8

Parallel Algorithms

Consider a program that performs the following computation:

```
y = f(x)
z = g(x)
```

In this example, the function $g(x)$ does not depend on the result of the function $f(x)$, and therefore the two functions could be computed independently and in parallel.

Often large problems can be divided into smaller computational problems, which can be solved concurrently ("in parallel") using different processing units (CPUs, cores). This is called *parallel computing*. Algorithms designed to work in parallel are called *parallel algorithms*.

In this chapter, we will refer to a processing unit as a node and to the code running on a node as a process. A parallel program consists of many processes running on as many nodes. It is possible for multiple processes to run on one and the same computing unit (node) because of the multitasking capabilities of modern CPUs, but that is not true parallel computing. We will use an emulator, `Psim`, which does exactly that.

Programs can be parallelized at many levels: bit level, instruction level, data, and task parallelism. Bit-level parallelism is usually implemented in hardware. Instruction-level parallelism is also implemented in hardware in modern multi-pipeline CPUs. Data parallelism is usually referred to as

SIMD. Task parallelism is also referred to as MIMD.

Historically, parallelism was found in applications in high-performance computing, but today it is employed in many devices, including common cell phones. The reason is heat dissipation. It is getting harder and harder to improve speed by increasing CPU frequency because there is a physical limit to how much we can cool the CPU. So the recent trend is keeping frequency constant and increasing the number of processing units on the same chip.

Parallel architectures are classified according to the level at which the hardware supports parallelism, with multicore and multiprocessor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task. Specialized parallel computer architectures are sometimes used alongside traditional processors for accelerating specific tasks.

Optimizing an algorithm to run on a parallel architecture is not an easy task. Details depend on the type of parallelism and details of the architecture.

In this chapter, we will learn how to classify architectures, compute running times of parallel algorithms, and measure their performance and scaling.

We will learn how to write parallel programs using standard programming patterns, and we will use them as building blocks for more complex algorithms.

For some parts of this chapter, we will use a simulator called `PSim`, which is written in Python. Its performances will only scale on multicore machines, but it will allow us to emulate various network topologies.

8.1 Parallel architectures

8.1.1 Flynn taxonomy

Parallel computer architecture classifications are known as Flynn's taxonomy [64] and are due to the work of Michael J. Flynn in 1966.

Flynn identified the following architectures:

- **Single instruction, single data stream (SISD)**

  A sequential computer that exploits no parallelism in either the instruction or data streams. A single control unit (CU) fetches a single instruction stream (IS) from memory. The CU then generates appropriate control signals to direct single processing elements (PE) to operate on a single data stream (DS), for example, one operation at a time.

  Examples of SISD architecture are the traditional uniprocessor machines like a PC (currently manufactured PCs have multiple processors) or old mainframes.

- **Single instruction, multiple data streams (SIMD)** A computer that exploits multiple data streams against a single instruction stream to perform operations that may be naturally parallelized (e.g., an array processor or GPU).

- **Multiple instruction, single data stream (MISD)**

  Multiple instructions operate on a single data stream. This is an uncommon architecture that is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result. Examples include the now retired Space Shuttle flight control computer.

- **Multiple instruction, multiple data streams (MIMD)** Multiple autonomous processors simultaneously executing different instructions on different data. Distributed systems are generally recognized to be MIMD architectures, either exploiting a single shared memory space (using threads) or a distributed memory space (using a message-passing protocol such as MPI).

MIMD can be further subdivided into the following:

- **Single program, multiple data (SPMD)** Multiple autonomous processors simultaneously executing the same program but at independent points not synchronously (as in the SIMD case). SPMD is the most common style of parallel programming.
- **Multiple program, multiple data (MPMD)** Multiple autonomous processors simultaneously operating at least two independent programs. Typically such systems pick one node to be the "host" ("the explicit host/node programming model") or "manager" (the "manager–worker" strategy), which runs one program that farms out data to all the other nodes, which all run a second program. Those other nodes then return their results directly to the manager. The Map-Reduce pattern also falls under this category.

An embarrassingly parallel workload (or embarrassingly parallel problem) is one for which little or no effort is required to separate the problem into a number of parallel tasks. This is often the case where there exists no dependency (or communication) between those parallel tasks.

The manager–worker node strategy, when workers do not need to communicate with each other, is an example of an "embarrassingly parallel" problem.

8.1.2 Network topologies

In the MIMD case, multiple copies of the same problem run concurrently (on different data subsets and branching differently, thus performing different instructions) on different processing units, and they exchange information using a network. How fast they can communicate depends on the network characteristics identified by the network topology and the latency and bandwidth of the individual links of the network.

Normally we classify network topologies based on the following taxonomy:

- **Completely connected:** Each node is connected by a directed link to each other node.

- **Bus topology:** All nodes are connected to the same single cable. Each computer can therefore communicate with each other computer using one and the same bus cable. The limitation of this approach is that the communication bandwidth is limited by the bandwidth of the cable. Most bus networks only allow two machines to communicate with each other at one time (with the exception of one too many broadcast messages). While two machines communicate, the others are stuck waiting. The bus topology is the most inexpensive but also slow and constitutes a single point of failure.
- **Switch topology (star topology):** In local area networks with a switch topology, each computer is connected via a direct link to a central device, usually a switch, and it resembles a star. Two computers can communicate using two links (to the switch and from the switch). The central point of failure is the switch. The switch is usually intelligent and can reroute the messages from any computer to any other computer. If the switch has sufficient bandwidth, it can allow multiple computers to talk to each other at the same time. For example, for a 10 Gbit/s links and an 80 Gbit/s switch, eight computers can talk to each other (in pairs) at the same time.
- **Mesh topology:** In a mesh topology, computers are assembled into an array (1D, 2D, etc.), and each computer is connected via a direct link to the computers immediately close (left, right, above, below, etc.). Next neighbor communication is very fast because it involves a single link and therefore low latency. For two computers not physically close to communicate, it is necessary to reroute messages. The latency is proportional to the distance in links between the computers. Some meshes do not support this kind of rerouting because the extra logic, even if unused, may be cause for extra latency. Meshes are ideal for solving numerical problems such as solving differential equations because they can be naturally mapped into this kind of topology.
- **Torus topology:** Very similar to a mesh topology (1D, 2D, 3D, etc.), except that the network wraps around the edges. For example, in one dimension node, i is connected to $(i+1)\%p$, where p is the total number of nodes. A one-dimensional torus is called a *ring network*.

- **Tree network:** The tree topology looks like a tree where the computer may be associated with every tree node or every leaf only. The tree links are the communication link. For a binary tree, each computer only talks to its parent and its two children nodes. The root node is special because it has no parent node.

  Tree networks are ideal for global operations such as broadcasting and for sharing IO devices such as disks. If the IO device is connected to the root node, every other computer can communicate with it using only $\log p$ links (where p is the number of computers connected). Moreover, each subset of a tree network is also a tree network. This makes it easy to distribute subtasks to different subsets of the same architecture.

- **Hypercube:** This network assumes 2^d nodes, and each node corresponds to a vertex of a hypercube. Nodes are connected by direct links, which correspond to the edges of the hypercube. Its importance is more academical than practical, although some ideas from hypercube networks are implemented in some algorithms.

If we identify each node on the network with a unique integer number called its rank, we write explicit code to determine if two nodes i and j are connected for each network topology:

Listing 8.1: in file: `psim.py`

```python
import os, string, pickle, time, math

def BUS(i,j):
    return True

def SWITCH(i,j):
    return True

def MESH1(p):
    return lambda i,j,p=p: (i-j)**2==1

def TORUS1(p):
    return lambda i,j,p=p: (i-j+p)%p==1 or (j-i+p)%p==1

def MESH2(p):
    q=int(math.sqrt(p)+0.1)
    return lambda i,j,q=q: ((i%q-j%q)**2,(i/q-j/q)**2) in [(1,0),(0,1)]

```

```python
def TORUS2(p):
    q=int(math.sqrt(p)+0.1)
    return lambda i,j,q=q: ((i%q-j%q+q)%q,(i/q-j/q+q)%q) in [(0,1),(1,0)] or \
                           ((j%q-i%q+q)%q,(j/q-i/q+q)%q) in [(0,1),(1,0)]
def TREE(i,j):
    return i==int((j-1)/2) or j==int((i-1)/2)
```

8.1.3 Network characteristics

- **Number of links**
- **Diameter:** The max distance between any two nodes measured as a minimum number of links connecting them. Smaller diameter means smaller latency. The diameter is proportional to the maximum time it takes for a message go from one node to another.
- **Bisection width:** The minimum number of links one has to cut to turn the network into two disjoint networks. Higher bisection width means higher reliability of the network.
- **Arc connectivity:** The number of different paths connecting any two nodes. Higher connectivity means higher bandwidth and higher reliability.

Here are values of this parameter for each type of network:

Network	Links	Diameter	Width	Connectivity
completely connected	1		$p^2/4$	$p(p-1)/2$
switch	$p(p-1)$	2	$n.d.$	1
1D mesh	$p-1$	$p-1$	1	1
2D mesh	$2(p^{\frac{1}{2}}-1)p^{\frac{1}{2}}$	$2(p^{\frac{1}{2}}-1)$	$p^{\frac{1}{2}}$	$p^{\frac{1}{2}}$
3D mesh	$3(p^{\frac{1}{3}}-1)p^{\frac{2}{3}}$	$3(p^{\frac{1}{3}}-1)$	$p^{\frac{2}{3}}$	$p^{\frac{2}{3}}$
1D torus	p	$p/2$	2	2
2D torus	$2p$	$2p^{\frac{1}{2}}$	$2p^{\frac{1}{2}}$	$2p^{\frac{1}{2}}$
3D torus	$3p$	$3/2p^{\frac{1}{3}}$	$2p^{\frac{1}{3}}$	$2p^{\frac{2}{3}}$
hypercube	$p/2\log p$	$\log p$	$p/2$	$\log 2$
tree	$p-1$	$\log p$	1	1

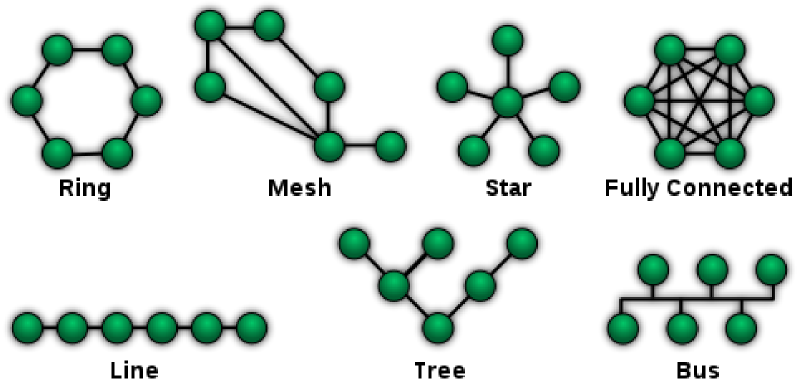

Figure 8.1: Examples of network topologies.

