Chapter 3

Software in HPC

3.1 Introduction

After presenting the rules of HPC and the hardware that compose the cluster, we introduce the most famous ways to target those architectures and supercomputers with programming models. Then, fitting those models, we present the possible options in the language, the API, the distribution and the accelerators code.

This chapter details the most important programming models and the software options for HPC programming and include the choices we made for our applications. Then it presents the software used to benchmark the supercomputers. We present here the most famous, the TOP500, GRAPH500, HPGC and GREEN500 to give their advantages and weaknesses.

3.2 Parallel and distributed programming Models

The Flynn taxonomy developed in chapter 1 was a characterization of the executions models. This model can be extended to programming models which are an extension of MIMD. We consider here a *Random Access Machine* (RAM). The memory of this machine consists of an unbounded sequence of registers each of which may hold an integer value. In this model the applications can access to every memory words directly in write or read manner. There is three main operations: load from memory to register; compute operation between data; store from register to memory. This model is use to estimate the complexity of sequential algorithms. If we consider the unit of time of each operation (like in cycle) we can have an idea of the overall time of the application. We identify two types of RAM, the Parallel-RAM using shared memory and the Distributed-RAM using distributed memory.

3.2.1 Parallel Random Access Machine

The Parallel Random Access Machine [FW78], PRAM, is a model in which the global memory is shared between the processes and each process have its own local memory/registers. The execution is synchronous, processes execute the same instructions at the same time. In this model each process is identify with its own index enabling to target different data. The problem in this model will be the concurrency in reading (R) and writing (W) data as the memory is shared between the processes. Indeed, mutual exclusion have to be set with exclusive (E) or concurrent (C) behaviors and we find 4 combinations: EREW, ERCW, CREW and CRCW. As the reading is not critical for data concurrency the standard model will be Concurrent Reading and Exclusive Writing: CREW.

3.2.2 Distributed Random Access Machine

For machine that base their memory model on NoRMA the execution model can be qualify of Distributed Random Access Machine, DRAM. It is based on NoRMA memories detailed in

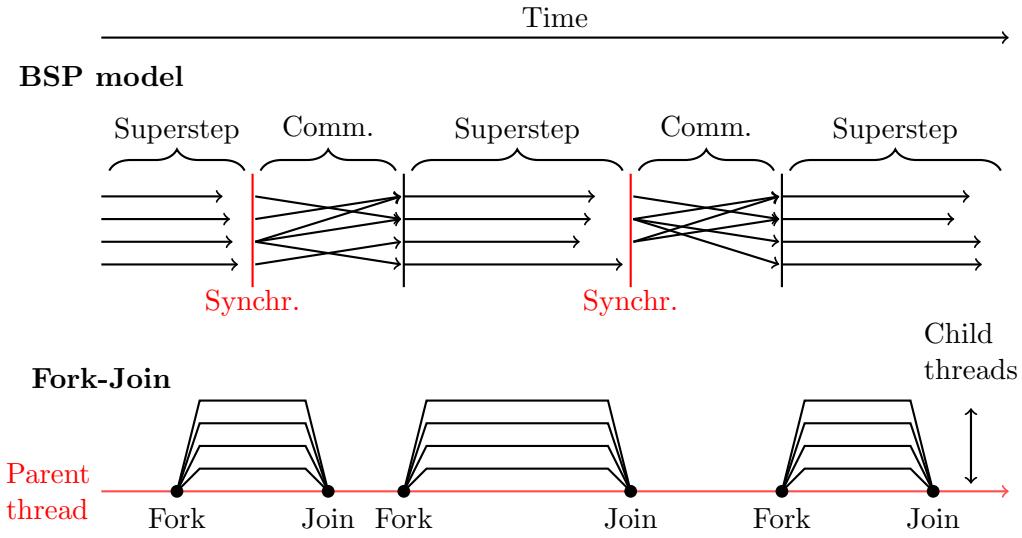

Figure 3.1: Bulk Synchronous Parallel model and Fork-Join model

part **??**. This model is in opposition to PRAM because the synchronization between processes is made by communications and messages. Those communications can be of several kind and depend of physical architecture, interconnection network and software used.

3.2.3 H-PRAM

A DRAM can be composed of an ensemble of PRAM system interconnected. Each of them working on their own data and instructions. This is an intermediate model between PRAM and DRAM having a set of shared memory and synchronous execution, the overall execution being asynchronous and having distributed memory.

