

**Topics in Parallel and Distributed Computing:
Introducing Concurrency in Undergraduate Courses^{1,2}**

**Chapter 10
Parallel Programming Illustrated Through Conway's
Game of Life**

Victor Eijkhout
University of Texas, Austin

¹How to cite this book: Prasad, Gupta, Rosenberg, Sussman, and Weems. *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses*, 1st Edition, Morgan Kaufmann, ISBN : 9780128038994, Pages: 360.

²Free preprint version of the CDER book: http://grid.cs.gsu.edu/~tcpp/curriculum/?q=cedr_book.

TABLE OF CONTENTS

| | |
|--|------------|
| LIST OF FIGURES | 448 |
| CHAPTER 10 PARALLEL PROGRAMMING ILLUSTRATED THROUGH CONWAY’S GAME OF LIFE | 449 |
| 10.1 Introduction | 449 |
| 10.1.1 Conway’s Game of Life | 450 |
| 10.1.2 Programming the Game of Life | 451 |
| 10.1.3 General thoughts on parallelism | 453 |
| 10.2 Parallel variants | 454 |
| 10.2.1 Data parallelism | 454 |
| 10.2.2 Loop-based parallelism | 460 |
| 10.2.3 Coarse-grained data parallelism | 462 |
| 10.3 Advanced topics | 470 |
| 10.3.1 Data partitioning | 470 |
| 10.3.2 Combining work, minimizing communication | 474 |
| 10.3.3 Load balancing | 475 |
| 10.4 Summary | 476 |
| 10.5 List of acronyms | 477 |

LIST OF FIGURES

| | | |
|--------------|---|-----|
| Figure 10.1 | Illustration of data parallelism: all points of the board get the same update treatment | 454 |
| Figure 10.2 | Four-wide vector instructions work on four operand pairs at the same time | 456 |
| Figure 10.3 | Illustration of shared memory: all processors access the same memory | 462 |
| Figure 10.4 | Illustration of distributed memory: every processor has its own memory and is connected to others through a network | 463 |
| Figure 10.5 | The Stampede supercomputer at the Texas Advanced Supercomputing Center | 464 |
| Figure 10.6 | Processor p receives a line of data from $p - 1$ and $p + 1$ | 465 |
| Figure 10.7 | Illustration of ‘ideal’ and ‘blocking’ send | 467 |
| Figure 10.8 | One-dimensional and two-dimensional distribution communication | 471 |
| Figure 10.9 | One-dimensional and two-dimensional distribution of a Life board | 473 |
| Figure 10.10 | Two steps of Life updates | 475 |

CHAPTER 10

PARALLEL PROGRAMMING ILLUSTRATED THROUGH CONWAY'S GAME OF LIFE

10.1 Introduction

There are many ways to approach parallel programming. Of course you need to start with the problem that you want to solve, but after that there can be more than one algorithm for that problem, you may have a choice of programming systems to use to implement that algorithm, and finally you have to consider the hardware that will run the software. Sometimes people will argue that certain problems are best solved on certain types of hardware or with certain programming systems. Whether this is so is indeed a question worth discussing, but hard to assess in all its generality.

In this tutorial we will look at one particular problem, Conway's *Game of Life*¹, and investigate how that is best implemented using different parallel programming systems and different hardware. That is, we will see how different types of parallel programming can all be used to solve the same problem. In the process, you will learn about most of the common parallel programming models and their characteristics.

This tutorial does not teach you to program in any particular system: you will only deal with *pseudo-code* and not run it on actual hardware. However, the discussion will go into detail on the implications of using different types of parallel computers.

(Note. At some points in this discussion there will be references to the book 'Introduction to High-Performance Scientific Computing' by the present author². Such references take the form 'HPSC-1.2.3' for section 1.2.3 of that book.)

¹Martin Gardner, 'Mathematical Games – the fantastic combinations of John Conway's new solitaire game Life', *Scientific American* 223, October 1970, pp 120–123.

²Victor Eijkhout, with Robert van de Geijn and Edmond Chow, *Introduction to High Performance Scientific Computing*, 2011.

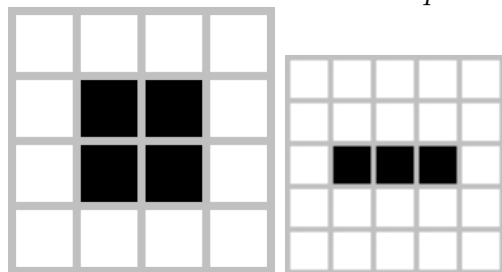
10.1.1 Conway's Game of Life

The Game of Life takes place on a two-dimensional board of *cells*. Each cell can be alive or dead, and it can switch its status from alive to dead or the other way around once per time interval, let's say a second. The rules for cells are as follows. In each time step, each cell counts how many live neighbours it has, where a neighbour is a cell that borders on it horizontally, vertically, or diagonally. Then:

- If a cell is alive, and it has fewer than two live neighbours, it dies of loneliness.
- A live cell with more than three live neighbours dies from overcrowding.
- A live cell with two or three live neighbours lives on to the next generation.
- A dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The 'game' is that you create an initial configuration of live cells, and then stand back and see what happens.

Exercise 1. Here are two simple Life configurations.



Go through the rules and show that the first figure is stationary, and the second figure morphs into something, then morphs back.

The Game of Life is hard to illustrate in a book, since it's so dynamic. If you search online you can find some great animations.

The rules of Life are very simple, but the results can be surprising. For instance, some simple shapes, called 'gliders', seem to move over the board; others, called 'puffers', move over the board leaving behind other groups of cells. Some configurations of cells quickly

disappear, others stay the same or alternate between a few shapes; for a certain type of configuration, called 'garden of Eden', you can prove that it could not have evolved from an earlier configuration. Probably most surprisingly, Life can simulate, very slowly, a computer!

