EBERHARD KARLS UNIVERSITY OF TÜBINGEN
INSTITUTE FOR ASTRONOMY AND ASTROPHYSICS
DEPARTMENT OF COMPUTATIONAL PHYSICS

Notes regarding master's thesis

Notes

Michael Staneker

Contents

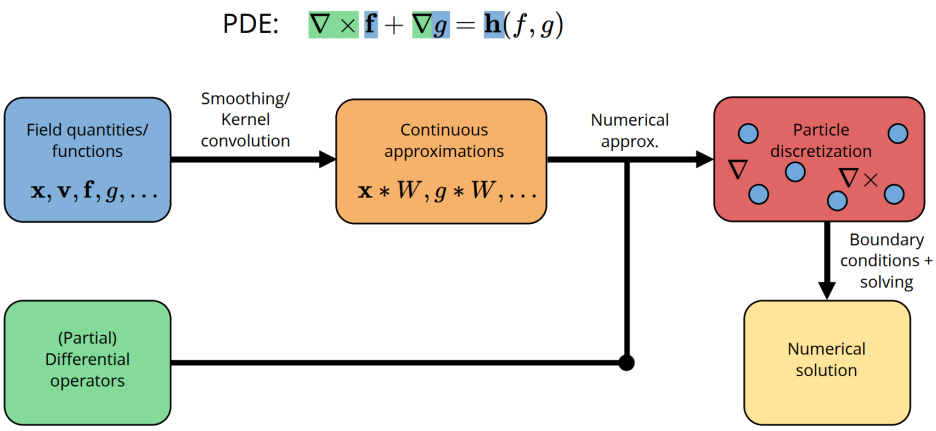
Chapter 1

Introduction

Smoothed particle hydrodynamics (SPH) is applicable to a wide variety of problems. The flexibility of SPH comes at the cost of higher computational costs compared to other methods. High-performance computing (HPC) and especially parallelization can significantly reduce the execution times for simulations.
However, SPH is a purely parallel method except for the interaction of every particle with its neighbors, as well as the need to search for a globally minimum timestep. That has to be considered, especially for multi-GPU SPH codes.

1.1 Smoothed Particle Hydrodynamics (SPH)

The following refers to [Kos+19] and `https://interactivecomputergraphics.github.io/SPH-Tutorial/slides/01_intro_foundations_neighborhood.pdf`.
Smoothed Particle Hydrodynamics (SPH) is a mesh-free method for the discretization of functions and partial differential operators. An SPH particle is a coefficient in the approximation or a sample that carries a field quantity.



1.1.1 Continuous approximation

The Dirac-δ function:

$$\delta(\boldsymbol{r}) = \begin{cases} \inf & \text{if } \boldsymbol{r} = 0 \\ 0 & \text{otherwise} \end{cases}, \int_{\mathbf{R}^d} \delta(\boldsymbol{x}) dv = 1 \,. \tag{1.1}$$

The Dirac-δ identity:

$$A(\boldsymbol{x}) = (A * \delta)(\boldsymbol{x}) = \int_{\mathbf{R}^d} A(\boldsymbol{x}')\delta(\boldsymbol{x} - \boldsymbol{x}')dv \,. \tag{1.2}$$

Find a kernel (function): $\boldsymbol{W} : \mathbf{R}^d \times \mathbf{R}^+ \to \mathbf{R}$ such that

$$\boldsymbol{A}(\boldsymbol{x}) \approx (\boldsymbol{A} * \boldsymbol{W})(\boldsymbol{x}) \tag{1.3}$$

Essential properties of the kernel:

- normalization condition: $\int_{\mathbf{R}^d} W(\boldsymbol{r}', h)dv' = 1$
- Dirac-δ condition: $\lim_{h' \to 0} W(\boldsymbol{r}, h') = \delta(r)$

Desired properties of the kernel:

- positivity condition: $W(\boldsymbol{r}, h) \geq 0$
- symmetry condition: $W(\boldsymbol{r}, h) = W(-\boldsymbol{r}, h)$
- compact support condition: $W(\boldsymbol{r}, h) = 0$ for $||\boldsymbol{r}|| \geq h$

A kernel fulfilling all conditions is e.g.:

$$W(\boldsymbol{r}, h) = \sigma_d \begin{cases} 6(q^3 - q^2) + 1 & \text{for} 0 \leq q \leq \frac{1}{2} \\ 2(1-q)^3 & \text{for} \frac{1}{2} < q \leq 1 \\ 0 & \text{otherwise} \end{cases}, \tag{1.4}$$

with $q = \frac{1}{h}||\boldsymbol{r}||$, the **smoothing length** h and a normalization constant σ_d.

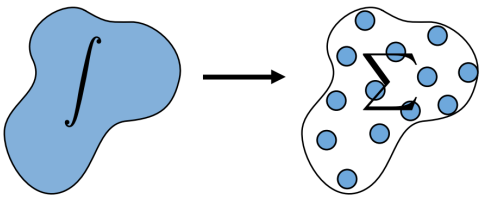
1.1.2 Field discretization

Numerical integration required, therefore discretization required:

$$(A * W)(\boldsymbol{x}_i) = \int \frac{A(\boldsymbol{x}')}{\rho(\boldsymbol{x}')} W(\boldsymbol{x}_i - \boldsymbol{x}', h) \underbrace{\rho(\boldsymbol{x}')dv'}_{dm'} \approx \sum A_j \frac{m_j}{\rho_j} W(\boldsymbol{x}_i - \boldsymbol{x}_j, h) := \langle A(\boldsymbol{x}_i) \rangle \tag{1.5}$$

Density does not have to be carried by particles, since it can be recovered/estimated using

$$\rho(\boldsymbol{x}_i) \approx \sum m_j W_{ij} \tag{1.6}$$



1.1.3 Discrete Differential Operators

$$\nabla A_i \approx \sum_j A_j \frac{m_j}{\rho_j} \nabla W_{ij} \tag{1.7}$$

$$\nabla \boldsymbol{A}_i \approx \sum_j \boldsymbol{A}_j \frac{m_j}{\rho_j} \otimes \nabla W_{ij} \tag{1.8}$$

$$\nabla \cdot \boldsymbol{A}_i \approx \sum_j \boldsymbol{A}_j \frac{m_j}{\rho_j} \cdot \nabla W_{ij} \tag{1.9}$$

$$\nabla \times \boldsymbol{A}_i \approx -\sum_j \boldsymbol{A}_j \frac{m_j}{\rho_j} \times \nabla W_{ij} \tag{1.10}$$

Attention: Direct methods leads to unstable simulations!

Instead, derive gradient from discrete Lagrangian and density estimate in hydrodynamics systems:

$$\nabla A_i = \rho_i \sum_j m_j \left(\frac{A_i}{\rho_i^2} + \frac{A_j}{\rho_j^2} \right) \nabla_i W_{ij} \ , \tag{1.11}$$

as well as the Laplacian:

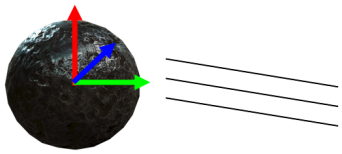
$$\nabla^2 A_i \approx -\sum_j \frac{m_j}{\rho_j} A_i j \frac{2||\nabla_i W_{ij}||}{||\boldsymbol{r}_{ij}||} \tag{1.12}$$

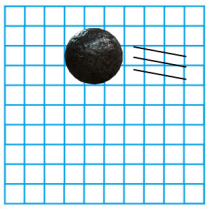
1.1.4 Continuum Mechanical Models

Continuum Mechanics describes continuously and infinitely often dividable masses, modeling physical phenomena as Partial differential equations (PDE).
The continuity equation

$$\frac{D\rho}{Dt} = -\rho(\nabla \boldsymbol{v}) \ , \tag{1.13}$$

describes the evolution of objects's mass density over time.





Lagrangian Coordinates

- Identify (or label) a material of the fluid
- Track material particle as it moves
- Monitor change in its properties
- Field: $A_p^L(t)$

$$\frac{DA^L}{Dt} = \frac{\partial A^L}{\partial t}$$

Eulerian Coordinates

- Identify (or label) fixed location
- Observe fixed location (like sensor)
- Monitor change in properties during flow
- Field: $A^E(t, \mathbf{x})$

$$\frac{DA^E}{Dt} = \frac{\partial A^E}{\partial t} + \mathbf{v} \cdot \nabla_{\mathbf{x}} A^E$$

1.1.5 Numerical solving

Steps:

1. Identify model
2. Split operators for simplification
3. Discretize fields and spatial differential operators using SPH
4. Discretize temporal derivatives (time integration)

1.1.6 Neighborhood Search

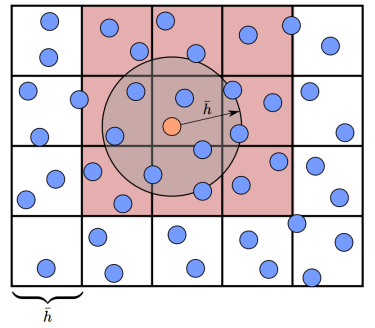
SPH requires lots of sums over particles. If computed for each particle this scales with the particle number squared. Since the kernel (and it derivatives) vanishes outside support radius, the summation can be reduced to a subset of the particles. Consequently a Neighborhood search is required.

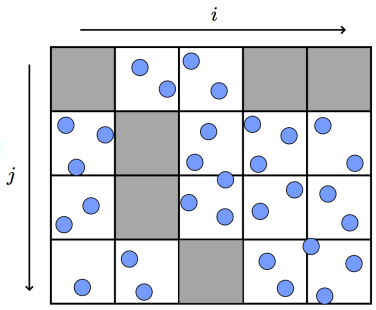
Grid-based Neighborhood search

Place grid with m cells over domain, choose cell size $\bar{h}$ and neighbors must lie within same or neighboring cell. Since many cells are empty this is memory intensive.

Spatial Hashing

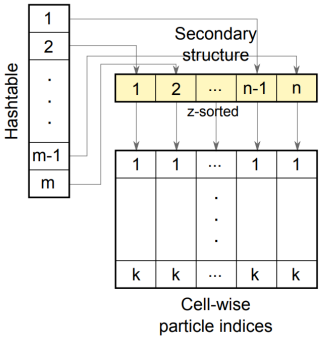
Use a sparse representation by only storing populated cells and use a hash map with a suitable hash function. The memory requirements are lower but the cache-hit rate is not optimal.





Compact Hashing

Only storing handles to secondary data structure in hash table and sorting cells in secondary structure. See `https://github.com/InteractiveComputerGraphics/CompactNSearch` for a (parallel) reference implementation.



1.2 N-Body problem/Self-gravitating systems

The n-body problem considers n point masses m_i, $i = 1, 2, .., n$ in an inertial reference frame moving under the influence of mutual gravitational attraction. Each mass m_i has a corresponding

position vector $\boldsymbol{x}_i$.
Newton's law of gravity gives the gravitational force felt on mass m_i by a single mass m_j:

$$\boldsymbol{F}_{ij} = G\frac{m_i m_j(\boldsymbol{x}_j - \boldsymbol{x}_i)}{||\boldsymbol{x}_j - \boldsymbol{x}_i||^3} \ . \tag{1.14}$$

Combining with Newton's second law and summing over all masses yields the n-body equations of motion:

$$m_i\frac{d^2\boldsymbol{x}_i}{dt^2} = \sum_{j=1}^{n} G\frac{m_i m_j(\boldsymbol{x}_j - \boldsymbol{x}_i)}{||\boldsymbol{x}_j - \boldsymbol{x}_i||^3} = -\frac{\partial U}{\partial \boldsymbol{x}_i} \ . \tag{1.15}$$

There are analytic solutions available for the classical two-body problem, as well as for selected configurations with $n > 2$. In general n-body problems must be solved or simulated numerically.

1.2.1 Simulation

For a **small number of bodies** the problem is solvable by *direct methods*, also called **particle-particle methods**, by numerically integrating the differential equations of motion.
For **large number of bodies** approximate methods have been developed:

- **Tree code methods** such as a Barnes–Hut simulation, are collisionless methods used when close encounters among pairs are not important and distant particle contributions do not need to be computed to high accuracy. The potential of a distant group of particles is computed using a multipole expansion of the potential. This approximation allows for a reduction in complexity to $O(n \log n)$
- **Fast multipole methods** take advantage of the fact that the multipole-expanded forces from distant particles are similar for particles close to each other. This further approximation reduces the complexity to $O(n)$
- **Particle mesh methods** divide up spatial (simulation) space into a three dimensional grid onto which the mass density of the particles is interpolated. Consequently the calculation of the potential corresponds to solving a Poisson equation on the grid. This can be computed in $O(n \log n)$ time using fast Fourier transform techniques
- **P^{3}M and PM-tree methods** are hybrid methods using the particle mesh approximation for distant particles, but using more accurate methods for close particles.
- **Mean field methods** approximate the system of particles with a time-dependent Boltzmann equation representing the mass density that is coupled to a self-consistent Poisson equation representing the potential

However, having strong gravitational fields (e.g. near the event horizon of black holes) **general relativity** needs to be considered. This is the domain of **numerical relativity**.

Chapter 2

High Performance Computing

2.1 HPC

High performance computing (HPC) is a general term referring to the practice of aggregating computing power in a way that delivers much higher performance than obtainable by a typical desktop computer or workstation, in order to solve large problems in science, engineering and business.

2.2 Architecture

In principle there is no general architecture for HPC. All the elements of high performance computers are elements of "normal" computers, like processors, memory, just more of them. High performance computers are really *clusters* of computers, often referring as the individual computers in the cluster as *node*.
The use of Graphic Processing Units (GPUs) is increasing for scientific research. This General-purpose GPU computing (GPGPU) uses a CPU and GPU together in a heterogeneous co-processing computing model. The sequential part of the application runs on the CPU and the computationally-intensive part is accelerated by the GPU.

