1

INTRODUCTION

This chapter is a self-contained tutorial which tells you how to get started with parallel programming and how to design and implement algorithms in a structured way. The chapter introduces a simple target architecture for designing parallel algorithms, the bulk synchronous parallel computer. Using the computation of the inner product of two vectors as an example, the chapter shows how an algorithm is designed hand in hand with its cost analysis. The algorithm is implemented in a short program that demonstrates the most important primitives of BSPlib, the main communication library used in this book. If you understand this program well, you can start writing your own parallel programs. Another program included in this chapter is a benchmarking program that allows you to measure the BSP parameters of your parallel computer. Substituting these parameters into a theoretical cost formula for an algorithm gives a prediction of the actual running time of an implementation.

1.1 Wanted: a gentle parallel programming model

Parallel programming is easy if you use the right programming model. All you have to do is find it! Today's developers of scientific application software must pay attention to the use of computer time and memory, and also to the accuracy of their results. Since this is already quite a burden, many developers view **parallelism**, the use of more than one processor to solve a problem, as just an extra complication that is better avoided. Nevertheless, parallel algorithms are being developed and used by many computational scientists in fields such as astronomy, biology, chemistry, and physics. In industry, engineers are trying to accelerate their simulations by using parallel computers. The main motivation of all these brave people is the tremendous computing power promised by parallel computers. Since the advent of the World Wide Web, this power lies only a tempting few mouse-clicks away from every computer desk. The Grid is envisioned to deliver this power to that desk, providing computational services in a way that resembles the workings of an electricity grid.

The potential of parallel computers could be realized if the practice of parallel programming were just as easy and natural as that of sequential

programming. Unfortunately, until recently this has not been the case, and parallel computing used to be a very specialized area where exotic parallel algorithms were developed for even more exotic parallel architectures, where software could not be reused and many man years of effort were wasted in developing software of limited applicability. Automatic parallelization by compilers could be a solution for this problem, but this has not been achieved yet, nor is it likely to be achieved in the near future. Our only hope of harnessing the power of parallel computing lies in actively engaging ourselves in parallel programming. Therefore, we might as well try to make parallel programming easy and effective, turning it into a natural activity for everyone who writes computer programs.

An important step forward in making parallel programming easier has been the development of **portability layers**, that is, communication software such as PVM [171] and MPI [137] that enable us to run the same parallel program on many different parallel computers without changing a single line of program text. Still, the resulting execution time behaviour of the program on a new machine is unpredictable (and can indeed be rather erratic), due to the lack of an underlying parallel programming model.

To achieve the noble goal of easy parallel programming we need a model that is simple, efficiently implementable, and acceptable to all parties involved: hardware designers, software developers, and end users. This model should not interfere with the process of designing and implementing algorithms. It should exist mainly in the background, being tacitly understood by everybody. Such a model would encourage the use of parallel computers in the same way as the Von Neumann model did for the sequential computer.

The bulk synchronous parallel (BSP) model proposed by Valiant in 1989 [177,178] satisfies all requirements of a useful parallel programming model: the BSP model is simple enough to allow easy development and analysis of algorithms, but on the other hand it is realistic enough to allow reasonably accurate modelling of real-life parallel computing; a portability layer has been defined for the model in the form of BSPlib [105] and this standard has been implemented efficiently in at least two libraries, namely the Oxford BSP toolset [103] and the Paderborn University BSP library [28,30], each running on many different parallel computers; another portability layer suitable for BSP programming is the one-sided communications part of MPI-2 [138], implementations of which are now appearing; in principle, the BSP model could be used in taking design decisions when building new hardware (in practice though, designers face other considerations); the BSP model is actually being used as the framework for algorithm design and implementation on a range of parallel computers with completely different architectures (clusters of PCs, networks of workstations, shared-memory multiprocessors, and massively parallel computers with distributed memory). The BSP model is explained in the next section.

1.2 The BSP model

The BSP model proposed by Valiant in 1989 [177,178] comprises a computer architecture, a class of algorithms, and a function for charging costs to algorithms. In this book, we use a variant of the BSP model; the differences with the original model are discussed at the end of this section.

A **BSP computer** consists of a collection of processors, each with private memory, and a communication network that allows processors to access memories of other processors. The architecture of a BSP computer is shown in Fig. 1.1. Each processor can read from or write to every memory cell in the entire machine. If the cell is local, the read or write operation is relatively fast. If the cell belongs to another processor, a message must be sent through the communication network, and this operation is slower. The access time for all nonlocal memories is the same. This implies that the communication network can be viewed as a black box, where the connectivity of the network is hidden in the interior. As users of a BSP computer, we need not be concerned with the details of the communication network. We only care about the remote access time delivered by the network, which should be uniform. By concentrating on this single property, we are able to develop **portable** algorithms, that is, algorithms that can be used for a wide range of parallel computers.

A **BSP algorithm** consists of a sequence of supersteps. A **superstep** contains either a number of computation steps or a number of communication steps, followed by a global barrier synchronization. In a **computation superstep**, each processor performs a sequence of operations on local data. In scientific computations, these are typically floating-point operations (flops). (A **flop** is a multiplication, addition, subtraction, or division of two floating-point numbers. For simplicity, we assume that all these operations take the same amount of time.) In a **communication superstep**, each processor

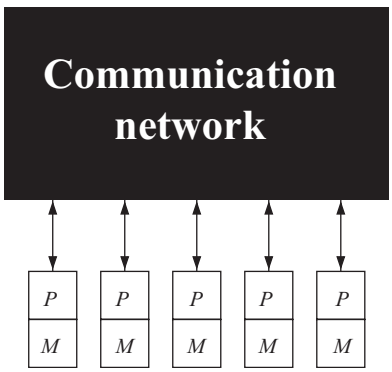

FIG. 1.1. Architecture of a BSP computer. Here, 'P' denotes a processor and 'M' a memory.

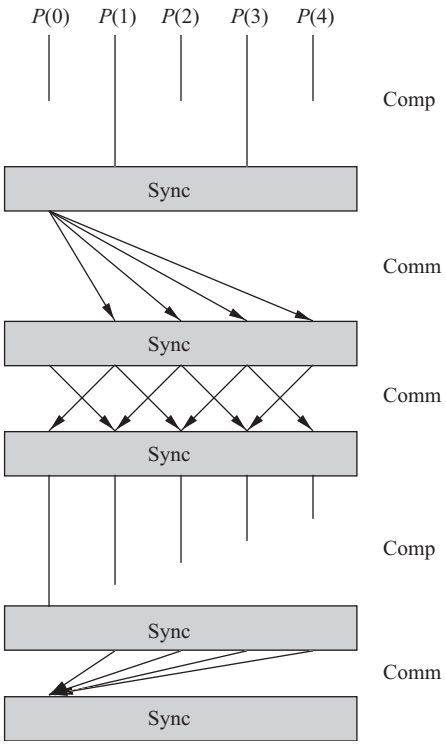

FIG. 1.2. BSP algorithm with five supersteps executed on five processors. A vertical line denotes local computation; an arrow denotes communication of one or more data words to another processor. The first superstep is a computation superstep. The second one is a communication superstep, where processor $P(0)$ sends data to all other processors. Each superstep is terminated by a global synchronization.

sends and receives a number of messages. At the end of a superstep, all processors synchronize, as follows. Each processor checks whether it has finished all its obligations of that superstep. In the case of a computation superstep, it checks whether the computations are finished. In the case of a communication superstep, it checks whether it has sent all messages that it had to send, and whether it has received all messages that it had to receive. Processors wait until all others have finished. When this happens, they all proceed to the next superstep. This form of synchronization is called **bulk synchronization**, because usually many computation or communication operations take place between successive synchronizations. (This is in contrast to pairwise synchronization, used in most message-passing systems, where each message causes a pair of sending and receiving processors to wait until the message has been transferred.) Figure 1.2 gives an example of a BSP algorithm.

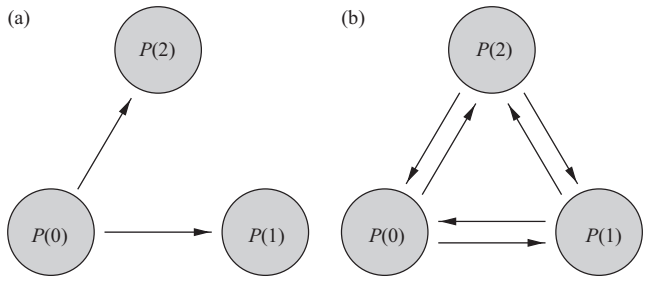

FIG. 1.3. Two different h-relations with the same h. Each arrow represents the communication of one data word. (a) A 2-relation with $h_s = 2$ and $h_r = 1$; (b) a 2-relation with $h_s = h_r = 2$.

The **BSP cost function** is defined as follows. An ***h*-relation** is a communication superstep where each processor sends at most h data words to other processors and receives at most h data words, and where at least one processor sends or receives h words. A data word is a real or an integer. We denote the maximum number of words sent by any processor by h_s and the maximum number received by h_r. Therefore,

$$h = \max\{h_s, h_r\}. \tag{1.1}$$

This equation reflects the assumption that a processor can send and receive data simultaneously. The cost of the superstep depends solely on h. Note that two different communication patterns may have the same h, so that the cost function of the BSP model does not distinguish between them. An example is given in Fig. 1.3. Charging costs on the basis of h is motivated by the assumption that the bottleneck of communication lies at the entry or exit of the communication network, so that simply counting the maximum number of sends and receives gives a good indication of communication time. Note that for the cost function it does not matter whether data are sent together or as separate words.

The time, or **cost**, of an h-relation is

$$T_{\text{comm}}(h) = hg + l, \tag{1.2}$$

where g and l are machine-dependent parameters and the time unit is the time of one flop. This cost is charged because of the expected linear increase of communication time with h. Since $g = \lim_{h\to\infty} T_{\text{comm}}(h)/h$, the parameter g can be viewed as the time needed to send one data word into the communication network, or to receive one word, in the asymptotic situation of continuous message traffic. The linear cost function includes a nonzero constant l because executing an h-relation incurs a fixed overhead, which includes the cost of global synchronization, but also fixed cost components of ensuring that all data have arrived at their destination and of starting up the communication. We lump all such fixed costs together into one parameter l.

Approximate values for g and l of a particular parallel computer can be obtained by measuring the execution time for a range of **full** h-relations, that is, h-relations where each processor sends and receives exactly h data words. Figure 1.3(b) gives an example of a full 2-relation. (Can you find another full 2-relation?) In practice, the measured cost of a full h-relation will be an upper bound on the measured cost of an arbitrary h-relation. For our measurements, we usually take 64-bit reals or integers as data words.

The cost of a computation superstep is

$$T_{\text{comp}}(w) = w + l, \tag{1.3}$$

where the amount of work w is defined as the maximum number of flops performed in the superstep by any processor. For reasons of simplicity, the value of l is taken to be the same as that of a communication superstep, even though it may be less in practice. As a result, the total synchronization cost of an algorithm can be determined simply by counting its supersteps. Because of (1.2) and (1.3), the total cost of a BSP algorithm becomes an expression of the form $a + bg + cl$. Figure 1.4 displays the cost of the BSP algorithm from Fig. 1.2.

A BSP computer can be characterized by four parameters: p, r, g, and l. Here, p is the **number of processors**. The parameter r is the **single-processor computing rate** measured in flop/s (floating-point operations per second). This parameter is irrelevant for cost analysis and algorithm design, because it just normalizes the time. (But if you wait for the result of your program, you may find it highly relevant!) From a global, architectural point of view, the parameter g, which is measured in flop time units, can be seen as the ratio between the computation throughput and the communication throughput of the computer. This is because in the time period of an h-relation, phg flops can be performed by all processors together and ph data words can be communicated through the network. Finally, l is called the **synchronization cost**, and it is also measured in flop time units. Here, we slightly abuse the language, because l includes fixed costs other than the cost of synchronization as well. Measuring the characteristic parameters of a computer is called **computer benchmarking**. In our case, we do this by measuring r, g, and l. (For p, we can rely on the value advertised by the computer vendor.) Table 1.1 summarizes the BSP parameters.

We can predict the execution time of an implementation of a BSP algorithm on a parallel computer by theoretically analysing the cost of the algorithm and, independently, benchmarking the computer for its BSP performance. The predicted time (in seconds) of an algorithm with cost $a+bg+cl$ on a computer with measured parameters r, g, and l equals $(a + bg + cl)/r$, because the time (in seconds) of one flop equals $t_{\text{flop}} = 1/r$.