3.2.4 Bulk Synchronous Parallelism

This model was presented in 1990 in [Val90]. Being the link of HRAM and PRAM The Bulk Synchronous Parallelism model is based on three elements:

- a set of processor and their local memory;
- a network for point-to-point communications between processors;
- a unit allowing global synchronization and barriers.

This model is the most common on HPC clusters. It can be present event on node themselves: a process can be assign on a core or set of cores and the shared memory is separated between the processes. The synchronization can be hardware but in most cases it is handle by the runtime used. A perfect example of runtime, presented later, is MPI.

In this model the applications apply a succession of *supersteps* separated by *synchronizations* steps and data exchanges.

At opposite to H-PRAM which represent the execution as a succession of independent blocks working synchronously, BSP propose independent blocks of asynchronous applications synchronized by synchronization steps.

In a communication/synchronization step we can consider the number of received messages h_r and the number of send ones h_s.

The time lost in communication in one synchronization step is:

$$T_{comm} = hg + I \tag{3.1}$$

With $h = max(h_s, h_r)$, g the time to transfer data and I the start-up latency of the algorithm. Indeed, the entry points and exit points of communications super-step can be a bottleneck

considered in I.

The time for computing a super-step is:

$$T_{comp} = \frac{w}{r} + I \tag{3.2}$$

With w the maximum number of flops in the computation of this super-step, r the speed of the CPU expressed in FLOPS and I the start-up latency of the algorithm. Indeed, the entry points and exit points of communications super-step can be a bottleneck considered in I.

The BSP model estimates the cost of one super-step with:

$$T_{comm} + T_{comp} = w + gh + 2l \tag{3.3}$$

With T a measure of time, a wall clock that measure elapsed time. We also note that usually g and I are function of the number of processes involved.

It can then be use to compute the overall cost in BSP model summing all super-steps s:

$$T = \sum_s \frac{max(w_s)}{r} + h_s g + I \tag{3.4}$$

The problem of performances in this model can come from unequal repartitions of work, the load balancing. The processes with less than w of work will be idle.

3.2.5 Fork-Join model

The Fork-Join model or pattern is presented in figure 3.1. A main thread pilot the overall execution. When requested by the application, typically following the idea of *divided-and-conquer* approach, the main thread will fork and then join other threads. The *Fork* operation, called by a logical thread parent, creates new logical threads children working in concurrency. There is no limitations in the model and we find nested fork-join where a child can also call fork to generate sub-child and so on. The *Join* can be called by both parents and child. Children call join when done and the parent join by wait until children completion. The Fork operation increase concurrency and join decrease concurrency.

3.3 Software/API

In this section we present the main runtime, API and frameworks use in HPC and in this study in particular. The considered language will be C/C++, the most present in HPC world along with Fortran.

3.3.1 Shared memory programming

On the supercomputers nodes we find one or several processors that access to UMA or NUMA memory. Several API and language provide tools to target and handle concurrency and data sharing in this context. The two main ones are PThreads and OpenMP for multi-core processors. We can also cite Cilk++ or TBB from Intel.

PThreads

The Portable Operating System Interface (POSIX) threads API is an execution model based on threading interfaces. It is developed by the IEEE Computer Society. It allows the user to define threads that will execute concurrently on the processor resources using shared/private memory. PThreads is the low level handling of threads and the user need to handle concurrency with semaphores, conditions variables and synchronization "by hand". This makes the PThreads hard to use in complex applications and used only for very fine-grained control over the threads management.

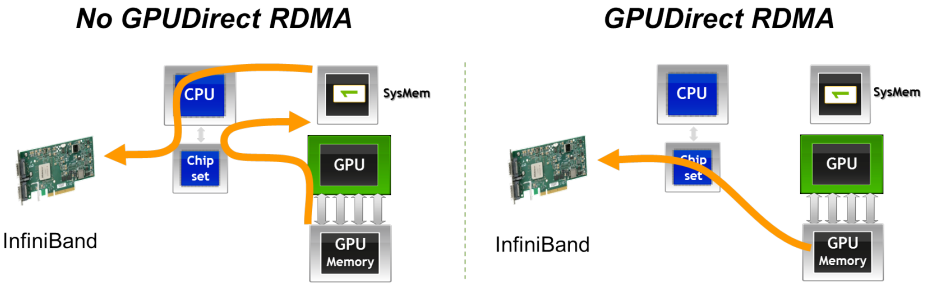

Figure 3.2: GPUDirect RDMA from NVIDIA Developer Blog, *An Introduction to CUDA-Aware MPI*

OpenMP

Open Multi-Processing, OpenMP[1] [Cha08, Sup17], is an API for multi-processing shared memory like UMA and CC-NUMA. It is available in C/C++ and Fortran. The user is provided with pragmas and functions to declare parallel loop and regions in the code. In this model the main thread, the first one before forks, command the fork-join operations.

