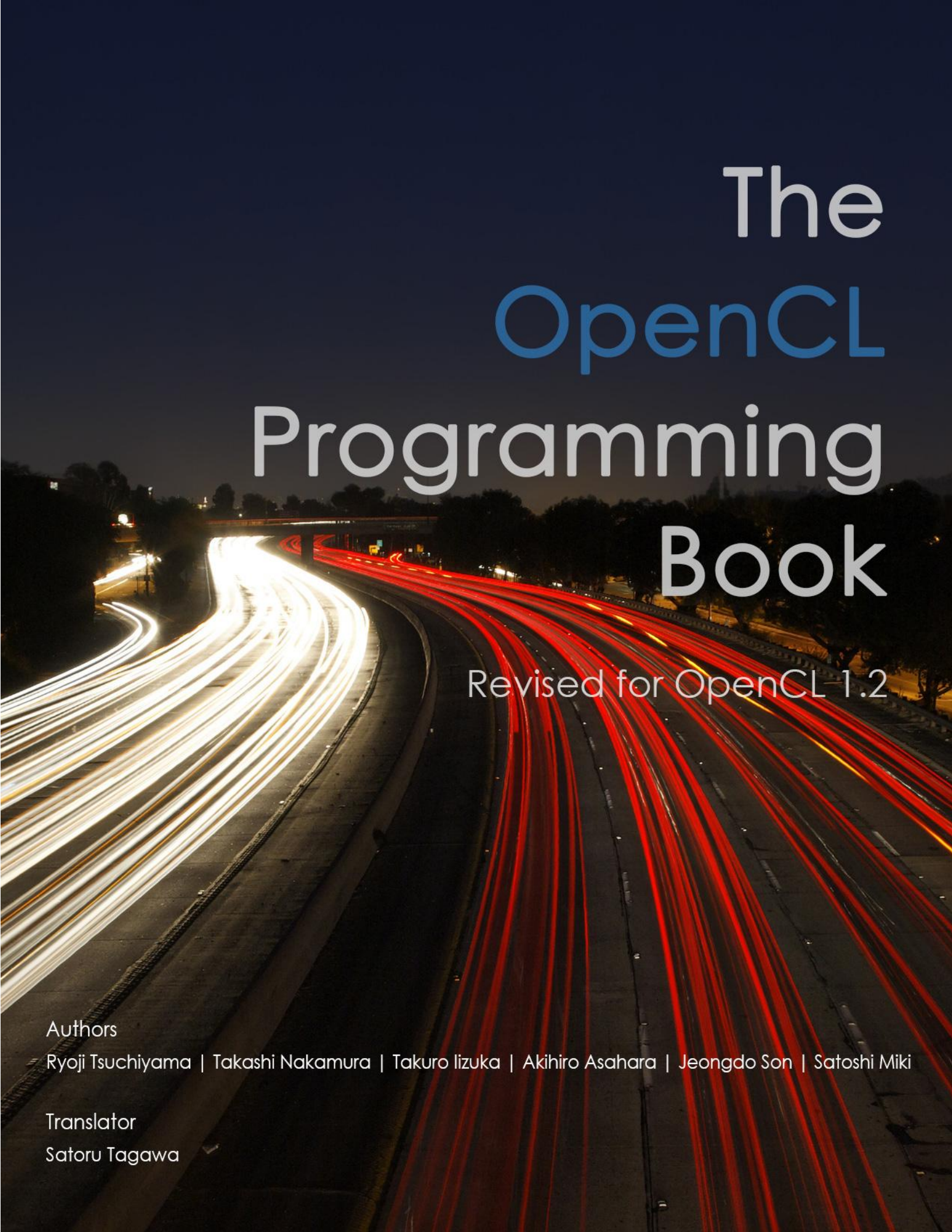


The OpenCL Programming Book

Revised for OpenCL 1.2

Authors

Ryoji Tsuchiyama | Takashi Nakamura | Takuro Iizuka | Akihiro Asahara | Jeongdo Son | Satoshi Miki

Translator

Satoru Tagawa

Based on The Impress Japan Corporation publication "The OpenCL Programming Book (OpenCL Nyumon) " by Fixstars Corporation.

Foreword

It is hard to believe that 3 years has already passed since the OpenCL 1.0 specification was announced. More and more multi-core processors are being introduced by the day, with no signs of dwindling momentum. NVIDIA's Fermi and Tegra, AMD's Fusion and Graphics Core Next, Intel's Sandy Bridge and Many Integrated Core (MIC), and IBM's POWER7 are some examples. Even smaller vendors such as Tilera and Adapteva have entered the market with their energy-efficient, high-performance multi-core processors. As far as software development is concerned, however, despite the research efforts in automatic parallelization and virtualization, no technology is in the horizon that will prevent programmers from having to write architecture-dependent, explicitly parallelized code. Although many chip vendors have released parallel-programming platforms independently, such as NVIDIA's CUDA, AMD's APP SDK, and Intel's ArBB, they also support OpenCL for cross-platform portability, leading us to believe that OpenCL is slowly but steadily becoming the standard for multi-core programming.

In the world of software development, the demand for OpenCL projects is increasing by the day. Our company, Fixstars Corporations, specializes in multi-core programming, and our programmers are commonly left with no other choice than to choose OpenCL since multi-platform execution is often mandatory. This all thanks to the fact that many chip vendors have decided to support OpenCL for their hardware devices. As long as OpenCL is supported, any new processor can be used without having to change the code, or at most with minimal changes. Software asset in a highly portable language increases competitiveness in the present and is a risk hedge for the future.

So, is OpenCL going be the standard platform for the ongoing multi-core era? Perhaps it is meaningless to predict what the future holds in this rapidly-advancing field, but as long as processor vendors continue with their innovations, and as long as an auto-parallelization compiler that optimizes any code to any architecture does not become a reality, the most probable option for a standard to conform to will be one from a highly reputable standard organization like Khronos. In other words, what else other than OpenCL?

January 2012
CEO, Fixstars Corporation
Satoshi Miki

Foreword

OpenCL is an exciting new open standard that is bringing the power of programming complex multi-processor systems to a much wider audience than ever before. Khronos is fully committed not only to creating the specification but also fostering a rich and diverse OpenCL ecosystem. A vital piece of any living standard is enabling the industry to understand it truly, letting it tap into the full potential power of the technology.

Fixstars is a skilled OpenCL practitioner and is ideally qualified to create state-of-the-art OpenCL educational materials. I recommend wholeheartedly this book to anyone seeking to understand and start using the amazing power of OpenCL.

Neil Trevett

President, The Khronos Group

Who Should Read this Book?

The book is intended for those interested in the new framework known as OpenCL. While we do not assume any preexisting knowledge of parallel programming, since we introduce most of what you need to know in Chapter 1, we do however assume that the reader has a good grasp of the C language. Those who are already experts in parallel programming can start at Chapter 2 and dive straight into the new world of OpenCL.

The official reference manual for OpenCL can be found online on Khronos' website at: http://www.khronos.org/registry/cl/

Sample programs in this book can be downloaded at http://www.fixstars.com/en/opencl/book

About the Authors

Fixstars Corporation

Fixstars Corporation is a software company focusing on the porting and optimization of programs for multi-core processors. Fixstars offers a complete solution to unleash the capability of multi-core processors for tasks that require high computing power such as image processing for medical and factory automation, CG rendering, and financial simulations. Fixstars is also currently developing "M-Cubed", which is the next-generation parallel computing platform for programming in Multi-core, Multi-node, Multi-architecture environments. For more information, please visit us at http://www.fixstars.com/.