One aim of the BSP model is to guarantee a certain performance of an implementation. Because of this, the model states costs in terms of upper

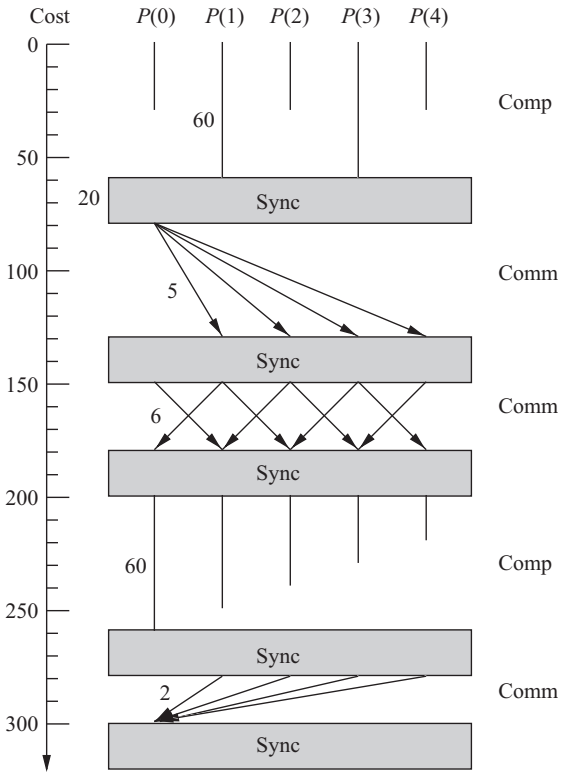

FIG. 1.4. Cost of the BSP algorithm from Fig. 1.2 on a BSP computer with $p = 5$, $g = 2.5$, and $l = 20$. Computation costs are shown only for the processor that determines the cost of a superstep (or one of them, if there are several). Communication costs are shown for only one source/destination pair of processors, because we assume in this example that the amount of data happens to be the same for every pair. The cost of the first superstep is determined by processors $P(1)$ and $P(3)$, which perform 60 flops each. Therefore, the cost is $60 + l = 80$ flop time units. In the second superstep, $P(0)$ sends five data words to each of the four other processors. This superstep has $h_s = 20$ and $h_r = 5$, so that it is a 20-relation and hence its cost is $20g + l = 70$ flops. The cost of the other supersteps is obtained in a similar fashion. The total cost of the algorithm is 320 flops.

TABLE 1.1. The BSP parameters

p	number of processors
r	computing rate (in flop/s)
g	communication cost per data word (in time units of 1 flop)
l	global synchronization cost (in time units of 1 flop)

bounds. For example, the cost function of an h-relation assumes a worst-case communication pattern. This implies that the predicted time is an upper bound on the measured time. Of course, the accuracy of the prediction depends on how the BSP cost function reflects reality, and this may differ from machine to machine.

Separation of concerns is a basic principle of good engineering. The BSP model enforces this principle by separating the hardware concerns from the software concerns. (Because the software for routing messages in the communication network influences g and l, we consider such routing software to be part of the hardware system.) On the one hand, the hardware designer can concentrate on increasing p, r and decreasing g, l. For example, he could aim at designing a BSP(p, r, g, l) computer with $p \geq 100$, $r \geq 1$ Gflop/s (the prefix G denotes Giga $= 10^9$), $g \leq 10$, $l \leq 1000$. To stay within a budget, he may have to trade off these objectives. Obviously, the larger the number of processors p and the computing rate r, the more powerful the communication network must be to keep g and l low. It may be preferable to spend more money on a better communication network than on faster processors. The BSP parameters help in quantifying these design issues. On the other hand, the software designer can concentrate on decreasing the algorithmic parameters a, b, and c in the cost expression $a + bg + cl$; in general, these parameters depend on p and the problem size n. The aim of the software designer is to obtain good scaling behaviour of the cost expression. For example, she could realistically aim at a decrease in a as $1/p$, a decrease in b as $1/\sqrt{p}$, and a constant c.

The BSP model is a distributed-memory model. This implies that both the computational work and the data are distributed. The work should be distributed evenly over the processors, to achieve a good load balance. The data should be distributed in such a way that the total communication cost is limited. Often, the data distribution determines the work distribution in a natural manner, so that the choice of data distribution must be based on two objectives: good load balance and low communication cost. In all our algorithms, choosing a data distribution is an important decision. By designing algorithms for a distributed-memory model, we do not limit ourselves to this model. Distributed-memory algorithms can be used efficiently on shared-memory parallel computers, simply by partitioning the shared memory among the processors. (The reverse is not true: shared-memory algorithms do not take data locality into account, so that straightforward distribution leads to inefficient algorithms.) We just develop a distributed-memory program; the BSP system does the rest. Therefore, BSP algorithms can be implemented efficiently on every type of parallel computer.

The main differences between our variant of the BSP model and the original BSP model [178] are:

1. The original BSP model allows supersteps to contain both computation and communication. In our variant, these operations must be split

into separate supersteps. We impose this restriction to achieve conceptual simplicity. We are not concerned with possible gains obtained by overlapping computation and communication. (Such gains are minor at best.)

2. The original cost function for a superstep with an h-relation and a maximum amount of work w is $\max(w, hg, L)$, where L is the **latency**, that is, the minimum number of time units between successive supersteps. In our variant, we charge $w + hg + 2l$ for the corresponding two supersteps. We use the synchronization cost l instead of the (related) latency L because it facilitates the analysis of algorithms. For example, the total cost in the original model of a 2-relation followed by a 3-relation equals $\max(2g, L) + \max(3g, L)$, which may have any of the outcomes $5g$, $3g + L$, and $2L$, whereas in our variant we simply charge $5g + 2l$. If g and l (or L) are known, we can substitute their values into these expressions and obtain one scalar value, so that both variants are equally useful. However, we also would like to analyse algorithms without knowing g and l (or L), and in that case our cost function leads to simpler expressions. (Valiant [178] mentions $w + hg + l$ as a possible alternative cost function for a superstep with both computation and communication, but he uses $\max(w, hg, l)$ in his analysis.)

3. The data distribution in the original model is controlled either directly by the user, or, through a randomizing hash function, by the system. The latter approach effectively randomizes the allocation of memory cells, and thereby it provides a shared-memory view to the user. Since this approach is only efficient in the rare case that g is very low, $g \approx 1$, we use the direct approach instead. Moreover, we use the data distribution as our main means of making computations more efficient.

4. The original model allowed for synchronization of a subset of all processors instead of all processors. This option may be useful in certain cases, but for reasons of simplicity we disallow it.

1.3 BSP algorithm for inner product computation

A simple example of a BSP algorithm is the following computation of the inner product α of two vectors $\mathbf{x} = (x_0, \ldots, x_{n-1})^{\mathrm{T}}$ and $\mathbf{y} = (y_0, \ldots, y_{n-1})^{\mathrm{T}}$,

$$\alpha = \sum_{i=0}^{n-1} x_i y_i. \tag{1.4}$$

In our terminology, vectors are column vectors; to save space we write them as $\mathbf{x} = (x_0, \ldots, x_{n-1})^{\mathrm{T}}$, where the superscript 'T' denotes transposition. The vector $\mathbf{x}$ can also be viewed as an $n \times 1$ matrix. The inner product of $\mathbf{x}$ and $\mathbf{y}$ can concisely be expressed as $\mathbf{x}^{\mathrm{T}}\mathbf{y}$.

The inner product is computed by the processors $P(0), \ldots, P(p-1)$ of a BSP computer with p processors. We assume that the result is needed by all processors, which is usually the case if the inner product computation is part of a larger computation, such as in iterative linear system solvers.

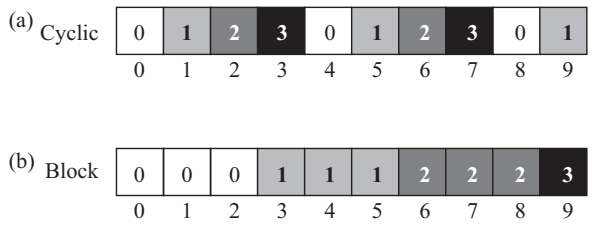

FIG. 1.5. Distribution of a vector of size ten over four processors. Each cell represents a vector component; the number in the cell and the greyshade denote the processor that owns the cell. The processors are numbered $0, 1, 2, 3$. (a) Cyclic distribution; (b) block distribution.

The data distribution of the vectors $\mathbf{x}$ and $\mathbf{y}$ should be the same, because in that case the components x_i and y_i reside on the same processor and they can be multiplied immediately without any communication. The data distribution then determines the work distribution in a natural manner. To balance the work load of the algorithm, we must assign the same number of vector components to each processor. Card players know how to do this blindly, even without counting and in the harshest of circumstances. They always deal out their cards in a cyclic fashion. For the same reason, an optimal work distribution is obtained by the **cyclic distribution** defined by the mapping

$$x_i \longmapsto P(i \bmod p), \quad \text{for } 0 \leq i < n. \tag{1.5}$$

Here, the mod operator stands for taking the remainder after division by p, that is, computing modulo p. Similarly, the div operator stands for integer division rounding down. Figure 1.5(a) illustrates the cyclic distribution for $n = 10$ and $p = 4$. The maximum number of components per processor is $\lceil n/p \rceil$, that is, n/p rounded up to the nearest integer value, and the minimum is $\lfloor n/p \rfloor = n \text{ div } p$, that is, n/p rounded down. The maximum and the minimum differ at most by one. If p divides n, every processor receives exactly n/p components. Of course, many other data distributions also lead to the best possible load balance. An example is the **block distribution**, defined by the mapping

$$x_i \longmapsto P(i \text{ div } b), \quad \text{for } 0 \leq i < n, \tag{1.6}$$

with block size $b = \lceil n/p \rceil$. Figure 1.5(b) illustrates the block distribution for $n = 10$ and $p = 4$. This distribution has the same maximum number of components per processor, but the minimum can take every integer value between zero and the maximum. In Fig. 1.5(b) the minimum is one. The minimum can even be zero: if $n = 9$ and $p = 4$, then the block size is $b = 3$, and the processors receive 3, 3, 3, 0 components, respectively. Since the computation cost is determined by the maximum amount of work, this is just as good as

Algorithm 1.1. Inner product algorithm for processor $P(s)$, with $0 \leq s < p$.

input: $\mathbf{x}, \mathbf{y}$: vector of length n,
distr($\mathbf{x}$) = distr($\mathbf{y}$) = ϕ,
with $\phi(i) = i \bmod p$, for $0 \leq i < n$.
output: $\alpha = \mathbf{x}^{\mathrm{T}}\mathbf{y}$.

```
(0)   α_s := 0;
      for i := s to n − 1 step p do
          α_s := α_s + x_i y_i;

(1)   for t := 0 to p − 1 do
          put α_s in P(t);

(2)   α := 0;
      for t := 0 to p − 1 do
          α := α + α_t;
```

the cyclic distribution, which assigns 3, 2, 2, 2 components. Intuitively, you may object to the idling processor in the block distribution, but the work distribution is still optimal!

Algorithm 1.1 computes an inner product in parallel. It consists of three supersteps, numbered (0), (1), and (2). The synchronizations at the end of the supersteps are not written explicitly. All the processors follow the same program text, but their actual execution paths differ. The path of processor $P(s)$ depends on the processor identity s, with $0 \leq s < p$. This style of programming is called **single program multiple data** (SPMD), and we shall use it throughout the book.

In superstep (0), processor $P(s)$ computes the local partial inner product

$$\alpha_s = \sum_{0 \leq i < n,\ i \bmod p = s} x_i y_i, \tag{1.7}$$

multiplying x_s by y_s, x_{s+p} by y_{s+p}, x_{s+2p} by y_{s+2p}, and so on, and adding the results. The data for this superstep are locally available. Note that we use global indices so that we can refer uniquely to variables without regard to the processors that own them. We access the local components of a vector by stepping through the arrays with a **stride**, or step size, p.

In superstep (1), each processor **broadcasts** its result α_s, that is, it sends α_s to all processors. We use the communication primitive 'put x in $P(t)$' in the program text of $P(s)$ to denote the one-sided action by processor $P(s)$ of storing a data element x at another processor $P(t)$. This completely determines the communication: both the source processor and the destination processor

of the data element are specified. The 'put' primitive assumes that the source processor knows the memory location on the destination processor where the data must be put. The source processor is the initiator of the action, whereas the destination processor is passive. Thus, we assume implicitly that each processor allows all others to put data into its memory. Superstep (1) could also have been written as 'put α_s in $P(*)$', where we use the abbreviation $P(*)$ to denote all processors. Note that the program includes a put by processor $P(s)$ into itself. This operation is simply skipped or becomes a local memory-copy, but it does not involve communication. It is convenient to include such puts in program texts, to avoid having to specify exceptions.