Most actual supercomputers implement a variety of taxonomies and topologies simultaneously. A modern supercomputer has many nodes, each node has many CPUs, each CPU has many cores, and each core implements SIMD instructions. Each core has it own cache, each CPU has its own cache, and each node has its own memory shared by all threads running on that one node. Nodes communicate with each other using multiple networks (typically a multidimensional mesh for point-to-point communications and a tree network for global communication and general disk IO).

This makes writing parallel programs very difficult. Parallel programs must be optimized for each specific architecture.

8.2 Parallel metrics

8.2.1 Latency and bandwidth

The time it takes for a message of size m (in bytes) over a wire can be broken into two components: a fixed overall time that does not depend on the size of the message, called *latency* (and indicated with t_s), and a time proportional to the message size, called *inverse bandwidth* (and indicated with t_w).

Think of a pipe of length L and section s, and you want to pump m liters of water through the pipe at velocity v. From the moment you start pumping, it takes L/v seconds before the water starts arriving at the other end of the pipe. From that moment, it will take m/sv for all the water to arrive at its destination. In this analogy, L/v is the latency t_s, sv is the bandwidth, and $t_w = 1/sv$.

The total time to send the message (or the water) is

$$T(m) = t_s + t_w m \tag{8.1}$$

From now on, we will use $T_1(n)$ to refer to the nonparallel running time of an algorithm as a function of its input m. We will use $T_p(n)$ to refer to its running time with p parallel processes.

As a practical case, in the following example, we consider a generic algorithm with the following parallel and nonparallel running times:

$$\begin{aligned} T_1(n) &= t_a n^2 && (8.2) \\ T_p(n) &= t_a n^2/p + 2p(t_s + t_w n/p) && (8.3) \end{aligned}$$

These formulas may come from example from the problem of multiplying a matrix times a vector.

Here t_a is the time to perform one elementary instruction; t_s and t_w are the latency and inverse bandwidth. The first term of T_p is nothing but T_1/p, while the second term is an overhead due to communications.

Typically $t_s >> t_w >> t_a$. In the following plots, we will always assume $t_a = 1$, $t_s = 0.2$, and $t_w = 0.1$. With these assumptions, fig. 8.2.1 shows how T_p changes with input size and number of parallel processes. Notice that while for small p, T_p decreases $\propto 1/p$, for large p, the communication overhead dominates over computation. This overhead is our example and is dominated by the latency contribution, which grows with p.

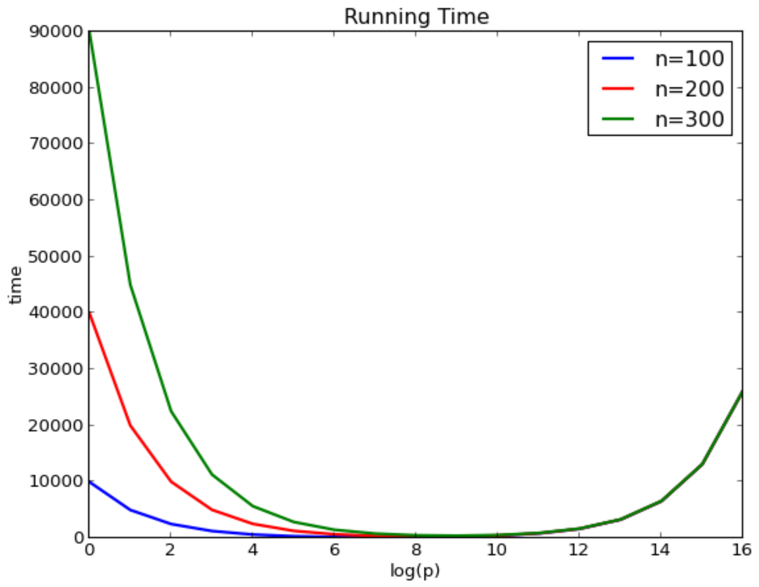

Figure 8.2: T_p as a function of input size n and number of processes p.

8.2.2 Speedup

The *speedup* is defined as

$$S_p(n) = \frac{T_1(n)}{T_p(n)} \tag{8.4}$$

where T_1 is the time it takes to run the algorithm on an input of size n on one processing unit (e.g., node), and T_p is the time it takes to run the same algorithm on the same input using p nodes in parallel. Fig. 8.2.2 shows an example of speedup. When communication overhead dominates, speedup decreases.

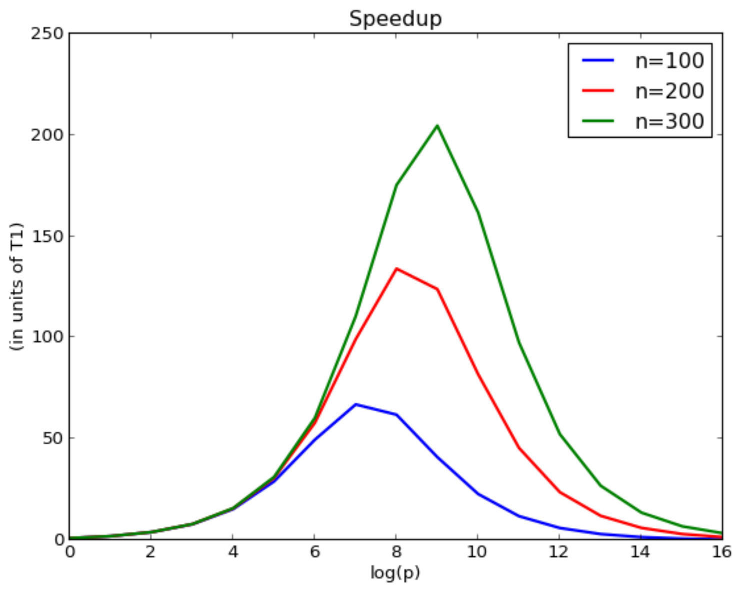

Figure 8.3: S_p as a function of input size n and number of processes p.

8.2.3 Efficiency

The *efficiency* is defined as

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T_1(n)}{pT_p(n)} \tag{8.5}$$

Notice that in case of perfect parallelization (impossible), $T_p = T_1/p$, and therefore $E_p(n) = 1$. Fig. 8.2.3 shows an example of efficiency. When communication overhead dominates, efficiency drops. Notice efficiency is always less than 1. We do not write parallel algorithms because they are more efficient. They are always less efficient and more costly than the nonparallel ones. We do it because we want the result sooner, and there is an economic value in it.

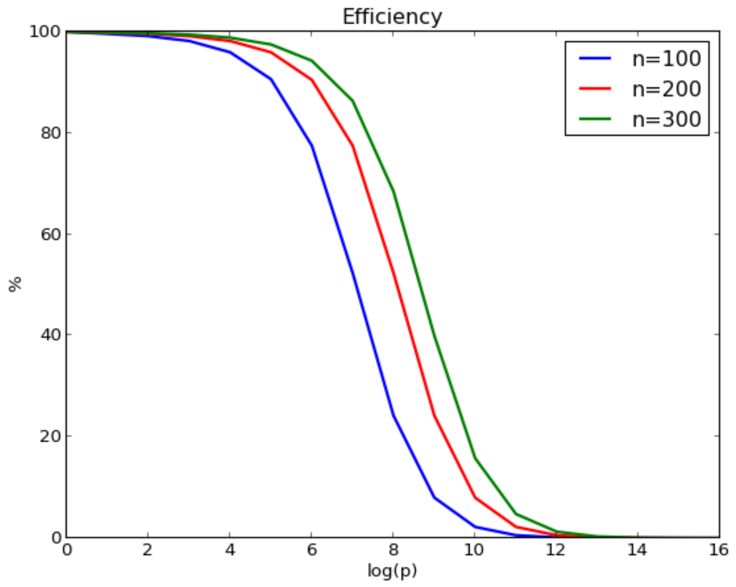

Figure 8.4: E_p as a function of input size n and number of processes p.

8.2.4 Isoefficiency

Given a value of efficiency that we choose as target, E, and a given number of nodes, p, we ask what is the maximum size of a problem that we can solve. The answer is found by solving n the following equation:

$$E_p(n) = E \tag{8.6}$$

For example T_p, we obtain

$$E_p = \frac{1}{1 + 2p^2(t_s + t_w n/p)/(n^2 t_a)} = E \tag{8.7}$$

which solved in n yields

$$n \simeq 2\frac{t_w}{t_a}\frac{E}{1-E}p \tag{8.8}$$

Isoefficiency curves for different values of E are shown in fig. 8.2.4. For our example problem, n is proportional to p. In general, this is not true, but n is monotonic in p.