Ryoji Tsuchiyama

I am currently involved in developing a parallel computing platform, in addition to optimizing CG renderers for multi-core processors at Fixstars. Since my first encounter with a computer (the PC-8001) as a child, my fascination with computers have gotten me through a master's degree with an emphasis on distributed computing, and to a career where I orchestrate a bunch of selfish CPUs and DSPs to work together in harmony with each other. Recently, I have been dealing with distributed computing on x86 clusters, as well as parallel processing using accelerators such as the Cell/B.E. and GPUs.

Takashi Nakamura

As a member of Fixstars' Multi-core Lab, I am involved with the research and development in the field of parallel computing frameworks. I enjoy long walks on the beach, writing automatic parallelization compilers, changing data structures to increase memory access efficiency, and filling in pipelines by hand. I get most of my vitamins and minerals from reading datasheets and manuals of chips from different vendors. I particularly enjoy glancing at the pages on latency/throughput, and being paid to do so is just an added bonus.

Takuro Iizuka

I am currently involved in developing a parallel computing platform at Fixstars. My current hobbies include writing a compiler in ML for my own amazingly incredible programming language that I am, and most likely will always be, in the process of perfecting, and messing with OCaml + LLVM. I can often be found drifting away in thought fantasizing about what a dream world it would be if

OpenCL kernels could be written in a much higher-level language than the OpenCL C. But alas, one step at a time, and yes, we must become the change we wish to see in the world.

Akihiro Asahara

I am currently in the midst of undertaking business development for Fixstars' products in Japan as well as in the Silicon Valley. As we await the arrival of numerous new hardware architectures to be introduced from major chip vendors this year (2012), my instincts are telling me that in time, OpenCL will become available on tablets and smart phones, and if WebCL picks up for real, the Silicon Valley geeks doing SNS are going to be all over this! The future holds so many exciting possibilities, and we just have to catch that wave!

Jeongdo Son

I am mainly involved in acceleration and optimization of embedded software here at Fixstars. The embedded world is full of complex heterogeneous architectures, each with its own unique development environment, and having to rewrite code for each processor from scratch is a major pain. Although unfortunately not covered in the scope of this book, it would save a lot of headaches for embedded engineers if more devices started supporting the OpenCL Embedded Profile.

Satoshi Miki

As one of the founding members, I currently serve as the CEO here at Fixstars. In 2004, I was convinced that the software developing process would have to change drastically for the multi-core era, and decided to change the direction of the company to focus on the research and development of multi-core programs. Since then, I have been working hard to help spread multi-core technology that is already becoming the standard. My debut as a programmer was at the age of 25, and as I enter my 10^{th} year as CEO this year, I am back to writing code regularly with my employees, which I am finding is a good way to keep myself humble and to keep learning.

Introduction to Parallelization

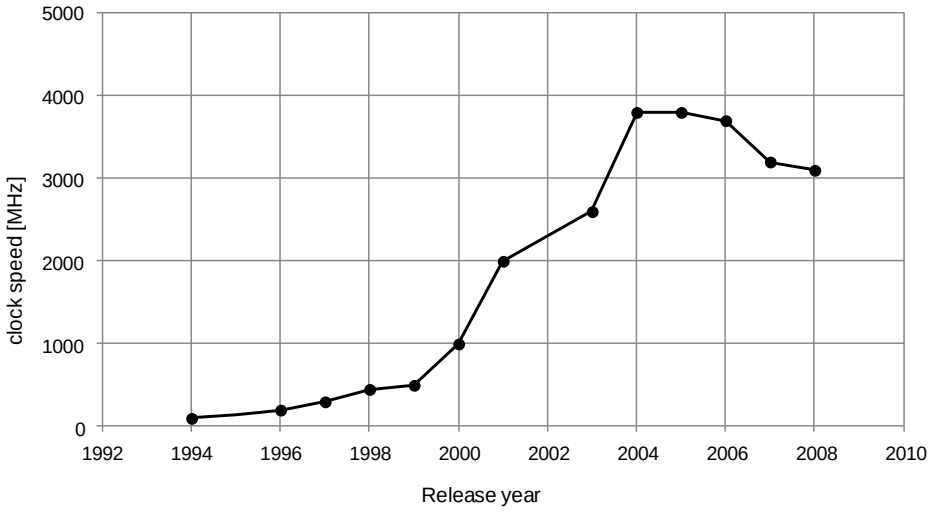
This chapter introduces the basic concepts of parallel programming from both the hardware and the software perspectives, leading up to the introduction of OpenCL in the chapters to follow.

Why Parallel

In the good old days, software speedup was achieved by using a CPU with a higher clock speed, which increased significantly with each passing year. However, around 2004 when Intel's CPU clock speed reached 4GHz, the increase in power consumption and heat dissipation formed what is now known as the "power wall", which effectively caused the CPU clock speed to level off (**Figure 1.1**[1]). The processor vendors were forced to give up their efforts in increasing the clock speed, and instead to adopt a new method of increasing the number of cores within the processor. Since the CPU clock speed has either remained the same or even slowed down in order to save the power, old software designed to run on a single processor does not get any faster just by upgrading the CPU to the newest model. To get the most out of the current processors, the software must be designed to take full advantage of the multiple cores and perform execution in parallel.

Today, dual-core CPUs are commonplace even for the basic consumer laptops. This shows that parallel processing is not just useful for performing advanced computations, but that it is becoming common in various applications.

Figure 1.1: Intel CPU Clock Speed by Year



Parallel Computing (Hardware)

First of all, what exactly is "parallel computing"? Wikipedia defines it as 'a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel")'. [1]
Many different hardware architectures exist today that can perform a single task using multiple processors. Some examples, in order of decreasing scale are:

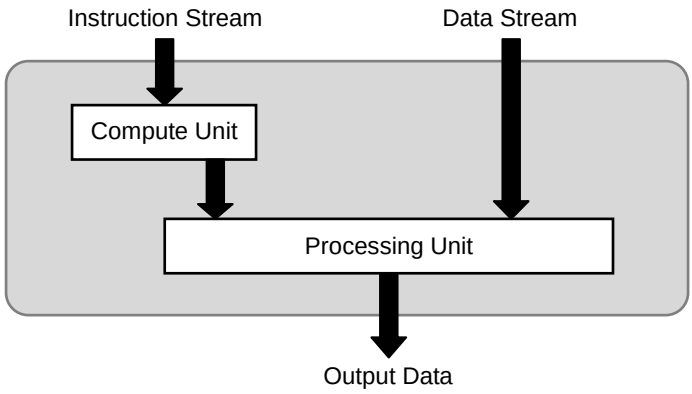
Grid computing - a combination of computer resources from multiple administrative domains applied to a common task.
MPP (Massively Parallel Processor) systems - known as the supercomputer architecture.
Cluster server system - a network of general-purpose computers.
SMP (Symmetric Multiprocessing) system - identical processors (grouped in powers of 2) connected together to act as one unit.
Multi-core processor - a single chip with numerous computing cores.

Flynn's Taxonomy

Flynn's Taxonomy is a classification of computer architectures proposed by Michael J. Flynn [2]. It is based on the concurrency of instruction and data streams available in the architecture. An instruction stream is the set of instructions that makes up a process, and a data stream is the set of data to be processed.

1. Single Instruction, Single Data stream (SISD)
SISD system is a sequential system where one instruction stream process one data stream. The pre-2004 PCs were of this type of system.

