\mathbf{P_{24}}$ $a_{4,7}$ $a_{4,8}$
$a_{5,1}$ $a_{5,2}$ $\mathbf{P_{31}}$ $a_{6,1}$ $a_{6,2}$	$a_{5,3}$ $a_{5,4}$ $\mathbf{P_{32}}$ $a_{6,3}$ $a_{6,4}$	$a_{5,5}$ $a_{5,6}$ $\mathbf{P_{33}}$ $a_{6,5}$ $a_{6,6}$	$a_{5,7}$ $a_{5,8}$ $\mathbf{P_{34}}$ $a_{6,7}$ $a_{6,8}$
$a_{7,1}$ $a_{7,2}$ $\mathbf{P_{41}}$ $a_{8,1}$ $a_{8,2}$	$a_{7,3}$ $a_{7,4}$ $\mathbf{P_{42}}$ $a_{8,3}$ $a_{8,4}$	$a_{7,5}$ $a_{7,6}$ $\mathbf{P_{43}}$ $a_{8,5}$ $a_{8,6}$	$a_{7,7}$ $a_{7,8}$ $\mathbf{P_{44}}$ $a_{8,7}$ $a_{8,8}$

(a) blocked distribution

$a_{1,1}$ $a_{1,5}$ $\mathbf{P_{11}}$ $a_{5,1}$ $a_{5,5}$	$a_{1,2}$ $a_{1,6}$ $\mathbf{P_{12}}$ $a_{5,2}$ $a_{5,6}$	$a_{1,3}$ $a_{1,7}$ $\mathbf{P_{13}}$ $a_{5,3}$ $a_{5,7}$	$a_{1,4}$ $a_{1,8}$ $\mathbf{P_{14}}$ $a_{5,4}$ $a_{5,8}$
$a_{2,1}$ $a_{2,5}$ $\mathbf{P_{21}}$ $a_{6,1}$ $a_{6,5}$	$a_{2,2}$ $a_{2,6}$ $\mathbf{P_{22}}$ $a_{6,2}$ $a_{6,6}$	$a_{2,3}$ $a_{2,7}$ $\mathbf{P_{23}}$ $a_{6,3}$ $a_{6,7}$	$a_{2,4}$ $a_{2,8}$ $\mathbf{P_{24}}$ $a_{6,4}$ $a_{6,8}$
$a_{3,1}$ $a_{3,5}$ $\mathbf{P_{31}}$ $a_{7,1}$ $a_{7,5}$	$a_{3,2}$ $a_{3,6}$ $\mathbf{P_{32}}$ $a_{7,2}$ $a_{7,6}$	$a_{3,3}$ $a_{3,7}$ $\mathbf{P_{33}}$ $a_{7,3}$ $a_{7,7}$	$a_{3,4}$ $a_{3,8}$ $\mathbf{P_{34}}$ $a_{7,4}$ $a_{7,8}$
$a_{4,1}$ $a_{4,5}$ $\mathbf{P_{41}}$ $a_{8,1}$ $a_{8,5}$	$a_{4,2}$ $a_{4,6}$ $\mathbf{P_{42}}$ $a_{8,2}$ $a_{8,6}$	$a_{4,3}$ $a_{4,7}$ $\mathbf{P_{43}}$ $a_{8,3}$ $a_{8,7}$	$a_{4,4}$ $a_{4,8}$ $\mathbf{P_{44}}$ $a_{8,4}$ $a_{8,8}$

(b) cyclic distribution

Figure 5.20: blocked and cyclic checkerboard matrix distribution for $A \in \mathbb{R}^{8\times 8}$.

Definition 5.13:
Let $n = \prod_{i=1}^{d} n_i$ be the total problem size and n_i the degrees of freedom in the i-th direction. Also p, as before, the number of processors in total. We call $\mathbf{p} = (p_1, \ldots, p_d)$ a *processor distribution* if it holds

$$p \leqslant \prod_{i=1}^{d} p_i.$$

On each processor we assume a *local data distribution* $\mathbf{b} = (b_1, \ldots, b_d)$ with

$$n \leqslant \prod_{i=1}^{d} p_i b_i.$$

Ideally we want to have equality in both cases to achieve optimal load balancing.

5.7.4 An Alternative for the LU Using Distributed BLAS and LAPACK

The PBLAS project (see Section 5.9) aims at providing a parallel distributed version of the BLAS library. In the previous Chapters we have investigated level 3 BLAS based block outer product versions of the LU decomposition (see, e.g. Algorithm 3.3).