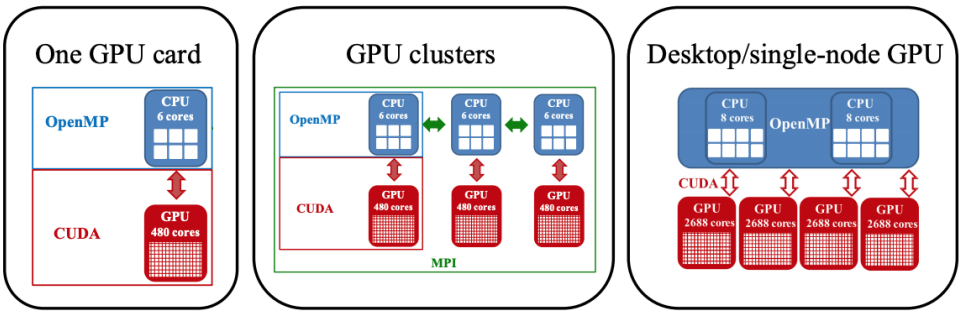


Figure 2.1: Parallel programming depending on architecture

2.2.1 Memory and communication

Main memory in parallel computer is

- **shared memory**: shared between all processing elements in a single address space. If each element of memory can be accessed with equal latency and bandwidth the computer's architecture is a **uniform memory access (UMA)** system.
- **distributed memory**: each processing element has its own local address. The distribution refers to logically distributed, which often implies that it is physically distributed as well. Distributed memory systems have **non-uniform memory access (NUMA)**.

Caches are small and fast memories located close to the processor for storing temporary copies of memory values. Parallel computers require cache coherency to avoid storing the same value in different caches possibly leading to incorrect program execution.
Parallel computers based on interconnected networks require some sort of **routing** to enable passing of messages between nodes that are not directly connected.

2.2.2 Classes of parallel computers

- **Multi-core computing:** Multi-core processors are processors including multiple processing units (cores) on the same chips
- **Symmetric multiprocessing:** A symmetric multiprocessor (SMP) is a computer system with multiple identical processors that share memory and connect via a bus.
- **Distributed computing:** A distributed computer (also known as a distributed memory multiprocessor) is a distributed memory computer system in which the processing elements are connected by a network. Distributed computers are highly scalable. The terms *concurrent computing*, *parallel computing*, and *distributed computing* have a lot of overlap, and no clear distinction exists between them. The same system may be characterized both as *parallel* and *distributed*; the processors in a typical distributed system run concurrently in parallel.
- **Cluster computing:** A cluster is a group of loosely coupled computers that work together closely, so that in some respects they can be regarded as a single computer. Clusters are composed of multiple standalone machines connected by a network. While machines in a cluster do not have to be symmetric.
- **Massively parallel computing (MPP):** A massively parallel processor (MPP) is a single computer with many networked processors. MPPs have many of the same characteristics as clusters, but MPPs have specialized interconnect networks (whereas clusters use commodity hardware for networking). In an MPP, each CPU contains its own memory and copy of the operating system and application. Each subsystem communicates with the others via a high-speed interconnect.
- **Grid computing:** Grid computing is the most distributed form of parallel computing. It makes use of computers communicating over the Internet to work on a given problem. Because of the low bandwidth and extremely high latency available on the Internet, distributed computing typically deals only with embarrassingly parallel problems.

There are some other classes/concepts or rather specialized parallel devices:

- **Field-programmable gate array (FPGA):** An FPGA is essentially a computer chip that can be rewired for a given task using hardware description languages like VHDL.
- **General-purpose computing on GPUs (GPGPU):** General-purpose computing on graphics processing units (GPGPU) is a fairly recent trend in computer engineering research. GPUs are co-processors that have been heavily optimized for computer graphics processing. Computer graphics processing is a field dominated by data parallel operations—particularly linear algebra matrix operations.
- **Application-specific Integrated circuits (ASIC):** Because an ASIC is (by definition) specific to a given application, it can be fully optimized for parallelism.
- **Vector processors:** A vector processor is a CPU or computer system that can execute the same instruction on large sets of data. Vector processors have high-level operations that work on linear arrays of numbers or vectors.

2.3 Programming concepts

2.3.1 Parallel computing

- https://en.wikipedia.org/wiki/Parallel_computing

Parallel programming/computing is a type of computation where many calculations or executions of processes are carried out simultaneously. Large problems can usually be divided into smaller ones, which can be solved at the same time.
Several different forms of parallel computing can be distinguished:

- **Bit-level parallelism:** Bit-level is parallelism in terms of increasing the processor word size, since the word size reduces the number of instructions the processor must execute in order to perform an operation on variables whose sizes are greater than the length of the word.
- **Instruction-level parallelism:** Computer programs are essentially streams of instructions executed by a processor. Having multi-stage instructions pipelines enables to have multiple instructions executed simultaneously.
- **Task parallelism:** Task parallelism involves the decomposition of a task into sub-tasks and then allocating each sub-task to a processor for execution, executing these sub-tasks concurrently and often cooperatively.
- **Superword level parallelism:** Superword level parallelism is a vectorization technique based on loop unrolling and basic block vectorization. It is distinct from loop vectorization algorithms in that it can exploit parallelism of inline code, such as manipulating coordinates, color channels or in loops unrolled by hand.

2.3.2 Concurrent computing

- https://en.wikipedia.org/wiki/Concurrent_computing

Concurrent computing is closely related to parallel computing, often conflated, though distinct. There is parallelism without concurrency (bit-level parallelism) and vice-versa (multitasking by time-sharing on a single-core CPU).

2.3.3 (Multithreading

- https://en.wikipedia.org/wiki/Multithreading_(computer_architecture)

Multithreading in terms of computer architectures describes the ability of the CPU (or a single core in a multi-core processor) to provide multiple threads of execution concurrently, supported by the operating system. A **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by scheduler.
Taking multithreading to the extreme leads to the **Agent-based model** (https://en.wikipedia.org/wiki/Agent-based_model).

2.3.4 Multiprocessing

- https://en.wikipedia.org/wiki/Multiprocessing

Multiprocessing is the use of multiple (two or more) CPUs within a single computer system, referring to the ability of a system to support more than one processor or the ability to allocate tasks between them.

2.3.5 Asynchronous programming

- https://en.wikipedia.org/wiki/Asynchrony_(computer_programming)

A **synchronous** program must complete each step before moving to the next. In contrast, a **asynchronous** starts a step, moving on to other steps (that don't require the result of the first step) and checks on the result of the first step when its result is needed.
Task based programming is kind of a asynchronous programming concept.

2.3.6 Task based programming

See 2.3.1 *task parallelism*.
The task based programming approach refers to the strategy in software engineering where dynamically tasks are created to be accomplished, by being picked up by a task manager assigning the tasks to threads. A finished task triggers the next task and so forth.
The implementation usually make use of a **thread-pool** (https://en.wikipedia.org/wiki/Thread_pool) and some **message-passing interface (MPI)** (https://en.wikipedia.org/wiki/Message_Passing_Interface) to communicate data and task contracts.

2.3.7 Object-oriented programming

Object-oriented programming (OOP) is a programming paradigm based on the concept of **objects**, which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).
Usually OOP languages use **classes** to define the data format and available procedures, allowing to create objects by making an instance of the class.

Distributed objects

- https://en.wikipedia.org/wiki/Distributed_object

In distributed computing, **distributed objects** are objects (in terms of object-oriented programming) that are distributed across different address spaces (event for multiple computers connected via a network), working together by sharing data and invoking methods. The main method for distributed object communication is remote method invocation by message-passing.

2.3.8 General-Purpose Computing on Graphics Processing Units (GPGPU)

- `https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units`

General-purpose computing on graphics processing units (GPGPU, rarely GPGP) is the use of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU).

2.4 APIs/frameworks

Application programming interfaces (APIs) supporting parallelism in host languages.

2.4.1 Message Passing Interface (MPI)

MPI is a message passing programming model.

- `https://de.wikipedia.org/wiki/Message_Passing_Interface`
- `https://www.mpi-forum.org/`

Message Passing Interface (MPI) is a standardized and portable message-passing standard designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran.

- Message-passing library specification for systems of shared memory (parallel computers and clusters)
- Several Processes are communicated by calling routines to send and receive messages
- Typically combined with OpenMP

MPI best solution to combine resources of multiple machines connected via network.
Current standard: MPI-3.1 Standard.

Implementations/Libraries

Implementation(s) exist (mainly) for **C** and **Fortran**.

- **MPICH** initial implementation of the MPI 1.x standard
- **Open MPI** `https://en.wikipedia.org/wiki/Open_MPI`

Language bindings

Bindings are libraries that extend MPI support to other languages by wrapping an existing MPI implementation.

- **C++** can use the C bindings by either using the Boost.MPI library or rolling a C++ wrapper
- **Python** MPI implementations include *pyMPI*, *mpi4py*, *pypar*, *MYMPI* and the MPI submodule in *Scipy*

2.4.2 Parallel Virtual Machine (PVM)

PVM is a message passing programming model.

- `https://en.wikipedia.org/wiki/Parallel_Virtual_Machine`

Parallel Virtual Machine (PVM) is a software tool for parallel networking of computers. It is designed to allow a network of heterogeneous Unix and/or Windows machines to be used as a single distributed parallel processor.

2.4.3 Open Multi-Processing (OpenMP)

OpenMP is a threaded shared memory programming model

- `https://en.wikipedia.org/wiki/OpenMP`
- `https://www.openmp.org/`

The application programming interface (API) OpenMP (Open Multi-Processing) supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran, on many platforms, instruction-set architectures and operating systems, including Solaris, AIX, HP-UX, Linux, macOS, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

- Parallel programming for systems of shared memory
- Portable and flexible programming interface
- Implementation relatively easy
- limited by number of cores/CPUs

OpenMP best solution to optimize performance of multiple cores.

2.4.4 POSIX Threads (PThreads)

OpenMP is a threaded shared memory programming model

- `https://en.wikipedia.org/wiki/POSIX_Threads`

POSIX Threads, usually referred to as pthreads, is an execution model that exists independently from a language, as well as a parallel execution model. It allows a program to control multiple different flows of work that overlap in time. Each flow of work is referred to as a thread, and creation and control over these flows is achieved by making calls to the POSIX Threads API.

2.4.5 Open Accelerators (OpenACC)

- https://en.wikipedia.org/wiki/OpenACC
- https://www.openacc.org/

OpenACC (for open accelerators) is a programming standard for parallel computing developed by Cray, CAPS, Nvidia and PGI. The standard is designed to simplify parallel programming of heterogeneous CPU/GPU systems.
As in OpenMP, the programmer can annotate C, C++ and Fortran source code to identify the areas that should be accelerated using compiler directives and additional functions. Like OpenMP 4.0 and newer, OpenACC can target both the CPU and GPU architectures and launch computational code on them.

- Programming standard for parallel computing
- simplify parallel programming of heterogeneous CPU/GPU systems

2.4.6 Open Computing Language (OpenCL)

- https://en.wikipedia.org/wiki/OpenCL
- https://www.khronos.org/opencl/

OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators.
OpenCL provides a standard interface for parallel computing using task- and data-based parallelism.

2.4.7 CUDA

- https://en.wikipedia.org/wiki/CUDA
- https://developer.nvidia.com/cuda-zone
- Multiple GPU: https://www.nvidia.com/docs/IO/116711/sc11-multi-gpu.pdf

CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by Nvidia. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing – an approach termed GPGPU (General-Purpose computing on Graphics Processing Units). The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

2.4.8 Open Graphics Library (OpenGL)

- `https://en.wikipedia.org/wiki/OpenGL`

OpenGL (Open Graphics Library[3]) is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering.

2.5 Programming languages

Regarding HPC and parallel programming.

2.5.1 CUDA

- `https://docs.nvidia.com/cuda/`

CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by Nvidia. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing – an approach termed GPGPU (General-Purpose computing on Graphics Processing Units). The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

2.5.2 C

MPI implementation

...

2.5.3 C++

OpenMP

See 2.4.3.

MPI

See 2.4.1.

pThreads

See 2.4.4.

C++ Standard (Template) Library

- Standard Library: https://en.wikipedia.org/wiki/C%2B%2B_Standard_Library#Thread_support_library
- Standard Template Library: https://en.wikipedia.org/wiki/Standard_Template_Library

CUDA C++

- C++ language support: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#c-cplusplus-language-support

Thrust

Thrust is a C++ template library for CUDA based on the STL, allowing to implement high performance parallel applications with minimal programming effort through a high-level interface that is fully interoperable with CUDA

- https://docs.nvidia.com/cuda/thrust/index.html
- https://developer.nvidia.com/thrust

Boost (C++ libraries)

Boost (https://en.wikipedia.org/wiki/Boost_(C%2B%2B_libraries)) is a set of libraries for the C++ programming language that provides support for tasks and structures such as linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions, and unit testing.
Relevant HPC Boost libraries (https://www.boost.org/doc/libs/1_64_0/libs/libraries.htm):

- Boost.MPI
- Boost.Thread
- Boost.Fiber
- Boost.Compute
- ...

Dlib

- https://en.wikipedia.org/wiki/Dlib
- http://dlib.net/

Dlib is a general purpose cross-platform software library written in the programming language C++. Its design is heavily influenced by ideas from design by contract and component-based software engineering. Thus it is, first and foremost, a set of independent software components.

Parallel Patterns Library

- https://en.wikipedia.org/wiki/Parallel_Patterns_Library

The Parallel Patterns Library is a Microsoft library designed for use by native C++ developers that provides features for multicore programming.

Stapl

- https://en.wikipedia.org/wiki/Stapl

STAPL (Standard Template Adaptive Parallel Library) is a library for C++, similar and compatible to STL. It provides parallelism support for writing applications for systems with shared or distributed memory.