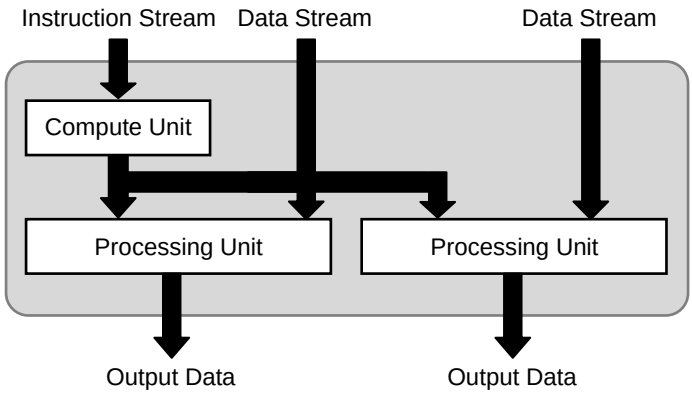
Figure 1.2: SISD architecture



2. Single Instruction, Multiple Data streams (SIMD)

One instruction is broadcasted across many compute units, where each unit processes the same instruction on different data. The vector processor, a type of supercomputer, is an example of this type of architecture. Recently, various micro-processors include SIMD processors. For example, SSE/AVX instructions on Intel CPUs, and SPE instructions on Cell Broadband Engines perform SIMD operations.

Figure 1.3: SIMD architecture



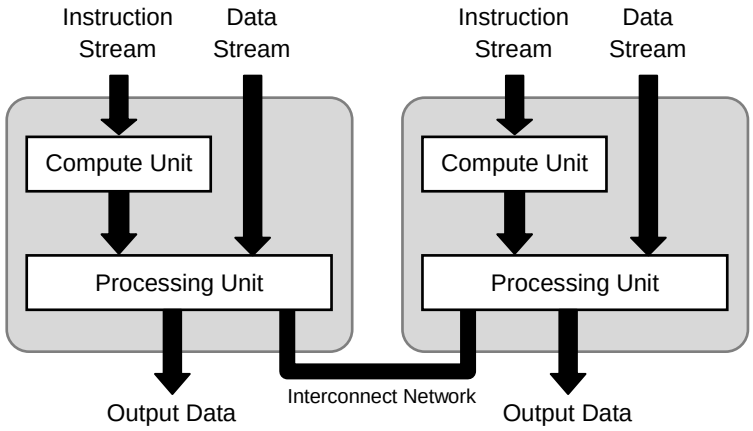
3. Multiple Instruction, Single Data stream (MISD)

Multiple instruction streams process a single data stream. Very few systems fit within this category, with the exception of fault tolerant systems.

4. Multiple Instruction, Multiple Data streams (MIMD)

Multiple processing units each process multiple data streams using multiple instruction streams.

Figure 1.4: MIMD architecture



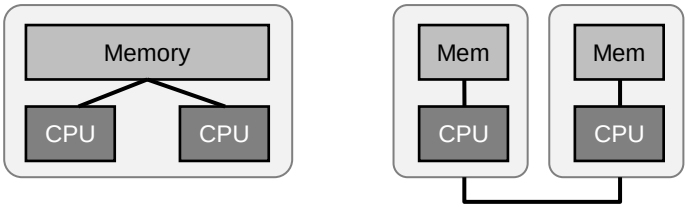
Using this classification scheme, most parallel computing hardware architectures, such as the SMP and cluster systems, fall within the MIMD category. For this reason, the MIMD architecture is further categorized by memory types.

The two main memory types used in parallel computing system are shared memory and distributed memory types. In shared memory type systems, each CPU that makes up the system is allowed access to the same memory space. In distributed memory type systems, each CPU that makes up the system uses a unique memory space.

Different memory types result in different data access methods. If each CPU is running a process, a system with shared memory type allows the two processes to communicate via read/write operations to the shared memory space. On the other hand, a system with distributed memory type requires data transfers to be explicitly performed by the user, since the two memory spaces are managed by two OS's.

The next sections explore the two parallel systems in detail.

Figure 1.5: The two widely used parallel processing system types



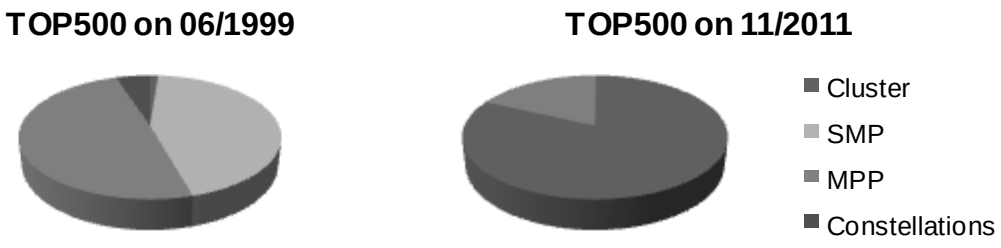
Distributed Memory Type

Tasks that take too long using one computer can be broken up to be performed in parallel using a network of processors. This is known as a cluster server system, which is perhaps the most commonly-seen distributed memory type system. This type of computing has been used for years in the HPC (High Performance Computing) field, which performs tasks such as large-scale simulations.

MPP (Massively Parallel Processor) system is also another commonly-seen distributed memory type system. It connects numerous nodes, which are made up of CPUs, memory, and a network port, connected via a specialized fast network. NEC's Earth Simulator and IBM's Blue Gene are some of the known MPP systems.

The main difference between a cluster system and a MPP system lies in the fact that a cluster does not use specialized hardware, giving it a much higher cost performance than the MPP systems. Due to this reason, many MPP systems, which used to be the leading supercomputer type, have been replaced by cluster systems. According to the TOP500 Supercomputer Sites [3], of the top 500 supercomputers as of June 2009, 17.6% are MPP systems, while 82% are cluster systems.

Figure 1.6: Architecture statistics from TOP500 Supercomputer Sites



One problem with cluster systems is the slow data transfer rates between the processors. This is due to the fact that these transfers occur via an external network. Some recent external networks include Myrinet, Infiniband, and 10Gbit Ethernet, which have become significantly faster compared to the traditional Gigabit Ethernet. However, even with these external networks, the transfer rates are still at least an order of magnitude slower than local memory access of each processor.

For the reason given above, cluster systems are suited for parallel algorithms where the CPUs do not have to communicate with each other too often. These types of algorithms are said to be "coarse-grained parallel". These algorithms are used often in simulations where many trials are required, but these trials do not depend on each other. An example of this is risk simulation used for the development of derivative products in the field of finance.

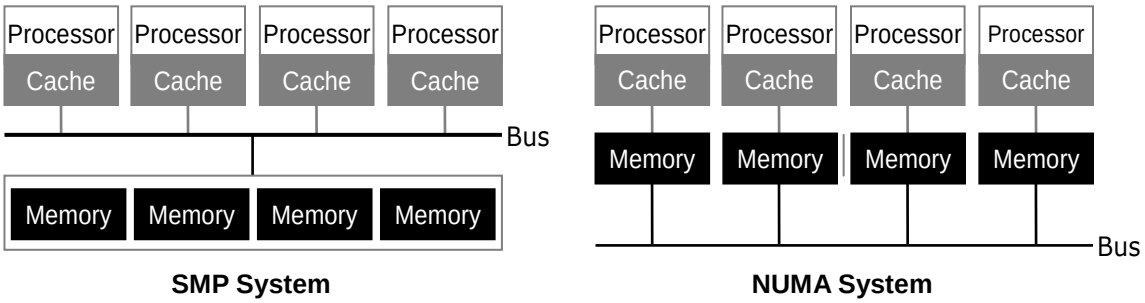
Shared Memory Type

In shared memory type systems, all processors share the same address space, allowing these processors to communicate with each other through read/writes to shared memory. Since data transfers/collections are unnecessary, this results in a much simpler system from the software perspective.

An example of a shared memory type system is the Symmetric Multiprocessing (SMP) system (**Figure 1.7**, left/[4]). The Intel Multiprocessor Specification Version 1.0 released back in 1994 describes the method for using x86 processors in a multi-processor configuration. 2-way workstations (where up to 2 CPUs can be installed) are commonly seen today [5]. However, increasing the number of processors naturally increases the number of accesses to the memory, making the bandwidth

between the processors and the shared memory a bottleneck. SMP systems are thus not scalable, and only effective up to a certain number of processors. Although 2-way servers are inexpensive and common, 32-way or 64-way SMP servers require specialized hardware, which can become expensive.