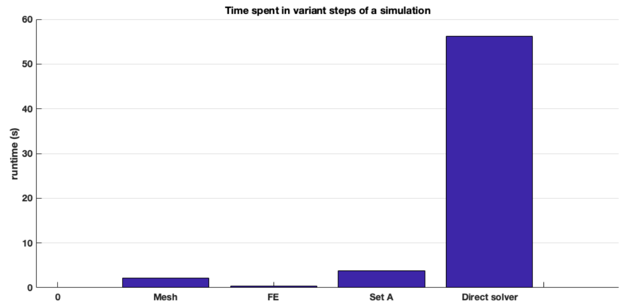
5.8 Data Distribution for other Problems

Basic idea of domain decomposition

Similar to the splitting of the matrix into blocks on which smaller subproblems are solved, in *domain decomposition*[8] methods for boundary value problems the objective domain on which the problem is to be solved is subdivided into smaller parts. Then on each part a smaller independent boundary value problem is solved. The interaction between subdomains is only necessary if their intersection is non empty, i.e., they have a common "boundary", the *interface*. In each iteration step both processes rely on the result of the prior step and exchange the data on the interface to make it fit in a post-processing procedure.

• The interface is sometimes also called *halo*.

[8] http://www.ddm.org



	Robustness	Memory consumption	Scalability
Direct methods	✓	X	X
Iterative methods	X	✓	✓

Table 5.3: Advantages and disadvantages of linear solver families for sparse systems of equations.

- The interface may be a single layer of unknowns, but can also be extended. One then speaks of *overlapping domain decomposition* methods.

Performing a simulation of a phenomenon governed by PDEs in a domain Ω, can be divided into the following steps

- discretize the domain Ω by using some (or a mixed) discretization scheme, FEM, VEM, FDM, DG, ...
- generate the discretized problem, assemble the matrix and the right hand side
- solve the linear system associated and obtain the solution

The most time-consuming step is the solution of the linear system of equations. Consider the linear system of equations (LSE)

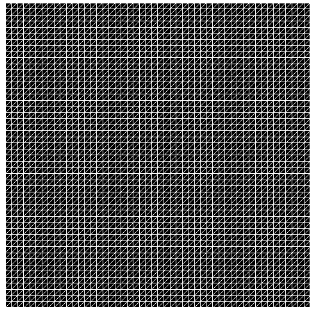
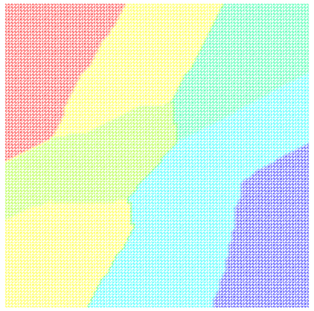
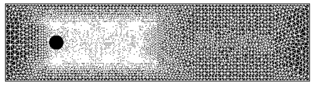
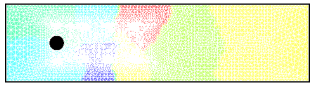
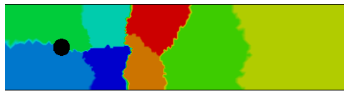
$$Ax = b,$$

where is $A \in \mathbb{R}^{n\times n}$ sparse and $b, x \in \mathbb{R}^n$. We have seen two classes of methods to solve such problem

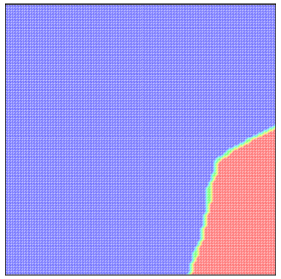
- sparse direct methods: LU, Cholesky, LDLT, QR, etc
- iterative methods: Krylov subspace, Jacobi, Schwarz, Gauss-Seidel, multigrid

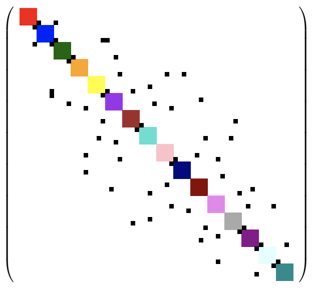
5.8.1 Algebraic Domain Decomposition

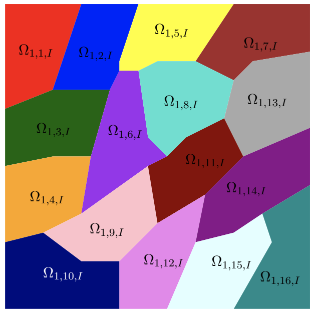
Given an undirected graph with n nodes, we decompose it into N nonoverlapping subdomains $\{\Omega_{j,I}\}_{1\leqslant j\leqslant N}, \bigcup_{j=1}^{N} \Omega_{j,I} = \Omega = \{1,\dots,n\}$. METIS, Par-METIS and PT-SCOTCH are widely known graph partitioning tools.