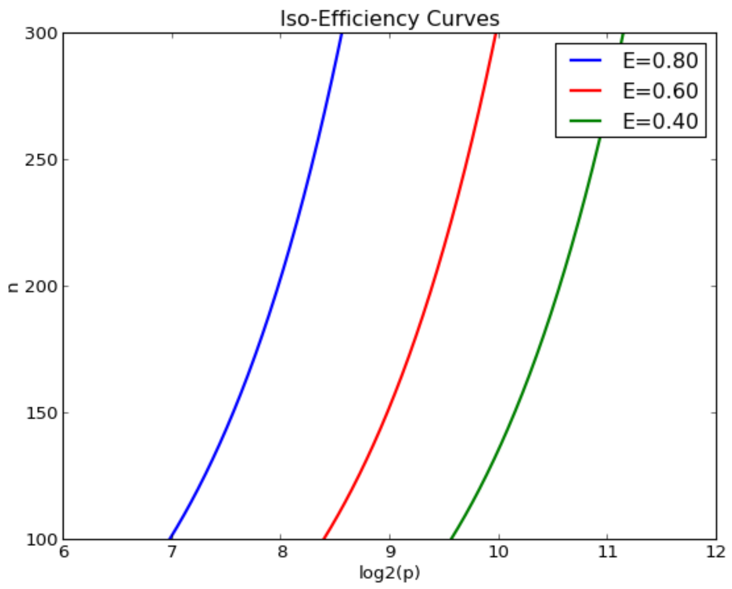

Figure 8.5: Isoefficiency curves for different values of the target efficiency.

8.2.5 Cost

The cost of a computation is equal to the time it takes to run on each node, multiplied by the number of nodes involved in the computation:

$$C_p(n) = pT_p(n) \tag{8.9}$$

Notice that in general

$$\frac{\mathrm{d}C_p(n)}{\mathrm{d}p} = \alpha T_1(n) > 0 \tag{8.10}$$

This means that for a fixed problem size n, the more an algorithm is parallelized, the more it costs to run it (because it gets less and less efficient).

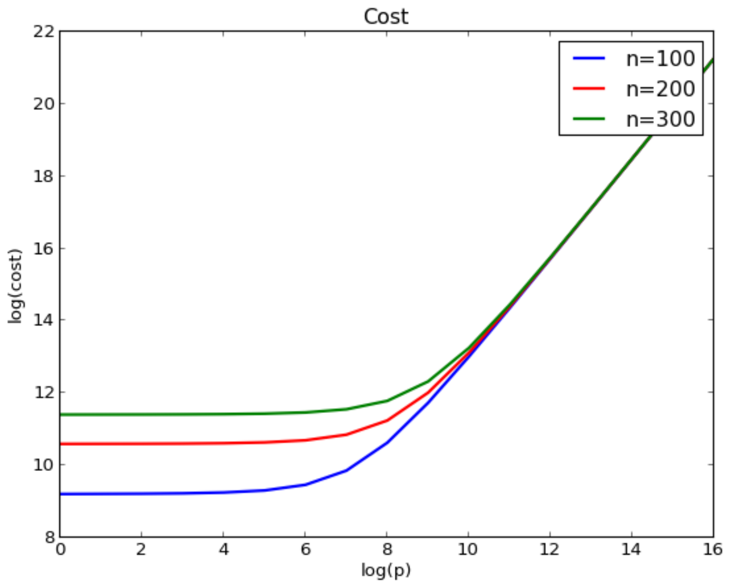

Figure 8.6: C_p as a function of input size n and a number of processes p.

8.2.6 Cost optimality

With the preceding disclaimer, we define cost optimality as the choice of p (as a function of n), which makes the cost scale proportional to $T_1(n)$:

$$pT_p(n) \propto T_1(n) \tag{8.11}$$

Or in other words,

$$p'T_p(n) + p\left(\frac{\partial T_p(n)}{\partial n} + p'\frac{\partial T_p(n)}{\partial p}\right) = \frac{\mathrm{d}T_1(n)}{\mathrm{d}n} \tag{8.12}$$

where $p' = \frac{\mathrm{d}p}{\mathrm{d}n}$.

8.2.7 Admahl's law

Consider an algorithm that can be parallelized, but one faction α of its total sequential running time αT_1 cannot be parallelized. That means that $T_p = \alpha T_1 + (1-\alpha)T_1/p$, and this yields [65]

$$S_p = \frac{1}{\alpha + (1-\alpha)/p} < \frac{1}{\alpha} \tag{8.13}$$

and

$$E_p = \frac{1}{p\alpha + (1-\alpha)} < \frac{1}{1-\alpha} \tag{8.14}$$

Therefore both the speedup and the efficiency are theoretically limited.

8.3 Message passing

Consider the following Python program:

```python
def f():
    import os
    if os.fork(): print True
    else: print False
f()
```

The output of the current program is

```
True
False
```

The function `fork` creates a copy of the current process (a child). The parent process returns the ID of the child process, and the child process returns 0. Therefore the `if` condition is both true and false, just on different processes.

We have created a Python module called `psim`, and its source code is listed here; `psim` forks the parallel processes, creates sockets connecting them, and provides API for communications. An example of `psim` usage will be given later.

Listing 8.2: in file: `psim.py`

```
class PSim(object):
    def log(self,message):
        """
        logs the message into self._logfile
        """
        if self.logfile!=None:
            self.logfile.write(message)

    def __init__(self,p,topology=SWITCH,logfilename=None):
        """
        forks p-1 processes and creates p*p
        """
        self.logfile = logfilename and open(logfile,'w')
        self.topology = topology
        self.log("START: creating %i parallel processes\n" % p)
        self.nprocs = p
        self.pipes = {}
        for i in xrange(p):
            for j in xrange(p):
                self.pipes[i,j] = os.pipe()
        self.rank = 0
        for i in xrange(1,p):
            if not os.fork():
                self.rank = i
                break
        self.log("START: done.\n")

    def send(self,j,data):
        """
        sends data to process #j
        """
        if not self.topology(self.rank,j):
            raise RuntimeError('topology violation')
        self._send(j,data)

    def _send(self,j,data):
        """
        sends data to process #j ignoring topology
        """
        if j<0 or j>=self.nprocs:
            self.log("process %i: send(%i,...) failed!\n" % (self.rank,j))
            raise Exception
        self.log("process %i: send(%i,%s) starting...\n" % \
                 (self.rank,j,repr(data)))
        s = pickle.dumps(data)
        os.write(self.pipes[self.rank,j][1], string.zfill(str(len(s)),10))
        os.write(self.pipes[self.rank,j][1], s)
        self.log("process %i: send(%i,%s) success.\n" % \
                 (self.rank,j,repr(data)))

    def recv(self,j):
        """
        returns the data received from process #j
        """
        if not self.topology(self.rank,j):
            raise RuntimeError('topology violation')
        return self._recv(j)

    def _recv(self,j):
        """
        returns the data received from process #j ignoring topology
        """
        if j<0 or j>=self.nprocs:
            self.log("process %i: recv(%i) failed!\n" % (self.rank,j))
            raise RuntimeError
        self.log("process %i: recv(%i) starting...\n" % (self.rank,j))
        try:
            size=int(os.read(self.pipes[j,self.rank][0],10))
            s=os.read(self.pipes[j,self.rank][0],size)
        except Exception, e:
            self.log("process %i: COMMUNICATION ERROR!!!\n" % (self.rank))
            raise e
        data=pickle.loads(s)
        self.log("process %i: recv(%i) done.\n" % (self.rank,j))
        return data
```

An instance of the class `PSim` is an object that can be used to determine the total number of parallel processes, the rank of each running process, to send messages to other processes, and to receive messages from them. It is usually called a *communicator*; `send` and `recv` represent the simplest type of communication pattern, point-to-point communication.

A PSim program starts by importing and creating an instance of the `PSim` class. The constructor takes two arguments, the number of parallel processes you want and the network topology you want to emulate. Before returning the `PSim` instance, the constructor makes $p-1$ copies of the running process and creates sockets connecting each two of them. Here is a simple example in which we make two parallel processes and send a message from process 0 to process 1:

```
from psim import *

```

```
comm = PSim(2,SWITCH)
if comm.rank == 0:
    comm.send(1, "Hello World")
elif comm.rank == 1:
    message = comm.recv(0)
    print message
```

Here is a more complex example that creates $p = 10$ parallel processes, and node 0 sends a message to each one of them:

```
from psim import *

p = 10

comm = PSim(p,SWITCH)
if comm.rank == 0:
    for other in range(1,p):
       comm.send(other, "Hello %s" % p)
else:
    message = comm.recv(0)
    print message
```

Following is a more complex example that implements a parallel scalar product. The process with rank 0 makes up two vectors and distributes pieces of them to the other processes. Each process computes a part of the scalar product. Of course, the scalar product runs in linear time, and it is very inefficient to parallelize it, yet we do it for didactic purposes.

Listing 8.3: in file: `psim_scalar.py`

```
import random
from psim import PSim

def scalar_product_test1(n,p):
    comm = PSim(p)
    h = n/p
    if comm.rank==0:
        a = [random.random() for i in xrange(n)]
        b = [random.random() for i in xrange(n)]
        for k in xrange(1,p):
            comm.send(k, a[k*h:k*h+h])
            comm.send(k, b[k*h:k*h+h])
    else:
        a = comm.recv(0)
        b = comm.recv(0)
    scalar = sum(a[i]*b[i] for i in xrange(h))
    if comm.rank == 0:
        for k in xrange(1,p):
```

```python
            scalar += comm.recv(k)
        print scalar
    else:
        comm.send(0,scalar)

scalar_product_test(10,2)
```

Most parallel algorithms follow a similar pattern. One process has access to IO. That process reads and scatters the data. The other processes perform their part of the computation; the results are reduced (aggregated) and sent back to the root process. This pattern may be repeated by multiple functions, perhaps in loops. Different functions may handle different data structure and may have different communication patterns. The one thing that must be constant throughout the run is the number of processes because one wants to pair each process with one computing unit.

In the following, we implement a parallel version of the `mergesort`. At each step, the code splits the problem into two smaller problems. Half of the problem is solved by the process that performed the split and assigns the other half to an existing free process. When there are no more free processes, it reverts to the nonparallel `mergesort` step. The `merge` function here is the same as the nonparallel `mergesort` of chapter 3.