Threading Building Blocks (TBB)

- https://en.wikipedia.org/wiki/Threading_Building_Blocks

Threading Building Blocks (TBB) is a C++ template library developed by Intel for parallel programming on multi-core processors. Using TBB, a computation is broken down into tasks that can run in parallel. The library manages and schedules threads to execute these tasks.

Integrated Performance Primitives (IPP)

- https://en.wikipedia.org/wiki/Integrated_Performance_Primitives

Intel Integrated Performance Primitives (Intel IPP) is a multi-threaded software library of functions for multimedia and data processing applications, produced by Intel. The library supports Intel and compatible processors and is available for Linux, macOS, Windows and Android operating systems.

C++ AMP

- https://en.wikipedia.org/wiki/C%2B%2B_AMP

C++ Accelerated Massive Parallelism (C++ AMP) is a native programming model that contains elements that span the C++ programming language and its runtime library. It provides an easy way to write programs that compile and execute on data-parallel hardware, such as graphics cards (GPUs).

SYCL

- https://en.wikipedia.org/wiki/SYCL

SYCL is a higher-level programming model for OpenCL as a single-source domain specific embedded language (DSEL) based on pure C++11 for SYCL 1.2.1 to improve programming productivity.

Vulkan (API)

- https://en.wikipedia.org/wiki/Vulkan_(API)

Vulkan is a low-overhead, cross-platform 3D graphics and computing API. Vulkan targets high-performance realtime 3D graphics applications such as video games and interactive media across all platforms. Compared to OpenGL, Direct3D 11 and Metal, Vulkan is intended to offer higher performance and more balanced CPU/GPU usage.

RaftLib

- https://en.wikipedia.org/wiki/RaftLib

RaftLib is a portable parallel processing system that aims to provide extreme performance while increasing programmer productivity. It enables a programmer to assemble a massively parallel program (both local and distributed) using simple iostream-like operators. RaftLib handles threading, memory allocation, memory placement, and auto-parallelization of compute kernels. It enables applications to be constructed from chains of compute kernels forming a task and pipeline parallel compute graph. Programs are authored in C++ (although other language bindings are planned).

Charm++

Charm++ is a parallel object-oriented programming paradigm based on C++ (https://en.wikipedia.org/wiki/Charm%2B%2B, https://charm.readthedocs.io/en/latest/quickstart.html).

2.5.4 Python

- https://developer.nvidia.com/how-to-cuda-python
- https://developer.nvidia.com/pycuda

2.5.5 M4

M4 or m4 is a general-purpose macro processor included in most Unix-lie operating systems.

- https://en.wikipedia.org/wiki/M4_(computer_language)

2.5.6 IDL

IDL (Interactive Data Language) is a vectorized, numerical and interactive programming language, commonly used for interactive processing of large amounts of data.

- https://en.wikipedia.org/wiki/IDL_(programming_language)

2.5.7 Parallel HDF5 (C++)

TODO...

2.5.8 AVX

Advanced Vector Extensions (AVX, also known as Sandy Bridge New Extensions) are extensions to the x86 instruction set architecture for microprocessors from Intel and AMD.
`https://en.wikipedia.org/wiki/Advanced_Vector_Extensions`

2.5.9 QuickSched

`https://gitlab.cosma.dur.ac.uk/swift/quicksched`

Chapter 3

Open Source SPH codes

3.1 SPlisHSPlasH

Description

SPlisHSPlasH is an open-source library for the physically-based simulation of fluids. The simulation in this library is based on the Smoothed Particle Hydrodynamics (SPH) method which is a popular meshless Lagrangian approach to simulate complex fluid effects. The SPH formalism allows an efficient computation of a certain quantity of a fluid particle by considering only a finite set of neighboring particles. One of the most important research topics in the field of SPH methods is the simulation of incompressible fluids. SPlisHSPlasH implements current state-of-the-art pressure solvers (WCSPH, PCISPH, PBF, IISPH, DFSPH, PF) to simulate incompressibility. Moreover, the library provides different methods to simulate viscosity, surface tension and vorticity.

Integrated Libraries:

- **PositionBasedDynamics** to simulate dynamic rigid bodies
- **Discregrid** to detect collisions between rigid bodies
- **CompactNSearch** to perform the neighborhood search
- **cuNSearch** to perform the neighborhood search on the GPU
- **GenericParameters** to handle generic parameters

Links

- `https://github.com/InteractiveComputerGraphics/SPlisHSPlasH`
- `http://www.interactive-graphics.de/index.php/downloads/106-splishsplash-library`
- `https://splishsplash.readthedocs.io/en/latest/index.html`
- License: MIT License

  `https://github.com/InteractiveComputerGraphics/SPlisHSPlasH/blob/master/LICENSE`

- `https://interactivecomputergraphics.github.io/SPH-Tutorial/slides/07_SPlisHSPlasH.pdf`

Implementation

C++ (90.0%) | Python (4.0%) | Objective-C (3.2%) | CMake (2.4%)

SPlisHSPlasH implements:

- an open-source SPH fluid simulation (2D & 3D)
- neighborhood search on CPU or GPU
- supports vectorization using AVX
- Python binding
- several implicit pressure solvers (WCSPH, PCISPH, PBF, IISPH, DFSPH, PF)

3.2 DualSPHysics

Description

DualSPHysics is based on the Smoothed Particle Hydrodynamics model named SPHysics (`https://wiki.manchester.ac.uk/sphysics/index.php/Main_Page`). The code is developed to study free-surface flow phenomena where Eulerian methods can be difficult to apply, such as waves or impact of damn-breaks on off-shore structures.

Links

- `https://github.com/DualSPHysics/DualSPHysics`
- `https://dual.sphysics.org/`
- License: GNU Lesser General Public License v2.1

  https://github.com/DualSPHysics/DualSPHysics/blob/master/LICENSE

Implementation

C++ (85.8%) | CUDA (12.5%) | C (1.2%)

DualSPHysics is a C++/CUDA/OpenMP based SPH solver.

3.3 Bonsai

Description

Bonsai is a GPU gravitational Barnes-Hut-tree code, including support for SPH in the *BonsaiSPH* branch.

Links

- `https://github.com/treecode/Bonsai`
- `https://arxiv.org/abs/1909.07439`
- License: apache-2.0 License

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

Implementation

C++ (55.6%) | C (23.0%) | CUDA (20.3%)

Bonsai-SPH is optimized for GPU accelerators using CUDA.

3.4 ChaNGa

Description

ChaNGa [Jet+10] (Charm N-body GrAvity solver) is a code to perform collisionless N-body simulations. It can perform cosmological simulations with periodic boundary conditions in comoving coordinates or simulations of isolated stellar systems. It also can include hydrodynamics using the Smooth Particle Hydrodynamics (SPH) technique. It uses a Barnes-Hut tree to calculate gravity, with hexadecapole expansion of nodes and Ewald summation for periodic forces.

Links

- `http://faculty.washington.edu/trq/hpcc/tools/changa.html`
- `https://github.com/N-BodyShop/changa`
- License: GPL-2.0 License

  `https://github.com/N-BodyShop/changa/blob/master/LICENSE.md`

Implementation

C++ (69.2%) | C (22.3%) | CUDA (4.7%) | IDL (1.9%) | M4 (0.7%)

ChaNGa uses Charm++ (see section 2.5.3).

The iteration steps are the following:

- Domain decomposition: Particles decomposition (into Charm++ objects)

  on the CPU, since not enough computation per particle to justify transfer to GPU

- Tree construction: Barnes-Hut tree construction

  on the CPU

- Tree traversal: traversal of the distributed Barnes-Hut tree

  on the CPU, since using the GPU would require repeated memory transfers between CPU and GPU

- (Gravitational) force computation

  on the GPU, since the gravitational force calculation routines exhibit a high intensity of floating point operations. Moreover a *work agglomeration module* is enhanced, collating the interaction lists of multiple buckets into a single work request, which is then transferred to the GPU execution.

3.5 GPUSPH

Description

GPUSPH is an implementation of weakly compressible Smoothed Particle Hydrodynamics (WCSPH), running fully on GPUs, using NVIDIA CUDA. Guided and funded by government agencies and industrial and academic partners.

Links

- `https://github.com/GPUSPH/gpusph`
- `http://www.gpusph.org/`
- License: -

Implementation

C++ (67.8%) | CUDA (25.0%) | Python (2.4%) | C (2.2%) | Makefile (1.9%)

Entirely on GPU using CUDA.
Parallelism features (see `http://www.gpusph.org/features/`):

- MultiGPU
- Multinode
- Arbitrary domain split
- Load balancing
- Asynchronous computations

3.6 GIZMO

Description

GIZMO is a flexible, massively-parallel, multi-physics simulation code, including SPH and self-gravity. The code is massively parallel. The code is descended from GADGET (`https://www.h-its.org/2014/10/29/gadget-code/`, `https://wwwmpa.mpa-garching.mpg.de/gadget/`). Physic modules included:

- Hydrodynamics
- Magnetic fields
- Cosmological integrations
- Radiative heating/cooling
- Non-standard cosmology
- ...

Links

- `https://bitbucket.org/phopkins/gizmo-public/src/master/`
- `http://www.tapir.caltech.edu/~phopkins/Site/GIZMO.html`
- `http://www.tapir.caltech.edu/~phopkins/Site/GIZMO_files/gizmo.pdf`
- License: GNU General Public License

  `https://www.gnu.org/licenses/gpl-3.0.html`

Implementation

C | MPI

Massively parallel, the portability of the code has been confirmed on a large number of systems ranging from a laptop to >1 million threads/cores on national super-computers.

3.7 SWIFT

Description

SWIFT is a hydrodynamics and gravity code for astrophysics and cosmology, aiming to simulate a whole universe.
Physic modules included:

- Gravity, using Fast Multipole Method (FFM). Combined with long-range forces provided by a mesh.
- LCDM cosmology model
- SPH (different modes, approaches, ...)
- ...

Links

- `http://swift.dur.ac.uk/`
- `https://gitlab.cosma.dur.ac.uk/swift/swiftsim`
- `http://swift.dur.ac.uk/docs/index.html`

Implementation

C | Python | M4 | Objective-C

SWIFT uses a hybrid MPI + threads parallelisation scheme with a modified version of the publicly available lightweight tasking library QuickSched (`https://gitlab.cosma.dur.ac.uk/swift/quicksched`) as its backbone. Communications between compute nodes are scheduled by the library itself and use asynchronous calls to MPI to maximize the overlap between communication and computation. The domain decomposition itself is performed by splitting the graph of all the compute tasks, using the METIS library (`http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`), to minimize the number of required MPI communications. The core calculations in SWIFT use hand-written SIMD intrinsics to process multiple particles in parallel and achieve maximal performance.

3.8 SPHERAL

Description

Spheral++ provides a steerable parallel environment for performing coupled hydrodynamical & gravitational numerical simulations. Hydrodynamics and gravity are modelled using particle based methods (SPH and N-Body).

Links

- `https://github.com/LLNL/spheral`
- `https://spheral.readthedocs.io/en/latest/`
- License: `https://github.com/LLNL/spheral/blob/develop/License.txt`

Implementation

Gnuplot (35.9%) | C++ (34%) | Python (28%)

The basic build includes a Python interface and MPI support. If the interface is not needed a C++ only build is also available. OpenMP/MPI is supported.

3.9 Miluphcuda

Description

Miluphcuda is the CUDA port of the original miluph code, therefore a SPH hydro and solid code, including self-gravity (via Barnes-Hut tree) and porosity models.

Links

`https://github.com/christophmschaefer/miluphcuda`

Implementation

CUDA (82.5%) | C (14.6%) | C++ (1.6%)

Miluphcuda is GPU-accelerated using CUDA for single NVIDIA GPUs with compute capability 5.0 and higher.

Chapter 4

Self gravity and Barnes-Hut tree

If gravity between SPH particles cannot be neglected, self-gravity has to be included. Self-gravity corresponds to the gravitational N-body problem (see 1.2) and the gravitational force

$$\boldsymbol{F}_G = -\nabla\phi_i = -G\sum_{j=1}^{n}\frac{m_j}{r_{ij}^2}\frac{r_{ij}}{r_{ij}} \tag{4.1}$$

needs to be included in the SPH formalism. The sum needs to be taken over all particles, which is extremely expensive. Consequently other approaches need to utilized in order to reduce the computational time:

- **Tree code methods** such as a Barnes–Hut simulation, are collisionless methods used when close encounters among pairs are not important and distant particle contributions do not need to be computed to high accuracy. The potential of a distant group of particles is computed using a multipole expansion of the potential. This approximation allows for a reduction in complexity to $O(n \log n)$
- **Fast multipole methods** take advantage of the fact that the multipole-expanded forces from distant particles are similar for particles close to each other. This further approximation reduces the complexity to $O(n)$
- **Particle mesh methods** divide up spatial (simulation) space into a three dimensional grid onto which the mass density of the particles is interpolated. Consequently the calculation of the potential corresponds to solving a Poisson equation on the grid. This can be computed in $O(n \log n)$ time using fast Fourier transform techniques
- **P^{3}M and PM-tree methods** are hybrid methods using the particle mesh approximation for distant particles, but using more accurate methods for close particles.
- **Mean field methods** approximate the system of particles with a time-dependent Boltzmann equation representing the mass density that is coupled to a self-consistent Poisson equation representing the potential

However, having strong gravitational fields (e.g. near the event horizon of black holes) **general relativity** needs to be considered. This is the domain of **numerical relativity**.

4.1 Tree Algorithms for Long-Range Potentials

Referring to *Numerical Simulation in Molecular Dynamics - Numerics, Algorithms, Parallelization, Application* by Michael Griebel, Stephan Knapek, Gerhard Zumbusch.