Figure 1.7: SMP and NUMA

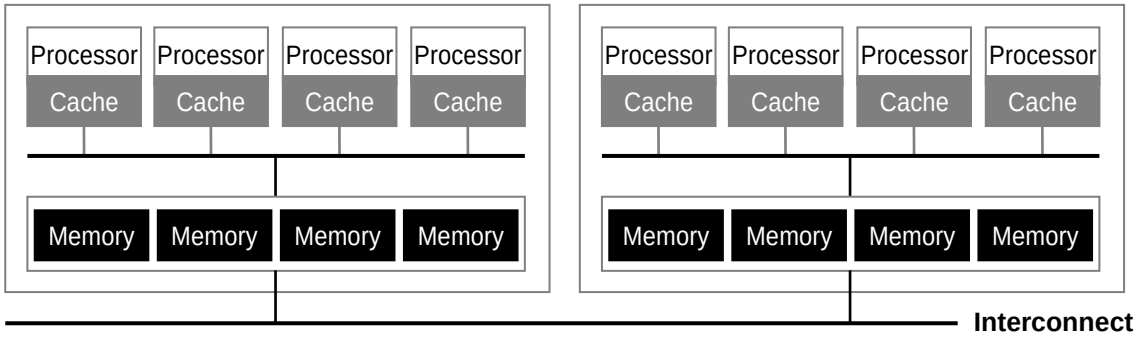


Another example of a shared memory type system is the Non-Uniform Memory Access (NUMA) system. The main difference from a SMP system is that the physical distance between the processor and the memory changes the access speeds. By prioritizing the usage of physically closer memory (local memory) over more distant memory (remote memory), the bottleneck in SMP systems can be minimized. To reduce the access cost to remote memory, a processor cache and a specialized hardware making sure the cache data is coherent has been added, and this system is known as a Cash Coherent NUMA (cc-NUMA).

Server CPUs such as AMD Opteron and Intel Xeon 5500 Series contain on-chip memory controllers. Thus, when these are used in multi-processor configurations, they become a NUMA system. The hardware to verify cache coherency is embedded into the CPU. In addition, NUMA replaces the Front Side Bus (FSB), which is a bus that connects multiple CPUs as well as other chipsets together, with an interconnect port that uses a point-to-point protocol. These ports are called Quick Path Interconnect (QPI) by Intel, and Hyper Transport by AMD.

Now that the basic concepts of SMP and NUMA are covered, let us look at an interesting fact of typical x86 server products. Dual-core and quad-core processors are SMP systems, since the processor cores all access the same memory space. Networking these multi-core processors actually end up creating a NUMA system. In other words, the mainstream 2-way or more x86 server products are "NUMA systems made from SMP systems" (**Figure 1.8**).

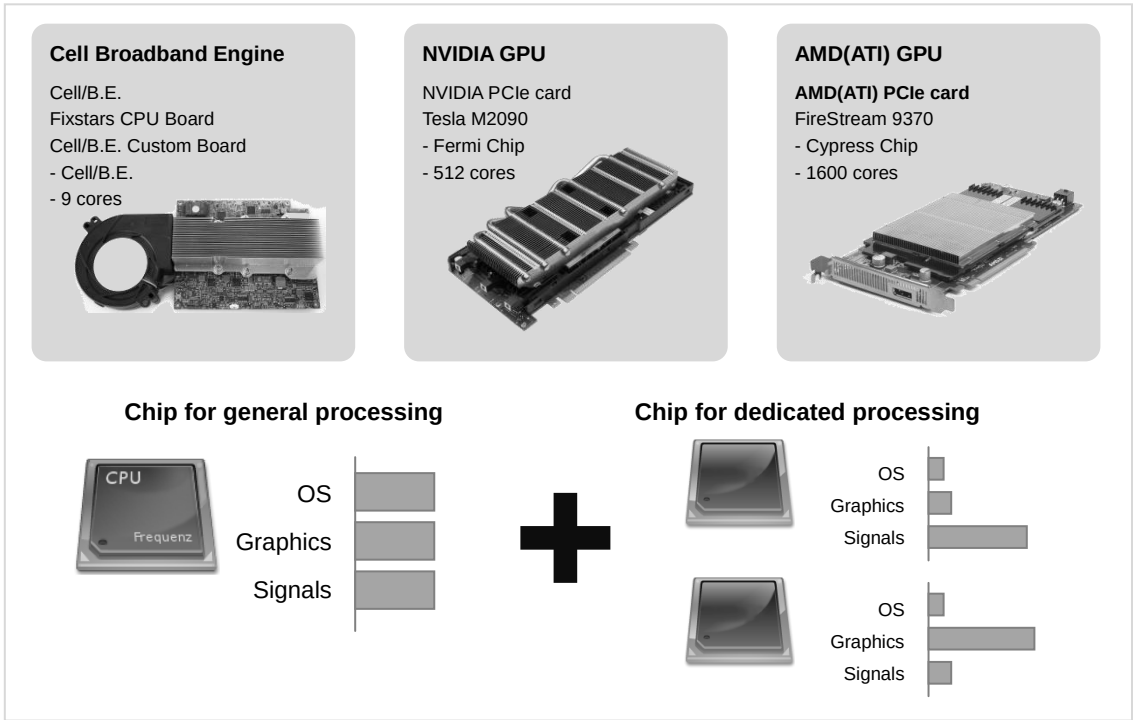
Figure 1.8: Typical 2-way x86 server



Accelerator

The parallel processing systems discussed in the previous sections are all made by connecting generic CPUs. Although this is an intuitive solution, another approach is to use as co-processor different hardware more suited for performing certain tasks. The non-CPU hardware in this configuration is known as an "accelerator".

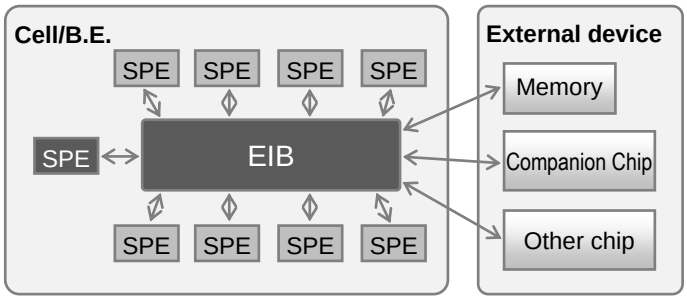
Figure 1.9: Example of accelerators



Some popular accelerators include the Cell Broadband Engine (Cell/B.E.) and GPUs. Accelerators typically contain cores optimized for performing floating-point arithmetic (or fixed-point arithmetic for some DSPs). Since these cores are relatively simple and thus do not take much space on the chip, numerous cores are typically available.

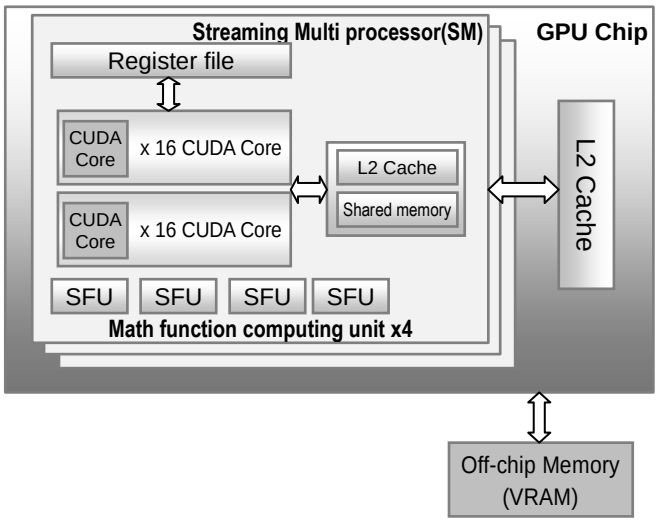
For example, the Cell/B.E. contains 1 PowerPC Processor Element (PPE), which is suited for processes requiring frequent thread switching, and 8 Synergistic Processor Elements (SPE) which are cores optimized for floating-point arithmetic. These 9 cores are connected using a high-speed bus called the Element Interconnect Bus (EIB), and placed on a single chip.