Listing 8.4: in file: `psim_mergesort.py`

```python
import random
from psim import PSim

def mergesort(A, x=0, z=None):
    if z is None: z = len(A)
    if x<z-1:
        y = int((x+z)/2)
        mergesort(A,x,y)
        mergesort(A,y,z)
        merge(A,x,y,z)

def merge(A,x,y,z):
    B,i,j = [],x,y
    while True:
        if A[i]<=A[j]:
            B.append(A[i])
            i=i+1
        else:
            B.append(A[j])
            j=j+1
        if i==y:
            while j<z:
                B.append(A[j])
                j=j+1
            break
        if j==z:
            while i<y:
                B.append(A[i])
                i=i+1
            break
    A[x:z]=B

def mergesort_test(n,p):
    comm = PSim(p)
    if comm.rank==0:
        data = [random.random() for i in xrange(n)]
        comm.send(1, data[n/2:])
        mergesort(data,0,n/2)
        data[n/2:] = comm.recv(1)
        merge(data,0,n/2,n)
        print data
    else:
        data = comm.recv(0)
        mergesort(data)
        comm.send(0,data)

mergesort_test(20,2)
```

More interesting patterns are global communication patterns implemented on top of `send` and `recv`. Subsequently, we discuss the most common: broadcast, scatter, collect, and reduce. Our implementation is not the most efficient, but it is the simplest. In principle, there should be a different implementation for each type of network topology to take advantage of its features.

8.3.1 Broadcast

The simplest type of broadcast is the one-2-all, which consists of one process (source) sending a message (value) to every other process. A more complex broadcast is when each process broadcasts a message simultaneously and each node receives the list of values ordered by the rank of the sender:

Listing 8.5: in file: `psim.py`

```
    def one2all_broadcast(self, source, value):
        self.log("process %i: BEGIN one2all_broadcast(%i,%s)\n" % \
                 (self.rank,source, repr(value)))
        if self.rank==source:
            for i in xrange(0, self.nprocs):
                if i!=source:
                    self._send(i,value)
        else:
            value=self._recv(source)
        self.log("process %i: END one2all_broadcast(%i,%s)\n" % \
                 (self.rank,source, repr(value)))
        return value

    def all2all_broadcast(self, value):
        self.log("process %i: BEGIN all2all_broadcast(%s)\n" % \
                 (self.rank, repr(value)))
        vector=self.all2one_collect(0,value)
        vector=self.one2all_broadcast(0,vector)
        self.log("process %i: END all2all_broadcast(%s)\n" % \
                 (self.rank, repr(value)))
        return vector
```

We have implemented the all-to-all broadcast using a trick. We send collected all values to node with rank 0 (via a function collect), and then we did a one-to-all broadcast of the entire list from node 0. In general, the implementation depends on the topology of the available network.

Here is an example of an application of broadcasting:

```
from psim import *

p = 10

comm = PSim(p,SWITCH)
message = "Hello World" if comm.rank==0 else None
message = comm.one2all_broadcast(0, message)
print message
```

Notice how before the broadcast, only the process with rank 0 has knowledge of the message. After broadcast, all nodes are aware of it. Also notice that `one2all_broadcast` is a global communication function, and all processes must call it. Its first argument is the rank of the broadcasting process (0), while the second argument is the message to be broadcasted (only the value from node 0 is actually used).

8.3.2 Scatter and collect

The all-to-one collect pattern works as follows. Every process sends a value to process `destination`, which receives the values in a list ordered according to the rank of the senders:

Listing 8.6: in file: `psim.py`

```python
    def one2all_scatter(self,source,data):
        self.log('process %i: BEGIN all2one_scatter(%i,%s)\n' % \
                     (self.rank,source,repr(data)))
        if self.rank==source:
             h, remainder = divmod(len(data),self.nprocs)
             if remainder: h+=1
             for i in xrange(self.nprocs):
                 self._send(i,data[i*h:i*h+h])
        vector = self._recv(source)
        self.log('process %i: END all2one_scatter(%i,%s)\n' % \
                     (self.rank,source,repr(data)))
        return vector

    def all2one_collect(self,destination,data):
        self.log("process %i: BEGIN all2one_collect(%i,%s)\n" % \
                     (self.rank,destination,repr(data)))
        self._send(destination,data)
        if self.rank==destination:
            vector = [self._recv(i) for i in xrange(self.nprocs)]
        else:
            vector = []
        self.log("process %i: END all2one_collect(%i,%s)\n" % \
                     (self.rank,destination,repr(data)))
        return vector
```

Here is a revised version of the previous scalar product example using scatter:

Listing 8.7: in file: `psim_scalar2.py`

```python
import random
from psim import PSim

def scalar_product_test2(n,p):
    comm = PSim(p)
    a = b = None
    if comm.rank==0:
        a = [random.random() for i in xrange(n)]
        b = [random.random() for i in xrange(n)]
    a = comm.one2all_scatter(0,a)
    b = comm.one2all_scatter(0,b)

    scalar = sum(a[i]*b[i] for i in xrange(len(a)))

    scalar = comm.all2one_reduce(0,scalar)
    if comm.rank == 0:
        print scalar

scalar_product_test2(10,2)
```

8.3.3 Reduce

The all-to-one reduce pattern is very similar to the collect, except that the `destination` does not receive the entire list of values but some aggregated information about the values. The aggregation must be performed using a commutative binary function $f(x,y) = f(y,x)$. This guarantees that the reduction from the values go down in any order and thus are optimized for different network topologies.

The all-to-all reduce is similar to reduce, but every process will get the result of the reduction, not just one `destination` node. This may be achieved by an all-to-one reduce followed by a one-to-all broadcast:

Listing 8.8: in file: `psim.py`

```
    def all2one_reduce(self,destination,value,op=lambda a,b:a+b):
        self.log("process %i: BEGIN all2one_reduce(%s)\n" % \
                 (self.rank,repr(value)))
        self._send(destination,value)
        if self.rank==destination:
            result = reduce(op,[self._recv(i) for i in xrange(self.nprocs)])
        else:
            result = None
        self.log("process %i: END all2one_reduce(%s)\n" % \
                 (self.rank,repr(value)))
        return result

    def all2all_reduce(self,value,op=lambda a,b:a+b):
        self.log("process %i: BEGIN all2all_reduce(%s)\n" % \
                 (self.rank,repr(value)))
        result=self.all2one_reduce(0,value,op)
        result=self.one2all_broadcast(0,result)
        self.log("process %i: END all2all_reduce(%s)\n" % \
                 (self.rank,repr(value)))
        return result
```

And here are some examples of a reduce operation that can be passed to the `op` argument of the `all2one_reduce` and `all2all_reduce` methods:

Listing 8.9: in file: `psim.py`

```
        @staticmethod
        def sum(x,y): return x+y
        @staticmethod
        def mul(x,y): return x*y
        @staticmethod
        def max(x,y): return max(x,y)
        @staticmethod
        def min(x,y): return min(x,y)
```

Graph algorithms can also be parallelized, for example, the Prim algorithm. One way to do it is to represent the graph using an adjacency matrix where term i, j corresponds to the link between vertex i and vertex j. The term can be `None` if the link does not exist. Any graph algorithm, in some order, loops over the vertices and over the neighbors. This step can be parallelized by assigning different columns of the adjacency matrix to different computing processes. Each process only loops over some of the neighbors of the vertex being processed. Here is an example of the Prim algorithm:

Listing 8.10: in file: `psim_prim.py`

```
from psim import PSim
import random

def random_adjacency_matrix(n):
    A = []
    for r in range(n):
        A.append([0]*n)
    for r in range(n):
        for c in range(0,r):
            A[r][c] = A[c][r] = random.randint(1,100)
    return A

class Vertex(object):
    def __init__(self,path=[0,1,2]):
        self.path = path

def weight(path=[0,1,2], adjacency=None):
    return sum(adjacency[path[i-1]][path[i]] for i in range(1,len(path)))

```

```python
def bb(adjacency,p=1):
    n = len(adjacency)
    comm = PSim(p)
    Q = []
    path = [0]
    Q.append(Vertex(path))
    bound = float('inf')
    optimal = None
    local_vertices = comm.one2all_scatter(0,range(n))
    while True:
        if comm.rank==0:
            vertex = Q.pop() if Q else None
        else:
            vertex = None
        vertex = comm.one2all_broadcast(0,vertex)
        if vertex is None:
            break
        P = []
        for k in local_vertices:
            if not k in vertex.path:
                new_path = vertex.path+[k]
                new_path_length = weight(new_path,adjacency)
                if new_path_length<bound:
                    if len(new_path)==n:
                        new_path.append(new_path[0])
                        new_path_length = weight(new_path,adjacency)
                        if new_path_length<bound:
                            bound = new_path_length # bcast
                            optimal = new_path      # bcast
                    else:
                        new_vertex = Vertex(new_path)
                        P.append(new_vertex) # fix this
                print new_path, new_path_length
        x = (bound,optimal)
        x = comm.all2all_reduce(x,lambda a,b: min(a,b))
        (bound,optimal) = x

        P = comm.all2one_collect(0,P)
        if comm.rank==0:
            for item in P:
                Q+=item
    return optimal, bound


m = random_adjacency_matrix(5)
print bb(m,p=2)
```

8.3.4 Barrier

Another global communication pattern is the barrier. It forces all processes when they reach the barrier to stop and wait until all the other processes have reached the barrier. Think of runners gathering at the starting line of a race; when all the runners are there, the race can start.

Here we implement it using a simple all-to-all broadcast:

Listing 8.11: in file: psim.py

```
    def barrier(self):
        self.log("process %i: BEGIN barrier()\n" % (self.rank))
        self.all2all_broadcast(0)
        self.log("process %i: END barrier()\n" % (self.rank))
        return
```

The use of `barrier` is usually a symptom of bad code because it forces parallel processes to wait for other processes without data actually being transferred.