Sometimes, it may be necessary to let the destination processor initiate the communication. This may happen in irregular computations, where the destination processor knows that it needs data, but the source processor is unaware of this need. In that case, the destination processor must fetch the data from the source processor. This is done by a statement of the form 'get x from $P(t)$' in the program text of $P(s)$. In most cases, however, we use the 'put' primitive. Note that using a 'put' is much simpler than using a matching 'send'/'receive' pair, as is done in message-passing parallel algorithms. The program text of such an algorithm must contain additional if-statements to distinguish between sends and receives. Careful checking is needed to make sure that pairs match in all possible executions of the program. Even if every send has a matching receive, this does not guarantee correct communication as intended by the algorithm designer. If the send/receive is done by the handshake (or kissing) protocol, where both participants can only continue their way after the handshake has finished, then it can easily happen that the sends and receives occur in the wrong order. A classic case is when two processors both want to send first and receive afterwards; this situation is called **deadlock**. Problems such as deadlock cannot happen when using puts.

In superstep (2), all processors compute the final result. This is done **redundantly**, that is, the computation is replicated so that all processors perform exactly the same operations on the same data. The complete algorithm is illustrated in Fig. 1.6.

The cost analysis of the algorithm is as follows. Superstep (0) requires a floating-point multiplication and an addition for each component. Therefore, the cost of (0) is $2\lceil n/p \rceil + l$. Superstep (1) is a $(p-1)$-relation, because each processor sends and receives $p-1$ data. (Communication between a processor and itself is not really communication and hence is not counted in determining h.) The cost of (1) is $(p-1)g + l$. The cost of (2) is $p + l$. The total cost of the inner product algorithm is

$$T_{\text{inprod}} = 2\left\lceil \frac{n}{p} \right\rceil + p + (p-1)g + 3l. \tag{1.8}$$

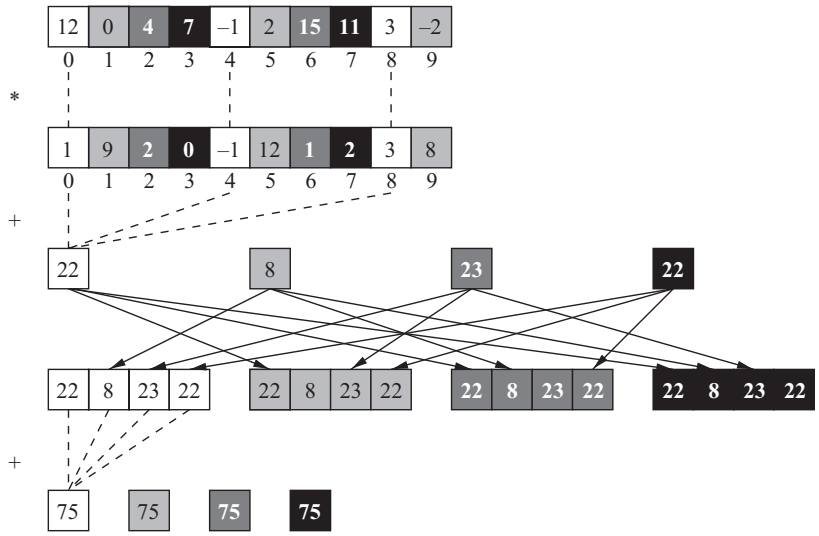

FIG. 1.6. Parallel inner product computation. Two vectors of size ten are distributed by the cyclic distribution over four processors. The processors are shown by greyshades. First, each processor computes its local inner product. For example, processor $P(0)$ computes its local inner product $12 \cdot 1 + (-1) \cdot (-1) + 3 \cdot 3 = 22$. Then the local result is sent to all other processors. Finally, the local inner products are summed redundantly to give the result 75 in every processor.

An alternative approach would be to send all partial inner products to one processor, $P(0)$, and let this processor compute the result and broadcast it. This requires four supersteps. Sending the partial inner products to $P(0)$ is a $(p-1)$-relation and therefore is just as expensive as broadcasting them. While $P(0)$ adds partial inner products, the other processors are idle and hence the cost of the addition is the same as for the redundant computation. The total cost of the alternative algorithm would be $2\lceil n/p \rceil + p + 2(p-1)g + 4l$, which is higher than that of Algorithm 1.1. The lesson to be learned: if you have to perform an h-relation with a particular h, you might as well perform as much useful communication as possible in that h-relation; other supersteps may benefit from this.

1.4 Starting with BSPlib: example program `bspinprod`

BSPlib is a standard library interface, which defines a set of primitives for writing bulk synchronous parallel programs. Currently, BSPlib has been implemented efficiently in two libraries, namely the Oxford BSP toolset [103] and the Paderborn University BSP library [28,30]. The BSPlib standard unifies and extends its two predecessors, the Oxford BSP library designed by Miller and Reed [140,141] and the Green BSP library by Goudreau *et al.* [81].

Implementations of BSPlib exist for many different architectures, including: a cluster of PCs running Linux; a network of UNIX workstations connected by an Ethernet and communicating through the TCP/IP or UDP/IP protocol; shared-memory multiprocessors running a variant of UNIX; and massively parallel computers with distributed memory such as the Cray T3E, the Silicon Graphics Origin, and the IBM SP. In addition, the library can also be used on an ordinary sequential computer to run a parallel program with $p = 1$ as a sequential program; in this case, a special version of BSPlib can be used that strips off the parallel overhead. This is advantageous because it allows us to develop and maintain only one version of the source program, namely the parallel version. It is also possible to simulate a parallel computer by running p processes in parallel on one processor, sharing the time of the common CPU. This is a useful environment for developing parallel programs, for example, on a PC.

The BSPlib library contains a set of primitive functions, which can be called from a program written in a conventional programming language such as C, C++, or Fortran 90. BSPlib was designed following the motto 'small is beautiful'. The primitives of BSPlib were carefully crafted and particular attention was paid to the question of what to exclude from the library. As a result, BSPlib contains only 20 primitive functions, which are also called the **core operations**. In this section, we present a small C program, which uses 12 different primitives. The aim of this tutorial program is to get you started using the library and to expose the main principles of writing a BSPlib program. The remainder of this book gives further examples of how the library can be used. Six primitives for so-called bulk synchronous message passing will be explained in Chapter 4, where they are first needed. Two primitives for high-performance communication are explained in an exercise in Chapter 2. They are primarily meant for programmers who want the ultimate in performance, in terms of memory and computing speed, and who are prepared to live on the edge and be responsible for the safety of their programs, instead of relying on the BSP system to provide this. A quick reference guide to the BSPlib primitives is given as Appendix B. An alternative to using BSPlib is using MPI. Appendix C discusses how to program in BSP style using MPI and presents MPI versions of all the programs from Chapters 1 to 4. For a full explanation of the BSPlib standard, see the definitive source by Hill *et al.* [105]. In the following, we assume that a BSPlib implementation has already been installed. To start with, you could install the Oxford BSP toolset [103] on your PC running Linux, or ask your systems administrator to install the toolset at your local network of workstations.

The parallel part of a BSPlib program starts with the statement

```
bsp_begin(reqprocs);
```

where `int reqprocs` is the number of processors requested. The function

bsp_begin starts several executions of the same subprogram, where each execution takes place on a different processor and handles a different stream of data, in true SPMD style. The parallel part is terminated by

```
bsp_end();
```

Two possible modes of operation can be used. In the first mode, the whole computation is SPMD; here, the call to bsp_begin must be the first executable statement in the program and the call to bsp_end the last. Sometimes, however, one desires to perform some sequential part of the program before and after the parallel part, for example, to handle input and output. For instance, if the optimal number of processors to be used depends on the input, we want to compute it before the actual parallel computation starts. The second mode enables this: processor $P(0)$ executes the sequential parts and all processors together perform the parallel part. Processor $P(0)$ preserves the values of its variables on moving from one part to the next. The other processors do not inherit values; they can only obtain desired data values by communication. To allow the second mode of operation and to circumvent the restriction of bsp_begin and bsp_end being the first and last statement, the actual parallel part is made into a separate function spmd and an initializer

```
bsp_init(spmd, argc, argv);
```

is called as the first executable statement of the main function. Here, int argc and char **argv are the standard arguments of main in a C program, and these can be used to transfer parameters from a command line interface. Funny things may happen if this is not the first executable statement. Do not even think of trying it! The initializing statement is followed by: a sequential part, which may handle some input or ask for the desired number of processors (depending on the input size it may be better to use only part of the available processors); the parallel part, which is executed by spmd; and finally another sequential part, which may handle output. The sequential parts are optional.

The rules for I/O are simple: processor $P(0)$ is the only processor that can read from standard input or can access the file system, but all processors can write to standard output. Be aware that this may mix the output streams; use an fflush(stdout) statement to empty the output buffer immediately and increase the chance of obtaining ordered output (sorry, no guarantees).

At every point in the parallel part of the program, one can enquire about the total number of processors. This integer is returned by

```
bsp_nprocs();
```

The function bsp_nprocs also serves a second purpose: when it is used in the sequential part at the start, or in the bsp_begin statement, it returns the available number of processors, that is, the size of the BSP machine used. Any desired number of processors not exceeding the machine size can be assigned to

the program by `bsp_begin`. The local processor identity, or processor number, is returned by

```
bsp_pid();
```

It is an integer between 0 and `bsp_nprocs()`-1. One can also enquire about the time in seconds elapsed on the local processor since `bsp_begin`; this time is given as a double-precision value by

```
bsp_time();
```

Note that in the parallel context the **elapsed time**, or wall-clock time, is often the desired metric and not the CPU time. In parallel programs, processors are often idling because they have to wait for others to finish their part of a computation; a measurement of elapsed time includes idle time, whereas a CPU time measurement does not. Note, however, that the elapsed time metric does have one major disadvantage, in particular to your fellow users: you need to claim the whole BSP computer for yourself when measuring run times.

Each superstep of the SPMD part, or **program superstep**, is terminated by a global synchronization statement

```
bsp_sync();
```

except the last program superstep, which is terminated by `bsp_end`. The structure of a BSPlib program is illustrated in Fig. 1.7. Program supersteps may be contained in loops and if-statements, but the condition evaluations of these loops and if-statements must be such that all processors pace through the same sequence of program supersteps. The rules imposed by BSPlib may seem restrictive, but following them makes parallel programming easier, because they guarantee that all processors are in the same superstep. This allows us to assume full data integrity at the start of each superstep.

The version of the BSP model presented in Section 1.2 does not allow computation and communication in the same superstep. The BSPlib system automatically separates computation and communication, since it delays communication until all computation is finished. Therefore the user does not have to separate these parts herself and she also does not have to include a `bsp_sync` for this purpose. In practice, this means that BSPlib programs can freely mix computation and communication. The automatic separation feature of BSPlib is convenient, for instance because communication often involves address calculations and it would be awkward for a user to separate these computations from the corresponding communication operations. A program superstep can thus be viewed as a sequence of computation, implicit synchronization, communication, and explicit synchronization. The computation part or the communication part may be empty. Therefore, a program superstep may contain one or two supersteps as defined in the BSP model, namely a computation superstep and/or a communication superstep. From

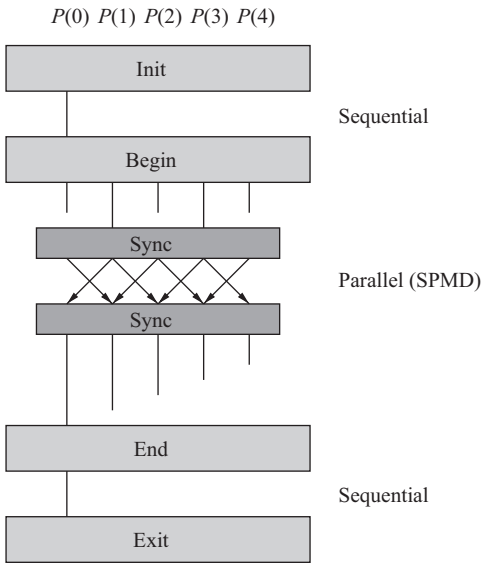

FIG. 1.7. Structure of a BSPlib program. The program first initializes the BSP machine to be used and then it performs a sequential computation on $P(0)$, followed by a parallel computation on five processors. It finishes with a sequential computation on $P(0)$.

now on, we use the shorter term 'superstep' to denote program supersteps as well, except when this would lead to confusion.