Figure 1.10: Architecture of the Cell Broadband Engine



Another example is NVIDIA's Tesla M2000, which is based on a GPU architecture known as Fermi. This chip contains 16 sets of 32-core Streaming Processors (SM), for a total of 512 cores on one chip.

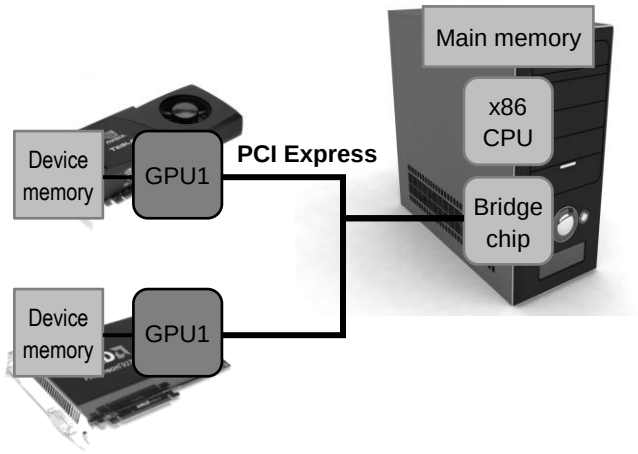
Figure 1.11: NVIDIA Fermi architecture



In recent years, these accelerators have been attracting a lot of attention. This is mainly due to the fact that the floating-point arithmetic capability of generic CPUs has leveled off at around 10 GFLOPS, while the Cell/B.E. and GPUs can perform between 100 GFLOPS and 1 TFLOPS for a relatively low price. They are also "greener", making them a better option than cluster server systems, as many factories and research labs are trying to cut back on power usage.

For example, the circuit board and semiconductor fields use automatic visual inspection. The tests increase in number and complexity every year, requiring faster image processing so that the rate of production is not compromised. Medical imaging devices such as ultrasonic diagnosing devices and CT scanners are producing every year higher and higher quality 2-D images as output, and generic CPUs are not capable of processing the images in a practical amount of time. Using a cluster server for these tasks requires a vast amount of space, as well as high power usage. Thus, the accelerators provide a portable and energy-efficient alternative to the cluster. These accelerators are typically used in conjunction with generic CPUs, creating what is known as a "hybrid system".

Figure 1.12: Hybrid system



In summary, an accelerator allows for a low-cost, low-powered, and high-performance system. However, the transfer speed between the host CPU and the accelerator can become a bottleneck, making it unfit for applications requiring frequent I/O operations. Thus, the decision to use a hybrid system, as well as what type of hybrid system, needs to be made wisely.

OpenCL, in brief, is a development framework to write applications that runs on these "hybrid systems".

Parallel Computing (Software)

Up until this section, hardware architectures that involve performing one task using numerous processors have been the focus. This section will focus on how to program in parallel on those discussed hardware architectures.

Sequential vs Parallel Processing

Let us take a look at the pseudocode below.

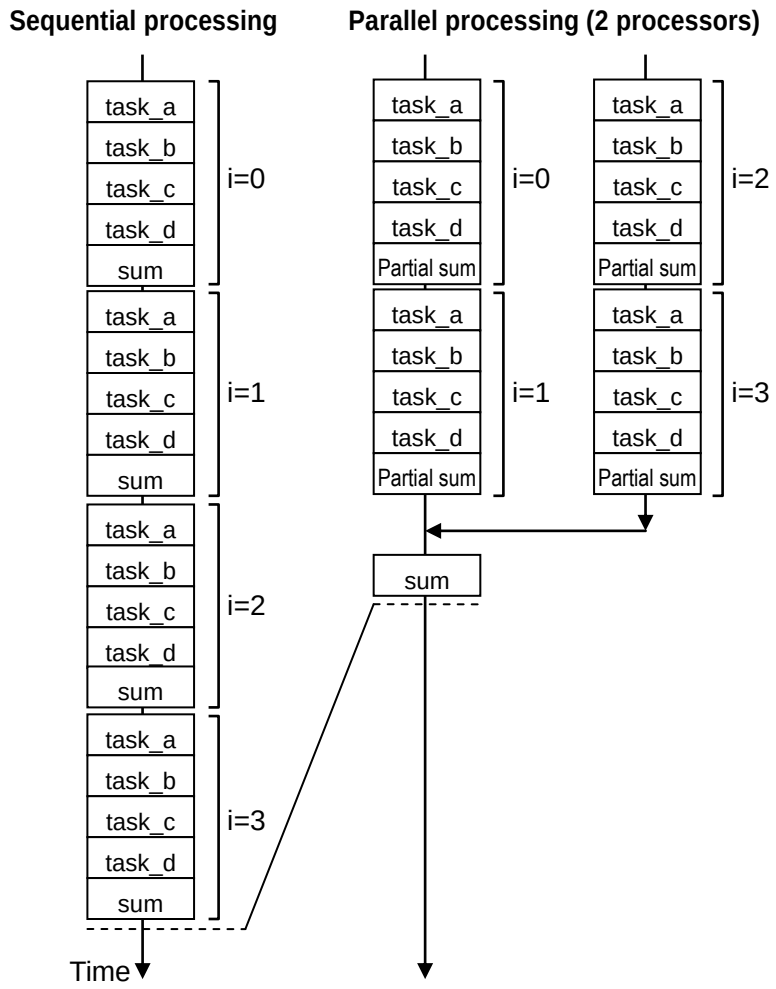
List 1.1: Pseudocode for parallel processing

```
001: for(i=0;i<N;i++){
002:        resultA    = task_a(i);
003:        resultB    = task_b(i);
004:        resultC    = task_c(i);
005:        resultD    = task_d(i);
006:        resultAll  += resultA + resultB + resultC + resultD;
007: }
```

Executing the code above on a single-core, single-CPU processor (SISD system), the 4 tasks would be run sequentially in the order task_a, task_b, task_c, task_d, and the returned values then summed up. This is repeated N times, incrementing i after each iteration. This type of method is called "sequential processing". The process of running the code with N=4 is shown on the left hand side of **Figure 1.3**.

This type of code can benefit from parallelization. If this code is compiled without adding any options for optimization on a dual-core system, the program would still run sequentially on only one core. In this scenario, since the other core has nothing to do, it idles. This is clearly inefficient, so the intuitive thing to do here is to split the task into 2 subtasks, and run each subtasks on one core each. This is the basis of parallel programming. As shown in **Figure 1.13**, the task is split up into 2 such that each subtask runs the loop N/2 times, and then performs the sum at the end.

Figure 1.13: Parallel processing example



For actual implementation of parallel programs, it is necessary to follow the following steps.

1. Analyze the dependencies within the data structures or within the processes, etc, in order to decide which sections can be executed in parallel.
2. Decide on the best algorithm to execute the code over multiple processors.
3. Rewrite the code using frameworks such as Message Passing Interface (MPI), OpenMP, or OpenCL.

In the past, these skills were required only by a handful of engineers, but since multi-core processors are becoming more common, the use of these distributed processing techniques is becoming

necessary more often. The following sections will introduce basic concepts required to implement parallel processing.

Where to Parallelize?