10.1.2 Programming the Game of Life

It is not hard to write a program for Life. Let's say we want to compute a certain number of time steps, and we have a square board of $N \times N$ cells. Also assume that we have a function `life_evaluation` that takes a 3×3 cells and returns the updated status of the center cell³:

```
def life_evaluation( cells ):
    # cells is a 3x3 array
    count = 0
    for i in [0,1,2]:
        for j in [0,1,2]:
            if i!=1 and j!=1:
                count += cells[i,j]
    return life_count_evaluation( cells[1,1],count )

def life_count_evaluation( cell,count )
    if count<2:
        return 0 # loneliness
    elif count>3:
        return 0 # overcrowding
    elif cell==1 and (count==2 or count==3):
        return 1 # live on
    elif cell==0 and count==3:
        return 1 # spontaneous generation
    else:
```

³We use a quasi-python syntax here, except that in arrays we let the upper bound be inclusive.

```
return 0 # no change in dead cells
```

The driver code would then be something like:

```
# create an initial board; we omit this code
life_board.create(final_time,N,N)

# now iterate a number of steps on the whole board
for t in [0:final_time-1]:
    for i in [0:N-1]:
        for j in [0:N-1]:
            life_board[t+1,i,j] =
                life_evaluation( life_board[t,i-1:i+1,j-1:j+1] )
```

where we don't worry too much about the edge of the board; we can for instance declare that points outside the range $0 \dots N - 1$ are always dead.

The above code creates a board for each time step, which is not strictly necessary. You can save yourself some space by creating only two boards:

```
life_board.create(N,N)
temp_board.create(N,N)

for t in [0:final_time-1]:
    life_generation( life_board,temp_board )

def life_generation( board,tmp ):
    for i in [0:N-1]:
        for j in [0:N-1]:
            tmp[i,j] = board[i,j]
    for i in [0:N-1]:
        for j in [0:N-1]:
```

```
board[i,j] = life_evaluation( tmp[i-1:i+1,j-1:j+1] )
```

We will call this the basic *sequential implementation*, since it does its computation in a long sequence of steps. We will now explore parallel implementations of this algorithm. You'll see that some look very different from this basic code.

Exercise 2. The second version used a whole temporary board. Can you come up with an implementation that uses just three temporary lines?

10.1.3 General thoughts on parallelism

In the rest of this tutorial we will use various types of parallelism to explore coding the Game of Life. We start with data parallelism, based on the observation that each point in a Life board undergoes the same computation. Then we go on to task parallelism, which is necessary when we start looking at distributed memory programming on large clusters. But first we start with some basic thoughts on parallelism.

If you're familiar with programming, you'll have read the above code fragments and agreed that this is a good way to solve the problem. You do one time step after another, and at each time step you compute a new version of the board, one line after another.

Most programming languages are very explicit about loop constructs: one iteration is done, and then the next, and the next, and so on. This works fine if you have just one processor. However, if you have some form of parallelism, meaning that there is more than one processing unit, you have to figure out which things really have to be done in sequence, and where the sequence is more an artifact of the programming language.

And by the way, *you* have to think about this yourself. In a distant past it was thought that programmers could write ordinary code, and the compiler would figure out parallelism. This has long proved impossible except in limited cases, so programmers these days accept that parallel code will look differently from sequential code, sometimes very much so.

So let's start looking at Life from a point of analyzing the parallelism. The Life program above used three levels of loops: one for the time steps, and two for the rows and columns of the board. While this is a correct way of programming Life, such explicit sequencing of

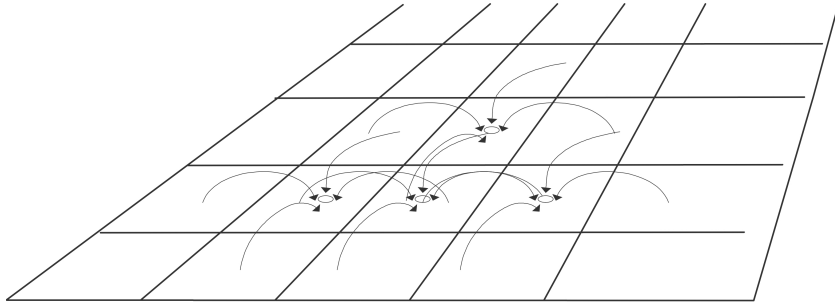


Figure (10.1) Illustration of data parallelism: all points of the board get the same update treatment

loop iterations is not strictly necessary for solving the Game of Life problem. For instance, all the cells in the new board are the result of independent computations, and so they can be executed in any order, or indeed simultaneously.

You can view parallel programming as the problem of how to tell multiple processors that they can do certain things simultaneously, and other things only in sequence.

10.2 Parallel variants

We will now discuss various specific parallel realizations of Life.

10.2.1 Data parallelism

In the sequential reference code for Life we updated the whole board in its entirety before we proceeded to the next step. That is, we did the time steps sequentially. We also observed that, in each time step, all cells can be updated independently, and therefore in parallel. If parallelism comes in such small chunks, we call it *data parallelism* or *fine-grained parallelism*: the parallelism comes from having lots of data points that are all treated identically. This is illustrated in figure 10.1.

The fine-grained data parallel model of computing is known as Single Instruction Multiple Data (SIMD): the same instruction is performed on multiple data elements. An actual computer will of course not have an instruction for computing a Life cell update. Rather, its instructions are things like additions and multiplications. Thus, you may need to restructure

your code a little for SIMD execution.

A parallel computer that is designed for doing lots of identical operations (on different data elements, of course) has certain advantages. For instance, there needs to be only one central instruction decoding unit that tells the processors what to do, so the design of the individual processors can be much simpler. This means that the processors can be smaller, more power efficient, and easier to manufacture.

In the 1980s and 1990s SIMD computers existed, such as the MasPar and the Connection Machine. They were sometimes called *array processors* since they could operate on an array of data simultaneously, up to 2^{16} elements. These days, SIMD still exists, but in slightly different guises and on much smaller scale; we will now explore what SIMD parallelism looks like in current architectures.

Vector instructions Modern processors have embraced the SIMD concept in an attempt to gain performance without complicating the processor design too much. Instead of operating on a single pair of inputs, you would load two or more pairs of operands, and execute multiple identical operations simultaneously.

Vector instructions constitute SIMD parallelism on a much smaller scale than the old array processors. For instance, Intel processors have had SIMD Streaming Extensions (SSE) instructions for quite some time, which are described as ‘two-wide’ since they work on two sets of (double precision floating point) operands. The current generation of Intel vector instructions is called Advanced Vector Extensions (AVX), and they can be up to ‘eight-wide’; see figure 10.2 for an illustration of four-wide instructions. Since with these instructions you can do four or eight operations per clock cycle, it becomes important to write your code such that the processor can actually use all that available parallelism.