8.3.5 Global running times

In the following table, we compute the order of growth or typical running times for the most common network topologies for typical communication algorithms:

Network	Send/Recv	One2All Bcast	Scatter
completely connected	1	1	1
switch	2	$\log p$	$2p$
1D mesh	$p-1$	$p-1$	p^2
2D mesh	$2(p^{\frac{1}{2}}-1)$	$\sqrt{p}$	p^2
3D mesh	$3(p^{\frac{1}{3}}-1)$	$p^{1/3}$	p^2
1D torus	$p/2$	$p/2$	p^2
2D torus	$2p^{\frac{1}{2}}$	$\sqrt{p}/2$	p^2
3D torus	$3/2p^{\frac{1}{3}}$	$p^{1/3}/2$	p^2
hypercube	$\log p$	$\log_2 p$	p
tree	$\log p$	$\log p$	p

It is obvious that the completely connected is the fastest network but also the most expensive to build. The tree is a cheap compromise. The switch tends to be faster for arbitrary point-to-point communication, but the switch comes to a premium. Multidimensional meshes and toruses become cost-effective when solving problems that are naturally defined on a grid because they only require next neighbor interaction.

8.4 mpi4py

The Psim emulator does not provide any actual speedup unless you have multiple cores or processors to execute the forked processes. A better approach would be to use `mpi4py` [66] because it allows running different processes on different machines on a network. `mpi4py` is a Python interface to the message passing interface (MPI). MPI is a standard protocol and API for interprocess communications. Its API are equivalent one by one to those of `PSim`, except that they have different names and different signatures.

Here is an example of using mpi4py:

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
   message = "Hello World"
   comm.send(message, dest=1, tag=11)
elif rank == 1:
   message = comm.recv(source=0, tag=11)
   print message
```

The `comm` object of class `MPI.COMM_WORLD` plays a similar role as the `PSim` object of the previous section. The MPI `send` and `recv` functions are very similar to the `PSim` equivalent ones, except that they require details about the type of the data being transferred and a communication tag. The tag allows node A to send multiple messages to B and allows B to receive them out of order. PSim does not allow tags.

8.5 Master-Worker and Map-Reduce

Map-Reduce [67] is a framework for processing highly distributable problems across huge data sets using a large number of computers (nodes). The group of computers is collectively referred to as a cluster (if all nodes use the same hardware) or a grid (if the nodes use different hardware). It comprises two steps:

"Map" (implemented here in a function `mapfn`): The master node takes the input data, partitions it into smaller subproblems, and distributes individual pieces of the data to worker nodes. A worker node may do this again in turn, leading to a multilevel tree structure. The worker node processes the smaller problem, computes a result, and passes that result back to its master node.

"Reduce" (implemented here in a function `reducefn`): The master node collects the partial results from all the subproblems and combines them in some way to compute the answer to the problem it needs.

Map-Reduce allows for distributed processing of the map and reduction operations. Provided each mapping operation is independent of the others, all maps can be performed in parallel—though in practice, it is limited by the number of independent data sources and/or the number of CPUs near each source. Similarly, a set of "reducers" can perform the reduction phase, provided all outputs of the map operation that share the same key are presented to the same reducer at the same time.

While this process can often appear inefficient compared to algorithms that are more sequential, Map-Reduce can be applied to significantly larger data sets than "commodity" servers can handle—a large server farm can use Map-Reduce to sort a petabyte of data in only a few hours, which would require much longer in a monolithic or single process system.

Parallelism also offers some possibility of recovering from partial failure of servers or storage during the operation: if one mapper or reducer fails, the work can be rescheduled—assuming the input data are still available.

Map-Reduce comprises of two main functions: `mapfn` and `reducefn`. `mapfn` takes a (key,value) pair of data with a type in one data domain and returns a list of (key,value) pairs in a different domain:

$$\text{mapfn}(k1, v1) \rightarrow (k2, v2) \tag{8.15}$$

The `mapfn` function is applied in parallel to every item in the input data set. This produces a list of (k2,v2) pairs for each call. After that, the Map-Reduce framework collects all pairs with the same key from all lists and groups them together, thus creating one group for each one of the different generated keys. The `reducefn` function is then applied in parallel to each group, which in turn produces a collection of values in the same domain:

$$\text{reducefn}(k2, [\text{list of } v2]) \rightarrow (k2, v3) \tag{8.16}$$

The values returned by `reducefn` are then collected into a single list. Each call to `reducefn` can produce one, none, or multiple partial results. Thus the Map-Reduce framework transforms a list of (key, value) pairs into a list of values. It is necessary but not sufficient to have implementations of the map and reduce abstractions to implement Map-Reduce. Distributed implementations of Map-Reduce require a means of connecting the processes performing the `mapfn` and `reducefn` phases.

Here is a nonparallel implementation that explains the data workflow better:

```python
def mapreduce(mapper,reducer,data):
    """
    >>> def mapfn(x): return x%2, 1
    >>> def reducefn(key,values): return len(values)
    >>> data = xrange(100)
    >>> print mapreduce(mapfn,reducefn,data)
    {0: 50, 1: 50}
    """
    partials = {}
    results = {}
    for item in data:
        key,value = mapper(item)
        if not key in partials:
            partials[key]=[value]
        else:
            partials[key].append(value)
    for key,values in partials.items():
        results[key] = reducer(key,values)
    return results
```

And here is an example we can use to find how many random DNA strings contain the subsequence "ACTA":

```
>>> from random import choice
>>> strings = [''.join(choice('ATGC') for i in xrange(10))
...            for j in xrange(100)]
>>> def mapfn(string): return ('ACCA' in string, 1)
>>> def reducefn(check, values): return len(values)
>>> print mapreduce(mapfn,reducefn,strings)
{False: ..., True: ...}
```

The important thing about the preceding code is that there are two loops in Map-Reduce. Each loop consists of executing tasks (map tasks and reduce tasks) which are independent from each other (all the maps are independent, all the reduce are independent, but the reduce depend on the maps). Because they are independent, they can be executed in parallel and by different processes.

A simple and small library that implements the map-reduce algorithm in Python is `mincemeat` [68]. The workers connect and authenticate to the server using a password and request tasks to executed. The server accepts connections and assigns the map and reduce tasks to the workers.

The communication is performed using asynchronous sockets, which means neither workers nor the master is ever in a wait state. The code is event based, and communication only happens when a socket connecting the master to a worker is ready for a write (task assignment) or a read (task completed).

The code is also failsafe because if a worker closes the connection prematurely, the task is reassigned to another worker.

Function `mincemeat` uses the python libraries `asyncore` and `asynchat` to implement the communication patterns, for which we refer to the Python documentation.

Here is an example of a `mincemeat` program:

```
import mincemeat
from random import choice
```

```

strings = [''.join(choice('ATGC') for i in xrange(10)) for j in xrange(100)]
def mapfn(k1, string): yield ('ACCA' in string, 1)
def reducefn(k2, values): return len(values)

s = mincemeat.Server()
s.mapfn = mapfn
s.reducefn = reducefn
s.datasource = dict(enumerate(strings))
results = s.run_server(password='changeme')
print results
```

Notice that in `mincemeat`, the data source is a list of key value dictionaries where the values are the ones to be processed. The key is also passed to the `mapfn` function as first argument. Moreover, the `mapfn` function can return more than one value using `yield`. This syntactical notation makes `minemeat` more flexible.

Execute this script on the server:

```
> python mincemeat_example.py
```

Run mincemeat.py as a worker on a client:

```
> python mincemeat.py -p changeme [server address]
```

You can run more than one worker, although for this example the server will terminate almost immediately.

Function `mincemeat` works fine for many applications, but sometimes one wishes for a more powerful tool that provides faster communications, support for arbitrary languages, and better scalability tools and monitoring tools. An example in Python is `disco`. A standard tool, written in Java but supporting Python, is **Hadoop**.

8.6 pyOpenCL

Nvidia should be credited for bringing GPU computing to the mass market. They have developed the CUDA [69] framework for GPU programming. CUDA programs consist of two parts: a host and a kernel. The host deploys the kernel on the available GPU core, and multiple copies of the kernel run in parallel.

Nvidia, AMD, Intel, and ARM have created the Kronos Group, and together they have developed the Open Common Language framework (OpenCL [70]), which borrows many ideas from CUDA and promises more portability. OpenCL supports Intel/AMD CPUs, Nvidia/ATI GPU, ARM chips, and the LLVM virtual machine.

OpenCL is a C99 dialect. In OpenCL, like in CUDA, there is a host program and a kernel. Multiple copies of the kernel are queued and run in parallel on available devices. Kernels running on the same device have access to a shared memory area as well as local memory areas.

A typical OpenCL program has the following structure:

```
find available devices (GPUs, CPUs)
copy data from host to device
run N copies of this kernel code on the device
copy data from device to host
```

Usually the kernel is written in C99, while the host is written in C++. It is also possible to write the host code in other languages, including Python. Here we will use the `pyOpenCL` [4] module for programming the host using Python. This produces no significative loss of performance compared to C++ because the actual computation is performed by kernel, not by the host. It is also possible to write the kernels using Python. This can be done using a library called Clyther [71] or one called `ocl` [5]. Here we will use the latter; `ocl` performs a one-time conversion of Python code for the kernel to C99 code. This conversion is done line by line and therefore also introduces no performance loss compared to writing native OpenCL kernels. It also provides an additional abstraction layer on top of `pyOpenCL`, which will make our examples more compact.

8.6.1 A first example with PyOpenCL

`pyOpenCL` uses `numpy` multidimensional arrays to store data. For example, here is a `numpy` example that performs the scalar product between two vectors, u and v:

```
import numpy as npy

size = 10000
```

```
u = npy.random.rand(size).astype(npy.float32)
v = npy.random.rand(size).astype(npy.float32)
w = npy.zeros(n,dtype=numpy.float32)

for i in xrange(0, n):
    w[i] = u[i] + v[i];

assert npy.linalg.norm(w - (u + v)) == 0
```

The program works as follows:

- It creates a two `numpy` arrays u and v of given size and filled with random numbers.
- It creates another `numpy` array w of the same size filled with zeros.
- It loops over all indices of w and adds, term by term, u and v storing the result into w.
- It checks the result using the `numpy linalg` submodule.

Our goal is to parallelize the part of the computation performed in the loop. Notice that our parallelization will not make the code faster because this is a linear algorithm, and algorithms linear in the input are never faster when parallelized because the communication has the same order of growth as the algorithm itself:

```
from ocl import Device
import numpy as npy

n = 100000
u = npy.random.rand(n).astype(npy.float32)
v = npy.random.rand(n).astype(npy.float32)

device = Device()
u_buffer = device.buffer(source=a)
v_buffer = device.buffer(source=b)
w_buffer = device.buffer(size=b.nbytes)

kernels = device.compile("""
    __kernel void sum(__global const float *u, /* u_buffer  */
                      __global const float *v, /* v_buffer  */
                      __global float *w) {     /* w_buffer  */
      int i = get_global_id(0);                /* thread id */
      w[i] = u[i] + v[i];
    }
    """)
```

```

kernels.sum(device.queue,[n],None,u_buffer,v_buffer,w_buffer)
w = device.retrieve(w_buffer)

assert npy.linalg.norm(w - (u+v)) == 0
```

This program performs the following steps in addition to the original non-OpenCL code: it declares a `device` object; it declares a buffer for each of the vectors u, v, and w; it declares and compiles the kernel; it runs the kernel; it retrieves the result.

The `device` object encapsulate the kernel(s) and a queue for kernel submission.

The line:

```
kernels.sum(...,[n],...)
```

submits to the queue `n` instances of the `sum` kernel. Each kernel instance can retrieve its own ID using the function `get_global_id(0)`. Notice that a kernel must be declared with the `__kernel` prefix. Arguments that are to be shared by all kernels must be `__global`.

The `Device` class is defined in the "ocl.py" file in terms of `pyOpenCL` API:

```
import numpy
import pyopencl as pcl

class Device(object):
    flags = pcl.mem_flags
    def __init__(self):
        self.ctx = pcl.create_some_context()
        self.queue = pcl.CommandQueue(self.ctx)
    def buffer(self,source=None,size=0,mode=pcl.mem_flags.READ_WRITE):
        if source is not None: mode = mode|pcl.mem_flags.COPY_HOST_PTR
        buffer = pcl.Buffer(self.ctx,mode, size=size, hostbuf=source)
        return buffer
    def retrieve(self,buffer,shape=None,dtype=numpy.float32):
        output = numpy.zeros(shape or buffer.size/4,dtype=dtype)
        pcl.enqueue_copy(self.queue, output, buffer)
        return output
    def compile(self,kernel):
        return pcl.Program(self.ctx,kernel).build()
```

Here `self.ctx` is the device context, `self.queue` is the device queue. The functions `buffer`, `retrieve`, and `compile` map onto the corresponding pyOpenCL functions `Buffer`, `enqueue_copy`, and `Program` but use a simpler syntax. For more details, we refer to the official `pyOpenCL` documentation.

8.6.2 Laplace solver

In this section we implement a two-dimensional Laplace solver. A three-dimensional generalization is straightforward. In particular, we want to solve the following differential equation known as a Laplace equation:

$$(\partial_x^2 + \partial_y^2)u(x,y) = q(x,y) \tag{8.17}$$

Here q is the input and u is the output.

This equation originates, for example, in electrodynamics. In this case, q is the distribution of electric charge in space and u is the electrostatic potential.

As we did in chapter 3, we proceed by discretizing the derivatives:

$$\begin{aligned} \partial_x^2 u(x,y) &= (u(x-h,y) - 2u(x,y) + u(x+h,y))/h^2 && (8.18)\\ \partial_y^2 u(x,y) &= (u(x,y-h) - 2u(x,y) + u(x,y+h))/h^2 && (8.19) \end{aligned}$$

Substitute them into eq. 8.17 and solve the equation in $u(x,y)$. We obtain

$$u(x,y) = 1/4(u(x-h,y)+u(x+h,y)+u(x,y-h)+u(x,y+h)-h^2q(x,y)) \tag{8.20}$$

We can therefore solve eq. 8.17 by iterating eq. 8.20 until convergence. The initial value of u will not affect the solution, but the closer we can pick it to the actual solution, the faster the convergence.

The procedure we utilized here for transforming a differential equation into an iterative procedure is a general one and applies to other differential equations as well. The iteration proceeds very much as the fixed point solver also examined in chapter 3.

Here is an implementation using `ocl`:

```
from ocl import Device
```

```python
from canvas import Canvas
from random import randint, choice
import numpy

n = 300
q = numpy.zeros((n,n), dtype=numpy.float32)
u = numpy.zeros((n,n), dtype=numpy.float32)
w = numpy.zeros((n,n), dtype=numpy.float32)

for k in xrange(n):
    q[randint(1, n-1),randint(1, n-1)] = choice((-1,+1))

device = Device()
q_buffer = device.buffer(source=q, mode=device.flags.READ_ONLY)
u_buffer = device.buffer(source=u)
w_buffer = device.buffer(source=w)


kernels = device.compile("""
       __kernel void solve(__global float *w,
                           __global const float *u,
                           __global const float *q) {
          int x = get_global_id(0);
          int y = get_global_id(1);
          int xy = y*WIDTH + x, up, down, left, right;
          if(y!=0 && y!=WIDTH-1 && x!=0 && x!=WIDTH-1) {
             up=xy+WIDTH; down=xy-WIDTH; left=xy-1; right=xy+1;
             w[xy] = 1.0/4.0*(u[up]+u[down]+u[left]+u[right] - q[xy]);
          }
       }
    """.replace('WIDTH',str(n)))

for k in xrange(1000):
    kernels.solve(device.queue, [n,n], None, w_buffer, u_buffer, q_buffer)
    (u_buffer, w_buffer) = (w_buffer, u_buffer)

u = device.retrieve(u_buffer,shape=(n,n))

Canvas().imshow(u).save(filename='plot.png')
```

We can now use the Python to C99 converter of `ocl` to write the kernel using Python:

```python
from ocl import Device
from canvas import Canvas
from random import randint, choice
import numpy

n = 300
```

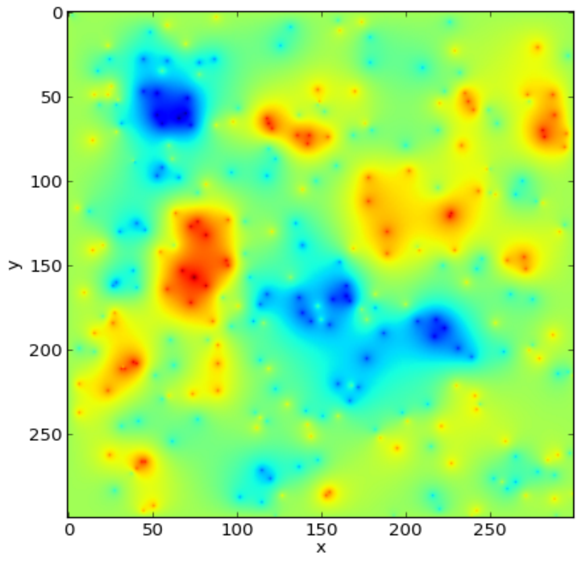

Figure 8.7: The image shows the output of the Laplace program and represents the two-dimensional electrostatic potential for a random charge distribution.

```
q = numpy.zeros((n,n), dtype=numpy.float32)
u = numpy.zeros((n,n), dtype=numpy.float32)
w = numpy.zeros((n,n), dtype=numpy.float32)

for k in xrange(n):
    q[randint(1, n-1),randint(1, n-1)] = choice((-1,+1))

device = Device()
q_buffer = device.buffer(source=q, mode=device.flags.READ_ONLY)
u_buffer = device.buffer(source=u)
w_buffer = device.buffer(source=w)

@device.compiler.define_kernel(
    w='global:ptr_float',
    u='global:const:ptr_float',
    q='global:const:ptr_float')
def solve(w,u,q):
    x = new_int(get_global_id(0))
    y = new_int(get_global_id(1))
    xy = new_int(x*n+y)
    if y!=0 and y!=n-1 and x!=0 and x!=n-1:
        up = new_int(xy-n)
        down = new_int(xy+n)
        left = new_int(xy-1)
        right = new_int(xy+1)
        w[xy] = 1.0/4*(u[up]+u[down]+u[left]+u[right] - q[xy])

kernels = device.compile(constants=dict(n=n))

for k in xrange(1000):
    kernels.solve(device.queue, [n,n], None, w_buffer, u_buffer, q_buffer)
    (u_buffer, w_buffer) = (w_buffer, u_buffer)

u = device.retrieve(u_buffer,shape=(n,n))

Canvas().imshow(u).save(filename='plot.png')
```

The output is shown in fig. 8.6.2.

One can pass constants to the kernel using

```
device.compile(..., constants = dict(n = n))
```

One can also pass include statements to the kernel:

```
device.compile(..., includes = ['#include <math.h>'])
```

where `includes` is a list of `#include` statements.

Notice how the kernel is line by line the same as the original C code. An important part of the new code is the `define_kernel` decorator. It tells `ocl` that the code must be translated to C99. It also declares the type of each argument, for example,

```
...define_kernel(... u='global:const:ptr_float' ...)
```

It means that:

```
global const float* u
```

Because in C, one must declare the type of each new variable, we must do the same in `ocl`. This is done using the pseudo-casting operators `new_int`, `new_float`, and so on. For example,