Wouldn't it be nice if we could compute and communicate at the same time? This tempting thought may have occurred to you by now. Indeed, processors could in principle compute while messages travel through the communication network. Exploiting this form of parallelism would reduce the total computation/communication cost $a + bg$ of the algorithm, but at most by a factor of two. The largest reduction would occur if the cost of each computation superstep were equal to the cost of the corresponding communication superstep, and if computation and communication could be overlapped completely. In most cases, however, either computation or communication dominates, and the cost reduction obtained by overlapping is insignificant. Surprisingly, BSPlib guarantees *not* to exploit potential overlap. Instead, delaying all communication gives more scope for optimization, since this allows the system to combine different messages from the same source to the same destination and to reorder the messages with the aim of balancing the communication traffic. As a result, the cost may be reduced by much more than a factor of two.

Processors can communicate with each other by using the `bsp_put` and `bsp_get` functions (or their high-performance equivalents `bsp_hpput` and

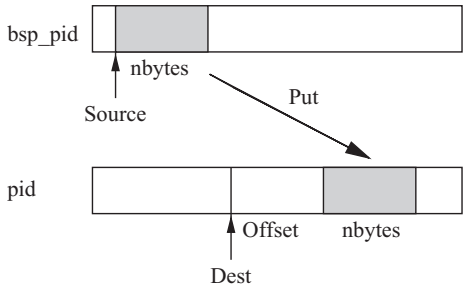

FIG. 1.8. Put operation from BSPlib. The `bsp_put` operation copies `nbytes` of data from the local processor `bsp_pid` into the specified destination processor `pid`. The pointer `source` points to the start of the data to be copied, whereas the pointer `dest` specifies the start of the memory area where the data is written. The data is written at `offset` bytes from the start.

`bsp_hpget`, see Exercise 10, or the `bsp_send` function, see Section 4.9). A processor that calls `bsp_put` reads data from its own memory and writes them into the memory of another processor. The function `bsp_put` corresponds to the put operation in our algorithms. The syntax is

```
bsp_put(pid, source, dest, offset, nbytes);
```

Here, `int pid` is the identity of the remote processor; `void *source` is a pointer to the source memory in the local processor from which the data are read; `void *dest` is a pointer to the destination memory in the remote processor into which the data are written; `int offset` is the number of bytes to be added to the address `dest` to obtain the address where writing starts; and `int nbytes` is the number of bytes to be written. The `dest` variable must have been registered previously; the registration mechanism will be explained soon. If `pid` equals `bsp_pid`, the put is done locally by a memory copy, and no data is communicated. The offset is determined by the local processor, but the destination address is part of the address space of the remote processor. The use of an offset separates the concerns of the local processor, which knows where in an array a data element should be placed, from the concerns of the remote processor, which knows the address of the array in its own address space. The `bsp_put` operation is illustrated in Fig. 1.8.

The `bsp_put` operation is safe in every sense, since the value to be put is first written into a local out-buffer, and only at the end of the superstep (when all computations in all processors are finished) it is transferred into a remote in-buffer, from which it is finally copied into the destination memory. The user can manipulate both the source and destination value without worrying about possible interference between data manipulation and transfer. Once the `bsp_put` is initiated, the user has got rid of the source data and can reuse the

variable that holds them. The destination variable can be used until the end of the superstep, when it will be overwritten. It is possible to put several values into the same memory cell, but of course only one value survives and reaches the next superstep. The user cannot know which value, and he bears the responsibility for ensuring correct program behaviour. Put and get operations do not block progress within their superstep: after a put or get is initiated, the program proceeds immediately.

Although a remote variable may have the same name as a local variable, it may still have a different physical memory address because each processor could have its own memory allocation procedure. To enable a processor to write into a remote variable, there must be a way to link the local name to the correct remote address. Linking is done by the registration primitive

```
bsp_push_reg(variable, nbytes);
```

where `void *variable` is a pointer to the variable being registered. All processors must simultaneously register a variable, or the `NULL` pointer; they must also deregister simultaneously. This ensures that they go through the same sequence of registrations and deregistrations. Registration takes effect at the start of the next superstep. From that moment, all simultaneously registered variables are linked to each other. Usually, the name of each variable linked in a registration is the same, in the right SPMD spirit. Still, it is allowed to link variables with different names.

If a processor wants to put a value into a remote address, it can do this by using the local name that is linked to the remote name and hence to the desired remote address. The second registration parameter, `int nbytes`, is an upper bound on the number of bytes that can be written starting from `variable`. Its sole purpose is sanity checking: our hope is to detect insane programs in their youth.

A variable is deregistered by a call to

```
bsp_pop_reg(variable);
```

Within a superstep, the variables can be registered and deregistered in arbitrary order. The same variable may be registered several times, but with different sizes. (This may happen for instance as a result of registration of the same variable inside different functions.) A deregistration cancels the last registration of the variable concerned. The last surviving registration of a variable is the one valid in the next superstep. For each variable, a stack of registrations is maintained: a variable is pushed onto the stack when it is registered; and it is popped off the stack when it is deregistered. A **stack** is the computer science equivalent of the hiring and firing principle for teachers in the Dutch educational system: Last In, First Out (LIFO). This keeps the average stack population old, but that property is irrelevant for our book. In a sensible program, the number of registrations is kept limited. Preferably, a registered variable is reused many times, to amortize the associated

overhead costs. Registration is costly because it requires a broadcast of the registered local variable to the other processors and possibly an additional synchronization.

A processor that calls the `bsp_get` function reads data from the memory of another processor and writes them into its own memory. The syntax is

```
bsp_get(pid, source, offset, dest, nbytes);
```

The parameters of `bsp_get` have the same meaning as those of `bsp_put`, except that the source memory is in the remote processor and the destination memory in the local processor and that the offset is in the source memory. The offset is again computed by the local processor. The `source` variable must have been registered previously. The value obtained by the `bsp_get` operation is the source value immediately *after* the computations of the present superstep have terminated, but *before* it can be modified by other communication operations.

If a processor detects an error, it can take action and bring down all other processors in a graceful manner by a call to

```
bsp_abort(error_message);
```

Here, `error_message` is an output string such as used by the `printf` function in C. Proper use of the abort facility makes it unnecessary to check periodically whether all processors are still alive and computing.

BSPlib contains only core operations. By keeping the core small, the BSPlib designers hoped to enable quick and efficient implementation of BSPlib on every parallel computer that appears on the market. Higher level functions such as broadcasting or global summing, generally called **collective communication**, are useful but not really necessary. Of course, users can write their own higher level functions on top of the primitive functions, giving them exactly the desired functionality, or use the predefined ones available in the Oxford BSP toolset [103]. Section 2.5 gives an example of a collective-communication function, the broadcast of a vector.

The best way of learning to use the library is to study an example and then try to write your own program. Below, we present the function `bspip`, which is an implementation of Algorithm 1.1 in C using BSPlib, and the test program `bspinprod`, which handles input and output for a test problem. Now, try to compile the program by the UNIX command

```
bspcc -o ip bspinprod.c bspedupack.c -lm
```

and run the resulting executable program `ip` on four processors by the command

```
bsprun -npes 4 ip
```

and see what happens for this particular test problem, defined by $x_i = y_i = i + 1$, for $0 \leq i < n$. (If you are not running a UNIX variant, you may have to follow a different procedure.) A listing of the file `bspedupack.c` can be found

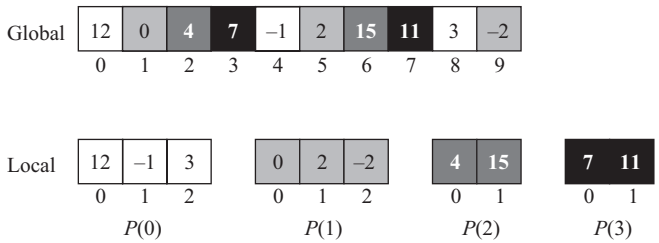

FIG. 1.9. Two different views of the same vector. The vector of size ten is distributed by the cyclic distribution over four processors. The numbers in the square cells are the numerical values of the vector components. The processors are shown by greyshades. The global view is used in algorithms, where vector components are numbered using global indices j. The local view is used in implementations, where each processor has its own part of the vector and uses its own local indices `j`.

in Appendix A. It contains functions for allocation and deallocation of vectors and matrices.

The relation between Algorithm 1.1 and the function `bspip` is as follows. The variables p, s, t, n, α of the algorithm correspond to the variables `p, s, t, n, alpha` of the function. The local inner product α_s of $P(s)$ is denoted by `inprod` in the program text of $P(s)$, and it is also put into `Inprod[s]` in all processors. The global index i in the algorithm equals `i*p+s`, where `i` is a local index in the program. The vector component x_i corresponds to the variable `x[i]` on the processor that owns x_i, that is, on processor $P(i \bmod p)$. The number of local indices on $P(s)$ is `nloc(p,s,n)`. The first $n \bmod p$ processors have $\lceil n/p \rceil$ such indices, while the others have $\lfloor n/p \rfloor$. Note the efficient way in which `nloc` is computed and check that this method is correct, by writing $n = ap + b$ with $0 \leq b < p$ and expanding the expression returned by the function.

It is convenient to describe algorithms such as Algorithm 1.1 in global variables, but to implement them using local variables. This avoids addressing by a stride and its worse alternative, the superfluous test '**if** $i \bmod p = s$' in a loop over the global index i. Using local, consecutive indices is most natural in an implementation because only a subarray of the original global array is stored locally. The difference between the global and local view is illustrated in Fig. 1.9. For all our distributions, we store the local vector components in order of increasing global index. This gives rise to a natural mapping between local and global indices.

The program printed below largely explains itself; a few additional explanations are given in the following. The included file `bspedupack.h` can be found in Appendix A. It contains inclusion statements for standard header files and also constants such as the size of a double `SZDBL`. The number of

processors is first stored as a global variable `P` (global in the C sense, that is, accessible to all functions in its file), so that we are able to transfer the value of `P` from the main program to the SPMD part. Values cannot be transferred other than by using global variables, because the SPMD function is not allowed to have parameters. The function `vecallocd` from `bspedupack.c` is used to allocate an array of doubles of length `p` dynamically and `vecfreed` is used to free the array afterwards.

The offset in the first `bsp_put` is `s*SZDBL`, since the local inner product of processor `s` is put into `Inprod`[`s`] on every processor `t`. The processors synchronize before the time measurements by `bsp_time`, so that the measurements start and finish simultaneously.

```
#include "bspedupack.h"

/*  This program computes the sum of the first n squares, for n>=0,
        sum = 1*1 + 2*2 + ... + n*n
    by computing the inner product of x=(1,2,...,n)^T and itself.
    The output should equal n*(n+1)*(2n+1)/6.
    The distribution of x is cyclic.
*/

int P; /* number of processors requested */

int nloc(int p, int s, int n){
    /* Compute number of local components of processor s for vector
       of length n distributed cyclically over p processors. */

    return  (n+p-s-1)/p ;

} /* end nloc */

double bspip(int p, int s, int n, double *x, double *y){
    /* Compute inner product of vectors x and y of length n>=0 */

    int nloc(int p, int s, int n);
    double inprod, *Inprod, alpha;
    int i, t;

    Inprod= vecallocd(p); bsp_push_reg(Inprod,p*SZDBL);
    bsp_sync();

    inprod= 0.0;
    for (i=0; i<nloc(p,s,n); i++){
        inprod += x[i]*y[i];
    }

    for (t=0; t<p; t++){
        bsp_put(t,&inprod,Inprod,s*SZDBL,SZDBL);
    }
    bsp_sync();
```

```
    alpha= 0.0;
    for (t=0; t<p; t++){
        alpha += Inprod[t];
    }
    bsp_pop_reg(Inprod); vecfreed(Inprod);

    return alpha;

} /* end bspip */

void bspinprod(){

    double bspip(int p, int s, int n, double *x, double *y);
    int nloc(int p, int s, int n);
    double *x, alpha, time0, time1;
    int p, s, n, nl, i, iglob;

    bsp_begin(P);
    p= bsp_nprocs(); /* p = number of processors obtained */
    s= bsp_pid();    /* s = processor number */
    if (s==0){
        printf("Please enter n:\n"); fflush(stdout);
        scanf("%d",&n);
        if(n<0)
            bsp_abort("Error in input: n is negative");
    }
    bsp_push_reg(&n,SZINT);
    bsp_sync();

    bsp_get(0,&n,0,&n,SZINT);
    bsp_sync();
    bsp_pop_reg(&n);

    nl= nloc(p,s,n);
    x= vecallocd(nl);
    for (i=0; i<nl; i++){
        iglob= i*p+s;
        x[i]= iglob+1;
    }
    bsp_sync();
    time0=bsp_time();

    alpha= bspip(p,s,n,x,x);
    bsp_sync();
    time1=bsp_time();

    printf("Processor %d: sum of squares up to %d*%d is %.lf\n",
            s,n,n,alpha); fflush(stdout);
    if (s==0){
        printf("This took only %.6lf seconds.\n", time1-time0);
        fflush(stdout);
    }
```

```
    vecfreed(x);
    bsp_end();

} /* end bspinprod */

int main(int argc, char **argv){

    bsp_init(bspinprod, argc, argv);

    /* sequential part */
    printf("How many processors do you want to use?\n");
    fflush(stdout);
    scanf("%d",&P);
    if (P > bsp_nprocs()){
        printf("Sorry, not enough processors available.\n");
        fflush(stdout);
        exit(1);
    }

    /* SPMD part */
    bspinprod();

    /* sequential part */
    exit(0);

} /* end main */
```

1.5 BSP benchmarking

Computer benchmarking is the activity of measuring computer performance by running a representative set of test programs. The performance results for a particular sequential computer are often reduced in some ruthless way to one number, the computing rate in flop/s. This allows us to rank different computers according to their computing rate and to make informed decisions on what machines to buy or use. The performance of parallel computers must be expressed in more than a single number because communication and synchronization are just as important for these computers as computation. The BSP model represents machine performance by a parameter set of minimal size: for a given number of processors p, the parameters r, g, and l represent the performance for computation, communication, and synchronization. Every parallel computer can be viewed as a BSP computer, with good or bad BSP parameters, and hence can also be benchmarked as a BSP computer. In this section, we present a method for BSP benchmarking. The aim of the method is to find out what the BSP computer looks like to an average user, perhaps you or me, who writes parallel programs but does not really want to spend much time optimizing

programs, preferring instead to let the compiler and the BSP system do the job. (The benchmark method for optimization enthusiasts would be very different.)