4.1.1 Series Expansion of the Potential

The integral form of a potential written for general kernels G with the particle density ρ in the domain Ω is given as

$$\phi(x) = \int_{\Omega} G(\boldsymbol{x}, \boldsymbol{y}) \rho(\boldsymbol{y}) d\boldsymbol{y} \ . \tag{4.2}$$

Using a Taylor Expansion the approximate formula for the fast computation of the energy and force

$$\phi(\boldsymbol{x}) = \sum_{||\boldsymbol{j}||_1 \leq p} \frac{1}{\boldsymbol{j}!} M_{\boldsymbol{j}}(\Omega, \boldsymbol{y}_0) G_{\boldsymbol{0},j}(\boldsymbol{x}, \boldsymbol{y}_0) \tag{4.3}$$

can be obtained, with the moments

$$M_{\boldsymbol{j}}(\Omega, \boldsymbol{y}_0) := \int_{\Omega} \rho(\boldsymbol{y}) (\boldsymbol{y} - \boldsymbol{y}_0)^{\boldsymbol{j}} d\boldsymbol{y} \ , \tag{4.4}$$

and the multi-indices $\boldsymbol{j} = (j_1, j_2, j_3)$.

4.1.2 Decomposition of the far field

How can a suitable decomposition of the far field can be constructed as efficiently as possibly? Especially regarding that the decomposition for the far field has to be done for all particles. Furthermore are the calculations of the moments rather expensive. Consequently, they should not only be usable for the evaluation of the potential of a single point, but for be reusable in the computation for other particle positions. Therefore, it is crucial that the moments of larger subdomains can be computed from those of the smaller subdomains subdividing them.
This can be achieved by a recursive decomposition of the entire domain Ω into a sequence of smaller and smaller cubic or cuboid subdomains.
Altogether, such an approach allows an efficient computation of the potential and the forces and the resulting structures can be described with geometric trees.

Tree structures

Defining **trees** in an abstract way:

- A **graph** is given as a set of **vertices/nodes** and a set of **edges**
- An **edge** is a connection between to vertices
- A **path** is a sequence of different vertices in which any two successive vertices are connected by an **edge**

- A **tree** is a graph in which there is exactly one **path** connecting any two **vertices**
- Whereas one of the **vertices** of the **tree** is the **root**
- Consequently, there is exactly one **path** from the **root** to each other **vertex** in the **tree**
- A node beneath another node is the **son node** of that **node**
- A node above another node is the **father node**
- The first **generation** of **nodes**, connected to the root belongs to **level one**
- accordingly, **nodes**, connected to the **root** by **paths** of **length** l, belongs to **level** l
- The **depth** of a **tree** is the maximum of the **levels** of all **nodes**, starting at the **root**
- **nodes** with **sons** are **inner nodes**. If not, they are called **leaf nodes**
- A **subtree** consists of one **node** of the original **tree** that has been designated as the **root** of the **subtree**, of all of its descendants, and of the **edges** of the original **tree** between them

Recursive decomposition

Steps to make a recursive partition of the entire domain into cells of different sizes:

- Assign entire cubic or cuboid domain Ω and all the particles contained to the root of the tree
- Partition Ω into disjoint subdomains, assigning them to the son nodes of the root
- Again divide every subdomain into smaller subdomains (recursively)
- Terminate the recursion if either one particle or no particle is left the corresponding subdomain

A simple approach of splitting into subdomains is to split a subdomain in each coordinate direction into two equal parts, resulting into eight equally sized smaller cubes for a cubic domain. Therefore, in the three-dimensional case, the inner nodes of the tree have eight sons, resulting into a type of tree called **octree**.
There are other ways to decompose the domain into cells, e.g. binary trees. Especially if decomposing into cells with different volumes is done, e.g. by dividing in dependence of the density, so that the number of particles in the resulting subcell is approximately equal.

4.1.3 Recursive Computation of the far field

Given:

- Every inner node of the tree represents a subdomain of Ω
- To each subdomain a designated expansion point $\boldsymbol{y}_0$ can be designated (center of cell or center of mass)
- For each vertex the size of the corresponding subdomain is known
- The size *diam* of the circumscribed sphere can be determined

Calculation:

- start with a particle i at the position $\boldsymbol{x}_i$ in the root of the tree
- descend recursively to the son nodes: compute $diam/||\boldsymbol{x}_i\boldsymbol{y}_0^\nu||$
- if this value is smaller than θ terminate recursion, if not, descend to the son nodes, and compute the ratios again
- is the recursive procedure finished, a decomposition for position $\boldsymbol{x}_i$ has been found

Note: This procedure has to be done for each particle separately, resulting in generally different decompositions for different particles, **but** all of the decompositions for *all particles* are contained within *one tree*!

4.1.4 Recursive Computation of the Moments

It is not efficient to directly compute the moments by numerical integration over the density or summing over the particles. Instead, the hierarchical tree structure can be employed to compute all moments for all subdomains, which are associated to the nodes of the tree.
Since it is possible to compute the moments for a father cell from the moments of its son cells, the computation of all moments can be done by proceeding recursively computing it for each node, starting from all leaves of the tree and ending in the root. The values for the leaves of the tree can be computed via

$$M_{\boldsymbol{j}}(\Omega_\nu^{leaf}, \boldsymbol{y}_0^\nu) = m_i(\boldsymbol{x}_i - \boldsymbol{y}_0^\nu)^{\boldsymbol{j}} \ , \tag{4.5}$$

since exactly one (or none) particle $\boldsymbol{x}_i$ with mass m_i is within a subdomain associated to a leaf.

4.2 Barnes-Hut tree

`https://en.wikipedia.org/wiki/Barnes%E2%80%93Hut_simulation`

Barnes-Hut tree allows for treatment of self-gravity and use the tree to determine the interaction partners without any additional helping grid.
The following refers to *Numerical Simulation in Molecular Dynamics - Numerics, Algorithms,*

Parallelization, Application by Michael Griebel, Stephan Knapek, Gerhard Zumbusch.
It is the simplest form of a tree-like method (using octrees) for the approximative computation of potentials, modeled by the gravitational potential

$$U(r_{ij}) = -G\frac{m_i m_j}{r_{ij}} \tag{4.6}$$

or modifications thereof. This method can be interpreted as a special case of the approximation by the Taylor series expansion, based on the idea that the effect of the gravitation of many particles in a cell far away can be modeled by one interaction with a so-called **pseudo-particle** positioned at the center of mass of the cell and the sum of the masses of the particles within.

4.2.1 The method

The method consists out of

- construction of the tree
- computation of the pseudoparticles
- computation of the forces

Pseudoparticles are cells with multiple particles in it, therefore presented by inner vertices of the tree, while real particles are only stored in the leaves of the tree.
Both the mass and the coordinate of the pseudoparticle can be recursively computed for all vertices of the tree in one traversal starting from the leaves according to

$$m^{\Omega_\nu} := \sum_{\mu=1}^{8} m^{\Omega_\nu^{son,\mu}} \tag{4.7}$$

$$\boldsymbol{y}_0^\nu := \frac{1}{m^{\Omega_\nu}} \sum_{\mu=1}^{8} m^{\Omega_\nu^{son,\mu}} \boldsymbol{y}_0^{son,\mu} \tag{4.8}$$

whereas Ω_ν denotes a cell associated to a vertex in the tree, $\Omega_\nu^{son,\mu}$ denotes the eight son cells associated to that vertex (if they exist), and $\boldsymbol{y}$ respectively the expansion points and m the associated masses. The expansion point of a non-trivial leaf, thus containing one particle, is the particle's position.
The computation of the forces for each given particle i at the position x_i starts at the root and the algorithm descends along the edges of the tree until the visited cells satisfy the selection criterion

$$\frac{diam}{r} \geq \theta \,, \tag{4.9}$$

with r as the distance of the associated pseudoparticle from the position $\boldsymbol{x}_i$ of particle i. The interactions of particle i with the pseudoparticles are computed and added to the global result. If the descent ends in a leaf, the interaction between the two particles is calculated directly, corresponding to the near field Ω^{near}.

4.2.2 GPU implementation

https://www.researchgate.net/publication/266523987_An_Efficient_CUDA_Implementation_of_the_Tree-Based_Barnes_Hut_n-Body_Algorithm

4.2.3 Multiple GPU implementation

- BEHALF: https://anaroxanapop.github.io/behalf/, https://github.com/bacook17/behalf

  Python implementation of multiple GPU Barnes-Hut tree

- rakau: https://github.com/bluescarni/rakau

  No multi-GPU implementation yet

- Gadget3: https://github.com/efarsarakis/gadget3

- StePS: multi-GPU algoirthm with MPI-OpenMP-CUDA hybrid parallelization [Rác+19] https://github.com/eltevo/StePS

- N-Body simulations on GPU cluster [Jet+10]: http://charm.cs.uiuc.edu/newPapers/10-16/paper.pdf, https://github.com/N-BodyShop/changa, https://github.com/N-BodyShop/changa/wiki/ChaNGa

- ECL-BH: simulating the gravitational forces in a star cluster using the Barnes-Hut n-body algorithm: https://userweb.cs.txstate.edu/~burtscher/research/ECL-BH/

- CUNBODY-1: implementation of accelerating particle-particle interaction in N-body simulation using GeForce8800GTX: https://github.com/thamada/cunbody1 [Ham+09]

- Multiple-walk parallel algoirthm for the Barnes-Hut treecode on GPUs [Ham+09]

  data of the multiple walks are prepared (number of walks determined by the resource limitations of a GPU)

  Multiple walks are sent to a GPU

  Calculations are performed (each GPU block evaluates only a single walk)

  Forces calculated by the different blocks in the GPU are received in a bundle

  ...

- NVIDIA fast N-Body simulation approach(es): https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-31-fast-n-body-simulation-cuda

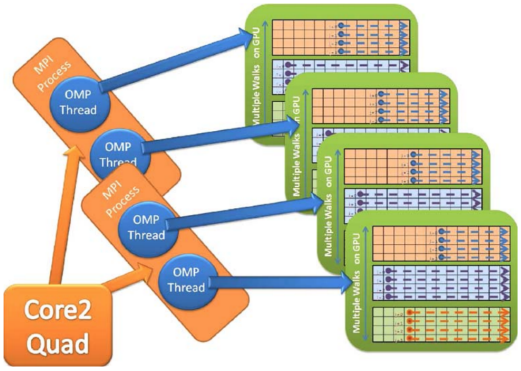


Figure 4.1: Strategy for parallelization in the host program. Each process/thread treats multiple walks in parallel using a shared GPU [Ham+09]

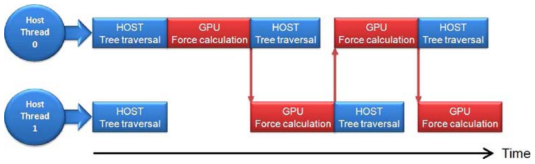


Figure 4.2: Multithread calculations using a host CPU and a GPU [Ham+09]

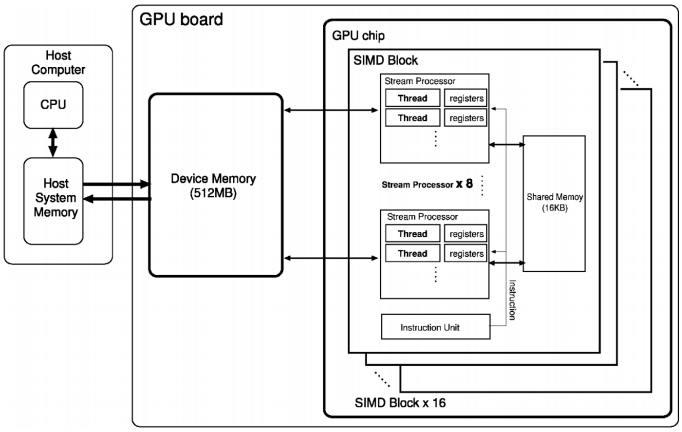


Figure 4.3: Basis structure of GeForce8800GTS GPU board [Ham+09]

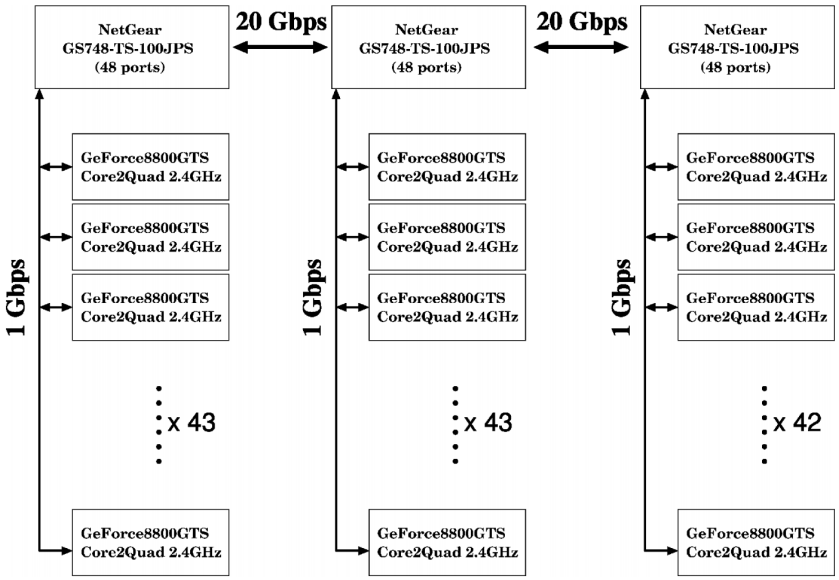


Figure 4.4: Block diagram of a GPU cluster. [Ham+09]

Chapter 5

Implementation

5.1 SPH implementation

5.1.1 Simple approach

- Initializing SPH-Particles
  position, velocity, forces ...
- Nearest neighbor search
- Calculation of density
- Calculation and Summation of Forces
  pressure, viscosity, gravity, ...
- time integration of vector force
- Updating (adding/deleting) SPH-particles
- Nearest neighbor search
- ...