The last versions of OpenMP also allow the user to target accelerators. During compilation the user specify on which processor or accelerator the code will be executed in parallel.

3.3.2 Distributed programming

In the cluster once the code have been developed locally and using the multiple cores available, the new step is to distribute it all over the nodes of the cluster. This step requires the processes to access NoRMA memory from a node to another. Several runtime are possible for this purpose and concerning our study. We should also cite HPX, the c++ standard distribution library, or AMPI for Adaptive MPI, Multi-Processor Computing (MPC) from CEA, etc.

MPI

The Message Passing Interface, MPI, is the most famous runtime for distributed computing [Gro14, Gro15]. Several implementations exists from Intel MPI[2] (IMPI), MVAPICH[3] by the Ohio State University and OpenMP[4] combining several MPI work like Los Alamos MPI (LA-MPI). Those implementation follow the MPI standards 1.0, 2.0 or the latest, 3.0.

This runtime provides directs, collectives and asynchronous functions for process(es) to process(es) communication. A process can be a whole node or one or several cores on a processor.

Some MPI implementations offer a support for accelerators targeting directly their memory through the network without multiple copies on host memory. The data go through one GPU to the other through network and PCIe. This feature is used in our code in part 2 and 3.

For NVIDIA this technology is called GPUDirect RDMA and presented on figure 3.2.

In term of development MPI can be very efficient if use carefully. Indeed, the collectives communications such as *MPI_Alltoall*, *MPI_Allgather*, etc. can be a bottleneck when scaling up to thousands of processes. A specific care have to be taken in those implementation with privilege to asynchronous communications to hide computation than synchronous idle CPU time.

[1] http://www.openmp.org
[2] https://software.intel.com/en-us/intel-mpi-library
[3] http://mvapich.cse.ohio-state.edu/
[4] http://www.open-mpi.org

Charm++

Charm++[5] is an API for distributed programming developed by the University of Illinois Urbana-Champaign. It is asynchronous messages paradigm driven. In contrary of runtime like MPI that are synchronous but can handle asynchronous, charm++ is natively asynchronous. It is based on *chare object* that can be activated in response to messages from other *chare objects* with triggered actions and callbacks. The repartition of data to processors is completely done by the API, the user just have to define correctly the partition and functions of the program. Charm++ also provides a GPU manager implementing data movement, asynchronous kernel launch, callbacks, etc.

A perfect example can be the hydrodynamics N-body simulation code Charm++ N-body Gravity Solver, ChaNGa [JWG+10], implemented with charm++ and GPU support.

Legion

Legion[6] is a distributed runtime support by Stanford University, Los Alamos National Laboratory (LANL) and NVIDIA. This runtime is data-centered targeting distributed heterogeneous architectures. Data-centered runtime focuses to keep the data dependency and locality moving the tasks to the data and moving data only if requested. In this runtime the user defines data organization, partitions, privileges and coherency. Many aspect of the distribution and parallelization are then handle by the runtime itself.

The FleCSI runtime develops at LANL provide a template framework for multi-physics applications and is built on top of Legion. We give more details on this project and Legion on part 3.

3.3.3 Accelerators

In order to target accelerators like GPU, several specific API have been developed. At first they were targeted for matrix computation with OpenGL or DirectX through specific devices languages to change the first purpose of the graphic pipeline. The GPGPUs arriving forced an evolution and new dedicated language to appear.

CUDA

The Compute Device Unified Architecture is the API develop in C/C++ Fortran by NVIDIA to target its GPGPUs. The API provide high and low level functions. The driver API allows a fine grain control over the executions.

The CUDA compiler is called NVidia C Compiler, NVCC. It converts the device code into Parallel Thread eXecution, PTX, and rely to the C++ host compiler for host code. PTX is a pseudo assembly language translated by the GPU in binary code that is then execute. As the ISA is simpler than CPU ones and able the user to work directly in assembly for very fine grain optimizations.