Then, add one layer of nodes to have overlapping subdomains Ω_j.







Example 5.14 (Illustrating Example)**:**

$$A = \begin{pmatrix} 2 & -1 & 0 & 0 \\ -1 & 3 & -1 & 0 \\ 0 & -1 & 4 & -1 \\ 0 & 0 & -1 & 5 \end{pmatrix}$$

$$\begin{aligned} R_1 &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, & R_2 &= \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \\ R_1 A R_1^\top &= \begin{pmatrix} 2 & -1 & 0 \\ -1 & 3 & -1 \\ 0 & -1 & 4 \end{pmatrix}, & R_2 A R_2^\top &= \begin{pmatrix} 3 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 5 \end{pmatrix}, \\ D_1 &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}. & D_2 &= \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \end{aligned}$$

We have,

- $R_j^\top R_j u = u \iff (u)_i = 0 \; \forall i \in \Omega \backslash \Omega_j$
- $(A R_j^\top D_j R_j u)_i = 0$ if $i \notin \Omega_j$

where $(u)_i$ is the element i of the vector u, [13, 5].

To perform SPMV, we have

$$\begin{aligned} v &= Au, \\ &= A \sum_{j=1}^{N} R_j^\top D_j R_j u, \\ &= \sum_{j=1}^{N} A R_j^\top D_j R_j u, \\ &= \sum_{j=1}^{N} R_j^\top R_j A R_j^\top D_j R_j u, \end{aligned}$$

To perform $\alpha = v^\top u$, we have

$$\begin{aligned} \alpha &= v^\top u, \\ &= v^\top \sum_{j=1}^{N} R_j^\top D_j R_j u, \\ &= \sum_{j=1}^{N} (R_j v)^\top D_j (R_j u). \end{aligned}$$

Suppose that there exists an invertible matrix M such that

- $M \approx A^{-1}$ in some sense and
- solving $Mu = v$ is relatively simple.

Then, solving

$$\begin{aligned} M^{-1}Ax &= M^{-1}b, \text{ rather than} \\ Ax &= b \end{aligned}$$

might be more appropriate for an iterative method, since often

$$\kappa_2(M^{-1}A) \ll \kappa_2(A).$$

To define a preconditioner based on the overlapping Schwarz method, we need the following elements:

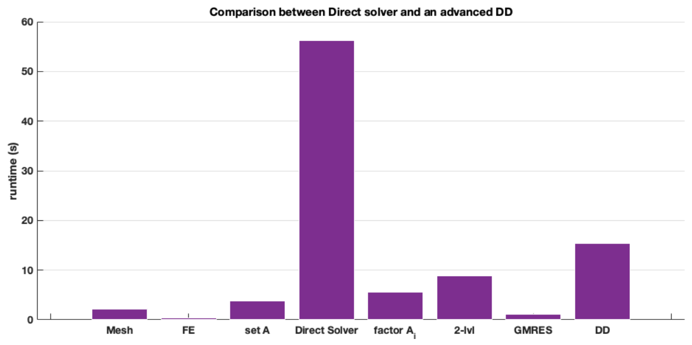
- restriction and plongoation operators R_j and $R_j^\top$, respectively
- partition of unity D_j

They fulfill the relation

$$R_j D_j R_j^\top = I.$$

The additive Schwarz preconditioner is:

$$M^{-1} = \sum_{j=1}^{N} R_j^\top (R_j A R_j^\top)^{-1} R_j$$



5.9 Relevant Software and Libraries

Implementations of the MPI Standard

- Open MPI, Current bugfix release 3.1.4 implements MPI-3.1[9]
- MPICH 3.3 (release of November 21, 2018) supports MPI-3.1[10]
- MVAPICH: The current MVAPICH2 2.3.1 is based on MPICH v3.2.1[11]
- Intel® MPI Library: version 2019 update 4 implements MPI-3.1[12]