During the planning phase of parallel programs, certain existing laws must be taken into account. The first law states that if a program spends y% of the time running code that cannot be executed in parallel, the expected speedup from parallelizing is at best a 1/y fold improvement. The law can be proven as follows. Let Ts represent the time required to run the portion of the code that cannot be parallelized, and let Tp represent the time required to run the portion of the code that can benefit from parallelization. Running the program on 1 processor, the processing time is:

```
T(1) = Ts + Tp
```

And the processing time when using N processors is:

```
T(N) = Ts + Tp/N
```

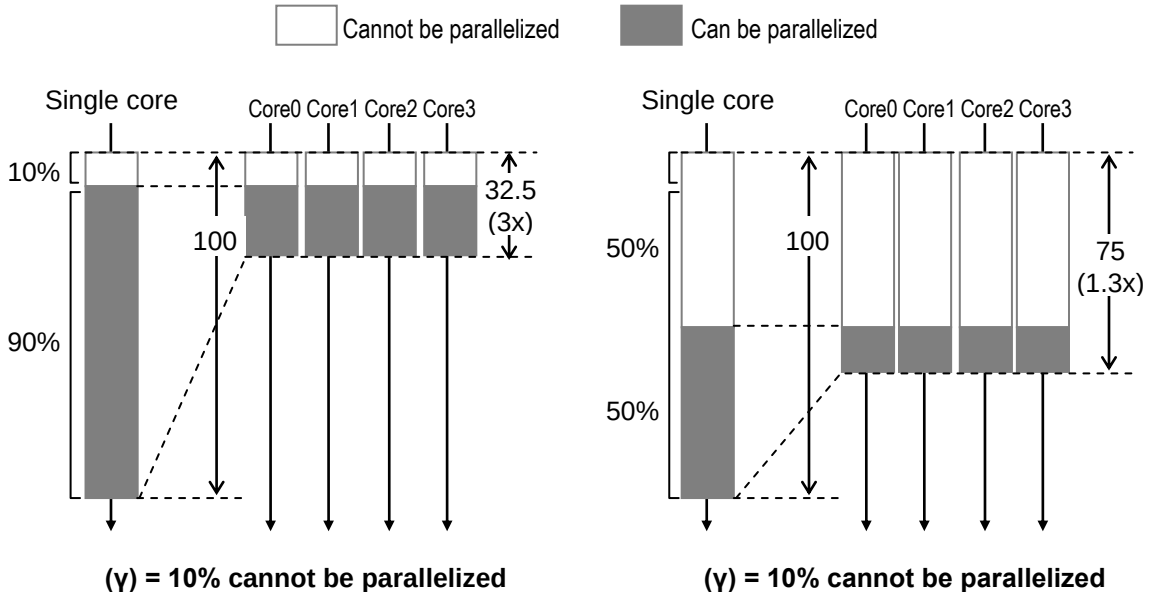
Using y, the proportion of time spent running code that cannot be parallelized, the speedup S achieved is:

```
S = T(1)/T(N) = T(1)/(Ts + Tp/N) = 1/(y+(1-y)/N)
```

Taking the limit as N goes to infinity, the highest speedup that can be achieved is S=1/y. This law is known as Amdahl's law [6].

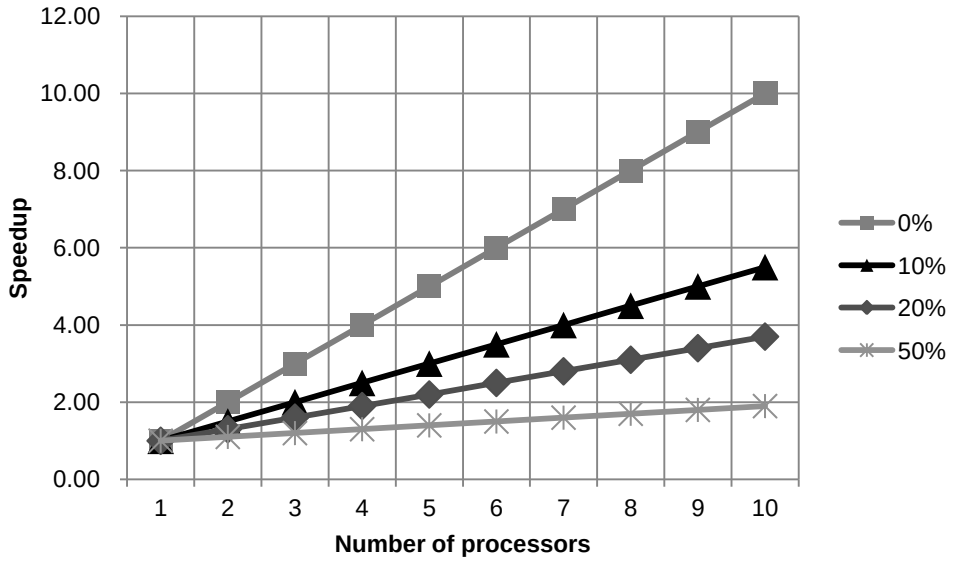
For a more concrete example, assume that some sequential code were being rewritten to run on a quad-core CPU. Ideally, a 4-times speedup would be achieved. However, as the law states, the speedup is limited by the portion of the code that must be run sequentially. **Figure 1.4** shows the 2 cases where the proportion of the code that cannot be run in parallel (y) is 10% and 50%. Even without taking overhead into account, the figure shows a difference of 3x and 1.3x speedup depending on y.

Figure 1.14: Example of Amdahl's law



This problem becomes striking as the number of processors increases. For a common 2-way server that uses Intel's Xeon 5500 Series CPUs which supports hyper-threading, the OS sees 16 cores. GPUs such as NVIDIA's Tesla can have more than 200 cores. **Figure 1.15** shows the speedup achieved as a function of the sequentially processed percentage y and the number of cores. The graph clearly shows the importance of reducing the amount of sequentially processed portions, especially as the number of cores is increased. This also implies that the effort used for parallelizing a portion of the code that does not take up a good portion of the whole process may be in vain. In summary, it is more important to reduce serially processed portions rather than parallelizing a small chunk of the code.

Figure 1.15: Result of scaling for different percentage of tasks that cannot be parallelized



While Amdahl's law gives a rather pessimistic impression of parallel processing, Gustafson's law provides a more optimistic view. This law states that as the program size increases, the fraction of the program that can run in parallel also increases, thereby decreasing the portion that must be performed sequentially. Recall the previously stated equation:

```
T(N) = Ts + Tp/N
```

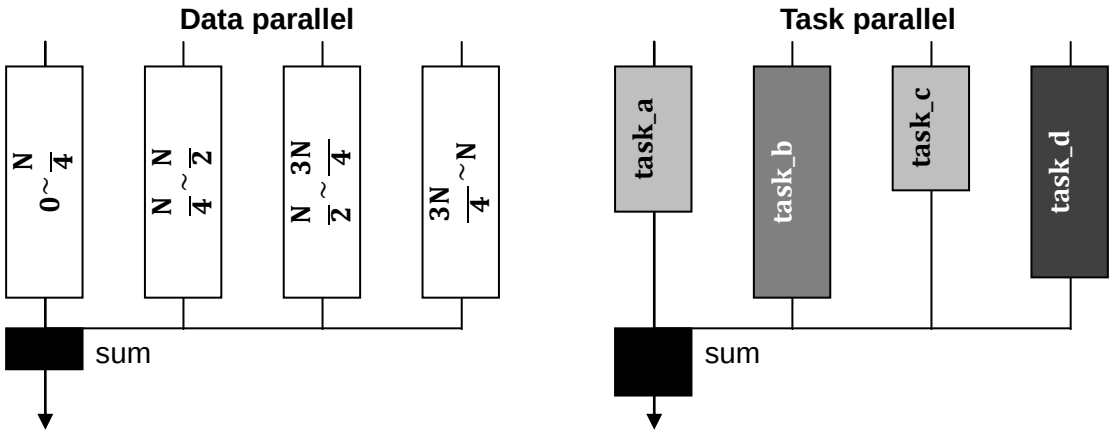
Gustafson states that of the changes in T(N), Ts is directly proportional to the program size, while Tp grows faster than Ts. For example, assume a program where a portion that must be run sequentially is limited to initialization and closing processes, and all other processes can be performed in parallel. By increasing the amount of data to process by the program, it is apparent that Gustafson's law holds true. In other words, Gustafson's law shows that in order to efficiently execute code over multiple processors, large-scale processing must take place. Development of parallel programming requires close attention to these 2 laws.