5.1.2 Self-gravity

Direct sum

$O(N^2)$

Barnes-Hut

$O(n \log n)$
The idea is that only nearby particles have to be treated individually, more distant particles can be merged to a single one at the center of mass, carrying the mass of all the particles. Therefore a hierarchical tree walk is performed.
This hierarchical tree can be used to determine the interaction partners.

5.2 Parallelization

5.2.1 GPU programming

Use CUDA to utilize (NVIDIA) GPU card for GPGPU accelerated programming.

5.2.2 Multi-GPU/ Inter-GPU communication

`https://www.nvidia.com/docs/IO/116711/sc11-multi-gpu.pdf`

5.2.3 CPU programming

Use OpenMP for optimization of multiple CPU-cores.

5.2.4 Cluster/ Multiple machine

OpenMP + CUDA + MPI (see fig. **??**)

5.2.5 Multi-GPU machine

OpenMP + CUDA (see fig. **??**)

Chapter 6

Accelerating SPH

6.1 Problems of SPH regarding computational effort

6.1.1 Interaction partners

Each SPH particle need to interact with its neighbors.

6.1.2 Globally minimum timestep

Need to search for a globally minimum timestep.

6.2 Possible solutions

6.3 Multi-GPU SPH

6.3.1 Splitting the problem/SPH-particles

- **pipeline**: assign each phase of the computation to a different device

  needs the entire set of particles to be transferred for every iteration step across all the devices

- **list**: all the particles (no matter of their spatial position) are enumerated. Split the list into subsets and assign every subset to a device

  it is not sure, that all neighbors of every particles are within the same subset of data and accessing single particles in different devices is expensive

- **spatial split**: use a helping mesh grid and split the particles on a spatial basis and assign to different devices

  it is necessary to handle a minimum of overlapping of subdomains

6.3.2 Kernels

...

Chapter 7

C++

C++ is a general-purpose programming language created by Bjarne Stroustrup as an extension of the C programming language, or "C with Classes". The language has expanded significantly over time, and modern C++ now has object-oriented, generic, and functional features in addition to facilities for low-level memory manipulation. It is almost always implemented as a compiled language, and many vendors provide C++ compilers, including the Free Software Foundation, LLVM, Microsoft, Intel, Oracle, and IBM, so it is available on many platforms.

7.1 Versions and Compiler

C++ is standardized by an ISO working group. The current standard or stable release is **C++17** (ISO/IEC 14882:2017).

7.1.1 Versions

See: `https://github.com/AnthonyCalandra/modern-cpp-features`.

- **C++98:** First version/release of C++.
- **C++03:** Introduction of value initialization.
- **C++11:** Introduction of lambda expressions, delegating constructors, uniform initialization syntax, null pointer, automatic type deduction and decltype, Rvalue references, ... (see `https://www.learncpp.com/cpp-tutorial/b-1-introduction-to-c11/`)
- **C++14:** Introduction of polymorphic lambdas, digit separators, generalized lambda capture, variable templates, binary integer literals, quoted strings, ... (see `https://www.learncpp.com/cpp-tutorial/b-2-introduction-to-c14/`)
- **C++17:** Introduction of fold expressions, hexadecimal floating point literals, selection statements with initializer, inline variables, ... (see `https://www.learncpp.com/cpp-tutorial/b-3-introduction-to-c17/`)
- **C++20:** Upcoming release.

7.1.2 Compiler/Implementations

`https://en.wikipedia.org/wiki/Category:C%2B%2B_compilers`

- Acorn C/C++
- AMD Optimizing C/C++ Compiler
- Borland C++
- C++/CX: extension to call Windows runtime APIs
- C++Builder
- Cfront: original compiler from 1983
- **Clang**: Clang is a compiler front end for the C, C++, Objective-C and Objective-C++ programming languages, as well as the OpenMP, OpenCL, RenderScript, CUDA and HIP frameworks
- CodeWarrior: for embedded systems
- Comeau C/C++
- Digital Mars
- **GNU Compiler Collection**: The GNU Compiler Collection (**GCC**) is a compiler system produced by the GNU Project supporting various programming languages. GCC is a key component of the GNU toolchain and the standard compiler for most projects related to GNU and Linux, including the Linux kernel.
- HP aC++
- IBM XL C/C++ Compilers
- Intel C++ Compiler
- Microsoft Visual C
- Microsoft Visual C++: Microsoft Visual C++ (MSVC) is an integrated development environment (IDE) product from Microsoft for the C, C++, and C++/CLI programming languages
- **MinGW**: Microsoft Visual C++ (MSVC) is an integrated development environment (IDE) product from Microsoft for the C, C++, and C++/CLI programming languages
- Mingw-w64
- Norcroft C compiler
- Open64
- Oracle Developer Studio

- PathScale
- The Portland Group
- ROSE (compiler framework)
- Shed Skin: Shed Skin is a Python to C++ programming language compiler. It is experimental, and can translate pure, but implicitly statically typed Python programs into optimized C++.
- Softune
- TenDRA Compiler
- THINK C
- Turbo C++
- Unity build
- VisualAge
- Watcom C/C++

7.2 Compilation of C++ programs (GNU compiler)

7.2.1 Compilation process

The compilation process is carried out in 4 steps:

1. **Preprocessor:** *cpp hello.cpp > hello.i*
2. **Compiler:** *g++ -S hello.i*
3. **Assemble:** *as -o hello.o hello.s*
4. **Linker:** *ld -o hello.out hello.o ...libraries...*

That's equivalent to: *gcc -o hello.out hello.cpp.*

7.2.2 Headers and Libraries

- **Static library:** file extension *.a* (*.lib*). Linking a static library, the machine code of external functions used is copied into the executable.
- **Shared library:** file extension *.so* (*.dll*). Linking against a shared library, only a small table is created in the executable. Before the executable starts running, the operating system loads the machine code needed for the external functions - a process known as dynamic linking. Dynamic linking makes executable files smaller and saves disk space, because one copy of a library can be shared between multiple programs. Furthermore,

most operating systems allows one copy of a shared library in memory to be used by all running programs, thus, saving memory. The shared library codes can be upgraded without the need to recompile your program.

Shared library linking is preferred over static linking if possible.
The

- compiler needs the header files to compile the source code
- the linker needs the libraries to resolve external references from other object files or libraries

the headers/libraries are not traceable, unless appropriate options are set.
For each of the headers used in your source (via #include directives), the compiler searches the so-called **include-paths** for these headers. The include-paths are specified via *-Idir*option (or environment variable CPATH). Since the header's filename is known only the directories are needed.
The linker searches the so-called **library-paths** for libraries needed to link the program into an executable. The library-path is specified via *-Ldir* option (uppercase "L" followed by the directory path) (or environment variable LIBRARY_PATH). In addition, you also have to specify the **library name**. In Unixes, the library *libxxx.a* is specified via -lxxx option (lowercase letter "l", without the prefix "lib" and ".a" extension). In Windows, provide the full name such as -lxxx.lib. The linker needs to know both the directories as well as the library names.
Include-paths, Library-paths and Libraries can be found with *cpp -v*.
GCC uses following environmental variables:

- **PATH** for searching the executables and run-time shared libraries
- **CPATH** for searching the *include-paths* for headers. Searched after paths specified by *-I<dir>* option. C_INCLUDE_PATH and CPLUS_INCLUDE_PATH can be used to specify C and C++ headers if the particular language was indicated in pre-processing.
- **LIBRARY_PATH** for searching *library-paths* for link libraries. It is searched after paths specified in *-L<dir>* options

7.2.3 Utilities

- *file* determines the type of a file
- *nm* list symbol table/name list
- *ldd* list dynamic-link libraries

7.2.4 GNU Make

For automation of this process *make* can be used by calling a *Makefile*, containing rules of an executable.

Listing 7.1: Simple Makefile

```
1  all: hello.exe
2
3  hello.exe: hello.o
4         gcc -o hello.exe hello.o
5
6  hello.o: hello.c
7         gcc -c hello.c
8
9  clean:
10        rm hello.o hello.exe
```

Variables begins with a $ and are enclosed within parentheses (...). Single character variables do not need parantheses.
Automatic variables are

- *$@* the target filename
- *$** the target filename without file extension
- *$<* the first prerequsite filename
- *$^* the filenames of all prerequisite, separated by spaces
- *$?* the names of all prerequisites that are newer than target, separated by spaces

Virtual paths

- *VPATH* can be used to specify the directory to search for dependencies and target files
- *vpath* can be used to be more precise about the file type and its search directory

Pattern rules, using the matching character % as the file name, can be applied to create a target, if there is no explicit rule.

7.3 C++ Programming language

https://devdocs.io/cpp/
https://en.cppreference.com/w/

7.3.1 Concepts

Objects

C++ offers OOP features:

- **abstraction**
- **encapsulation**
- **inheritance**
- **polymorphism**

Templates

C++ templates enable **generic programming** (in terms of types to be specified later), implemented by instantiation at compile-time, and **template metaprogramming**.
A template is a compile-time parameterized function or class written without knowledge of the specific arguments used to instantiate it. After instantiation, the resulting code is equivalent to code written specifically for the passed arguments. In this manner, templates provide a way to decouple generic, broadly applicable aspects of functions and classes (encoded in templates) from specific aspects (encoded in template parameters) without sacrificing performance due to abstraction.

Listing 7.2: Function template

```
1  template <typename T> T max(T x, T y) {
2      T value;
3
4      if (x < y)
5          value = y;
6      else
7          value = x;
8
9      return value;
10 }
```

Lambda expressions

C++ provides lambda expressions/anonymous funtions:

Listing 7.3: Function template

```
[capture](parameters) -> return_type { function_body } // general form

[](int x, int y) { return x + y; } // inferred
[](int x, int y) -> int { return x + y; } // explicit
```

Exception handling

Listing 7.4: Function template

```
#include <iostream>
#include <vector>
#include <stdexcept>

int main() {
    try {
        std::vector<int> vec{3, 4, 3, 1};
        int i{vec.at(4)}; // Throws an exception, std::out_of_range (indexing
            for vec is from 0-3 not 1-4)
    }
    // An exception handler, catches std::out_of_range, which is thrown by vec
        .at(4)
    catch (std::out_of_range &e) {
        std::cerr << "Accessing a non-existent element: " << e.what() << '\n';
    }
    // To catch any other standard library exceptions (they derive from std::
        exception)
    catch (std::exception &e) {
        std::cerr << "Exception thrown: " << e.what() << '\n';
    }
    // Catch any unrecognised exceptions (i.e. those which don't derive from
        std::exception)
    catch (...) {
        std::cerr << "Some fatal error\n";
    }
```

7.4 Libraries

A list of (open-source) C++ libraries:

- https://en.cppreference.com/w/cpp/links/libs
- https://github.com/fffaraz/awesome-cpp

7.4.1 Standard (Template) library (STL)

The C++ Standard library is a collection of classes and functions, which are written in the core language and part of the C++ ISO standard itself.

- `https://en.wikipedia.org/wiki/C%2B%2B_Standard_Library`
- `https://en.cppreference.com/w/cpp/header`
- `http://www.cplusplus.com/reference/`

STL containers

The STL contains many different container classes useful in different situations. Generally container classes fall into three basic categories

- **Sequence Containers:** maintaining the ordering of elements in container (e.g.: *std::deque*, *std:array*, *std::list*, *std::forward_list*, and *std::basic_string*)
- **Associative Containers:** automatically sorting their inputs when inserted (by default using <). A **set** is a container that stores unique elements (no duplicate), sorted according to their values. A **multiset** is a set with possible duplicates. A **map** (or **associative array** is a set where each element is a pair, called a *key/value* pair, whereas the key is used for sorting and must be unique. A **multimap** (or **dictionary**) is a map allowing duplicate keys.
- **Container Adapters:** are special predefined containers that are adapted to specific uses, thus, the sequence container to be used can be chosen. A **stack** is a container where elements operate in a LIFO (Last in, First out) context and elements can be inserted (pushed) or removed (popped). A **queue** is a container where elements operate in FIFO (First in, First out) context. A **priority queue** is a type of queue where the elements are kept sorted, consequently a push removes the item with the highest priority (sorted via <).

STL iterators

An **iterator** is an object that can traverse (iterate over) a container class, without the need to know how the container is implemented.
(Overloaded) operators for iterators:

- *****: Dereferencing the iterator returns the element the iterator is currently pointing at.
- ++: Moves the iterator to the next element in the container (correspondingly moves − to the previous one).
- == and **!**=: basic comparison operators for determining whether two iterators point to the same element (first dereference).
- =: Assign the iterator to a new position (first dereference).

Basic member functions of iterators:

- **begin()**: returns an iterator representing the beginning of the elements
- **end()**: returns an iterator representing the element just past the end of elements
- **cbegin()**: gives an const (read-only) iterator corresponding to begin()
- **cend()**: const iterator corresponding to end()

All containers provide (at least) two types of iterators:

- **container::iterator** providing a read/write iterator
- **container::const_iterator** providing a read-only iterator

STL algorithm overview

STL also provides a number of generic algorithms for working with the elements of a container class, allowing you to

- search
- sort
- insert
- reorder
- remove
- copy
- ...

simply by including the algorithm header file.

std::string

Since many problems occur from (c-style) strings, the **std::string** and **std::wstring** classes are reasonable to use.

I/O streams

Input and output functionality is not defined in the core C++-language, but provided through the C++ standard library (resides in std namespace) e.g. with the **iostream** library.
Standard streams in C++:

- **cin** is an istream_withassign class tied to the standard input (keyboard)
- **cout** is an ostream_withassign class tied to the standard output (monitor)

- **cerr** is an ostream_withassign class tied to the standard error, providing unbuffered output
- **clog** is an ostream_withasign class tied to the standard error, providing buffered output

For basic file I/O three classes are provided

- ifstream
- ofstream
- fstream

C POSIX library

A specification of a C standard library for POSIX systems.

The GNU C Library

...

7.4.2 Boost library

Large collection of generic C++ libraries.