Scientific Software

- BLACS (Basic Linear Algebra Communication Subprograms) "is an ongoing investigation whose purpose is to create a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms."[13]
- ScaLAPACK a BLACS-based scalable distributed implementation of LAPACK (current version 2.0.2 of May 1, 2012)[14]
- PBLAS (Parallel Basic Linear Algebra Subprograms) subproject of the above[15]

[9] http://www.openmpi.org
[10] http://www.mpich.org/
[11] http://mvapich.cse.ohio-state.edu/
[12] http://software.intel.com/en-us/intel-mpi-library
[13] http://www.netlib.org/blacs/
[14] http://www.netlib.org/scalapack/
[15] http://www.netlib.org/scalapack/pblas_qref.html

- Boost starting with version 1.35 has a boost.MPI module providing a C++ friendly MPI framework. (current version 1.70.0 April 17, 2019)[16]
- PETSC "*is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations.*"[17]
- SLEPC is the **S**calable **L**ibrary for **E**igenvalue **P**roblem **C**omputations.[18]
- PARPACK an extension to the ARPACK for eigenvalue computations using MPI and BLACS for parallel execution.[19]

[16] http://www.boost.org/
[17] http://www.mcs.anl.gov/petsc/
[18] http://www.grycap.upv.es/slepc/
[19] http://www.caam.rice.edu/software/ARPACK/

References

[1] S. Akhter and J. Roberts, *Multicore Programmierung - Performance erhöhen durch Software-Multithreading*, Intel Press, 2008.

[2] C. Breshears, *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*, O'Reilly Media, Inc., 2009.

[3] *CUDA C Programming Guide*, version 5.0 ed., http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[4] E. W. Dijkstra, *Solution of a problem in concurrent programming control*, Commun. ACM, 8 (1965), pp. 569–, https://doi.org/10.1145/365559.365617.

[5] V. Dolean, P. Jolivet, and F. Nataf, *An introduction to domain decomposition methods*, Algorithms, theory, and parallel implementation, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2015, ch. 1. Schwarz Methods, pp. 1–34, https://doi.org/10.1137/1.9781611974065.ch1.

[6] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, *An extended set of FORTRAN Basic Linear Algebra Subprograms*, ACM Trans. Math. Softw., 14 (1988), pp. 1–17.

[7] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, *A set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Softw., 16 (1990), pp. 1–17.

[8] A. B. Downey, *The Little Book of Semaphores*, Green Tea Press, 2008, http://greenteapress.com/semaphores/.

[9] M. Flynn, *Some computer organizations and their effectiveness*, Computers, IEEE Transactions on, C-21 (1972), pp. 948–960, https://doi.org/10.1109/TC.1972.5009071.

[10] M. A. Francisco D. Igual, *Exploring the capabilities of multicore dsps for dense linear algebra*. Posters, February 2012, "http://www3.uji.es/~figual/files/Posters/TI_Poster%20copy.pdf". Poster presented at SIAM Conference on Parallel Processing for Scientific Computing. Savannah, Georgia.

[11] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, Chapman & Hall, 2010, http://www.rrze.fau.de/dienste/arbeiten-rechnen/hpc/HPC4SE/.

[12] ISO, *ISO/IEC 9899:1999: Programming Languages — C*, International Organization for Standardization, Geneva, Switzerland, Dec. 1999, http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf.

[13] P. Jolivet, *Méthodes de décomposition de domaine. Application au calcul haute performance*, PhD thesis, Université Grenoble Alpes, 2014, http://www.theses.fr/2014GRENM040.

[14] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, *Basic linear algebra subprograms for fortran usage*, ACM Trans. Math. Softw., 5 (1979), pp. 308–323, https://doi.org/10.1145/355841.355847, http://doi.acm.org/10.1145/355841.355847.

[15] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*, O'Reilly and Associates, Inc., 1996.

[16] OpenMP Archtiecture Review Board, *OpenMP Application Program Interface*, version 3.1 ed., July 2011. available from http://www.openmp.org/mp-documents/OpenMP3.1.pdf.

[17] OpenMP Archtiecture Review Board, *OpenMP Summary Card*, version 3.1 ed., July 2011, http://www.openmp.org/mp-documents/OpenMP3.1-CCard.pdf.

[18] T. Rauber and G. Rünger, *Parallel Programming for Multicore and Cluster Systems*, Springer, 1st edition ed., 2010. ISBN 978-3-642-04817-3.

[19] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, 1st ed., 2010.