Now suppose that you are coding the Game of Life, which is SIMD in nature, and you want to make sure that is executed with these vector instructions.

First of all the code needs to have the right structure. The original code does not have a lot of parallelism in the inner loop, where it can be exploited with vector instruction:

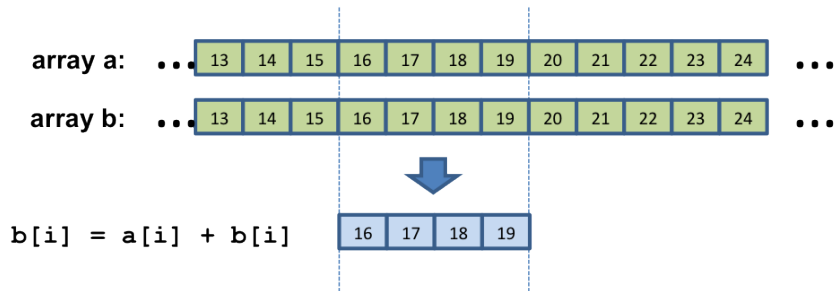


Figure (10.2) Four-wide vector instructions work on four operand pairs at the same time

```
for i in [0:N]:
  for j in [0:N]:
    count = 0
    for ii in {-1,0,+1}:
      for jj in {-1,0,+1}:
        if ii!=0 and jj!=0:
          count += board[i+ii,j+jj]
```

Instead, we have to exchange loops as:

```
for i in [0:N]:
  for j in [0:N]:
    count[j] = 0
    for ii in {-1,0,+1}:
      for jj in {-1,0,+1}:
        if ii!=0 and jj!=0:
          for j in [0:N]:
            count[j] += board[i+ii,j+jj]
```

Note that the `count` variable now has become an array. This is one of the reasons that compilers are unable to make this transformation.

Regular programming languages have no way of saying ‘do the following operation with vector instructions’. That leaves you with two options:

1. You can start coding in assembly language, or use your compiler's facility for using 'in-line assembly'; or
2. You can hope that the compiler understands your code enough to generate the vector instructions for you.

The first option is no fun, and beyond the capabilities of most programmers, so you'll probably rely on the compiler.

Compilers are pretty smart, but they cannot read your mind. If your code is too sophisticated, they may not figure out that vector instructions can be used. On the other hand, you can sometimes help the compiler. For instance, the operation

```
for i in [0:N]:  
    count[i,j] += board[i,j+1]
```

can be written as

```
for ii in [0:N/2]:  
    i = 2*ii  
    count[i,j] += board[i,j+1]  
    count[i+1,j] += board[i+1,j+1]
```

Here we perform half the number of iterations, but each new iteration comprises two old ones. In this version the compiler will have no trouble concluding that there are two operations that can be done simultaneously. This transformation of a loop is called *loop unrolling*, in this case, unrolling by 2.

Exercise 3. The second code is not actually equivalent to the first. (Hint: consider the case that N is odd.) How can you repair that code? One way of repairing this code is to add a few lines of 'clean-up code' after the unrolled loop. Give the pseudo-code for this.

Now consider the case of unrolling by 4. What does the unrolled code look like now? Think carefully about the clean-up code.

Vector pipelining In the previous section you saw that modern CPUs can deal with applying the same operation to a sequence of data elements. In the case of vector instructions (above), or in the case of GPUs (next section), these identical operations are actually done simultaneously. In this section we will look at *pipelining*, which is a different way of dealing with identical instructions.

Imagine a car being put together on an assembly line: as the frame comes down the line one worker puts on the wheels, another the doors, another puts on the steering wheel, et cetera. Thus, the final product, a car, is gradually being constructed; since more than one car is being worked on simultaneously, this is a form of parallelism. And while it is possible for one worker to go through all these steps until the car is finished, it is more efficient to let each worker specialize in just one of the partial assembly operations.

We can do a similar story for computations in a CPU. Let's say we're dealing with floating point numbers of the form $a.b \times 10^c$. Now if we add 5.8×10^1 and 3.2×10^2 , we

1. first bring them to the same power of ten: $0.58 \times 10^2 + 3.2 \times 10^2$,
2. do the addition: 3.88×10^2 ,
3. round to get rid of that last decimal place: 3.9×10^2

So now we can apply the assembly line principle to arithmetic: we can let the processor do each piece in sequence, but a long time ago it was recognized that operations can be split up like that, letting the sub-operations take place in different parts of the processor. The processor can now work on multiple operations at the same time: we start the first operation, and while it is under way we can start a second one, et cetera. In the context of computer arithmetic we call this assembly line the *pipeline*.

If the pipeline has four stages, after filling the pipeline there will be four operations partially completed at any time. Thus, the pipeline operation is roughly equivalent to, in this example, a fourfold parallelism. You would hope that this corresponds to a fourfold speedup; the following exercise lets you analyze this precisely.

Exercise 4. Assume that all the sub-operations take the same amount of time t . If there are s sub-operations (and assume $s > 1$), how much time does it take for one full calculation? And how much time for two? Recognize that the time for two operations is less than twice the time for a single operation, since the second is started while the first is still in progress.

How much time does it take to do n operations? How much time would n operations take if the processor was not pipelined? What is the asymptotic improvement in speed of a pipelined processor over a non-pipelined one?

Around the 1970s this was the definition of a supercomputer: a machine with a single processor that could do floating point operations several times faster than other processors, as long as these operations were delivered as a stream of identical operations. This type of supercomputer essentially died out in the 1990s, but by that time micro-processors had become so sophisticated that they started to include pipelined arithmetic. So the idea of pipelining lives on.

Pipelining has similarities with array operations as described above: they both apply to sequences of identical operations, and they both apply the same operation to all operands. Because of this, pipelining is sometimes also considered *SIMD*.

GPUs Graphics has always been an important application of computers, since everyone likes to look at pictures. With computer games, the demand for very fast generation of graphics has become even bigger. Since graphics processing is often relatively simple and structured, with for instance the same blur operation executed on each pixel, or the same rendering on each polygon, people have made specialized processors for doing just graphics. These can be cheaper than regular processors, since they only have to do graphics-type of operations, and they take the load off the main CPU of your computer.

Wait. Did we just say ‘the same operation on each pixel/polygon’? That sounds a lot like SIMD, and in fact it is something very close to it.