The sequential computing rate r is determined by measuring the time of a so-called DAXPY operation ('Double precision A times X Plus Y'), which has the form $\mathbf{y} := \alpha\mathbf{x} + \mathbf{y}$, where $\mathbf{x}$ and $\mathbf{y}$ are vectors and α is a scalar. A DAXPY with vectors of length n contains n additions and n multiplications and some overhead in the form of $\mathcal{O}(n)$ address calculations. We also measure the time of a DAXPY operation with the addition replaced by subtraction. We use 64-bit arithmetic throughout; on most machines this is called **double-precision arithmetic**. This mixture of operations is representative for the majority of scientific computations. We measure the time for a vector length, which on the one hand is large enough so that we can ignore the startup costs of vector operations, but on the other hand is small enough for the vectors to fit in the cache; a choice of $n = 1024$ is often adequate. A **cache** is a small but fast intermediate memory that allows immediate reuse of recently accessed data. Proper use of the cache considerably increases the computing rate on most modern computers. The existence of a cache makes the life of a benchmarker harder, because it leads to two different computing rates: a flop rate for in-cache computations and a rate for out-of-cache computations. Intelligent choices should be made if the performance results are to be reduced to a single meaningful figure.

The DAXPY measurement is repeated a number of times, both to obtain a more accurate clock reading and to amortize the cost of bringing the vector into the cache. We measure the sequential computing rate of each processor of the parallel computer, and report the minimum, average, and maximum rate. The difference between the minimum and the maximum indicates the accuracy of the measurement, except when the processors genuinely differ in speed. (One processor that is slower than the others can have a remarkable effect on the overall time of a parallel computation!) We take the average computing rate of the processors as the final value of r. Note that our measurement is representative for user programs that contain mostly hand-coded vector operations. To realize top performance, system-provided matrix–matrix operations should be used wherever possible, because these are often efficiently coded in assembler language. Our benchmark method does not reflect that situation.

The communication parameter g and the synchronization parameter l are obtained by measuring the time of full h-relations, where each processor sends and receives exactly h data words. To be consistent with the measurement of r, we use double-precision reals as data words. We choose a particularly demanding test pattern from the many possible patterns with the same h, which reflects the typical way most users would handle communication in their programs. The destination processors of the values to be sent

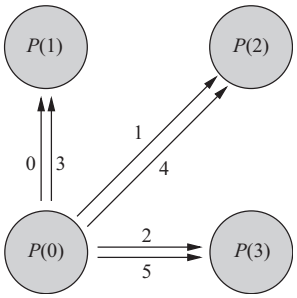

FIG. 1.10. Communication pattern of the 6-relation in the BSP benchmark. Processors send data to the other processors in a cyclic manner. Only the data sent by processor $P(0)$ are shown; other processors send data in a similar way. Each arrow represents the communication of one data word; the number shown is the index of the data word.

are determined in a cyclic fashion: $P(s)$ puts h values in remote processors $P(s+1), P(s+2), \ldots, P(p-1), P(0), \ldots, P(s-1), P(s+1), \ldots$, wrapping around at processor number p and skipping the source processor to exclude local puts that do not require communication, see Fig. 1.10. In this communication pattern, all processors receive the same number, h, of data words (this can easily be proven by using a symmetry argument). The destination processor of each communicated value is computed before the actual h-relation is performed, to prevent possibly expensive modulo operations from influencing the timing results of the h-relation. The data are sent out as separate words, to simulate the typical situation in an application program where the user does not worry about the size of the data packets. This is the task of the BSP system, after all! Note that this way of benchmarking h-relations is a test of both the machine and the BSP library for that machine. Library software can combine several data words into one packet, if they are sent in one superstep from the same source processor to the same destination processor. An efficient library will package such data automatically and choose an optimal packet size for the particular machine used. This results in a lower value of g, because the communication startup cost of a packet is amortized over several words of data.

Our variant of the BSP model assumes that the time $T_{\text{comm}}(h)$ of an h-relation is linear in h, see (1.2). In principle, it would be possible to measure $T_{\text{comm}}(h)$ for two values of h and then determine g and l. Of course, the results would then be highly sensitive to measurement error. A better way of doing this is by measuring $T_{\text{comm}}(h)$ for a range of values h_0–h_1, and then finding the best least-squares approximation, given by the values of g and l that minimize

the error

$$E_{\mathrm{LSQ}} = \sum_{h=h_0}^{h_1} (T_{\mathrm{comm}}(h) - (hg + l))^2. \tag{1.9}$$

(These values are obtained by setting the partial derivatives with respect to g and l to zero, and solving the resulting 2×2 linear system of equations.) We choose $h_0 = p$, because packet optimization becomes worthwhile for $h \geq p$; we would like to capture the behaviour of the machine and the BSP system for such values of h. A value of $h_1 = 256$ will often be adequate, except if $p \geq 256$ or if the asymptotic communication speed is attained only for very large h.

Timing parallel programs requires caution since ultimately it often relies on a system timer, which may be hidden from the user and may have a low resolution. Always take a critical look at your timing results and your personal watch and, in case of suspicion, plot the output data in a graph! This may save you from potential embarrassment: on one occasion, I was surprised to find that according to an erroneous timer the computer had exceeded its true performance by a factor of four. On another occasion, I found that g was negative. The reason was that the particular computer used had a small g but a huge l, so that for $h \leq h_1$ the measurement error in $gh + l$ was much larger than gh, thereby rendering the value of g meaningless. In this case, h_1 had to be increased to obtain an accurate measurement of g.

1.6 Example program `bspbench`

The program `bspbench` is a simple benchmarking program that measures the BSP parameters of a particular computer. It is an implementation of the benchmarking method described in Section 1.5. In the following, we present and explain the program. The least-squares function of `bspbench` solves a 2×2 linear system by subtracting a multiple of one equation from the other. Dividing by zero or by a small number is avoided by subtracting the equation with the largest leading coefficient. (Solving a 2×2 linear system is a prelude to Chapter 2, where large linear systems are solved by LU decomposition.)

The computing rate r of each processor is measured by using the `bsp_time` function, which gives the elapsed time in seconds for the processor that calls it. The measurements of r are independent, and hence they do not require timer synchronization. The number of iterations `NITERS` is set such that each DAXPY pair (and each h-relation) is executed 100 times. You may decrease the number if you run out of patience while waiting for the results.

The h-relation of our benchmarking method is implemented as follows. The data to be sent are put into the array `dest` of the destination processors. The destination processor `destproc[i]` is determined by starting with the next

processor $s+1$ and allocating the indices `i` to the $p-1$ remote processors in a cyclic fashion, that is, by adding `i` mod $(p-1)$ to the processor number $s+1$. Taking the resulting value modulo p then gives a valid processor number unequal to s. The destination index `destindex`[`i`] is chosen such that each source processor fills its own part in the `dest` arrays on the other processors: $P(s)$ fills locations $s, s+p, s+2p$, and so on. The locations are defined by `destindex`[`i`] $= s+$(`i` div $(p-1))p$, because we return to the same processor after each round of $p-1$ puts into different destination processors. The largest destination index used in a processor is at most $p-1+((h-1)$ div $(p-1))p < p+2 \cdot$ `MAXH`, which guarantees that the array `dest` is sufficiently large.

```
#include "bspedupack.h"

/*  This program measures p, r, g, and l of a BSP computer
    using bsp_put for communication.
*/

#define NITERS 100     /* number of iterations */
#define MAXN 1024      /* maximum length of DAXPY computation */
#define MAXH 256       /* maximum h in h-relation */
#define MEGA 1000000.0

int P; /* number of processors requested */

void leastsquares(int h0, int h1, double *t, double *g, double *l){
    /* This function computes the parameters g and l of the
       linear function T(h)= g*h+l that best fits
       the data points (h,t[h]) with h0 <= h <= h1. */

    double nh, sumt, sumth, sumh, sumhh, a;
    int h;

    nh= h1-h0+1;
    /* Compute sums:
        sumt  =  sum of t[h] over h0 <= h <= h1
        sumth =         t[h]*h
        sumh  =         h
        sumhh =         h*h     */
    sumt= sumth= 0.0;
    for (h=h0; h<=h1; h++){
        sumt  += t[h];
        sumth += t[h]*h;
    }
    sumh= (h1*h1-h0*h0+h1+h0)/2;
    sumhh= ( h1*(h1+1)*(2*h1+1) - (h0-1)*h0*(2*h0-1))/6;

    /* Solve      nh*l +  sumh*g =  sumt
                sumh*l + sumhh*g = sumth */
```

```
    if(fabs(nh)>fabs(sumh)){
        a= sumh/nh;
        /* subtract a times first eqn from second eqn */
        *g= (sumth-a*sumt)/(sumhh-a*sumh);
        *l= (sumt-sumh* *g)/nh;
    } else {
        a= nh/sumh;
        /* subtract a times second eqn from first eqn */
        *g= (sumt-a*sumth)/(sumh-a*sumhh);
        *l= (sumth-sumhh* *g)/sumh;
    }

} /* end leastsquares */

void bspbench(){
    void leastsquares(int h0, int h1, double *t, double *g, double *l);
    int p, s, s1, iter, i, n, h, destproc[MAXH], destindex[MAXH];
    double alpha, beta, x[MAXN], y[MAXN], z[MAXN], src[MAXH], *dest,
           time0, time1, time, *Time, mintime, maxtime,
           nflops, r, g0, l0, g, l, t[MAXH+1];

    /**** Determine p ****/
    bsp_begin(P);
    p= bsp_nprocs(); /* p = number of processors obtained */
    s= bsp_pid();    /* s = processor number */

    Time= vecallocd(p); bsp_push_reg(Time,p*SZDBL);
    dest= vecallocd(2*MAXH+p); bsp_push_reg(dest,(2*MAXH+p)*SZDBL);
    bsp_sync();

    /**** Determine r ****/
    for (n=1; n <= MAXN; n *= 2){
        /* Initialize scalars and vectors */
        alpha= 1.0/3.0;
        beta= 4.0/9.0;
        for (i=0; i<n; i++){
            z[i]= y[i]= x[i]= (double)i;
        }
        /* Measure time of 2*NITERS DAXPY operations of length n */
        time0=bsp_time();
        for (iter=0; iter<NITERS; iter++){
            for (i=0; i<n; i++)
                y[i] += alpha*x[i];
            for (i=0; i<n; i++)
                z[i] -= beta*x[i];
        }
        time1= bsp_time();
        time= time1-time0;
        bsp_put(0,&time,Time,s*SZDBL,SZDBL);
        bsp_sync();

        /* Processor 0 determines minimum, maximum, average
```

```
            computing rate */
        if (s==0){
            mintime= maxtime= Time[0];
            for(s1=1; s1<p; s1++){
                mintime= MIN(mintime,Time[s1]);
                maxtime= MAX(maxtime,Time[s1]);
            }
            if (mintime>0.0){
                /* Compute r = average computing rate in flop/s */
                nflops= 4*NITERS*n;
                r= 0.0;
                for(s1=0; s1<p; s1++)
                    r += nflops/Time[s1];
                r /= p;
                printf("n= %5d min= %7.3lf max= %7.3lf av= %7.3lf Mflop/s ",
                       n, nflops/(maxtime*MEGA),nflops/
                        (mintime*MEGA), r/MEGA);
                fflush(stdout);
                /* Output for fooling benchmark-detecting compilers */
                printf(" fool=%7.1lf\n",y[n-1]+z[n-1]);
            } else
                printf("minimum time is 0\n"); fflush(stdout);
        }
    }

    /**** Determine g and l ****/
    for (h=0; h<=MAXH; h++){
        /* Initialize communication pattern */
        for (i=0; i<h; i++){
            src[i]= (double)i;
            if (p==1){
                destproc[i]=0;
                destindex[i]=i;
            } else {
                /* destination processor is one of the p-1 others */
                destproc[i]= (s+1 + i%(p-1)) %p;
                /* destination index is in my own part of dest */
                destindex[i]= s + (i/(p-1))*p;
            }
        }

        /* Measure time of NITERS h-relations */
        bsp_sync();
        time0= bsp_time();
        for (iter=0; iter<NITERS; iter++){
            for (i=0; i<h; i++)
                bsp_put(destproc[i],&src[i],dest,destindex[i]*SZDBL,
                        SZDBL);
            bsp_sync();
        }
        time1= bsp_time();
        time= time1-time0;
```

```
        /* Compute time of one h-relation */
        if (s==0){
            t[h]= (time*r)/NITERS;
            printf("Time of %5d-relation= %lf sec= %8.0lf flops\n",
                   h, time/NITERS, t[h]); fflush(stdout);
        }
    }

    if (s==0){
        printf("size of double = %d bytes\n",(int)SZDBL);
        leastsquares(0,p,t,&g0,&l0);
        printf("Range h=0 to p   : g= %.1lf, l= %.1lf\n",g0,l0);
        leastsquares(p,MAXH,t,&g,&l);
        printf("Range h=p to HMAX: g= %.1lf, l= %.1lf\n",g,l);
        printf("The bottom line for this BSP computer is:\n");
        printf("p= %d, r= %.3lf Mflop/s, g= %.1lf, l= %.1lf\n",
               p,r/MEGA,g,l);
        fflush(stdout);
    }
    bsp_pop_reg(dest); vecfreed(dest);
    bsp_pop_reg(Time); vecfreed(Time);

    bsp_end();
} /* end bspbench */

int main(int argc, char **argv){

    bsp_init(bspbench, argc, argv);
    printf("How many processors do you want to use?\n");
    fflush(stdout);
    scanf("%d",&P);
    if (P > bsp_nprocs()){
        printf("Sorry, not enough processors available.\n");
        exit(1);
    }
    bspbench();
    exit(0);

} /* end main */
```