```
a = new_int(b+c)
```

is converted into

```
int a = b+c;
```

The converter also checks the types for consistency. The return type is determined automatically from the type of the object that is returned. Python objects that have no C99 equivalent like lists, tuples, dictionaries, and sets are not supported. Other types are converted based on the following table:

ocl	**C99/OpenCL**
`a = new_`*type*`(...)`	*type* `a = ...;`
`a = new_prt_`*type*`(...)`	*type* `*a = ...;`
`a = new_prt_prt_`*type*`(...)`	*type* `**a = ...;`
`None`	`null`
`ADDR(x)`	`&x`
`REFD(x)`	`*x`
`CAST(prt_`*type*`,x)`	`(`*type*`*)x`

8.6.3 Portfolio optimization (in parallel)

In a previous chapter, we provided an algorithm for portfolio optimization. One critical step of that algorithm was the knowledge of all-to-all correlations among stocks. This step can efficiently be performed on a GPU.

In the following example, we solve the same problem again. For each time series `k`, we compute the arithmetic daily returns, `r[k,t]`, and the average returns, `mu[k]`. We then compute the covariance matrix, `cov[i,j]`, and the correlation matrix, `cor[i,j]`. We use different kernels for each part of the computation.

Finally, to make the application more practical, we use MPT [34] to compute a tangency portfolio that maximizes the Sharpe ratio under the assumption of Gaussian returns:

$$\max_x \frac{\mu^T x - r_{\text{free}}}{\sqrt{x^T \Sigma x}} \tag{8.21}$$

Here μ is the vector of average returns (`mu`), Σ is the covariance matrix (`cov`), and r_{free} is the input risk-free interest rate. The tangency portfolio is identified by the vector x (array `x` in the code) whose terms indicate the amount to be invested in each stock (must add up to \$1). We perform this maximization on the CPU to demonstrate integration with the `numpy` linear algebra package.

We use the symbols `i` and `j` to identify the stock time series and the symbol `t` for time (for daily data `t` is a day); `n` is the number of stocks, and `m` is the number of trading days.

We use the `canvas` [11] library, based on the Python `matplotlib` library, to display one of the stock price series and the resulting correlation matrix. Following is the complete code. The output from the code can be seen in fig. 8.6.3.

```python
from ocl import Device
from canvas import Canvas
import random
import numpy
from math import exp

n = 1000 # number of time series
m = 250  # number of trading days for time series
p = numpy.zeros((n,m), dtype=numpy.float32)
r = numpy.zeros((n,m), dtype=numpy.float32)
mu = numpy.zeros(n, dtype=numpy.float32)
cov = numpy.zeros((n,n), dtype=numpy.float32)
cor = numpy.zeros((n,n), dtype=numpy.float32)

for k in xrange(n):
    p[k,0] = 100.0
    for t in xrange(1,m):
        c = 1.0 if k==0 else (p[k-1,t]/p[k-1,t-1])
        p[k,t] = p[k,t-1]*exp(random.gauss(0.0001,0.10))*c

device = Device()
p_buffer = device.buffer(source=p, mode=device.flags.READ_ONLY)
r_buffer = device.buffer(source=r)
mu_buffer = device.buffer(source=mu)
cov_buffer = device.buffer(source=cov)
cor_buffer = device.buffer(source=cor)

@device.compiler.define_kernel(p='global:const:ptr_float',
                               r='global:ptr_float')
def compute_r(p, r):
    i = new_int(get_global_id(0))
    for t in xrange(0,m-1):
        r[i*m+t] = p[i*m+t+1]/p[i*m+t] - 1.0

@device.compiler.define_kernel(r='global:ptr_float',
                               mu='global:ptr_float')
def compute_mu(r, mu):
    i = new_int(get_global_id(0))
    sum = new_float(0.0)
    for t in xrange(0,m-1):
        sum = sum + r[i*m+t]
    mu[i] = sum/(m-1)

@device.compiler.define_kernel(r='global:ptr_float',
      mu='global:ptr_float', cov='global:ptr_float')
def compute_cov(r, mu, cov):
    i = new_int(get_global_id(0))
    j = new_int(get_global_id(1))
    sum = new_float(0.0)
    for t in xrange(0,m-1):
        sum = sum + r[i*m+t]*r[j*m+t]
    cov[i*n+j] = sum/(m-1)-mu[i]*mu[j]

@device.compiler.define_kernel(cov='global:ptr_float',
                               cor='global:ptr_float')
def compute_cor(cov, cor):
    i = new_int(get_global_id(0))
    j = new_int(get_global_id(1))
    cor[i*n+j] = cov[i*n+j] / sqrt(cov[i*n+i]*cov[j*n+j])

program = device.compile(constants=dict(n=n,m=m))

q = device.queue
program.compute_r(q, [n], None, p_buffer, r_buffer)
program.compute_mu(q, [n], None, r_buffer, mu_buffer)
program.compute_cov(q, [n,n], None, r_buffer, mu_buffer, cov_buffer)
program.compute_cor(q, [n,n], None, cov_buffer, cor_buffer)

r = device.retrieve(r_buffer,shape=(n,m))
mu = device.retrieve(mu_buffer,shape=(n,))
cov = device.retrieve(cov_buffer,shape=(n,n))
cor = device.retrieve(cor_buffer,shape=(n,n))

points = [(x,y) for (x,y) in enumerate(p[0])]
Canvas(title='Price').plot(points).save(filename='price.png')
Canvas(title='Correlations').imshow(cor).save(filename='cor.png')