Starting from the realization that graphics processing has a lot in common with tra-

ditional parallelism, people have tried to use Graphics Processing Units (GPUs) for SIMD-type numerical computations. Doing so was cumbersome, until *NVIDIA* came out with the *Compute Unified Device Architecture (CUDA)* language. CUDA is a way of explicitly doing data parallel programming: you write a piece of code called a *kernel*, which applies to a single data element. You then indicate a two-dimensional or three-dimensional grid of points on which the kernel will be applied.

In pseudo-CUDA, a kernel definition for the game of life and its invocation would look like:

```
kerneldef life_step( board ):
    i = my_i_number()
    j = my_j_number()
    board[i,j] = life_evaluation( board[i-1:i+1,j-1:j+1] )

for t in [0:final_time]:
    <<N,N>>life_step( board )
```

where the $\langle\langle N,N \rangle\rangle$ notation means that the processors should arrange themselves in an $N \times N$ grid. Every processor has a way of telling its own coordinates in that grid.

There are aspects to CUDA that make it different from SIMD, namely its threading, and for this reason NVIDIA uses the term *Single Instruction Multiple Thread (SIMT)*. We won't go into that here. The main purpose of this section was to remark on the similarities between GPU programming and SIMD array programming.

10.2.2 Loop-based parallelism

The above strategies of parallel programming were all based on assigning certain board locations to certain processors. Since the locations on the board can be updated independently, the processors can then all work in parallel.

There is a slightly different way of looking at this. Rather than going back to basics and reasoning about the problem abstractly, you can take the code of the basic, sequential,

implementation of Life. Since the locations can be updated independently, the *iterations* of the loop are *independent* and can be executed in any order, or in fact simultaneously. So is there a way to tell the compiler that the iterations are independent, and let the compiler decide how to execute them?

The popular *OpenMP* system lets the programmer supply this information in comments:

```
def life_generation( board,tmp ):
    # OMP parallel for
    for i in [0:N-1]:
        for j in [0:N-1]:
            tmp[i,j] = board[i,j]
    # OMP parallel for
    for i in [0:N-1]:
        for j in [0:N-1]:
            board[i,j] = life_evaluation( tmp[i-1:i+1,j-1:j+1] )
```

The comments here state that both the `for i` loops are parallel, and therefore their iterations can be executed by whatever parallel resources are available.

In fact, all N^2 iterations of the `i,j` loop nest are independent, which we express as

```
def life_generation( board,tmp ):
    # OMP parallel for collapse(2)
    for i in [0:N-1]:
        for j in [0:N-1]:
            tmp[i,j] = board[i,j]
```

This approach of annotating the loops of a naively written sequential implementation is a good way of getting started with parallelism. However, the structure of the resulting parallel execution may not be optimally suited to a given computer architecture. In the next section we will look at different ways of getting task parallelism, and why they may computationally be preferable.

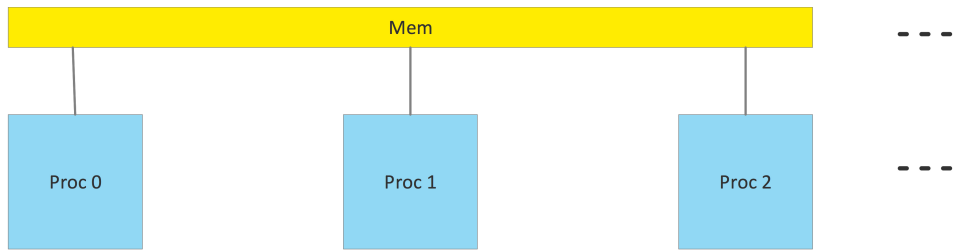


Figure (10.3) Illustration of shared memory: all processors access the same memory

10.2.3 Coarse-grained data parallelism

So far we have looked at implications of the fact that each cell in a step of Life can be updated independently. This view leads to tiny grains of computing, which are a good match to the innermost components of a processor core. However, if you look at parallelism on the level of the cores of a processor there are disadvantages to assigning small-grained computations randomly to the cores. Most of these have something to do with the way memory is structured: moving such small amounts of work around can be more costly than executing them (for a detailed discussion see section HPSC-??). Therefore, we are motivated to look at computations in larger chunks than a single cell update.

For instance, we can divide the Life board in lines or square patches, and formulate the algorithm in terms of operations on such larger units. This is called *coarse-grained parallelism*, and we will look at several variants of it.

Shared memory parallelism In the approaches to parallelism mentioned so far we have implicitly assumed that a processing element can actually get hold of any data element it needs. Or look at it this way: a program has a set of instructions and so far we have assumed that any processor can execute any instruction.

This is certainly the case with multicore processors, where all cores can equally easily read any element from memory. We call this *shared memory*; see figure 10.3.

In the CUDA example each processing element essentially reasoned ‘this is my number, and therefore I will work on this element of the array’. In other words, each processing element assumes that it can work on any data element, and this works because a GPU has

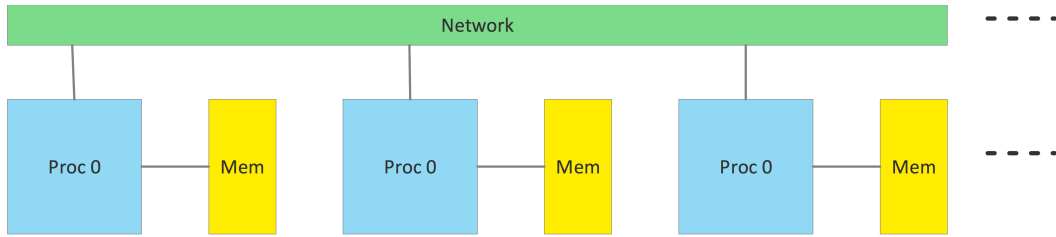


Figure (10.4) Illustration of distributed memory: every processor has its own memory and is connected to others through a network

a form of shared memory.

While it is convenient to program this way, it is not possible to make arbitrarily large computers with shared memory. The shared memory approaches discussed so far are limited by the amount of memory you can put in a single PC, at the moment about 1 terabyte (which costs a lot of money!), or the processing power that you can associate with shared memory, at the moment around 48 cores.

If you need more processing power, you need to look at clusters, and ‘distributed memory programming’.