As presented in figure 3.3, NVIDIA GPUs include many *Streaming Multiprocessors* (SM), each of which is composed of many *Streaming Processors* (SP). In the Kepler architecture, the SM new generation is called SMX. Grouped into *blocks*, *threads* execute *kernels* functions synchronously. Threads within a block can cooperate by sharing data on an SMX and synchronizing their execution to coordinate memory accesses; inside a block, the scheduler organizes *warps* of 32 threads which execute the instructions simultaneously. The blocks are distributed over the GPU SMXs to be executed independently.

In order to use data in a device kernel, it has to be first created on the CPU, allocated on the GPU and then transferred from the CPU to the GPU; after the kernel execution, the results have to be transferred back from the GPU to the CPU. GPUs consist of several memory categories,

[5] http://charmplusplus.org/
[6] http://legion.stanford.edu/

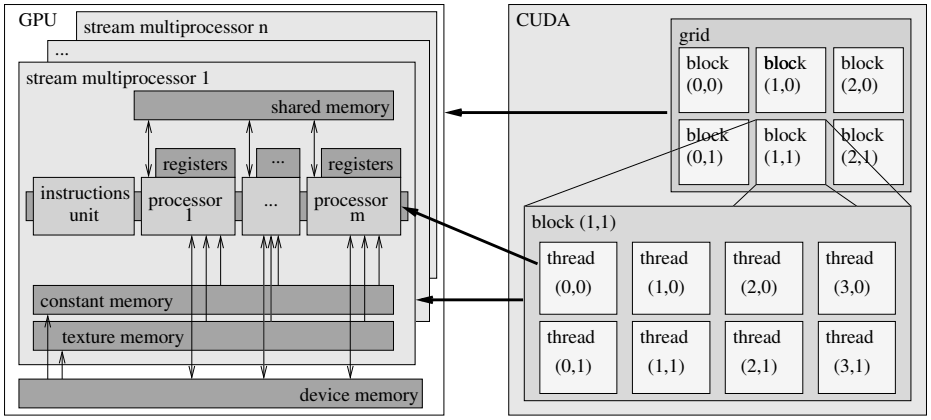

Figure 3.3: NVIDIA GPU and CUDA architecture overview

organized hierarchically and differing by size, bandwidth and latency. On the one hand, the device's main memory is relatively large but has a slow access time due to a huge latency. On the other hand, each SMX has a small amount of shared memory and L1 cache, accessible by its SPs, with faster access, and registers organized as an SP-local memory. SMXs also have a constant memory cache and a texture memory cache. Reaching optimal computing efficiency requires considerable effort while programming. Most of the global memory latency can then be hidden by the threads scheduler if there is enough computational effort to be executed while waiting for the global memory access to complete. Another way to hide this latency is to use streams to overlap kernel computation and memory load.

It is also important to note that branching instructions may break the threads synchronous execution inside a warp and thus affect the program efficiency. This is the reason why test-based applications, like combinatorial problems that are inherently irregular, are considered as bad candidates for GPU implementation.

Specific tools have been made for HPC in the NVIDIA GPGPUs.

Dynamic Parallelism This feature allow the GPU kernels to run other kernels themselves. This feature

Hyper-Q This technology enable several CPU threads to execute kernels on the same GPU simultaneously. This can help to reduce the synchronization time and idle time of CPU cores for specific applications.

NVIDIA GPU-Direct GPUs' memory and CPU ones are different and the Host much push the data on GPU before allowing it to compute. GPU-Direct allows direct transfers from GPU devices through the network. Usually implemented using MPI.

OpenCL

OpenCL is a multi-platform framework targeting a large part of nowadays architectures from processors to GPUs, FPGAs, etc. A large group of company already provided conform version of the OpenCL standard: IBM, Intel, NVIDIA, AMD, ARM, etc. This framework allows to produce a single code that can run in all the host or device architectures. It is quite similar to NVIDIA CUDA Driver API and based on kernels that are written and can be used in On-line/Off-line compilation meaning Just In Time (JIT) or not. The idea of OpenCL is great by rely on the vendors wrapper. Indeed, one may wonder, what is the level of work done by NVIDIA on its own CUDA framework compare to the one done to implement OpenCL standards? What is the advantage for NVIDIA GPU to be able to be replace by another component and compare on the same level? Those questions are still empty but many tests prove that OpenCL can be as comparable as CUDA but rarely better[KDH10, FVS11].