rf = 0.05/m # input daily risk free interest rate
x = numpy.linalg.solve(cov,mu-rf) # cov*x = (mu-rf)
x *= 1.00/sum(x) # assumes 1.00 dollars in total investment
open('optimal_portfolio','w').write(repr(x))
```

Notice how the memory buffers are always one-dimensional, therefore the `i,j` indexes have to be mapped into a one-dimensional index `i*n+j`. Also notice that while kernels `compute_r` and `compute_mu` are called `[n]` times (once per stock `k`), kernels `compute_cov` and `compute_cor` are called `[n,n]` times, once per each couple of stocks `i,j`. The values of `i,j` are retrieved by `get_global_id(0)` and `(1)`, respectively.

In this program, we have defined multiple kernels and complied them at once. We call one kernel at the time to make sure that the call to the previous kernel is completed before running the next one.

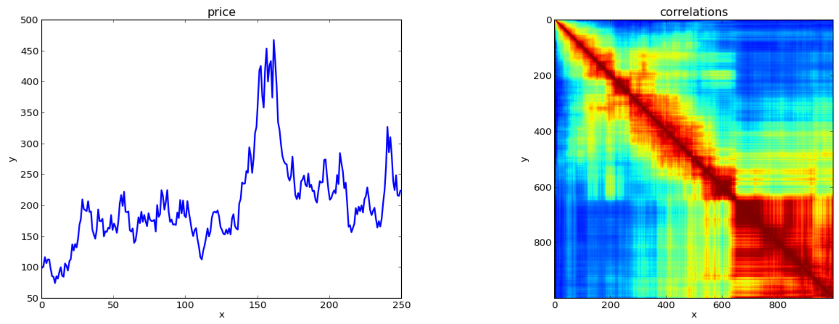

Figure 8.8: The image on the left shows one of the randomly generated stock price histories. The image on the right represents the computed correlation matrix. Rows and columns correspond to stock returns, and the color at the intersection is their correlation (red for high correlation and blue for no correlation). The resulting shape is an artifact of the algorithm used to generate random data.

9
Appendices

9.1 Appendix A: Math Review and Notation

9.1.1 Symbols

∞	infinity
$\wedge$	and
$\vee$	or
$\cap$	intersection
$\cup$	union
$\in$	element or In
$\forall$	for each
$\exists$	exists
$\Rightarrow$	implies
:	such that
iff	if and only if

(9.1)

9.1.2 Set theory

Important sets

$\mathbf{0}$	empty set
$\mathbb{N}$	natural numbers {0,1,2,3,...}
$\mathbb{N}^+$	positive natural numbers {1,2,3,...}
$\mathbb{Z}$	all integers {...,-3,-2,-1,0,1,2,3,...}
$\mathbb{R}$	all real numbers
$\mathbb{R}^+$	positive real numbers (not including 0)
$\mathbb{R}^0$	positive numbers including 0

(9.2)

Set operations

$\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are some generic sets.

- **Intersection**

$$\mathcal{A} \cap \mathcal{B} \equiv \{x : x \in \mathcal{A} \text{ and } x \in \mathcal{B}\} \tag{9.3}$$

- **Union**

$$\mathcal{A} \cup \mathcal{B} \equiv \{x : x \in \mathcal{A} \text{ or } x \in \mathcal{B}\} \tag{9.4}$$

- **Difference**

$$\mathcal{A} - \mathcal{B} \equiv \{x : x \in \mathcal{A} \text{ and } x \notin \mathcal{B}\} \tag{9.5}$$

Set laws

- Empty set laws

$$\mathcal{A} \cup \mathbf{0} = \mathcal{A} \tag{9.6}$$
$$\mathcal{A} \cap \mathbf{0} = \mathbf{0} \tag{9.7}$$

- Idempotency laws

$$\mathcal{A} \cup \mathcal{A} = \mathcal{A} \tag{9.8}$$
$$\mathcal{A} \cap \mathcal{A} = \mathcal{A} \tag{9.9}$$

- Commutative laws

$$\mathcal{A} \cup \mathcal{B} = \mathcal{B} \cup \mathcal{A} \quad (9.10)$$
$$\mathcal{A} \cap \mathcal{B} = \mathcal{B} \cap \mathcal{A} \quad (9.11)$$

- Associative laws

$$\mathcal{A} \cup (\mathcal{B} \cup \mathcal{C}) = (\mathcal{A} \cup \mathcal{B}) \cup \mathcal{C} \quad (9.12)$$
$$\mathcal{A} \cap (\mathcal{B} \cap \mathcal{C}) = (\mathcal{A} \cap \mathcal{B}) \cap \mathcal{C} \quad (9.13)$$

- Distributive laws

$$\mathcal{A} \cap (\mathcal{B} \cup \mathcal{C}) = (\mathcal{A} \cap \mathcal{B}) \cup (\mathcal{A} \cap \mathcal{C}) \quad (9.14)$$
$$\mathcal{A} \cup (\mathcal{B} \cap \mathcal{C}) = (\mathcal{A} \cup \mathcal{B}) \cap (\mathcal{A} \cup \mathcal{C}) \quad (9.15)$$

- Absorption laws

$$\mathcal{A} \cap (\mathcal{A} \cup \mathcal{B}) = \mathcal{A} \quad (9.16)$$
$$\mathcal{A} \cup (\mathcal{A} \cap \mathcal{B}) = \mathcal{A} \quad (9.17)$$

- DeMorgan laws

$$\mathcal{A} - (\mathcal{B} \cup \mathcal{C}) = (\mathcal{A} - \mathcal{B}) \cap (\mathcal{A} - \mathcal{C}) \quad (9.18)$$
$$\mathcal{A} - (\mathcal{B} \cap \mathcal{C}) = (\mathcal{A} - \mathcal{B}) \cup (\mathcal{A} - \mathcal{C}) \quad (9.19)$$

More set definitions

- $\mathcal{A}$ is a **subset** of $\mathcal{B}$ iff $\forall x \in \mathcal{A}, x \in \mathcal{B}$
- $\mathcal{A}$ is a **proper subset** of $\mathcal{B}$ iff $\forall x \in \mathcal{A}, x \in \mathcal{B}$ and $\exists x \in \mathcal{B}, x \notin \mathcal{A}$
- $P = \{S_i, i = 1, ..., N\}$ (a set of sets S_i) is a **partition** of $\mathcal{A}$ iff $S_1 \cup S_2 \cup ... \cup S_N = \mathcal{A}$ and $\forall i, j, S_i \cap S_j = \mathbf{0}$
- The number of elements in a set $\mathcal{A}$ is called the **cardinality** of set $\mathcal{A}$.
- cardinality($\mathbb{N}$)=countable infinite (∞)
- cardinality($\mathbb{R}$)=uncountable infinite (∞) !!!

Relations

- A **Cartesian Product** is defined as

$$\mathcal{A} \times \mathcal{B} = \{(a, b) : a \in \mathcal{A} \text{ and } b \in \mathcal{B}\} \tag{9.20}$$

- A **binary relation** R between two sets $\mathcal{A}$ and $\mathcal{B}$ if a subset of their Cartesian product.
- A binary relation is **transitive** is aRb and bRc implies aRc
- A binary relation is **symmetric** if aRb implies bRa
- A binary relation is **reflexive** if aRa if always true for each a.

Examples:

- $a < b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive)
- $a > b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive)
- $a = b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive, symmetric and reflexive)
- $a \leq b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive, and reflexive)
- $a \geq b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive, and reflexive)
- A relation R that is transitive, symmetric and reflexive is called an **equivalence relation** and is often indicated with the notation $a \sim b$.

An equivalence relation is the same as a partition.

Functions

- A **function** between two sets $\mathcal{A}$ and $\mathcal{B}$ is a binary relation on $\mathcal{A} \times \mathcal{B}$ and is usually indicated with the notation $f : \mathcal{A} \longmapsto \mathcal{B}$
- The set $\mathcal{A}$ is called **domain** of the function.
- The set $\mathcal{B}$ is called **codomain** of the function.
- A function **maps** each element $x \in \mathcal{A}$ into an element $f(x) = y \in \mathcal{B}$
- The **image** of a function $f : \mathcal{A} \longmapsto \mathcal{B}$ is the set $\mathcal{B}' = \{y \in \mathcal{B} : \exists x \in \mathcal{A}, f(x) = y\} \subseteq \mathcal{B}$

- If $\mathcal{B}'$ is $\mathcal{B}$ then a function is said to be **surjective**.
- If for each x and x' in $\mathcal{A}$ where $x \neq x'$ implies that $f(x) \neq f(x')$ (e.g., if not two different elements of $\mathcal{A}$ are mapped into different element in $\mathcal{B}$) the function is said to be a **bijection**.
- A function $f : \mathcal{A} \longmapsto \mathcal{B}$ is **invertible** if it exists a function $g : \mathcal{B} \longmapsto \mathcal{A}$ such that for each $x \in \mathcal{A}, g(f(x)) = x$ and $y \in \mathcal{B}, f(g(y)) = y$. The function g is indicated with f^{-1}.
- A function $f : \mathcal{A} \longmapsto \mathcal{B}$ is a surjection and a bijection iff f is an invertible function.

Examples:

- $f(n) \equiv n\text{mod}2$ with domain $\mathbb{N}$ and codomain $\mathbb{N}$ is not a surjection nor a bijection.
- $f(n) \equiv n\text{mod}2$ with domain $\mathbb{N}$ and codomain $\{0, 1\}$ is a surjection but not a bijection
- $f(x) \equiv 2x$ with domain $\mathbb{N}$ and codomain $\mathbb{N}$ is not a surjection but is a bijection (in fact it is not invertible on odd numbers)
- $f(x) \equiv 2x$ with domain $\mathbb{R}$ and codomain $\mathbb{R}$ is not a surjection and is a bijection (in fact it is invertible)
-

9.1.3 Logarithms

If $x = a^y$ with $a > 0$, then $y = \log_a x$ with domain $x \in (0, \infty)$ and codomain $y = (-\infty, \infty)$. If the base a is not indicated, the natural log $a = e = 2.7183...$ is assumed.

Properties of logarithms:

$$\begin{aligned}
\log_a x &= \frac{\log x}{\log a} && (9.21)\\
\log xy &= (\log x) + (\log y) && (9.22)\\
\log \frac{x}{y} &= (\log x) - (\log y) && (9.23)\\
\log x^n &= n \log x && (9.24)
\end{aligned}$$

9.1.4 Finite sums

Definition

$$\sum_{i=0}^{i<n} f(i) \equiv f(0) + f(1) + ... + f(n-1) \qquad (9.25)$$

Properties

- **Linearity I**

$$\begin{aligned}
\sum_{i=0}^{i\leq n} f(i) &= \sum_{i=0}^{i<n} f(i) + f(n) && (9.26)\\
\sum_{i=a}^{i\leq b} f(i) &= \sum_{i=0}^{i\leq b} f(i) - \sum_{i=0}^{i<a} f(i) && (9.27)
\end{aligned}$$

- **Linearity II**

$$\sum_{i=0}^{i<n} af(i) + bg(i) = a\left(\sum_{i=0}^{i<n} f(i)\right) + b\left(\sum_{i=0}^{i<n} g(i)\right) \qquad (9.28)$$

Proof:

$$\begin{aligned}
\sum_{i=0}^{i<n} af(i) + bg(i) &= (af(0) + bg(0)) + ... + (af(n-1) + bg(n-1)) \\
&= af(0) + ... + af(n-1) + bg(0) + ... + bg(n-1) \\
&= a\,(f(0) + ... + f(n-1)) + b\,(g(0) + ... + g(n-1)) \\
&= a\left(\sum_{i=0}^{i<n} f(i)\right) + b\left(\sum_{i=0}^{i<n} g(i)\right) \qquad (9.29)
\end{aligned}$$

Examples:

$$\begin{aligned}
\sum_{i=0}^{i<n} c &= cn \text{ for any constant } c \qquad (9.30) \\
\sum_{i=0}^{i<n} i &= \frac{1}{2}n(n-1) \qquad (9.31) \\
\sum_{i=0}^{i<n} i^2 &= \frac{1}{6}n(n-1)(2n-1) \qquad (9.32) \\
\sum_{i=0}^{i<n} i^3 &= \frac{1}{4}n^2(n-1)^2 \qquad (9.33) \\
\sum_{i=0}^{i<n} x^i &= \frac{x^n-1}{x-1} \text{ (geometric sum)} \qquad (9.34) \\
\sum_{i=0}^{i<n} \frac{1}{i(i+1)} &= 1 - \frac{1}{n} \text{ (telescopic sum)} \qquad (9.35)
\end{aligned}$$

9.1.5 Limits ($n \to \infty$)

In these section we will only deal with limits ($n \to \infty$) of positive functions.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = ? \qquad (9.36)$$

First compute limits of the numerator and denominator separately:

$$\lim_{n\to\infty} f(n) = a \tag{9.37}$$

$$\lim_{n\to\infty} g(n) = b \tag{9.38}$$

- If $a \in \mathbb{R}$ and $b \in \mathbb{R}^+$ then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{a}{b} \tag{9.39}$$

- If $a \in \mathbb{R}$ and $b = \infty$ then

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0 \tag{9.40}$$

- If $(a \in \mathbb{R}^+$ and $b = 0)$ or $(a = \infty$ and $b \in \mathbb{R})$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \tag{9.41}$$

- If $(a = 0$ and $b = 0)$ or $(a = \infty$ and $b = \infty)$ use de l'Hopital rule

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)} \tag{9.42}$$

  and start again!
- Else ... the limit does not exist (typically oscillating functions or non-analytic functions).

For any $a \in \mathbb{R}$ or $a = \infty$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = a \Rightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} = 1/a \tag{9.43}$$

Table of derivatives

$f(x)$	$f'(x)$
c	0
ax^n	anx^{n-1}
$\log x$	$\frac{1}{x}$
e^x	e^x
a^x	$a^x \log a$
$x^n \log x, n > 0$	$x^{n-1}(n \log x + 1)$

(9.44)

Practical rules to compute derivatives

$$\frac{d}{dx}\left(f(x)+g(x)\right) = f'(x)+g'(x) \tag{9.45}$$

$$\frac{d}{dx}\left(f(x)-g(x)\right) = f'(x)-g'(x) \tag{9.46}$$

$$\frac{d}{dx}\left(f(x)g(x)\right) = f'(x)g(x)+f(x)g'(x) \tag{9.47}$$

$$\frac{d}{dx}\left(\frac{1}{f(x)}\right) = -\frac{f'(x)}{f(x)^2} \tag{9.48}$$

$$\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) = \frac{f'(x)}{g(x)} - \frac{f(x)g'(x)}{g(x)^2} \tag{9.49}$$

$$\frac{d}{dx}f(g(x)) = f'(g(x))g'(x) \tag{9.50}$$