1.7 Benchmark results

What is the cheapest parallel computer you can buy? Two personal computers connected by a cable. What performance does this configuration give you? Certainly $p = 2$, and perhaps $r = 122$ Mflop/s, $g = 1180$, and $l = 138\,324$. These BSP parameters were obtained by running the program `bspbench` on two PCs from a cluster that fills up a cabinet at the Oxford office of Sychron. This cluster consists of 11 identical Pentium-II PCs with a clock rate of 400 MHz running the Linux operating system, and a connection network of four Fast

Ethernets. Each Ethernet of this cluster consists of a Cisco Catalyst switch with 11 cables, each ending in a Network Interface Card (NIC) at the back of a different PC. (A **switch** connects pairs of PCs; pairs can communicate independently from other pairs.) Thus, each Ethernet connects all PCs. Having four Ethernets increases the communication capacity fourfold and it gives each PC four possible routes to every other PC, which is useful if some Ethernets are tied down by other communicating PCs. A Fast Ethernet is capable of transferring data at the rate of 100 Mbit/s (i.e. 12.5 Mbyte/s or 1 562 500 double-precision reals per second—ignoring overheads). The system software running on this cluster is the Sychron Virtual Private Server, which guarantees a certain specified computation/communication performance on part or all of the cluster, irrespective of use by others. On top of this system, the **portability layers** MPI and BSPlib are available. In our experiments on this machine, we used version 1.4 of BSPlib with communication optimization level 2 and the GNU C compiler with computation optimization level 3 (our compiler flags were `-flibrary-level 2 -O3`).

Figure 1.11 shows how you can build a parallel computer from cheap commodity components such as PCs running Linux, cables, and simple switches. This figure gives us a look inside the black box of the basic BSP architecture shown in Fig. 1.1. A parallel computer built in DIY (do-it-yourself) fashion is often called a Beowulf, after the hero warrior of the Old English epic, presumably in admiration of Beowulf's success in slaying the supers of his era. The Sychron cluster is an example of a small Beowulf; larger ones of hundreds of PCs are now being built by cost-conscious user groups in industry and in academia, see [168] for a how-to guide.

Figure 1.12 shows the time of an h-relation on two PCs of the Sychron Beowulf. The benchmark was run with `MAXN` decreased from the default

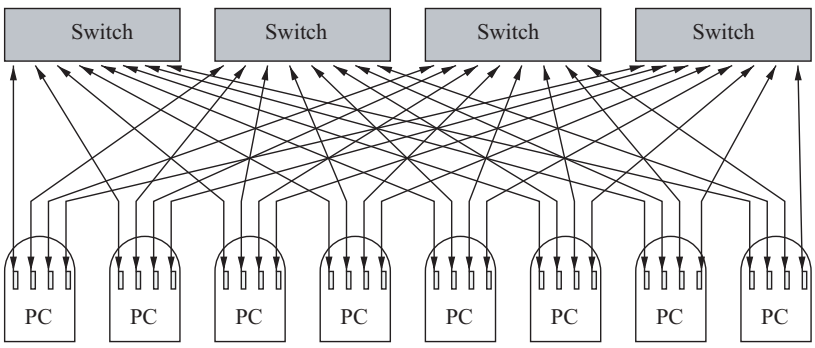

FIG. 1.11. Beowulf cluster of eight PCs connected by four switches. Each PC is connected to all switches.

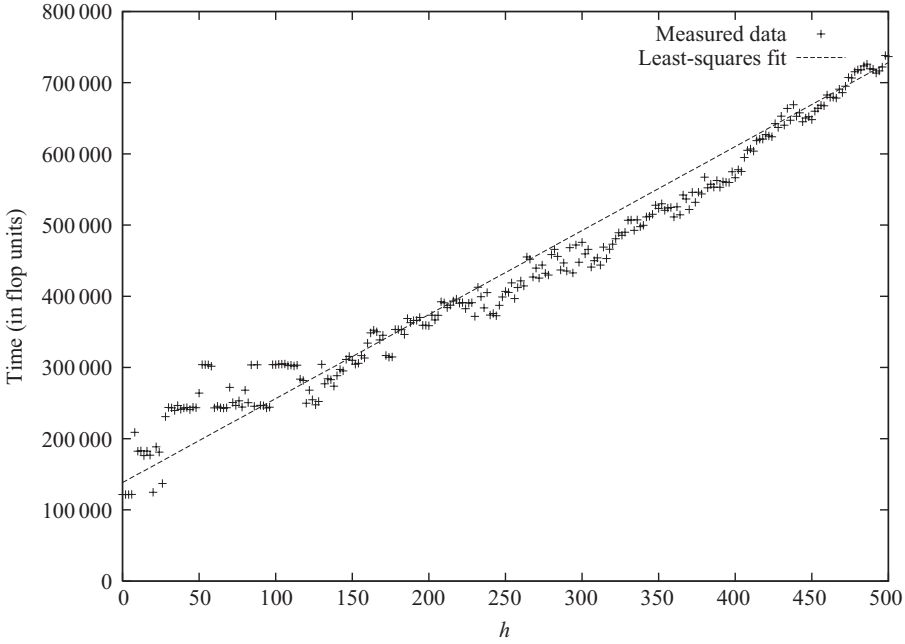

FIG. 1.12. Time of an h-relation on two connected PCs. The values shown are for even h with $h \leq 500$.

1024 to 512, to make the vectors fit in primary cache (i.e. the fastest cache); for length 1024 and above the computing rate decreases sharply. The value of `MAXH` was increased from the default 256 to 800, because in this case $g \ll l$ and hence a larger range of h-values is needed to obtain an accurate value for g from measurements of $hg + l$. Finding the right input parameters `MAXN`, `MAXH`, and `NITERS` for the benchmark program may require trial and error; plotting the data is helpful in this process. It is unlikely that one set of default parameters will yield sensible measurements for every parallel computer.

What is the most expensive parallel computer you can buy? A supercomputer, by definition. Commonly, a **supercomputer** is defined as one of today's top performers in terms of computing rate, communication/synchronization rate, and memory size. Most likely, the cost of a supercomputer will exceed a million US dollars. An example of a supercomputer is the Cray T3E, which is a massively parallel computer with distributed memory and a communication network in the form of a three-dimensional **torus** (i.e. a mesh with wraparound links at the boundaries). We have benchmarked up to 64 processors of the 128-processor machine called Vermeer,

after the famous Dutch painter, which is located at the HPαC supercomputer centre of Delft University of Technology. Each node of this machine consists of a DEC Alpha 21164 processor with a clock speed of 300 MHz, an advertised peak performance of 600 Mflop/s, and 128 Mbyte memory. In our experiments, we used version 1.4 of BSPlib with optimization level 2 and the Cray C compiler with optimization level 3.

The measured single-processor computing rate is 35 Mflop/s, which is much lower than the theoretical peak speed of 600 Mflop/s. The main reason for this discrepancy is that we measure the speed for a DAXPY operation written in C, whereas the highest performance on this machine can only be obtained by performing matrix–matrix operations such as DGEMM (Double precision GEneral Matrix–Matrix multiplication) and then only when using well-tuned subroutines written in assembler language. The BLAS (Basic Linear Algebra Subprograms) library [59,60,126] provides a portable interface to a set of subroutines for the most common vector and matrix operations, such as DAXPY and DGEMM. (The terms 'DAXPY' and 'DGEMM' originate in the BLAS definition.) Efficient BLAS implementations exist for most machines. A complete BLAS list is given in [61,Appendix C]. A Cray T3E version of the BLAS is available; its DGEMM approaches peak performance. A note of caution: our initial, erroneous result on the Cray T3E was a computing rate of 140 Mflop/s. This turned out to be due to the Cray timer IRTC, which is called by BSPlib on the Cray and ran four times slower than it should. This error occurs only in version 1.4 of BSPlib, in programs compiled at BSPlib level 2 for the Cray T3E.

Figure 1.13 shows the time of an h-relation on 64 processors of the Cray T3E. The time grows more or less linearly, but there are some odd jumps, for instance the sudden significant decrease around $h = 130$. (Sending more data takes less time!) It is beyond our scope to explain every peculiarity of every benchmarked machine. Therefore, we feel free to leave some surprises, like this one, unexplained.

Table 1.2 shows the BSP parameters obtained by benchmarking the Cray T3E for up to 64 processors. The results for $p = 1$ give an indication of the overhead of running a bulk synchronous parallel program on one processor. For the special case $p = 1$, the value of `MAXH` was decreased to 16, because $l \approx g$ and hence a smaller range of h-values is needed to obtain an accurate value for l from measurements of $hg + l$. (Otherwise, even negative values of l could appear.) The table shows that g stays almost constant for $p \leq 16$ and that it grows slowly afterwards. Furthermore, l grows roughly linearly with p, but occasionally it behaves strangely: l suddenly decreases on moving from 16 to 32 processors. The explanation is hidden inside the black box of the communication network. A possible explanation is the increased use of wrap-around links when increasing the number of processors. (For a small number of processors, all boundary links of a subpartition connect to other subpartitions, instead of wrapping around to the subpartition itself; thus, the subpartition

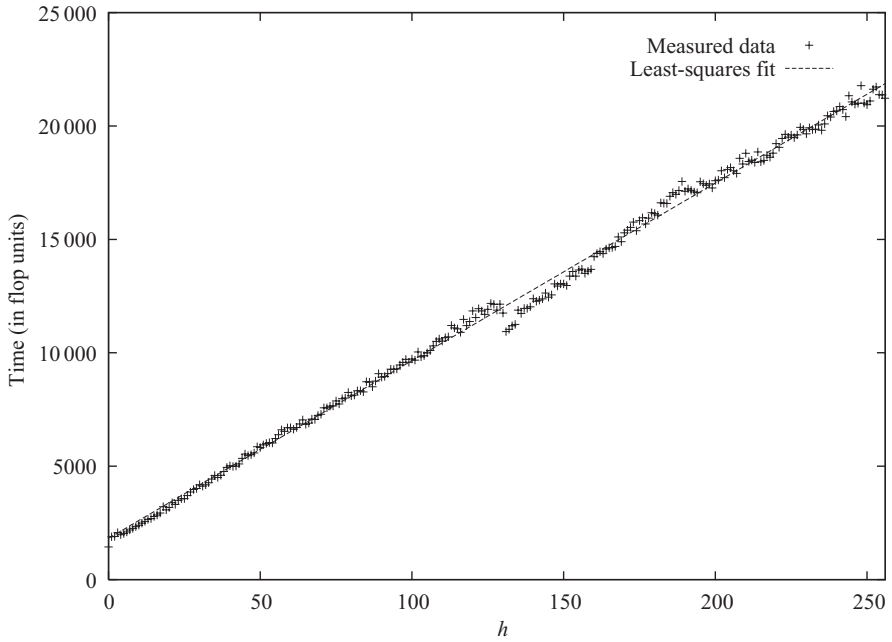

FIG. 1.13. Time of an h-relation on a 64-processor Cray T3E.