- `https://www.boost.org/`

7.4.3 Abseil

`https://github.com/abseil/abseil-cpp`
Abseil is an open-source collection of C++ code (compliant to C++11) designed to augment the C++ standard library.

7.4.4 Dlib

`https://github.com/davisking/dlib`

A general purpose cross-platform C++ library designed using contract programming and modern C++ techniques. Dlib is a modern C++ toolkit containing machine learning algorithms and tools for creating complex software in C++ to solve real world problems.

7.5 Modern C++ programming

7.5.1 Core Guidelines

`https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`

Philosophy

Summary:

- Express ideas directly in code
- Write ISO Standard C++
- Express intent
- Ideally, a program should be statically type safe
- Prefer compile-time checking over run-time checking
- What cannot be checked at compile time should be checkable at run time
- Catch run-time errors early
- Don't leak any resources
- Don't waste time or space
- Prefer immutable data to mutable data
- Encapsulate constructs, rather than spreading through the code
- Use supporting tools as appropriate
- Use support libraries as appropriate

7.6 Unit-testing

In programming, **unit testing** (see: `https://en.wikipedia.org/wiki/Unit_testing`) is a software testing method by which individual units of source code with associated control data, usage procedures and operating procedures are tested to determine whether they are fit to use. Typically are unit tests automated tests controlling that a section of an application (therefore a *unit*) behaves as intended. Some programming languages support directly unit testing, some other languages including C++ have very goo unit testing libraries/frameworks.
In OOP, unit testing is often combined with **mock objects**, simulated objects mimicking the behavior of real objects in controlled ways. A mock object is typically created to test the behavior of some other object.

7.6.1 Google Test

See **Google Test** - google Testing and Mocking Framework (`https://github.com/google/googletest`) on Github.
The **Google Test Framework** consists out of

- **Google Test** for writing unit tests
- **Google Mock** for writing mock classes

Links

- https://www.eriksmistad.no/getting-started-with-google-test-on-ubuntu/
- https://github.com/google/googletest
- https://rvarago.medium.com/introduction-to-google-c-unit-testing-3d564c30f3b0

Installation

On Debian (see https://rvarago.medium.com/introduction-to-google-c-unit-testing-3d564c30f3b0:

- `sudo apt install libgtest-dev`
- compile using *cmake* and *make*: `cd /usr/src/gtest sudo && cmake CMakeLists.txt && sudo make`
- copy to lib directory: `sudo cp libgtest.a libgtest_main.a /usr/lib`

Example

```
// calc.hpp
#ifndef CALC_HPP_
#define CALC_HPP_

int add(int op1, int op2);
int sub(int op1, int op2);

#endif // CALC_HPP_

// calc.cpp
#include "calc.hpp"

int add(int op1, int op2) {
    return op1 + op2;
}

int sub(int op1, int op2) {
    return op1 - op2;
}

// calc_test.cpp
#include <gtest/gtest.h>
#include "calc.hpp"

TEST(CalcTest, Add) {
    ASSERT_EQ(2, add(1, 1));
    ASSERT_EQ(5, add(3, 2));
```

```
    ASSERT_EQ(10, add(7, 3));
}

TEST(CalcTest, Sub) {
    ASSERT_EQ(3, sub(5, 2));
    ASSERT_EQ(-10, sub(5, 15));
}

int main(int argc, char **argv) {
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

To compile: `g++ -o calc_test calc_test.cpp calc.cpp -lgtest -lpthread`. GoogleTest (*gtest*) and POSIX Thread (*pthread*) libs must be linked.

7.6.2 Catch

See Catch Org (`https://github.com/catchorg`) and **Catch2** (`https://github.com/catchorg/Catch2`) for a modern, C++ native, header only test framework for unit-tests, TDD and BDD. Catch has no external dependencies.

Links

- `https://github.com/catchorg`
- `https://github.com/catchorg/Catch2`
- `https://github.com/catchorg/Catch2/blob/devel/docs/tutorial.md#top`
- Get the most out of Catch2: `https://github.com/catchorg/Catch2/blob/devel/docs/Readme.md#top`

Installation

```
$ git clone https://github.com/catchorg/Catch2.git
$ cd Catch2
$ cmake -Bbuild -H. -DBUILD_TESTING=OFF
$ sudo cmake --build build/ --target install
```

Example

```
#define CATCH_CONFIG_MAIN // This tells Catch to provide a main() - only do this in one cpp fil
#include "catch.hpp"

unsigned int Factorial( unsigned int number ) {
  return number > 1 ? Factorial(number-1)*number : 1;
}
```

```
TEST_CASE( "Factorials␣are␣computed", "[factorial]" ) {
    REQUIRE( Factorial(1) == 1 );
    REQUIRE( Factorial(2) == 2 );
    REQUIRE( Factorial(3) == 6 );
    REQUIRE( Factorial(10) == 3628800 );
}
```

7.6.3 Boost.Test

See the **Boost.test** (`https://www.boost.org/doc/libs/1_66_0/libs/test/doc/html/index.html`) for the C++ Boost.Test library, providing both an easy to use and flexible set of interfaces for writing test programs, organizing tests into simple test cases and test suites, and controlling their runtime execution.

Links

- `https://www.boost.org/doc/libs/1_66_0/libs/test/doc/html/index.html`
- `https://www.boost.org/doc/libs/1_45_0/libs/test/doc/html/utf.html`
- `https://beroux.com/english/articles/boost_unit_testing/`

Usage/Installation

- **Single-header usage**

```
#define BOOST_TEST_MODULE test module name
#include <boost/test/included/unit_test.hpp>
```

- **Static library usage**

```
#include <boost/test/unit_test.hpp>
#define BOOST_TEST_MODULE test module name
#include <boost/test/unit_test.hpp>
```

- **Shared library usage**

```
#define BOOST_TEST_DYN_LINK
#include <boost/test/unit_test.hpp>
#define BOOST_TEST_MODULE test module name
#define BOOST_TEST_DYN_LINK
#include <boost/test/unit_test.hpp>
```

Example

```
#include "StdAfx.h"
#include "../MyFoo.h"
#include <boost/test/unit_test.hpp>

using namespace std;

struct CMyFooTestFixture
{
    CMyFooTestFixture()
    : m_configFile("test.tmp")
    {
        // TODO: Common set-up each test case here.
        fclose( fopen(m_configFile.c_str(), "w+") );
    }

    ~CMyFooTestFixture()
    {
        // TODO: Common tear-down for each test case here.
        remove(m_configFile.c_str());
    }

    // TODO: Possibly put some common tests.
    void TestSaveLoad(CMyFoo& foo, bool asBinary)
    {
        BOOST_CHECK(foo.Save(asBinary));
    }

    // TODO: Declare some common values accesses in tests here.
    string m_configFile;
}

BOOST_FIXTURE_TEST_SUITE(MyFooTest, CMyFooTestFixture);

BOOST_AUTO_TEST_CASE(LoadTestConfigFile)
{
    CMyFoo foo;
    BOOST_REQUIRE(foo.IsValid()); // Stop here if it fails.
    TestSaveLoad(foo, true);
    TestSaveLoad(foo, false);
        BOOST_CHECK_THROW(foo.Save(nullptr), exception);
}
```

```
BOOST_AUTO_TEST_CASE(Name)
{
    CMyFoo foo;
    foo.SetName("␣␣foooooo␣");
        BOOST_CHECK_EQUAL(foo.GetName, "␣␣foooooo␣");
}

BOOST_AUTO_TEST_SUITE_END();
```

7.6.4 Doctest

Doctest (`https://github.com/onqtam/doctest`) is a new C++ testing framework but is by far the fastest both in compile times (by orders of magnitude) and runtime compared to other feature-rich alternatives. It brings the ability of compiled languages such as D / Rust / Nim to have tests written directly in the production code thanks to a fast, transparent and flexible test runner with a clean interface.

Links

- `https://github.com/onqtam/doctest`
- `https://github.com/onqtam/doctest/blob/master/doc/markdown/tutorial.md`
- `https://blog.jetbrains.com/rscpp/2019/07/10/better-ways-testing-with-doctest/`

Installation/Usage

To get started with doctest the latest version (`https://raw.githubusercontent.com/onqtam/doctest/master/doctest/doctest.h`) which is just a single header and include it in your source files (or add the repository as a git submodule).

Example

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include "doctest.h"

int factorial(int number) { return number > 1 ? factorial(number - 1) * number : 1; }

TEST_CASE("testing the factorial function") {
    CHECK(factorial(1) == 1);
    CHECK(factorial(2) == 2);
    CHECK(factorial(3) == 6);
    CHECK(factorial(10) == 3628800);
}
```

Chapter 8

CUDA

8.1 Links

- https://developer.nvidia.com/cuda-zone
- https://docs.nvidia.com/cuda/
- https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- https://developer.nvidia.com/blog/even-easier-introduction-cuda/
- https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
- http://www.mat.unimi.it/users/sansotte/cuda/CUDA_by_Example.pdf
- https://nvidia.github.io/libcudacxx/

8.2 Introduction

CUDA is a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs.

8.3 Programming model

The following refers to https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

8.3.1 Kernels

Kernels are functions running on the device (GPU) and are executed N ties in parallel by N different CUDA threads. A Kernel can be defined using __*global*__ and the number of threads that execute that kernel can be specified using «<...»>. For example:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
```

```
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Whereas, *threadIDx* is a 3-component vector, o that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume. There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads.
However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks.

8.3.2 Memory model

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 5. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory.

8.3.3 Execution model

The programming model assumes that CUDA threads execute on a physically separate device that operators s a coprocessor, both having their own separate memory spaces, including device memory allocation/deallocation as well as data transfer between host and device memory.

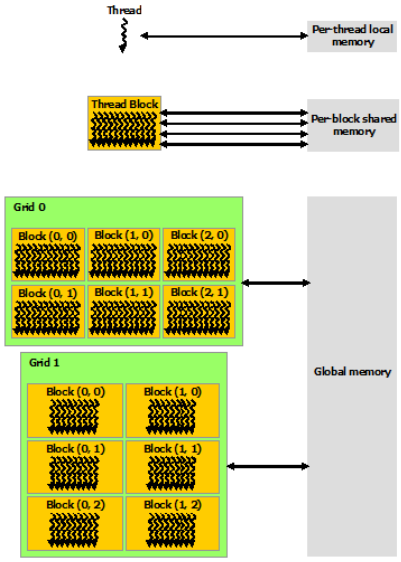


Figure 8.1: Memory hierarchy

8.4 C++ Language Extensions

8.4.1 Function Execution Space Specifiers

Defining whether a function executes on the host or on the device.

__global__

The **__global__** execution space specifier declares a function to be a kernel:

- executed on the device
- callable from the host
- callable from a device for devices
- must have void return type
- cannot be member of a class
- a call is asynchronous, meaning it returns before the device has completed its execution

__device__

The **__device__** execution space specifier declares a function that is

- executed on the device
- callable from the device only

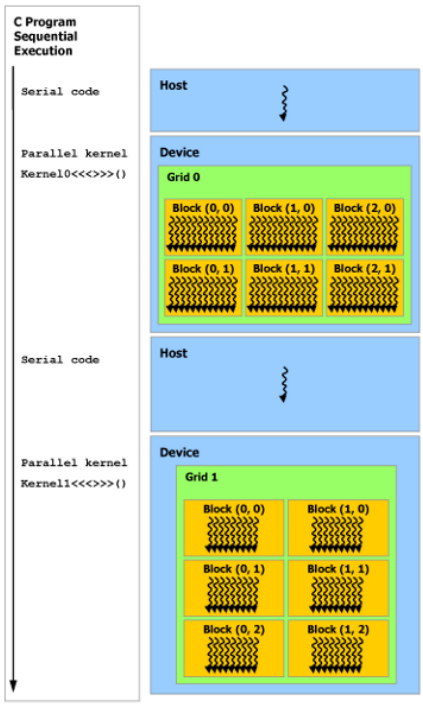


Figure 8.2: Execution hierarchy

- not usable in combination with __*global*__

__**host**__

The __**host**__ execution space specifier declares a function that is

- executed on the host
- callable from the host only
- usable in combination with __device__, in which case the function is compiled for both the host and the device

Others

The compiler inlines any __device__ function when deemed appropriate

- __**noinline**__ is a hint for the compiler to not inline if possible
- __**forceinline**__ forces the compiler to inline

8.4.2 Variable Memory Space specifiers

Variable memory space specifiers denote the memory location on the device of a variable.

__device__

The __device__ memory device specifier declares a variable on the device. If not declared otherwise, the variable

- resides in global memory space
- has the lifetime of the CUDA context in which created
- has a distinct object per device
- accessible from all the threads within the grid
- accessible from the host through runtime library using *cudaGetSymbolAddress(), cudaGetSymbolSize(), cudaMemcpyToSymbol, cudaMemcpyFromSymbol()*

__constant__

The __constant__ memory device specifier (optionally in combination with __device__)declares a variable that

- resides in constant memory space
- has the lifetime of the CUDA context in which created
- has a distinct object per device
- extitcudaGetSymbolAddress(), cudaGetSymbolSize(), cudaMemcpyToSymbol, cudaMemcpyFromSymbol()

__shared__

The __shared__ memory device specifier optionally in combination with __device__)declares a variable that

- resides in the shared memory space of a thread block
- has the lifetime of the block
- has a distinct object per block
- only accessible from all the threads within the block
- does not have a constant address

__managed__

The __managed__ memory device specifier optionally in combination with __device__)declares a variable that

- can be referenced from both device and host code
- has the lifetime of an application

__restrict__

nvcc supports restricted pointers via __restrict__ keyword.

8.4.3 Built-in Variables

Built-in variables are only valid within functions that are executed on the device.