In this study most of the code had been developed using CUDA to have the best benefit

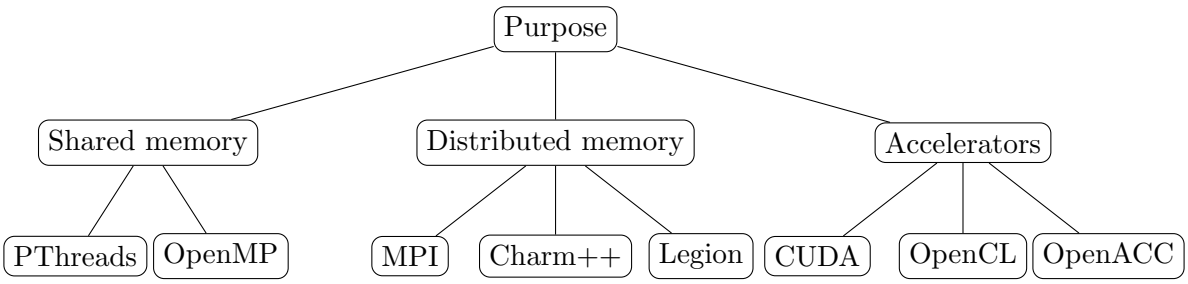

Figure 3.4: Runtimes, libraries, frameworks or APIs

of the NVIDIA GPUs present in the ROMEO Supercomputer. Also the long time partnership of the University of Reims Champagne-Ardenne and NVIDIA since 2003 allows us to exchange directly with the support and NVIDIA developers.

OpenACC

Open ACCelerators is a "user-driven directive-based performance-portable parallel programming model"[7] developed with Cray, AMD, NVIDIA, etc. This programming model propose, in a similar way to OpenMP, pragmas to define the loop parallelism and the device behavior. As the device memory is separated specific pragmas are use to define the memory movements. Research works[WSTaM12] tend to show that OpenACC performances are good regarding the time spend in the implementation itself compare to fine grain CUDA or OpenCL approaches. The little lack of performances can also be explain by the current contribution to companies in the wrapper for their architectures and devices.

The runtime, libraries, frameworks and APIs are summarized in figure 3.4. They are used in combination. The usual one is MPI for distribution, OpenMP and CUDA to target processors and GPUs.

3.4 Benchmarks

All those models, theory, hardware and software leads to better understanding and characterization of machines to produce better algorithm and solve bigger and harder problems. The question that arise is: how to know if a machine is better than another? We answer that question with FLOPS, IPC, OPS or just the frequency of the machine. The models like BSP or law's like Amdahl and Gustafson ones propose to find the best/worst case during the execution.

In real application the only way to really know what will be the behavior of a supercomputer is to try, test real code on it. This is call benchmarking. Several kind of benchmarks exists and target a specific application of supercomputers. We present here the most famous benchmarks of HPC and their specificities.

3.4.1 TOP500

The most famous benchmark is certainly the TOP500[8]. It gives the ranking of the 500 most powerful, known, supercomputers of the world as its name indicates. Since 1993 the organization assembles and maintains this list updated twice a year in June and November.

This benchmark is based on the LINPACK[DMS+94] a benchmark introduced by Jack J. Dongarra. This benchmark rely on solving dense system of linear equations. As specified in this document this benchmark is just one of the tools to define the performance of a supercomputer. It reflects "the performance of a dedicated system for solving a dense system of linear equations". This kind of benchmark is very regular in computation giving high results for FLOPS.

[7] https://www.openacc.org/
[8] http://www.top500.org

In 1965 the Intel co-fonder Gordon Moore made an observation[Pre00] on the evolution of devices. He pointed the fact that the number of transistors in a dense integrated circuit doubles approximately every eighteen months. This is know as the Moore's law. Looking at the last TOP500 figure presented on figure **??**, in the introduction of this document, we saw that nowadays machines does not fit in the law anymore. This is due to the size of transistor and the energy needed to reach more powerful machines. The Moore's law have been sustains by the arrival of many-cores architectures such as GPU or Xeon Phi. Tomorrow machines architectures will have to be based on hybrid with more paradigms and tools to take part of massive parallelism.