TABLE 1.2. Benchmarked BSP parameters p, g, l and the time of a 0-relation for a Cray T3E. All times are in flop units ($r = 35$ Mflop/s)

p	g	l	$T_{\text{comm}}(0)$
1	36	47	38
2	28	486	325
4	31	679	437
8	31	1193	580
16	31	2018	757
32	72	1145	871
64	78	1825	1440

is a mesh, rather than a torus. Increasing the number of processors makes the subpartition look more like a torus, with richer connectivity.) The time of a 0-relation (i.e. the time of a superstep without communication) displays a smoother behaviour than that of l, and it is presented here for comparison. This time is a lower bound on l, since it represents only part of the fixed cost of a superstep.

Another prominent supercomputer, at the time of writing these lines of course since machines come and go quickly, is the IBM RS/6000 SP. This machine evolved from the RS/6000 workstation and it can be viewed as a tightly coupled cluster of workstations (without the peripherals, of course). We benchmarked eight processors of the 76-processor SP at the SARA supercomputer centre in Amsterdam. The subpartition we used contains eight so-called thin nodes connected by a switch. Each node contains a 160 MHz PowerPC processor, 512 Mbyte of memory, and a local disk of 4.4 Gbyte. The theoretical peak rate of a processor is about 620 Mflop/s, similar to the Cray T3E above. Figure 1.14 shows the time of an h-relation on $p = 8$ processors of the SP. Note the effect of optimization by BSPlib: for $h < p$, the time of an h-relation increases rapidly with h. For $h = p$, however, BSPlib detects that the pth message of each processor is sent to the same destination as the first message, so that it can combine the messages. Every additional message can also be combined with a previous one. As a result, the number of messages does not increase further, and g grows only slowly. Also note the statistical outliers, that is, those values that differ considerably from the others. All outliers are high, indicating interference by message traffic from other users. Although the system guarantees exclusive access to the processors used, it does not guarantee the same for the communication network. This makes communication

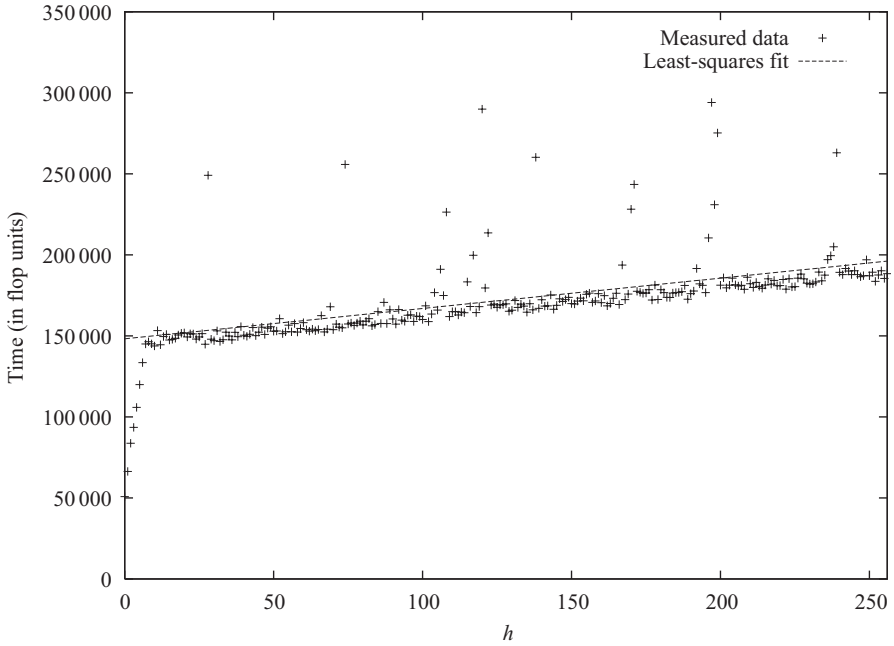

FIG. 1.14. Time of an h-relation on an 8-processor IBM SP.

benchmarking difficult. More reliable results can be obtained by repeating the benchmark experiment, or by organizing a party for one's colleagues (and sneaking out in the middle), in the hope of encountering a traffic-free period. A plot will reveal whether this has happened. The resulting BSP parameters of our experiment are $p = 8$, $r = 212$ Mflop/s, $g = 187$, and $l = 148\ 212$. Note that the interference from other traffic hardly influences g, because the slope of the fitted line is the same as that of the lower string of data (which can be presumed to be interference-free). The value of l, however, is slightly overestimated.

The last machine we benchmark is the Silicon Graphics Origin 2000. This architecture has a physically distributed memory, like the other three benchmarked computers. In principle, this promises scalability because memory, communication links, and other resources can grow proportionally with the number of processors. For ease of use, however, the memory can also be made to look like shared memory. Therefore, the Origin 2000 is often promoted as a 'Scalable Shared Memory Multiprocessor'. Following BSP doctrine, we ignore the shared-memory facility and use the Origin 2000 as a distributed-memory machine, thereby retaining the advantage of portability.

The Origin 2000 that we could lay our hands on is Oscar, a fine 86-processor parallel computer at the Oxford Supercomputer Centre of Oxford University. Each processing element of the Origin 2000 is a MIPS R10000 processor with a clock speed of 195 MHz and a theoretical peak performance of 390 Mflop/s. We compiled our programs using the SGI MIPSpro C-compiler with optimization flags switched on as before. We used eight processors in our benchmark. The systems managers were kind enough to empty the system from other jobs, and hence we could benchmark a dedicated system. The resulting BSP parameters are $p = 8$, $r = 326$ Mflop/s, $g = 297$, and $l = 95\ 686$. Figure 1.15 shows the results.

Table 1.3 presents benchmark results for three different computers with the number of processors fixed at eight. The parameters g and l are given not only in flops but also in raw microseconds to make it easy to compare machines with widely differing single-processor performance. The Cray T3E is the best balanced machine: the low values of g and l in flops mean that communication/synchronization performance of the Cray T3E is excellent, relative to the computing performance. The low values of l in microseconds tell us that in absolute terms synchronization on the Cray is still cheap. The main drawback of the Cray T3E is that it forces the user to put effort into optimizing programs, since straightforward implementations such as our benchmark do not attain top performance. For an unoptimized program, eight processors of the T3E are slower than a single processor of the Origin 2000. The SP and the Origin 2000 are similar as BSP machines, with the Origin somewhat faster in computation and synchronization.

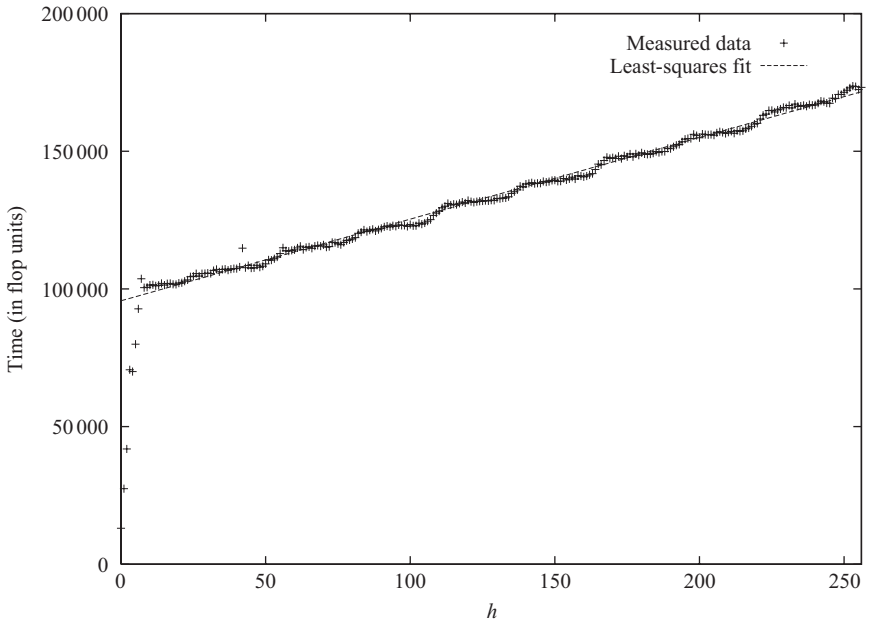

FIG. 1.15. Time of an h-relation on an 8-processor SGI Origin.

TABLE 1.3. Comparing the BSP parameters for three different parallel computers with $p = 8$

Computer	r (Mflop/s)	(flop)		(μs)	
		g	l	g	l
Cray T3E	35	31	1 193	0.88	34
IBM RS/6000 SP	212	187	148 212	0.88	698
SGI Origin 2000	326	297	95 686	0.91	294

1.8 Bibliographic notes

1.8.1 *BSP-related models of parallel computation*

Historically, the Parallel Random Access Machine (PRAM) has been the most widely studied general-purpose model of parallel computation. In this model, processors can read from and write to a shared memory. Several variants of the PRAM model can be distinguished, based on the way concurrent memory access is treated: the concurrent read, concurrent write (CRCW) variant allows full concurrent access, whereas the exclusive read, exclusive

write (EREW) variant allows only one processor to access the memory at a time. The PRAM model ignores communication costs and is therefore mostly of theoretical interest; it is useful in establishing lower bounds for the cost of parallel algorithms. The PRAM model has stimulated the development of many other models, including the BSP model. The BSP variant with automatic memory management by randomization in fact reduces to the PRAM model in the asymptotic case $g = l = \mathcal{O}(1)$. For an introduction to the PRAM model, see the survey by Vishkin [189]. For PRAM algorithms, see the survey by Spirakis and Gibbons [166,167] and the book by JáJá [113].

The BSP model has been proposed by Valiant in 1989 [177]. The full description of this 'bridging model for parallel computation' is given in [178]. This article describes the two basic variants of the model (automatic memory management or direct user control) and it gives a complexity analysis of algorithms for fast Fourier transform, matrix–matrix multiplication, and sorting. In another article [179], Valiant proves that a hypercube or butterfly architecture can simulate a BSP computer with optimal efficiency. (Here, the model is called XPRAM.) The BSP model as it is commonly used today has been shaped by various authors since the original work by Valiant. The survey by McColl [132] argues that the BSP model is a promising approach to general-purpose parallel computing and that it can deliver both scalable performance and architecture independence. Bisseling and McColl [21,22] propose the variant of the model (with pure computation supersteps of cost $w + l$ and pure communication supersteps of cost $hg+l$) that is used in this book. They show how a variety of scientific computations can be analysed in a simple manner by using their BSP variant. McColl [133] analyses and classifies several important BSP algorithms, including dense and sparse matrix–vector multiplication, matrix–matrix multiplication, LU decomposition, and triangular system solution.

The LogP model by Culler *et al.* [49] is an offspring of the BSP model, which uses four parameters to describe relative machine performance: the latency L, the overhead o, the gap g, and the number of processors P, instead of the three parameters l, g, and p of the BSP model. The LogP model treats messages individually, not in bulk, and hence it does not provide the notion of a superstep. The LogP model attempts to reflect the actual machine architecture more closely than the BSP model, but the price to be paid is an increase in the complexity of algorithm design and analysis. Bilardi *et al.* [17] show that the LogP and BSP model can simulate each other efficiently so that in principle they are equally powerful.