- **gridDim** (type: dim3) contains the dimension of the grid
- **blockIdx** (type: unit3) contains the block index within the grid
- **blockDim** (type: dim3) contains the dimensions of the block
- **threadIdx** (type: uint3) contains the thread index within the block
- **warpSize** (type: int) contains the warp size in threads

8.4.4 Memory Fence Functions

The CUDA programming model assumes a device with a weakly-ordered memory model, that is the order in which a CUDA thread writes data to shared memory, global memory, page-locked host memory, or the memory of a peer device is not necessarily the order in which the data is observed being written by another CUDA or host thread.
Memory fence functions can be used to enforce some ordering on memory accesses.

- `void __threadfence_block()`
- `void __threadfence()`
-
- `void __threadfence_system()`

8.4.5 Synchronization Functions

`void __syncthreads()` waits until all threads in the thread block have reached this point.

8.5 CUDA Dynamic Parallelism

Supported API Functions:

- **cudaDeviceSynchronize**: Synchronizes on work launched from thread's own block only
- **cudaGetLastError**: Last error is per-thread state, not per-block state
- **cudaDeviceGetAttribute**: Return attributes for any device
- **cudaGetDevice**: Always returns current device ID as would be seen from host
- **cudaMemcpyAsync**
- **cudaFuncGetAttributes**
- **cudaRuntimeGetVersion**
- **cudaMalloc**
- **cudaFree**

8.6 CUDA C++ programming

Chapter 9

MPI

9.1 Links

- Open MPI: `https://www.open-mpi.org/software/ompi/v4.0/`
- MPICH: `https://www.mpich.org/downloads/`

9.2 MPI introduction

Referring to:

- `https://mpitutorial.com/tutorials/`
-

A **communicator** defines a group of processes that have the ability to communicate with another in dependence of their **ranks**. Communication is based on sending and receiving operations among processes. If one sender and receiver is involved, this refers to **point-to-point** communication. If a process need to communicate with everyone else **collective** communication involves all processes.
First the MPI header files need to be included *#include <mpi.h>* and the MPI environment must be initialized with

```
MPI_Init(int* argc, char*** argv)
```

constructing all of MPI's global and internal variables. After that

```
MPI_Comm_size(MPI_Comm communicator, int* size)
```

returns the size of a communicator and

```
MPI_Comm_rank(MPI_Comm communicator, int* rank)
```

returns the rank of a process in a communicator. The ranks of the processes are (primarily) used for identification purposes when sending and receiving messages.
Using

```
MPI_Get_processor_name(char* name, int* name_length)
```

gives the actual name of the processor on which the process is executing.
The final call is

`MPI_Finalize()`

used to clean up the MPI environment and no more MPI calls can be done afterwards.

Chapter 10

OpenMP

10.1 Links

- https://www.openmp.org//wp-content/uploads/OpenMP4.0.0.pdf
- https://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf
- https://computing.llnl.gov/tutorials/openMP/
- Compiler support C++: https://www.openmp.org//resources/openmp-compilers-tools/
- https://bisqwit.iki.fi/story/howto/openmp/

10.2 Introduction

Referring to:

- https://www.rc.fas.harvard.edu/wp-content/uploads/2016/04/Introduction-to-OpenMP.pdf
- http://www.red-ricap.org/documents/1071192/1486573/OpenMP_01_Introduction.pdf/2a2c91a8-60cf-4716-9ae4-57684a56e4b8

OpenMP (Open Multi-Processing as a Shared-Memory Parallel Programming Model) is the de-facto standard for shared-memory parallelization, thus for multicore systems. OpenMP programs start with one thread (the *Master*) and *Worker threads* are spawned at **Parallel Regions** together with the *Master* forming a *Team of threads*. In between Parallel Regions Worker threads are put to sleep.
OpenMP establishes a simple and limited set of directives for programming shared memory machines, implemented by just a few directives.

The programming model includes:

- **Shared Memory Model:** designed for multi-processor/core, shared memory machines
- **Thread Based Parallelism:** programs accomplish parallelism exclusively through the use of threads
- **Explicit Parallelism:** provides explicit (not automatic) parallelism, offering full control over parallelization

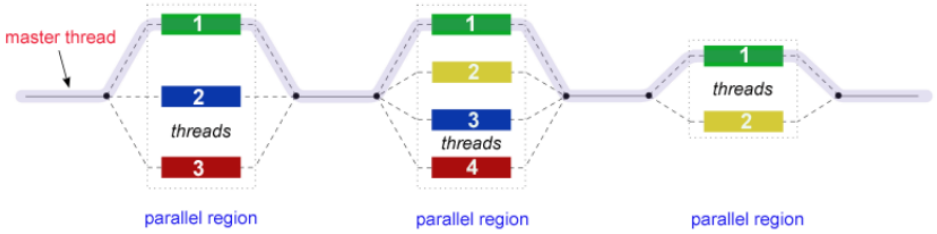


Figure 10.1: Fork-join model

- **Compiler Directive Based:** Parallelism is specified through the use of compiler directives embedded in the code
- **I/O:** OpenMP does not specify parallel I/O, therefore it is up to the programmer to ensure that I/O is conducted correctly
- **MPI:** MPI and OpenMP can interoperate to create a hybrid model of parallelism

Components of OpenMP embraces:

- **Compiler Directives (Pragma)** appear as comments in the code and are completely ignored by compilers unless otherwise stated by specifying appropriate compiler flags (GNU: *-fopenmp*). The compiler directives are used to spawn a parallel region, dividing blocks of code among threads, distributing loop iterations among threads, synchronize work among threads
- **Runtime Library Routines** are used to set and query number of threads, query threads unique identifier and the thread pool size
- **Environment Variables** are used to control execution of parallel code at run-time, thus for setting the number of threads, specifying how loop iterations are divided and enabling/disabling dynamic threads

10.3 OpenMP programming

The following refers to C/C++.
All directives begin with *#pragma omp* via

```
#pragma omp directive
{
        [structured block of code]
}
```

10.3.1 Parallel region construct

The fundamental OpenMP parallel construct is

```
#pragma omp parallel [clause ...]
       if (scalar_expression)
       private (list)
       shared (list)
       default (shared | none)
       firstprivate (list)
       reduction (operator: list)
       copyin (list)
       num_threads (integerexpression)
```

10.3.2 Important Runtime Routines

- *OMP_SET_NUM_THREADS* sets the number of threads for the application
- *OMP_GET_NUM_THREADS* polls the current setting for number of threads
- *OMP_GET_THREAD_NUM* tells which thread number you are
- *OMP_GET_WTIME* times the routine

10.3.3 DO/FOR directives

If a parallel region has already been initiated the do/for directive executes the iterations of the loop parallel by the team

```
#pragma omp for [clause ...]
       schedule (type [,chunk])
       ordered
       private (list)
       firstprivate (list)
       lastprivate (list)
       shared (list)
       reduction (operator: list)
       collapse (n)
       nowait
```

Scheduling

The scheduling algorithm for the for-loop can be controlled via `pragma omp for schedule(static)`, with the scheduling types

- **static** (default): each thread decides independently which chunk of the loop they will process

- **dynamic**: no predictable order in which the loop items are assigned to the threads, most useful in combination with *ordered* or when the different iterations of the loop take different time to execute. The chunk size can be specified via `pragma omp for schedule (dynamic, 3`
- **guided**: behave like static with the shortcomings of static fixed with dynamic-like traits
- **auto**
- **runtime**: runtime library chooses the scheduling options at runtime

and the scheduling modifiers

- **monotonic**: each thread executes chunks in increasing iteration order
- **nonmonotonic**: each thread executes chunks in an unspecified order
- **simd**: if the loop is a simd loop, this controls the chunk size for scheduling in a manner that is optimal for the hardware limitations (according to the compiler)

which can be added to the clause via `pragma omp for schedule (nonmonotonic:dynamic)`.

Ordering

The order in which the loop iterations are executed is unspecified and depends on the runtime conditions. It is possible to force that certain events within the loop happen in a predicted order using the *ordered* clause

```
#pragma omp for ordered schedule(dynamic)
 for(int n=0; n<100; ++n)
 {
   files[n].compress();

   #pragma omp ordered
   send(files[n]);
 }
```

In the above example are all files "sent" in a strictly sequential order.
Only one *ordered* block per an ordered loop, no less or more.

Collapse

When having nested loops, *collapse* can be used to apply the threading to multiple nested iterations. E.g.:

```
#pragma omp parallel for collapse(2)
 for(int y=0; y<25; ++y)
   for(int x=0; x<80; ++x)
   {
     tick(x,y);
   }
```

Reduction

The reduction clause is a special directive that instructs the compiler to generate code that accumulates values from different loop iterations together in a certain manner. E.g.:

```
int sum=0;
 #pragma omp parallel for reduction(+:sum)
 for(int n=0; n<1000; ++n) {
       sum += table[n];
 }
```

10.3.4 Sections

It may be useful to define *sections* to indicate, that this and this can be run in parallel

```
//#pragma omp parallel {
#pragma omp (parallel) sections
 {
   { Work1(); }
   #pragma omp section
   { Work2();
     Work3(); }
   #pragma omp section
   { Work4(); }
 }
 //}
```

10.3.5 Single Instruction, Multiple-Data (SIMD)

SIMD parallelism (or **vector** parallelism) means that multiple calculations will be performed simultaneously by the processor doing the same calculation to multiple values at once.
`pragma omp simd` can be used to declare that a loop will be utilizing SIMD.
`pagma omp declare simd` can be used to indicate a function or procedure that is designed to take advantage of SIMD parallelism

```
#pragma omp declare simd aligned(a,b:16)
  void add_arrays(float *__restrict__ a, float *__restrict__ b)
  {
    #pragma omp simd aligned(a,b:16)
    for(int n=0; n<8; ++n) a[n] += b[n];
  }
```

Following clauses can be used/added:

- *collapse* clause can be added to allow multiple nested loops with SIMD
- *reduction* clause can be used just like with parallel loops

- *aligned* (e.g. aligned(a,b:16)) hints the compiler that each element listed is aligned to the given number of bytes
- *safelen* controlling over pointer aliasing
- *simdlen* limiting how many elements of an array are passed to the SIMD register
- *uniform* declares one or more arguments to have an invariant value for all concurrent invocations
- *linear*
- *inbranch* called from inside a conditional statement of a SIMD loop
- *notinbranch* never be called from inside

The *for simd* construct

For and *simd* can be combined to divide the execution of a loop into multiple threads and then execute those loop slices in parallel using SIMD

```
float sum(float* table)
  {
    float result=0;
    #pragma omp parallel for simd reduction(+:result)
    for(int n=0; n<1000; ++n) result += table[n];
    return result;
  }
```

10.3.6 Task (construct)

If *for* and *sections* are to cumbersome, *task* can be used.

```
struct node { node *left, *right; };
extern void process(node* );
void postorder_traverse(node* p)
{
    if (p->left)
        #pragma omp task // p is firstprivate by default
        postorder_traverse(p->left);
    if (p->right)
        #pragma omp task // p is firstprivate by default
        postorder_traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

10.3.7 Offloading (support)

Offloading means that parts of the program can be executed in other hardware (e.g. GPU). Using **declare target** and **end declare target** a section of source code can be delimited wherein all declarations are compiled for a device. Using

- **target data** construct, creates a device data environment
- **target** construct executed the construct on a device

```
#pragma omp target // device()... map()... if()...
{
  <<statements...>>
}
//or (equally)
#pragma omp target data // device()... map()... if()...
{
  #pragma omp target
  {
    <<statements...>>
  }
}
```

It is important to mention, that *target* does not add any parallelism, it only transfers the execution into another device and executes the code there in a single thread. To utilize parallelism on the device, **teams** constructs inside the *target* construct are necessary.
Clauses to use with *target*:

- *if* decides whether code is only executed on the host or device
- *device* specifies the particular device to execute the code

  omp_set_default_device, *omp_get_default_device*, *omp_get_num_devices* and *omp_is_initial_device* can be used to acquire device numbers
- *map* controls how data is exchanged between the host and the device via (*map(alloc:variables)*, *map(from:variables)*), *map(to:variables)*, *map(tofrom:variable)*)
- `pragma omp target enter data map(from:var)`
- `pragma omp target exit data map(to:var)`
- *target update* can be used to synchronize data between device memory and host memory (without deallocation)

10.3.8 Teams

Only within *target*. While *parallel* creates a **team of threads**, the *teams* construct creates **league of teams**.

10.3.9 Synchronization Constructs

Each thread is independent, thus different threads may complete different sections at different times, getting them out of sync. There are several concepts provided for synchronization, like the **barrier** directive

```
#pragma omp barrier
```

10.3.10 Thread safety

Atomicity

Using `pragma omp atomic` means that something is inseparable, it either happens completely or it does not happen at all and another thread cannot intervene during the execution.

- *atomic read*
- *atomic write*
- *atomic update*
- *atomic capture*

Critical (construct)

Using the *critical* construct restricts the execution of the associated block to a single thread at time. No to threads can execute a *critical* construct of the same name at the same time, giving a name via `pragma omp critical (name)`.

Locks

OpenMP provides a lock type *omp_lock_t* with the manipulator functions

- *omp_init_lock* initializing the lock (unset lock)
- *omp_destroy_lock* destroys the lock (must be unset)
- *omp_set_lock* attempts to set the lock, if already set, the thread wait until no longer set and then sets it
- *omp_unset_lock* unsets the lock, only callable by same thread that set the lock
- *omp_test_lock* attempts to set the lock. If the lock is already set by another thread, it returns 0; if it managed to set the lock, it returns 1

E.g.:

```
#ifdef _OPENMP
 # include <omp.h>
 #endif
 #include <set>
```

```
class data
{
private:
  std::set<int> flags;
#ifdef _OPENMP
  omp_lock_t lock;
#endif
public:
  data() : flags()
  {
#ifdef _OPENMP
    omp_init_lock(&lock);
#endif
  }
  ~data()
  {
#ifdef _OPENMP
    omp_destroy_lock(&lock);
#endif
  }

  bool set_get(int c)
  {
  #ifdef _OPENMP
    omp_set_lock(&lock);
  #endif
    bool found = flags.find(c) != flags.end();
    if(!found) flags.insert(c);
  #ifdef _OPENMP
    omp_unset_lock(&lock);
  #endif
    return found;
  }
};
```

Flush

The *flush* directive can be used to ensure that the value observed in one thread is also the value observed by other threads. Needed if one needs to write and read from the same data in different threads. **Attention:** If the program appears to work correctly without the flush directive, it does not mean that the flush directive is not required.

10.3.11 Controlling which data to share between threads

In the parallel section, it is possible to specify which variables are shared between the different threads and which are not. By default, all variables are shared except those declared within the parallel block.

- **private** each thread has their own copy
- **shared** each thread access the same variable

10.3.12 Thread affinity

The thread affinity can be controlled via *proc_ bind*:

- `pragma omp parallel proc_bind(master)`
- `pragma omp parallel proc_bind(close)`
- `pragma omp parallel proc_bind(spread)`

10.3.13 Execution synchronization

Barrier and nowait

The *barrier* directive causes threads encountering the barrier to wait until all the other threads in the same team have encountered the barrier.
There is an implicit barrier at the end of each parallel block and at the end of each section, for and single statement, unless the *nowait* directive is used.