Distributed memory parallelism *Clusters*, also called *distributed memory* computers, can be thought of as a large number of PCs with network cabling between them. This design can be scaled up to a much larger number of processors than shared memory. In the context of a cluster, each of these PCs is called a *node*. The network can be *Ethernet* or something more sophisticated like *Infiniband*.

Since all nodes work together, a cluster is in some sense one large computer. Since the nodes are also to an extent independent, this type of parallelism is called *Multiple Instruction Multiple Data (MIMD)*: each node has its own data, and executes its own program. However, most of the time the nodes will all execute the same program, so this model is often called *Single Program Multiple Data (SPMD)*; see figure 10.4. The advantage of this design is that tying together thousands of processors allows you to run very large problems. For instance, the almost 13 thousand processors of the Stampede supercomputer⁴ (figure 10.5)

⁴Stampede has more than 6400 nodes, each with 2 Intel Sandy Bridge processors. Each node also has an



Figure (10.5) The Stampede supercomputer at the Texas Advanced Supercomputing Center

have almost 200 terabytes of memory. Parallel programming on such a machine is a little harder than what we discussed above. First of all we have to worry about how to partition the problem over this *distributed memory*. But more importantly, our above assumption that each processing element can get hold of every data element no longer holds.

It is clear that each cluster node can access its local problem data without any problem, but this is not true for the ‘remote’ data on other nodes. In the former case the program simply reads the memory location; in the latter case accessing data is only possible because there is a network between the processors: in the Stampede picture you can see yellow cabling connecting the nodes in each cabinet, and orange cabling overhead that connects the cabinets. Accessing data over the network probably involves an operating system call and accessing the network card, both of which are slow operations.

Distributed memory programming By far the most popular way for programming distributed memory machines is by using the Message Passing Interface (MPI) library. This

Intel Xeon Phi co-processor, but we don’t count those for the moment.

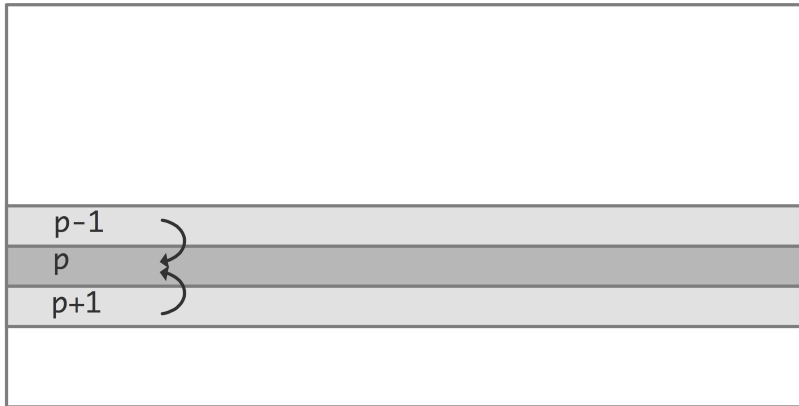


Figure (10.6) Processor p receives a line of data from $p - 1$ and $p + 1$

library adds functionality to an otherwise normal C or Fortran program for exchanging data with other processors. The name derives from the fact that the technical term for exchanging data between distributed memory nodes is *message passing*.

Let's explore how you would program with MPI. We start with the case that each processor stores the cells of a single line of the Life board, and that processor p stores line p . In that case, to update that line it needs the lines above and below it, which come from processors $p - 1$ and $p + 1$ respectively. In MPI terms, the processor needs to receive a message from each of these processors, containing the state of their line.

Let's build up the basic structure of an MPI program. Throughout this example, keep in mind that we are working in SPMD mode: all processes execute the same program. As illustrated in figure 10.6 a process needs to get data from its neighbours. The first step is for each process to find out what its number is, so that it can name its neighbours.

```
p = my_processor_number()
```

Then the process can actually receive data from those neighbours (we ignore complications from the first and last line of the board here).

```
high_line = MPI_Receive(from=p-1,cells=N)
```

```
low_line = MPI_Receive(from=p+1,cells=N)
```

With this, it is possible to update the data stored on this process:

```
tmp_line = my_line.copy()
my_line = life_line_update(high_line,tmp_line,low_line,N)
```

(We omit the code for `life_line_update`, which computes the updated cell values on a single line.) Unfortunately, there is more to MPI than that. The most common way of using the library is through *two-sided communication*, where for each receive action there is a corresponding send action: a process cannot just receive data from its neighbours, the neighbours have to send the data.

But now we recall the SPMD nature of the computation: if your neighbours send to you, you are someone else's neighbour and need to send to them. So the program code will contain both send and receive calls.

The following code is closer to the truth.

```
p = my_processor_number()

# send my data
my_line.MPI_Send(to=p-1,cells=N)
my_line.MPI_Send(to=p+1,cells=N)

# get data from neighbours
high_line = MPI_Receive(from=p-1,cells=N)
low_line = MPI_Receive(from=p+1,cells=N)
tmp_line = my_line.copy()

# do the local computation
my_line = life_line_update(high_line,tmp_line,low_line,N)
```

Since this is a general tutorial, and not a course in MPI programming, we'll leave the example phrased in pseudo-MPI, ignoring many details. However, this code is still not entirely correct conceptually. Let's fix that.

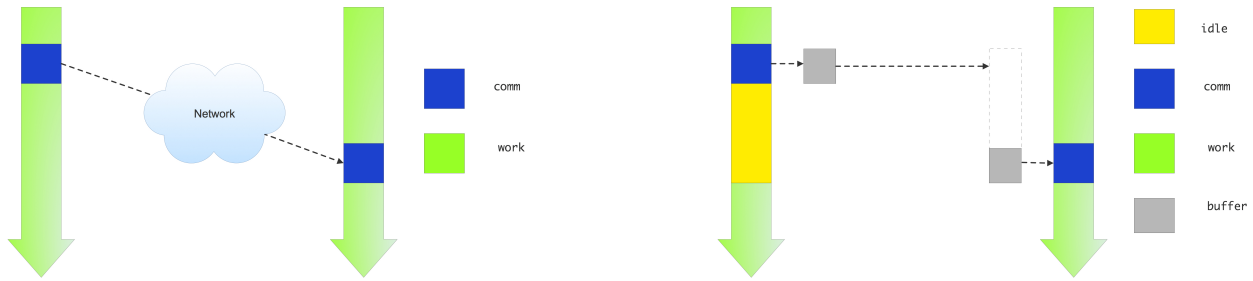


Figure (10.7) Illustration of ‘ideal’ and ‘blocking’ send

Conceptually, a process would send a message, which disappears somewhere in the network, and goes about its business. The receiving process would at some point issue a receive call, get the data from the network, and do something with it. This idealized behaviour is illustrated in the left half of figure 10.7. Practice is different.