Types of Parallelism

After scrutinizing the algorithm and deciding where to parallelize, the next step is to decide on the type of parallelism to use.

Parallel processing requires splitting up the data to be handled, and/or the process itself. Please refer back to the code of **List 1.1**. This time, assume a quad-core CPU running 4 processes simultaneously. An intuitive approach is to let each processor perform N/4 iterations of the loop, but since there are 4 tasks within the loop, it also makes just as much sense to run each of these tasks in each processor. The former method is called "data parallel", while the latter is called "task parallel" (**Figure 1.16**).

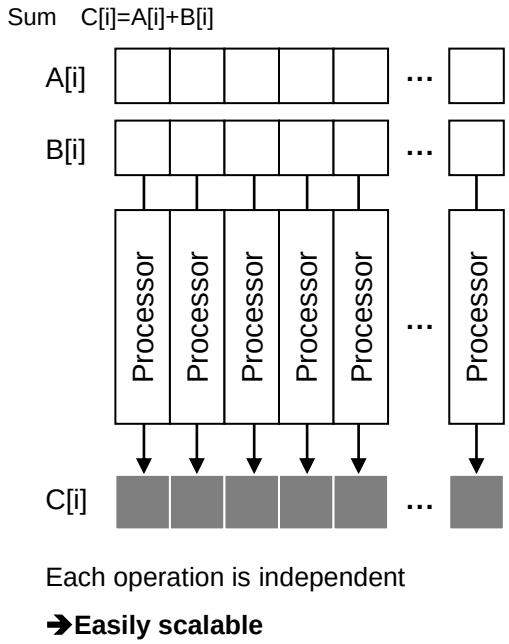
Figure 1.16: Data parallel vs task parallel



Data Parallel

The main characteristics of the data parallel method are that programming is relatively simple since multiple processors are all running the same program, and that all processors finish their tasks at about the same time. This method is efficient when the dependencies between the data being processed by each processor are minimal. For example, vector addition can benefit greatly from this method. As illustrated in **Figure 1.17**, the addition at each index can be performed completely independently of each other. For this operation, the number of processors is directly proportional to the speedup that may be achieved if overhead from parallelization can be ignored. Another, more concrete example where this method can be applied is in image processing. The pixels can be split up into blocks, and each of these blocks can be filtered in parallel by each processor.

Figure 1.17: Vector addition

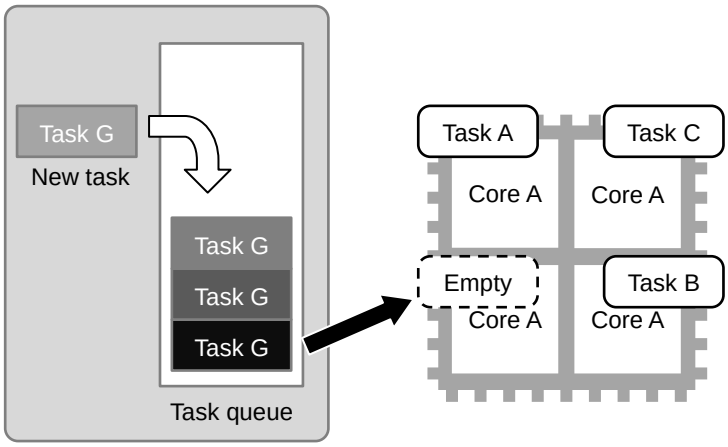


Task Parallel

The main characteristic of the task parallel method is that each processor executes different commands. This increases the programming difficulty when compared to the data parallel method. Since the processing time may vary depending on how the task is split up, it is actually not suited for the example shown in **Figure 1.16**. Also, since task_a and task_c do nothing until task_b and task_d finish, the processor utilization is decreased. The task parallel method requires a way of balancing the tasks to take full advantage of all the cores. One way is to implement a load balancing function on one

of the processors. In the example of **Figure 1.18**, the load balancer maintains a task queue, and assigns a task to a processor that has finished its previous task.

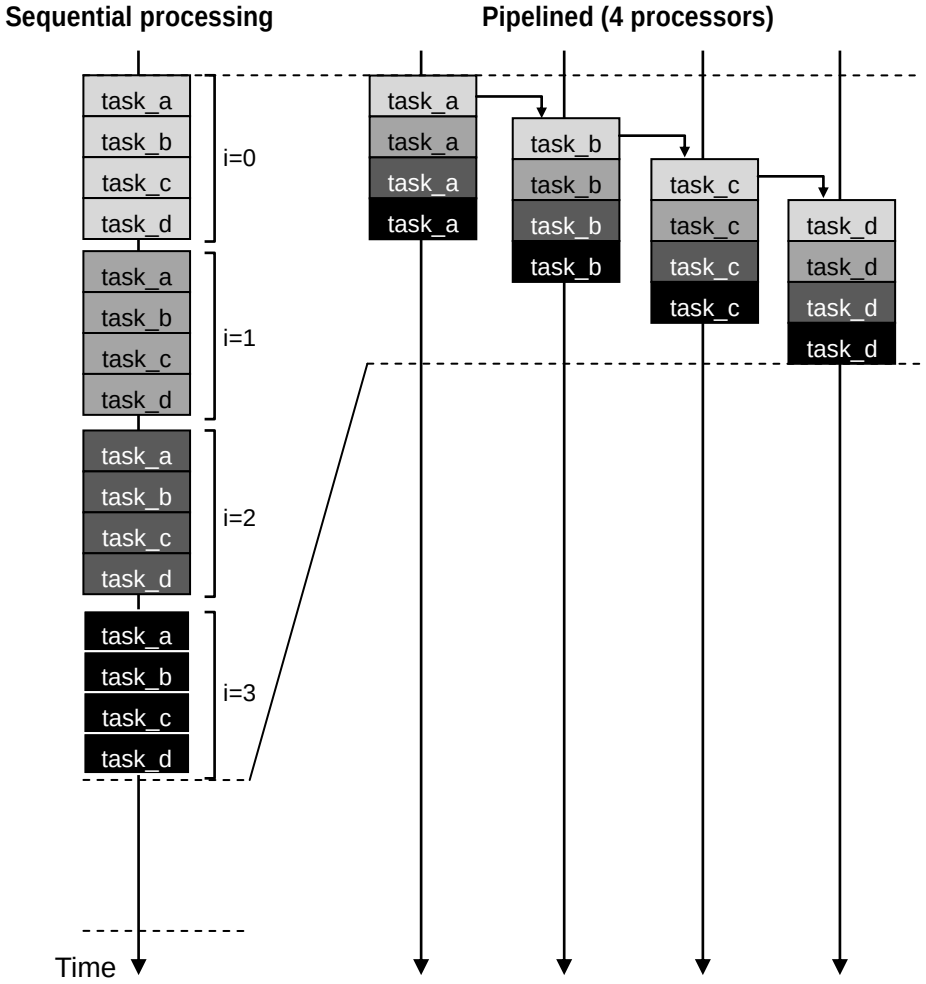
Figure 1.18: Load balancing



Another method for task parallelism is known as pipelining. Pipelining is usually in reference to the "instruction pipeline" where multiple instructions, such as instruction decoding, arithmetic operation, and register fetch, are executed in an overlapped fashion over multiple stages. This concept can be used in parallel programming as well.