3.4.2 Green500

In conjunction of the TOP500, the Green500[9] focus on the energy consumption of supercomputers. The scale is based on FLOPS per watts [FC07]. Indeed the energy wall is the main limitation for next generation and exascale supercomputers. In the last list, November 2017, the TOP3 machines are accelerated with PEZY-SC many-core devices. The TOP20 supercomputers are all equipped with many-cores architectures: 5 with PEZY-SC, 14 with NVIDIA P100 and 1 with the Sunway many-core devices. This show clearly that the nowadays energy efficient solutions resides in many-core architecture and more than that, hybrid supercomputers.

3.4.3 GRAPH500

The GRAPH500[10] benchmark[MWBA10] focus on irregular memory accesses, and communications. The authors try to find ways to face the futures large-scale large-data problems and data-driven analysis. This can be see as a complement of the TOP500 for data intensive applications. The aim is to generate a huge graph to fill all the maximum memory on the machine and then operate either:

BFS: A Breadth-First Search which is an algorithm starting from a root and exploring recursively all the neighbors. This requires a lot of irregular communications and memory accesses.

SSSP: A Single Source Shortest Path which is an algorithm searching the shortest path from one node to the others. Like the BFS it has an irregular behavior but also requires to keep more data during the computation.

This benchmark will be detailed in Part II Chapter II in our benchmark suite.

3.4.4 HPCG

The High Performance Conjugate Gradient benchmark[11] is a new benchmark created in 2015 and presented for the first time at SuperComputing 2015. The last list, November 2017 contains 115 supercomputers ranked. The list also offer to compare the results of Linpack compared to Conjugate Gradient. This benchmark is a first implementation of having both computation and communications aspects of HPC in the same test.

This benchmark is presented and features:
- Sparse matrix-vector multiplication;
- Vector updates;
- Global dot products;
- Local symmetric Gauss-Seidel smoother;
- Sparse triangular solve (as part of the Gauss-Seidel smoother);

[9] https://www.top500.org/green500/
[10] https://www.graph500.org/
[11] http://www.hpcg-benchmark.org/

- Driven by multigrid preconditioned conjugate gradient algorithm that exercises the key kernels on a nested set of coarse grids;
- Reference implementation is written in C++ with MPI and OpenMP support.

3.5 Conclusion

In this chapter we presented the most used software tools for HPC. From inside node with shared memory paradigms, accelerators and distributed memory using message passing runtime with asynchronous or synchronous behavior.

The tools to target accelerators architectures tend to be less architecture dependent with API like OpenMP, OpenCL or OpenACC targeting all the machines architectures. Unfortunately the vendor themselves have to be involve to provide the best wrapper for their architecture. In the mean time vendor dependent API like CUDA for NVIDIA seems to deliver the best performances.

We show through the different benchmark that hybrid architecture start to have their place even in computation heavy and communication heavy context. They are the opportunity to reach exascale supercomputers in horizon 2020.

Bibliography

[Cha08] Barbara Chapman. *Using OpenMP : portable shared memory parallel programming.* MIT Press, Cambridge, Mass, 2008.

[DMS+94] Jack J Dongarra, Hans W Meuer, Erich Strohmaier, et al. Top500 supercomputer sites, 1994.

[FC07] Wu-chun Feng and Kirk Cameron. The green500 list: Encouraging sustainable supercomputing. *Computer*, 40(12), 2007.

[FVS11] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE, 2011.

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM, 1978.

[Gro14] William Gropp. *Using MPI : portable parallel programming with the Message-Passing-Interface.* The MIT Press, Cambridge, MA, 2014.

[Gro15] William Gropp. *Using advanced MPI : modern features of the Message-Passing-Interface.* The MIT Press, Cambridge, MA, 2015.

[JWG+10] Pritish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V Kalé, and Thomas R Quinn. Scaling hierarchical n-body simulations on gpu clusters. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[KDH10] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*, 2010.

[MWBA10] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 19:45–74, 2010.

[Pre00] I Present. Cramming more components onto integrated circuits. *Readings in computer architecture*, 56, 2000.

[Sup17] Bronis Supinski. *Scaling OpenMP for Exascale Performance and Portability : 13th International Workshop on OpenMP, IWOMP 2017, Stony Brook, NY, USA, September 20-22, 2017, Proceedings.* Springer International Publishing, Cham, 2017.

[Val90] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[WSTaM12] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc—first experiences with real-world applications. In *European Conference on Parallel Processing*, pages 859–870. Springer, 2012.