Suppose a process sends a large message, something that takes a great deal of memory. Since the only memory in the system is on the processors, the message has to stay in the memory of one processor, until it is copied to the other. We call this behaviour *blocking communication*: a send call will wait until the receiving processor is indeed doing a receive call. That is, the sending code is blocked until its message is received.

But this is a problem: if every process p starts sending to $p - 1$, everyone is waiting for someone else to do a receive, and no one is actually doing a receive. This sort of situation is called *deadlock*.

Exercise 5. Do you now see why the code fragment leads to deadlock? Can you come up with a clever rearrangement of the sends and receives so that there is no deadlock?

Finding a ‘clever rearrangement’ is usually not the best way to solve deadlock problems. A common solution is to use *non-blocking communication* calls. Here the send or receive instruction only indicates to the system the buffer with send data, or in which to receive data. You then need a second call to ensure that the operation is actually completed.

In pseudo-code:

```

send( buffer1, to=neighbour1, result=request1 );
send( buffer2, to=neighbour2, result=request2 );
// maybe execute some other code
wait( request1 ); wait( request2 ); // make sure the operations are done

```

Task scheduling All parallel realizations of Life you have seen so far were based on taking a single time step, and applying parallel computing to the updates in that time step. This was based on the fact that the points in the new time step can be computed independently. But the outer iteration has to be done in that order. Right?

Well...

Let's suppose you want to compute the board two timesteps from now, without explicitly computing the next timestep. Would that be possible?

Exercise 6. Life expresses the value in i, j at time $t+1$ as a simple function of the 3×3 patch $i-1:i+1, j-1:j+1$ at time t . Convince yourself that the value in i, j at $t+2$ can be computed as a function of a 5×5 patch at t .

Can you formulate rules for this update over two timesteps? Are these rules as elegant as the old ones, just expressed in a count of live and dead cells? If you would code the new rules as a case statement, how many clauses would there be?

Let's not pursue this further...

This exercise makes an important point about dependence and independence. If the value at i, j depends on 3×3 previous points, and each of these have a similar dependence, we can compute the value at i, j if we know 5×5 points two steps away, et cetera. The conclusion is that you do not need to finish a whole time step before you can start the next: for each point update only certain other points are needed, and not the whole board. If multiple processors are updating the board, they do not need to be working on the same timestep. This is sometimes called *asynchronous computing*. It means that processors do not have to synchronize what time step they are working on: within restrictions they can be working on different time steps.

Exercise 7. Just how independent can processors be? If processor i, j is working on time t , can processor $i + 1, j$ be working on $t + 2$? Can you give a formal description of how far out of step processor i, j and i', j' can be?

The previous sections were supposed to be about task parallelism, but we didn't actually define the concept of task. Informally, a processor receiving border information and then updating its local data sounds like something that could be called a task. To make it a little more formal, we define a task as some operations done on the same processor, plus a list of other tasks that have to be finished before this task can be finished.

This concept of computing is also known as *dataflow*: data flows as output of one operation to another; an operation can start executing when all its inputs are available. Another concept connected to this definition of tasks is that of a Directed Acyclic Graph (DAG): the dependencies between tasks form a graph, and you cannot have cycles in this graph, otherwise you could never get started. . .

You can interpret the MPI examples in terms of tasks. The local computation of a task can start when data from the neighbouring tasks is available, and a task finds out about that by the messages from those neighbours coming in. However, this view does not add much information.

On the other hand, if you have shared memory, and tasks that do not all take the same amount of running time, the task view can be productive. In this case, we adopt a *master-worker model*: there is one master process that keeps a list of tasks, and there are a number of worker processors that can execute the tasks. The master executes the following program:

1. The master finds which running tasks are finished;
2. For each scheduled task, if it needs the data of a finished task, mark that the data is available;
3. Find a task that can now execute, find a processor for it, and execute it there.

The pseudo-code for this is:

```

while there_are_tasks_left():
    for r in running_tasks:
        if r.finished():
            for t in scheduled_tasks:
                t.mark_available_input(r)
t = find_available_task()
p = find_available_processor()
schedule(t,p)

```

The master-worker model assumes that in general there are more available tasks than processors. In the Game of Life we can easily get this situation if you divide the board in more parts than there are processing elements. (Why would you do that? This mostly makes sense if you think about the memory hierarchy and cache sizes; see section HPSC-??.) So with $N \times N$ divisions of the board and T time steps, we define the queue of tasks:

```

for t in [0:T]:
    for i in [0:N]:
        for j in [0:N]:
            task( id=[t+1,i,j],
                prereqs=[ [t,i,j],[t,i-1,j],[t,i+1,j] # et cetera
                          ] )

```

Exercise 8. Argue that this model mostly makes sense on shared memory.
Hint: if you would execute this model on distributed memory, how much data needs to be moved in general when you start a task?

10.3 Advanced topics

10.3.1 Data partitioning

The previous sections approached parallelization of the Game of Life by taking the sequential implementation and the basic loop structure. For instance, in section 10.2.3

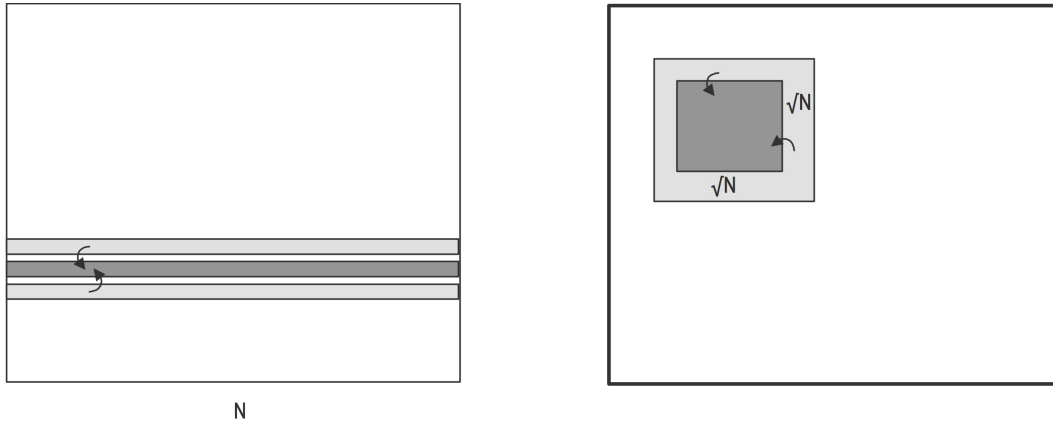


Figure (10.8) One-dimensional and two-dimensional distribution communication

we assigned a number of lines to each processor. This corresponds to a *one-dimensional partitioning* of the data. Sometimes, however, it is a good idea to use a two-dimensional one instead. (See figure 10.8 for an illustration of the basic idea.) In this section you'll get the flavour of the argument.