Figure 1.19 shows a case where each processor is assigned its own specialized task type. In this example, the start of each task set is shifted in the time domain, such that task_b, task_c, task_d takes in the output of task_a, task_b, task_c as an input, respectively. The data moves as a stream across the processors. This method is not suited for the case where only one set of tasks is performed, but can be effective when processing, for example, video data, where frames to process are taken as inputs one after the other.

Figure 1.9: Pipelining



Hardware Dependencies

When porting sequential code to parallel code, the hardware must be evaluated carefully. Programs usually have sections suited for data parallelism as well as for task parallelism, while any given hardware is usually suited for only one or the other.

For example, the GPU is suited for data parallel algorithms, due to the existence of many cores. However, the GPU is not suited for performing different tasks in parallel, unlike the CPU, which is more suited for performing task parallel algorithms, since its fewer cores are capable of working independently of each other.

OpenCL allows the same code to be executed on either platforms, but since it cannot change the

nature of the hardware, the hardware and the parallelization method must be chosen wisely.

Implementing a Parallel Program

After deciding on the parallelization method, the next step is the implementation. In decreasing order of user involvement:

1. Writing parallel code using the operating system's functions.
2. Using a parallelization framework for porting programs.
3. Using an automatic parallelization compiler.

This section will explore the different methods.

Parallelization by Using the OS System Calls

Implementing parallel programs using the OS system call requires, at minimum, a call to execute and close a program, and some way of transferring data between the executed programs. If this is done on a cluster system, the data transfer between programs is performed using network transfer APIs such as socket system calls, but this is commonly done using a framework instead.

For parallel instructions performed within the processor itself, however, OS system calls may be used instead of the framework. The code can be further broken down into "parallel processes" and "parallel threads" to be run on the processor. The difference between processes and threads are as follows.

A program executing in a process is given its own address space by the operating system. In general, the operating system performs execution, closing, and interruption within these process units, making sure each of these processes' distinct resources do not interfere with each other. Data transfer between programs may be performed by a system call to the operating system. For example, UNIX provides a system call shmget() that allocates shared memory that can be accessed by different processes.

A thread is a subset of a process that is executed in parallel within the program. These threads share the same address space as the processes. In general, since these threads execute in the same memory space, the overhead from starting and switching is much smaller than when compared to processes. In general, the operating system provides an API to create and manage these threads. For example,

UNIX provides a library called Pthreads, which is a POSIX-approved thread protocol. POSIX is a standard API specified by IEEE.

Whether to use parallel processes or parallel threads within a processor depends on the case at hand, but in general, parallel threads are used if the goal is speed optimization, due to the smaller overhead.

List 1.2 shows an example where each member of an array is incremented using multithreading.

List 1.2: Pthreads example

```
001: #include <stdio.h>
002: #include <stdlib.h>
003: #include <pthread.h>
004:
005: #define TH_NUM 4
006: #define N 100
007:
008: static void *thread_increment(void *array)
009: {
010: int i;
011: int *iptr;
012:
013: iptr = (int *)array;
014: for(i=0;i< N / TH_NUM;i++) iptr[i] += 1;
015:
016: return NULL;
017: }
018:
019: int main(void)
020: {
021: int i;
022: pthread_t thread[TH_NUM];
023:
024: int array[N];
```

```
025:
026: /* initialize array*/
027: for(i=0;i<N;i++){
028: array[i] = i;
029: }
030:
031: /* Start parallel process */
032: for(i=0;i<TH_NUM;i++){
033: if (pthread_create(&thread[i], NULL, thread_increment, array + i * N / TH_NUM) != 0)
034: {
035: return 1;
036: }
037: }
038:
039: /* Synchronize threads*/
040: for(i=0;i<TH_NUM;i++){
041: if (pthread_join(thread[i], NULL) != 0)
042: {
043: return 1;
044: }
045: }
046:
047: return 0;
048: }
```

The code is explained below.

003: Include file required to use the Pthreads API.
008-017: The code run by each thread. It increments each array element by one. The start index is passed in as an argument.
022: Declaration of the pthread_t variable for each thread. This is used on line 033.
032-037: Creation and execution of threads. On line 033, the third argument is the name of the function to be executed by the thread, and the fourth argument is the argument passed to the thread.

039-045: Waits until all threads finish executing.

Parallelization by Using a Framework

Many frameworks exist to aid in parallelization, but the ones used in practical applications, such as in research labs and retail products, are limited. The most widely used frameworks are Message Passing Interface (MPI) for cluster servers, OpenMP for shared memory systems (SMP, NUMA), and the parallelization APIs in Boost C++. These frameworks require the user to specify the section as well as the method used for parallelization, but take care of tasks such as data transfer and execution of the threads, allowing the user to focus on the main code of the program.

List 1.3 shows an example usage of OpenMP, which is supported by most mainstream compilers, such as GCC, Intel C, and Microsoft C. From the original sequential code, the programmer inserts a "#pragma" directive, which explicitly tells the compiler which sections to run in parallel. The compiler then takes care of the thread creation and the commands for thread execution.

List 1.3: OpenMP example

```
001: #include<stdio.h>
002: #include<stdlib.h>
003: #include<omp.h>
004: #define N 100
005: #define TH_NUM 4
006:
007: int main ()
008: {
009: int i;
010: int rootBuf[N];
011:
012: omp_set_num_threads(TH_NUM);
013:
014: /* Initialize array*/
015: for(i=0;i<N;i++){
016: rootBuf[i] = i;
017: }
```

```
018:
019: /* Parallel process */
020: #pragma omp parallel for
021: for (i = 0; i < N; i++) {
022: rootBuf[i] = rootBuf[i] + 1;
023: }
024:
025: return(0);
026: }
```

When compiling the code above, the OpenMP enabling options must be specified. GCC (Linux) requires "-fopenmp", Intel C (Linux) requires "-openmp", and Microsoft Visual C++ requires /OpenMP. The code is explained below.

003: Include file required to use OpenMP.
004: Size of the array, and the number of times to run the loop. In general, this number should be somewhat large to benefit from parallelism.
012: Specifies the number of threads to be used. The argument must be an integer.
020: Breaks up the for loops that follows this directive, into the number of threads specified in 012.

This example shows how much simpler the programming becomes when we use the OpenMP framework. Compare with the code of **List 1.2** that uses Pthreads.

Automatic Parallelization Compiler

Compilers exist that examine for loops to decide automatically which sections can be executed in parallel, as well as how many threads should be used.

Intel C/C++ is an example of a compiler that does this when an option is set.

(On Linux)

```
> icc -parallel -par-report3 -par-threshold0 -03 o parallel_test parallel_test.c
```

(On Windows)

```
> Icc /Qparallel /Qpar-report3 /Q-par-threshold0 -03 o parallel_test parallel_test.c
```

The explanations of the options are given below.

• -parallel: Enables automatic parallelization.
• -par-report3: Reports which section of the code was parallelized. There are 3 verbosity levels, which can be specified this way: -par-report[n].
• -par-threshold0: Sets the threshold to decide when loops get parallelized. In order to benefit from parallelization, enough computations must be performed within each loop to hide the overhead from process/thread creation. This is specified this way: -par-threshold[n]. The value for n must be between 0 ~ 100, with a higher number implying a higher number of computations. When this value is 0, all sections that can be parallelized get parallelized. The default value is 75.

At a glance, the automatic parallelization compiler seems to be the best solution since it does not require the user to do anything, but in reality, as the code becomes more complex, the compiler has difficulty finding what can be parallelized, reducing the performance. As of January 2012, no existing compilers (at least no commercial ones) can auto-generate parallel code for hybrid systems such as the accelerator[1].

Conclusion

This section discussed the basics of parallel processing from both the hardware and the software perspectives. The content here is not limited to OpenCL, so those interested in parallel processing in general should have some familiarity with the content discussed above. The next chapter will introduce the basic concepts of OpenCL.