```
#pragma omp parallel
 {
   /* All threads execute this. */
   SomeCode();

   #pragma omp barrier

   /* All threads execute this, but not before
    * all threads have finished executing SomeCode().
    */
   SomeMoreCode();
 }
 #pragma omp parallel
 {
   #pragma omp for nowait
   for(int n=0; n<10; ++n) Work();

   // This line may be reached while some threads are still executing the for-loop.
```

```
  SomeMoreCode();
}

// This line is reached only after all threads from
// the previous parallel block are finished.
CodeContinues();
```

10.3.14 Single and Master

The *single* construct specifies that the given block is executed by only one thread (unspecified which thread). The *master* construct is similar, but the thread is no longer unspecified, since the master thread needs to run this block.

10.3.15 Thread cancellation

Using OpenMP parallel constructs, *return*, *goto*, *break* and *throw* are not allowed. To do something similar **cancellation points** need to be invoked, using

- `pragma omp cancellation point`
- `pragma omp cancellation point for`

The library-internal global variable *OMP_CANCELLATION* can be checked with the *omp_get_cancellation()* function.

10.4 Notes regarding C++

Performance

Compared to a naive use of C++11 threads, OpenMP threads are often more efficient, since OpenMP uses **thread pools**.

STL

The STL is not thread-safe, therefore when using STL containers in a parallel context, concurrent access (using locks or other mechanism) must be excluded.

Exceptions

Exceptions may not be thrown and caught across OpenMP constructs.

Chapter 11

OpenCL

11.1 Links

- `https://www.khronos.org/opencl/`
- `https://www.khronos.org/opencl/resources`
- `https://developer.nvidia.com/opencl`
-

11.2 Introduction

Referring to

- `https://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/opencl/opencl-03-basics.pdf?__blob=publicationFile`

OpenCL is a programming framework for CPUs, GPUs, DSPs and FPGAs. Compared to CUDA, OpenCL is not as mature and widely used, but supports heterogeneous platforms and various processor types.

11.3 Concept of OpenCL

11.3.1 Platform model

The basic structure embraces a **host** (computational unit, usually a CPU) connected to several **devices** (CPUs, GPUs, DSPs, FPGAs).

Every OpenVL implementation (with underlying OpenCL library) defines a so-called **platform**. Each specific platform enables the host to control the devices belonging to it.

11.3.2 Execution model

A **Kernel** is a function for execution on the device (e.g. GPU). Since many kernel instantiations running simultaneously in parallel threads, a proper thread management is required.
Hierarchical thread organization:

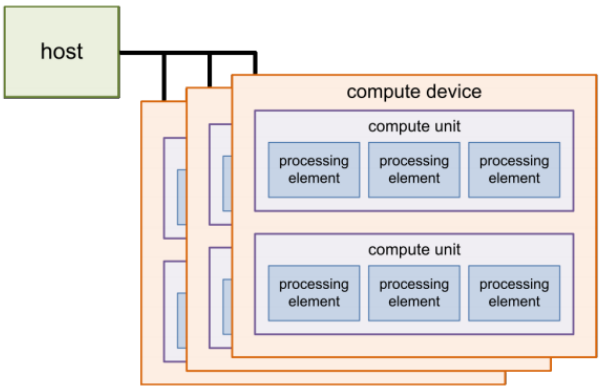


Figure 11.1: Platform model

- Upper level: **Grid**
- Medium level: **Block**
- Lower level: **Thread**

A **Block Scheduler** distributes groups of work-items to perform load-balancing. **Warps** or **Wavefronts** are groups of work-items scheduled and executed together.
Thus, the **host** executes the host program and the **device** the device kernel. The host manages the **queues**, therefore the kernel execution, operations on memory objects and synchronization.

11.3.3 Memory model

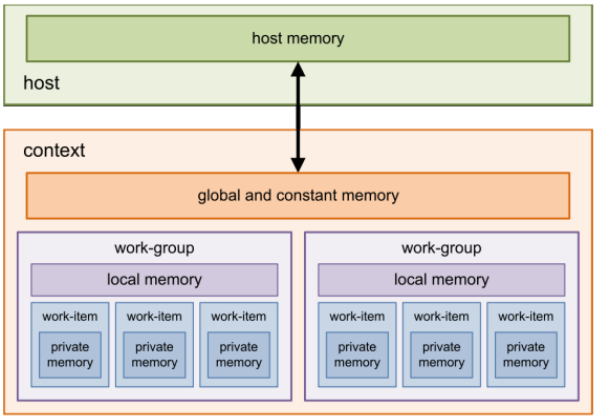


Figure 11.2: Memory model

11.3.4 Programming model

Supported approaches are **Data Parallelization** (e.g. for GPUs) and **Task Parallelization** (e.g. for Multi-core CPUs or multi-CPU systems).
The basic programming steps embraces

- determine components of heterogeneous system
- query specific properties of each component to adapt program execution dynamically during runtime
- compile and configure the OpenCL kernels
- create and initialize memory objects
- execution of the kernels in the correct order with the best suited device for each kernel
- collection of results

which can be done using functions from the **OpenCL Platform and Runtime API**.

11.4 Nomenclature (OpenCL vs. CUDA)

OpenCL	CUDA
Work–Item	Thread
Work–Group	Block
NDRange (Workspace)	Grid
Local Memory	Shared Memory
Private Memory	Registers/Local Memory
Image	Texture
Queue	Stream
Event	Event

Figure 11.3: Nomenclature (OpenCL vs. CUDA

Chapter 12

Hybrid Parallelization

12.1 CUDA-Aware MPI

- https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/
- https://www.open-mpi.org/faq/?category=building#build-cuda
- https://stackoverflow.com/questions/27908813/requirements-for-use-of-cuda-aware-mpi
- https://fisica.cab.cnea.gov.ar/gpgpu/images/charlas/multi_gpu_programming_with_mpi.pdf
- http://nowlab.cse.ohio-state.edu/static/media/publications/abstract/IWOPH_19_MPI_on_POWER.pdf

12.2 OpenMP on GPUs (CUDA)

- https://on-demand.gputechconf.com/gtc/2018/presentation/s8344-openmp-on-gpus-first-experi pdf
-

Chapter 13

Build, Test and Package Software

13.1 CMake

13.1.1 Links

See

- `https://cmake.org/cmake/help/v3.19/`
- `https://www.jetbrains.com/help/clion/quick-cmake-tutorial.html`
- `https://tuannguyen68.gitbooks.io/learning-cmake-a-beginner-s-guide/content/index.html`

13.1.2 Introduction

CMake is a **build system generator** and useful if its wanted to

- avoid hard-coding paths
- build a package on more than one computer
- support different OSs
- support multiple compilers
- describe the logic of the program
- use a library

CMake generates input files for build-generators like Make, Ninja, Visual Studio, XCode. It is cross-platform and supports multiple languages.
Some terms:

- **Build-requirements**: everything that is need to build the target (source files, preprocessor macros, compiler/linker-options, link dependencies, ...)
- **Usage-requirements**: everything that is needed to use the target (include search-paths, link dependencies, compiler/linker-options, ...)

Global variables

The following global variables can be set via `set(<global variable> <setting>)`.

- CMAKE_BINARY_DIR
- CMAKE_SOURCE_DIR
- EXECUTABLE_OUTPUT_PATH
- LIBRARY_OUTPUT_PATH
- PROJECT_NAME
- PROJECT_SOURCE_DIR

Usage

For a minimal project the following **CMakeLists.txt** is required

```
cmake_minimum_required(VERSION 3.10)
# set the project name
project(Tutorial)
# add the executable
add_executable(Tutorial tutorial.cxx)
# Set the output folder where your program will be created
set(CMAKE_BINARY_DIR ${CMAKE_SOURCE_DIR}/bin)
set(EXECUTABLE_OUTPUT_PATH ${CMAKE_BINARY_DIR})
set(LIBRARY_OUTPUT_PATH ${CMAKE_BINARY_DIR})
```

Specifying the C++ standard can be done using `set(CMAKE_CXX_STANDARD 11` and `set(CMAKE_CXX_STANDARD_R`

Build and Test

Navigate to the build directory and run CMake to configure the project and generate a native build system:

```
cd Step1_build
cmake ../Step1
```

and call the build system to actually compile/link the project

```
cmake --build
```

Adding a library

Assuming a library in a subdirectory *MathFuncitons* containing a header file *MathFunctions.h* and a source file *mysqrt.cxx*:

```
add_library(MathFunctions mysqrt.cxx)
# add the MathFunctions library
# making the library optional
option(USE_MYMATH "Use tutorial provided math implementation" ON)
#configure a header file to pass some of the CMake settings to the source code
configure_file(TutorialConfig.h.in TutorialConfig.h)
if (USE_MYMATH)
       add_subdirectory(MathFunctions)
       list(APPEND EXTRA LIBS MathFunctions)
       list(APPEND EXTRA_INCLUDES "${PROJECT_SOURCE_DIR}/MathFunctions")
endif()
# add the executable
add_executable(Tutorial tutorial.cxx)
target_link_libraries(Tutorial PUBLIC MathFunctions)
# add the binary tree to the search path for include files
target_include_directories(Tutorial PUBLIC "${PROJECT_BINARY_DIR}" "${PROJECT_SOURCE_DIR}/MathF
# or rather: target_include_directories(Tutorial PUBLIC "${PROJECT_BINARY_DIR}" ${EXTRA_INCLUDE
```

Since the source code now requires *USE_MYMATH*, add it to *TutorialConfig.h.in* via `#cmakedefine USE_MYMATH`.

Usage Requirements for Library

The primary commands leveraging usage requirements:

- *target_compile_defintions()*
- *target_compile_options()*
- *target_include_directories()*
- *target_link_libraries()*

Installing and Testing

Install rules can be done via

```
install(TARGETS <targets> DESTINATION <destination (lib)>)
install(FILES <files> DESTINATION <destination (include)>)
```

Install via

```
cmake --install . --prefix "<installation prefix>"
```

By adding tests in *CMakeLists.txt* via

```
enable_testing()
add_test(NAME <name> COMMAND <command>)
set_tests_properties(...)
# result based tests
do_test(<project> ...)
```

System Introspection

For testing the availability of functions `check_symbol_exists(...)` can be used.

Chapter 14

Thesis title

14.1 Thesis title

14.1.1 Thesis content

- SPH
- Self gravity
- (Barnes-Hut) tree (code)
- C++
- (Multi-)(GP)GPU
- Parallelization
- Cluster
- High performance computing (HPC)

14.1.2 Possible titles

- Parallel tree code for self-gravitating systems
- Parallel tree code for n-body problems
- Parallel tree code for SPH
- SPH with self-gravity parallelized for HPC
-

Chapter 15

Concepts to reduce computational time

15.1 Self-Gravity only for dense regions

Apply gravity only for dense regions. ...

15.2 Updating neighbor list

Updating the neighbor list only every k iterations. ...

15.3 ...

...

Chapter 16

Todo

- correct plagiarism disclaimer (see examination regulations)

Chapter 17

Links

https://arxiv.org/pdf/1909.07439.pdf

link	content	description
https://arxiv.org/abs/1909.07439	Bonsai-SPH	A GPU accelerated astrophysical Smoothed Particle Hydrodynamics code
https://github.com/treecode/Bonsai	Github Bonsai code	see branch *BonsaiSPH* for SPH version of code
https://github.com/christophmschaefer/miluphcuda	miluphcuda Github	-

Bibliography

[Ham+09] T. Hamada, K. Nitadori, K. Benkrid, Y. Ohno, G. Morimoto, T. Masada, Y. Shibata, K. Oguri, and M. Taiji. "A novel multiple-walk parallel algorithm for the Barnes–Hut treecode on GPUs –towardscost effective, high performance N-body simulation." In: *Computer Science - Research and Development* 24.1 (2009), pp. 21–31. DOI: 10.1007/s00450-009-0089-1.

[Jet+10] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn. "Scaling Hierarchical N-body Simulations on GPU Clusters." In: *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 2010, pp. 1–11. DOI: 10.1109/SC.2010.49.

[Kos+19] D. Koschier, J. Bender, B. Solenthaler, and M. Teschner. "Smoothed Particle Hydrodynamics Techniques for the Physics Based Simulation of Fluids and Solids." In: *Eurographics 2019 - Tutorials*. Ed. by W. Jakob and E. Puppo. The Eurographics Association, 2019. DOI: 10.2312/egt.20191035.

[Rác+19] G. Rácz, I. Szapudi, L. Dobos, I. Csabai, and A. S. Szalay. *StePS: A Multi-GPU Cosmological N-body Code for Compactified Simulations*. 2019. arXiv: 1811.05903 [astro-ph.CO].