Suppose each processor stores one line of the Life board. As you saw in the previous section, to update that line it needs to receive two lines worth of data, and this takes time. In fact, receiving one item of data from another node is much slower than reading one item from local memory. If we inventory the cost of one timestep in the distributed case, that comes down to

1. Receiving $2N$ Life cells from other processors⁵; and
2. Adding $8N$ values together to get the counts.

For most architectures, the cost of sending and receiving data will far outweigh the computation.

Let us now assume that we have N processors, each storing a $\sqrt{N} \times \sqrt{N}$ part of the Life board. We sometimes call this the processor's *subdomain*. To update this, a processor now needs to receive data from the lines above, under, and to the left and right of its part (we are

⁵For now we only count the transmission cost per item; there is also a one-time cost for each transmission, called the *latency*. For large enough messages we can ignore this; for details see HPSC-??.

ignoring the edge of the board here). That means four messages, each of size $\sqrt{N} + 2$. On the other hand, the update takes $8N$ operations. For large enough N , the communication, which is slow, will be outweighed by the computation, which is much faster.

Our analysis here was very simple, based on having exactly N processors. In practice you will have fewer processors, and each processor will have a subdomain rather than a single point. However, a more refined analysis gives the same conclusion: a two-dimensional distribution is to be preferred over a one-dimensional one; see for instance section HPSC-?? for the analysis of the matrix-vector product algorithm.

Let's do just a little analysis on the following scenario:

- You have a parallel machine where each processor has an amount M of memory to store the Life board.
- You can buy extra processors for this machine, thereby expanding both the processing power (in operations per second) and the total memory.
- As you buy more processors, you can store a larger Life board: we're assuming that the amount M of memory per processor is kept constant. (This strategy of scaling up the problem as you scale up the computer is called *weak scaling*. The scenario where you only increase the number of processors, keeping the problem fixed and therefore putting less and less Life cells on each processor, is called *strong scaling*.)

Let P be the number of processors, and N the size of the board. In terms of the amount of memory M you then have:

$$M = N^2/P.$$

Let's now consider a one-dimensional distribution. (Left half of figure 10.9.) Every processor but the first and last one needs to communicate two whole lines, meaning $2N$ elements. If you express this in terms of M you find a formula that contains the variable P . This means that as you buy more processors, and can store a larger problem, the amount of communication becomes a function of the number of processors.

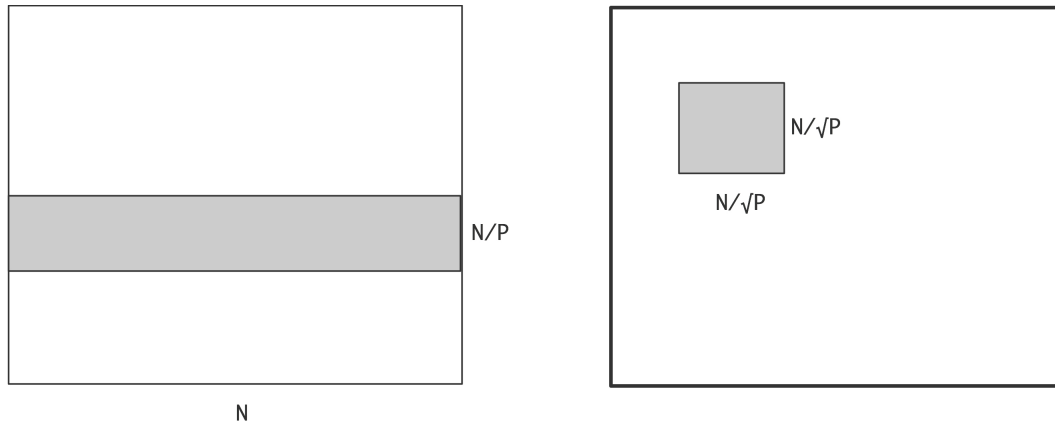


Figure (10.9) One-dimensional and two-dimensional distribution of a Life board

Exercise 9. Show that the amount of communication goes up with the number of processors. On the other hand, show that the amount of work stays constant, and that it corresponds to a perfect distribution of the work over the processors.

Now consider a two-dimensional distribution. (Right half of figure 10.9.) Every processor that is not on the edge of the board will communicate with eight others. With the four ‘corner’ processors only a single item is exchanged.

Exercise 10. What is the amount of data exchanged with the processors left/right and top/bottom? Show that, expressed in terms of M , this formula does not contain the variable P . Show that, again, the work is constant in N and P .

The previous two exercises demonstrate an important point: different parallelization strategies can have different overhead and therefore different efficiencies. Both the one and two-dimensional distribution lead to a perfect parallelization of the work. On the other hand, with the two-dimension distribution the communication is constant, while with the one-dimensional distribution the communication cost goes up with the number of processors, so the algorithm becomes less and less efficient.

10.3.2 Combining work, minimizing communication

In most of the above discussion we have considered the parallel update of the Life board as one bulk operation that is executed in sequence: you do all communication for one update step, and then the communication for the next, et cetera.

Now, the time for a communication between two processes has two components: there is a startup time (known as ‘latency’), and then there is a time per item communicated. This is usually rendered as

$$T(n) = \alpha + \beta \cdot n$$

where the α is the startup time and β the per-item time.

Exercise 11. Show that sending two messages of length n takes longer than one message of length $2n$, in other words $T(2n) < 2T(n)$. For what value of n is the overhead 50%? 10%?

If the ratio between α and β is large there is clearly an incentive to combine messages. For the naive parallelization strategies considered above there is no easy way to do this. However, there is a way to communicate only once every *two* updates. This communication will be larger, but there will be savings in the startup cost.

First we must observe that to update a single Life cell by one time step we need the eight cells around it. So to update a cell by two time steps we need those eight cells plus the ones around them. This is illustrated in figure 10.10. If a processor has the responsibility for updating a subsection of the board, it needs the *halo region* around it. For a single update, this is a halo of width one, and for two updates this is a halo of width two.

Let’s analyze the cost of this scheme. We assume that the board is square of size $N \times N$, and that there are $P \times P$ processors, so that each processor is computing an $(N/P) \times (N/P)$ part of the board.

In the one-step-at-a-time implementation a processor does the following:

1. Receives four messages of length N and four of length 1; and

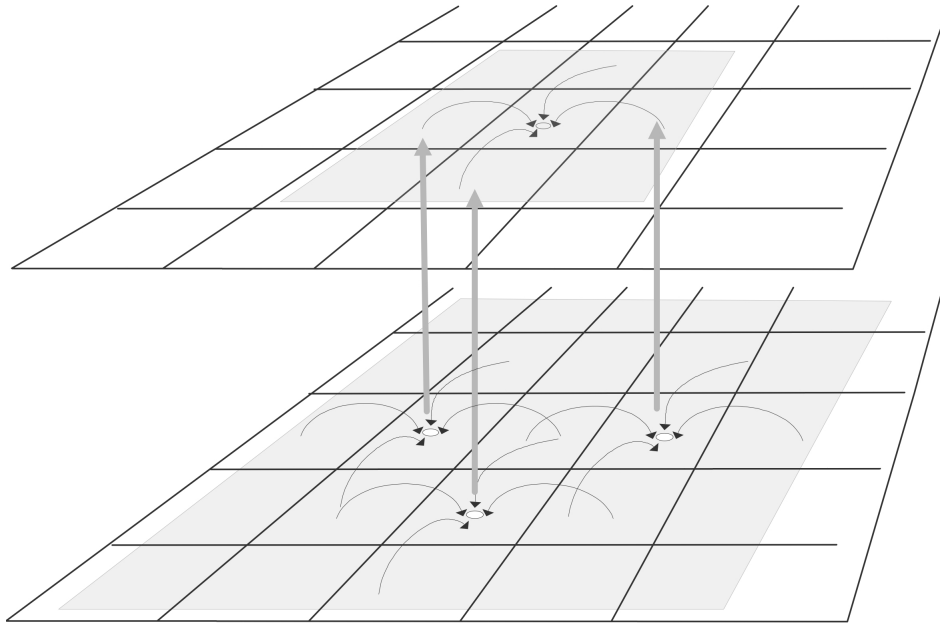


Figure (10.10) Two steps of Life updates

2. Then updates the part of the board it owns to the next time step.

In order to update its subdomain two timesteps, the following is needed:

1. Receive four messages of size $2N$ and four of size 4;
2. Compute the updated values at time $t + 1$ of the subdomain plus a locally stored border of thickness 1 around it;
3. Update precisely the owned subdomain to its state at $t + 2$.

So now you send slightly more data, and you compute a little more, but you save half the latency cost. Since communication latency can be quite high, this scheme can be faster overall.

10.3.3 Load balancing

The basic motivation of parallel computing is to be able to compute faster. Ideally, having p processors would make your computation p times faster (see section HPSC-?? for definition and discussion of speedup), but practice doesn't always live up to that ideal. There

are many reasons for this, but one is that the work may not be evenly divided between the processors. If some processors have more work than others, they will still be computing while the others have finished and are sitting idle. This is called *load imbalance*.

Exercise 12. Compute the speedup from using p processors if one processor has a fraction ϵ more work than the others; the others are assumed to be perfectly balanced. Also compute the speedup from the case where one processor has ϵ less work than all others. Which of the two scenarios is worse?

Clearly, there is a strong incentive for having a well-balanced load between the processors. How to do this depends on what sort of parallelism you are dealing with.

In section 10.2.3 you saw that in shared memory it can make sense to divide the work in more units than there are processors. Statistically, this evens out the work balance. On the other hand, in distributed memory (section 10.2.3) such dynamic assignment is not possible, so you have to be careful in dividing up the work. Unfortunately, sometimes the workload changes during the run of a program, and you want to rebalance it. Doing so can be tricky, since it requires problem data to be moved, and processors have to reallocate and rearrange their data structures. This is a very advanced topic, and not at all simple to do.

10.4 Summary

In this chapter you have seen a number of parallel programming concepts through the example of the Game of Life. Like many scientific problems, you can view this as having parallelism on more than one level, and you can program it accordingly, depending on what sort of computer you have.

- The ‘data parallel’ aspect of the Life board can be addressed with the vector instructions in even laptop processors.
- Just about every processor in existence is also pipelined, and you can express that in your code.

- A GPU can handle fairly fine-grained parallelism, so you would write a ‘kernel’ that expresses the operations for updating a single cell of the board.
- If you have a multicore processor, which has shared memory, you can use a loop-based model such as OpenMP to parallelize the naive code.
- On both a multicore processor and on distributed memory clusters you could process the Life board as a set of sub-boards, each of which is handled by a core or by a cluster node. This takes some rearranging of the reference code; in the distributed memory case you need to insert MPI library calls.

In all, you see that, depending on your available machine, some rearranging of the naive algorithm for Game of Life is needed. Parallelism is not a magic ingredient that you can easily add to existing code, it needs to be integral part of the design of your program.

10.5 List of acronyms

| | |
|---|--|
| AVX Advanced Vector Extensions | MIMD Multiple Instruction Multiple Data |
| BSP Bulk Synchronous Parallel | MPI Message Passing Interface |
| CAF Co-array Fortran | PGAS Partitioned Global Address Space |
| CUDA Compute Unified Device Architecture | SIMD Single Instruction Multiple Data |
| | SIMT Single Instruction Multiple Thread |
| DAG Directed Acyclic Graph | SMP Symmetric Multi Processing |
| GPU Graphics Processing Unit | SPMD Single Program Multiple Data |
| HPC High-Performance Computing | SSE SIMD Streaming Extensions |