The YPRAM model by de la Torre and Kruskal [54] characterizes a parallel computer by its latency, bandwidth inefficiency, and recursive decomposability. The decomposable BSP (D-BSP) model [55] is the same model expressed in BSP terms. In this model, a parallel computer can be decomposed into submachines, each with their own parameters g and l. The parameters g and l of submachines will in general be lower than those of the complete machine. The

scaling behaviour of the submachines with p is described by functions $g(p)$ and $l(p)$. This work could provide a theoretical basis for subset synchronization within the BSP framework.

The BSPRAM model by Tiskin [174] replaces the communication network by a shared memory. At the start of a superstep, processors read data from the shared memory into their own local memory; then, they compute independently using locally held data; and finally they write local data into the shared memory. Access to the shared memory is in bulk fashion, and the cost function of such access is expressed in g and l. The main aim of the BSPRAM model is to allow programming in shared-memory style while keeping the benefits of data locality.

1.8.2 *BSP libraries*

The first portable BSP library was the Oxford BSP library by Miller and Reed [140,141]. This library contains six primitives: put, get, start of superstep, end of superstep, start of program, and end of program. The Cray SHMEM library [12] can be considered as a nonportable BSP library. It contains among others: put, strided put, get, strided get, and synchronization. The Oxford BSP library is similar to the Cray SHMEM library, but it is available for many different architectures. Neither of these libraries allows communication into dynamically allocated memory. The Green BSP library by Goudreau *et al.* [80,81] is a small experimental BSP library of seven primitives. The main difference with the Oxford BSP library is that the Green BSP library communicates by bulk synchronous message passing, which will be explained in detail in Chapter 4. The data sent in a superstep is written into a remote receive-buffer. This is one-sided communication, because the receiver remains passive when the data is being communicated. In the next superstep, however, the receiver becomes active: it must retrieve the desired messages from its receive-buffer, or else they will be lost forever. Goudreau *et al.* [80] present results of numerical experiments using the Green BSP library in ocean eddy simulation, computation of minimum spanning trees and shortest paths in graphs, n-body simulation, and matrix–matrix multiplication.

Several communication libraries and language extensions exist that enable programming in BSP style but do not fly the BSP flag. The Split-C language by Culler *et al.* [48] is a parallel extension of C. It provides put and get primitives and additional features such as global pointers and spread arrays. The Global Array toolkit [146] is a software package that allows the creation and destruction of distributed matrices. It was developed in first instance for use in computational chemistry. The Global Array toolkit and the underlying programming model include features such as one-sided communication, global synchronization, relatively cheap access to local memory, and uniformly more expensive access to remote memory. Co-Array Fortran [147],

formerly called F^{--}, is a parallel extension of Fortran 95. It represents a strict version of an SPMD approach: all program variables exist on all processors; remote variables can be accessed by appending the processor number in square brackets, for example, `x(3)[2]` is the variable `x(3)` on $P(2)$. A put is concisely formulated by using such brackets in the left-hand side of an assignment, for example, `x(3)[2]=y(3)`, and a get by using them on the right-hand side. Processors can be synchronized in subsets or even in pairs. The programmer needs to be aware of the cost implications of the various types of assignments.

BSPlib, used in this book, combines the capabilities of the Oxford BSP and the Green BSP libraries. It has grown into a *de facto* standard, which is fully defined by Hill *et al.* [105]. These authors also present results for fast Fourier transformation, randomized sample sorting, and n-body simulation using BSPlib. Frequently asked questions about BSP and BSPlib, such as 'Aren't barrier synchronizations expensive?' are answered by Skillicorn, Hill, and McColl [163].

Hill and Skillicorn [106] discuss how to implement BSPlib efficiently. They demonstrate that postponing communication is worthwhile, since this allows messages to be reordered (to avoid congestion) and combined (to reduce startup costs). If the natural ordering of the communication in a superstep requires every processor to put data first into $P(0)$, then into $P(1)$, and so on, this creates congestion at the destination processors, even if the total h-relation is well-balanced. To solve this problem, Hill and Skillicorn use a $p \times p$ Latin square, that is, a matrix with permutations of $\{0, \ldots, p-1\}$ in the rows and columns, as a schedule for the communication. An example is the 4×4 Latin square

$$R = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 0 \\ 2 & 3 & 0 & 1 \\ 3 & 0 & 1 & 2 \end{bmatrix}. \tag{1.10}$$

The communication of the superstep is done in p rounds. In round j, processor $P(i)$ sends all data destined for $P(r_{ij})$. In another article [107], Hill and Skillicorn discuss the practical issues of implementing a global synchronization on different machines. In particular, they show how to abuse the cache-coherence mechanism of a shared-memory parallel computer to obtain an extremely cheap global synchronization. (In one case, 13.4 times faster than the vendor-provided synchronization.) Donaldson, Hill, and Skillicorn [57] show how to implement BSPlib efficiently on top of the TCP/IP and UDP/IP protocols for a Network Of Workstations (NOW) connected by an Ethernet. They found that it is important to release data packets at an optimal rate into the Ethernet. Above this rate, data packets will collide too often, in which case they must be resent; this increases g. Below the optimal rate, the network

capacity is underused. Hill, Donaldson, and Lanfear [102] present an implementation of BSPlib that continually migrates p processes around a NOW, taking care to run them on the least used workstations. After the work loads change, for example, because the owner returns from lunch, the next global synchronization provides a natural point to stop the computation, dump all data onto disk, and restart the computation on a less busy set of workstations. As an additional benefit, this provides fault tolerance, for instance the capability to recover from hardware failures.

The Oxford BSP toolset [103] contains an implementation of the C and Fortran 90 versions of the BSPlib standard. This library is used in most numerical experiments of this book. The toolset also contains profiling tools [101,104] that can be used to measure and visualize the amount of computation and the amount of incoming and outgoing data of each processor during a sequence of supersteps.

The Paderborn University BSP (PUB) library [28,30] is an implementation of the C version of the BSPlib standard with extensions. One extension of PUB is **zero-cost synchronization** [5], also called oblivious synchronization, which exploits the fact that for certain regular computations each receiving processor $P(s)$ knows the number of data words $h_{\mathrm{r}}(s)$ it will receive. Processor $P(s)$ performs a `bsp_oblsync(`$h_{\mathrm{r}}(s)$`)` operation at the end of the superstep, instead of a `bsp_sync`. This type of synchronization is cheap because no communication is needed to determine that processors can move on to the next superstep. Another extension is partitioning (by using the primitive `bsp_partition`), which is decomposing a BSP machine into submachines, each of them again a BSP machine. The submachines must be rejoined later (by using the primitive `bsp_done`). Each submachine can again be partitioned, in a recursive fashion. The processors of a submachine must be numbered consecutively. The partitioning mechanism provides a disciplined form of subset synchronization. The PUB library also includes a large set of collective-communication functions. Bonorden *et al.* [29] compare the performance on the Cray T3E of PUB with that of the Oxford BSP toolset and MPI.

The one-sided communications added by MPI-2 [138] to the orginal MPI standard can also be viewed as comprising a BSP library. The MPI-2 primitives for one-sided communications are put, get, and accumulate; their use is demonstrated in the fifth program, `mpimv`, of Appendix C. For the complete MPI-2 standard with annotations, see [83]. For a tutorial introduction, see the book by Gropp, Lusk, and Thakur [85].

1.8.3 *The non-BSP world: message passing*

In contrast to the one-sided communication of BSP programming, traditional message passing uses two-sided communication, which involves both an active sender and an active receiver. The underlying model is that of communicating sequential processes (CSP) by Hoare [108]. Several libraries support this

type of communication. The first portable communication library, parallel virtual machine (PVM), was developed by Sunderam [171]. PVM is a message-passing library that enables computing on **heterogeneous networks**, that is, networks of computers with different architectures. PVM has evolved into a standard [75] and PVM implementations are available for many different parallel computers. The two strong points of PVM compared with other systems such as BSPlib and MPI-1 are its support of heterogeneous architectures and its capability of dynamic process creation. This means that processors can be added (or removed) during a computation. These features make PVM attractive for certain Beowulf clusters, in particular heterogeneous ones. Geist, Kohl, and Papadopoulus [76] compare PVM with MPI-1.

The message-passing interface (MPI) [137], based on traditional message passing, was defined in 1994 by the MPI Forum, a committee of users and manufacturers of parallel computers. The initial definition is now known as MPI-1. For the complete, most recent MPI-1 standard with annotations, see [164]. For a tutorial introduction, see the book by Gropp, Lusk, and Skjellum [84]. An introduction to parallel programming that uses MPI-1 is the textbook by Pacheco [152]. An introduction that uses PVM and MPI-1 for distributed-memory parallel programming and Pthreads for shared-memory programming is the textbook by Wilkinson and Allen [191]. An introduction to parallel computing that uses MPI-1 for distributed-memory parallel programming and Pthreads and OpenMP for shared-memory programming is the textbook by Grama *et al.* [82]. Today, MPI-1 is the most widely used portability layer for parallel computers.

1.8.4 *Benchmarking*

The BSPlib definition [105] presents results obtained by the optimized benchmarking program `bspprobe`, which is included in the Oxford BSP toolset [103]. The values of r and l that were measured by `bspprobe` agree well with those of `bspbench`. The values of g, however, are much lower: for instance, the value $g = 1.6$ for 32-bit words at $r = 47$ Mflop/s given in [105,Table 1] corresponds to $g = 0.07\,\mu$s for 64-bit words, which is 12.5 times less than the 0.88 μs measured by `bspbench`, see Table 1.3. This is due to the high optimization level of `bspprobe`: data are sent in blocks instead of single words and high-performance puts are used instead of buffered puts. The goal of `bspprobe` is to measure communication performance for optimized programs and hence its bottom line takes as g-value the asymptotic value for large blocks. The effect of such optimizations will be studied in Chapter 2. The program `bspprobe` measures g for two different h-relations with the same h: (i) a local cyclic shift, where every processor sends h data to the next higher-numbered processor; (ii) a global all-to-all procedure where every processor sends $h/(p-1)$ data to every one of the others. In most cases, the difference between the two g-values is small. This validates the basic assumption of the BSP model,

namely that costs can be expressed in terms of h. The program `bspprobe` takes as l-value the cost of a synchronization in the absence of communication, that is, $T_{\text{comm}}(0)$, see Table 1.2. BSP parameters obtained for the Green BSP library are given by Goudreau *et al.* [80].

Benchmark results for machines ranging from personal computers to massively parallel supercomputers are collected and regularly updated by Dongarra [58]. These results represent the total execution rates for solving a dense $n \times n$ linear system of equations with $n = 100$ and $n = 1000$ by the LINPACK software and with unlimited n by any suitable software. The slowest machine included is an HP48 GX pocket calculator which achieves 810 flop/s on the $n = 100$ benchmark. A PC based on the 3.06 GHz Intel Pentium-IV chip achieves a respectable 1.41 Gflop/s for $n = 100$ and 2.88 Gflop/s for $n = 1000$ (1 Gflop = 1 Gigaflop = 10^9 flop). Most interesting is the unrestricted benchmark, see [58,Table 3], which allows supercomputers to show off and demonstrate their capabilities. The table gives: $r_{\max}$, the maximum rate achieved; r_{peak}, the theoretical peak rate; $n_{\max}$, the size of the system at which $r_{\max}$ is achieved; and $n_{1/2}$, the size at which half of $r_{\max}$ is obtained. The $n_{1/2}$ parameter is widely used as a measure of startup overhead. Low $n_{1/2}$ values promise top rates already for moderate problem sizes, see *The Science of Computer Benchmarking* by Hockney [109]. The value of $r_{\max}$ is the basis for the Top 500 list of supercomputer sites, see `http://www.top500.org`.

To be called a supercomputer, at present a computer must achieve at least 1 Tflop/s (1 Tflop = 1 Teraflop = 10^{12} flop). The fastest existing number cruncher is the Earth Simulator, which was custom-built by NEC for the Japan Marine Science and Technology Center. This computer occupies its own building, has 5120 processors, and it solves a linear system of size $n = 1\,075\,200$ at $r_{\max} = 36$ Tflop/s. Of course, computing rates increase quickly, and when you read these lines, the fastest computer may well have passed the Pflop/s mark (1 Pflop = 1 Petaflop = 10^{15} flop).

1.9 Exercises

1. Algorithm 1.1 can be modified to combine the partial sums into one global sum by a different method. Let $p = 2^q$, with $q \geq 0$. Modify the algorithm to combine the partial sums by repeated pairing of processors. Take care that every processor obtains the final result. Formulate the modified algorithm exactly, using the same notation as in the original algorithm. Compare the BSP cost of the two algorithms. For which ratio l/g is the pairwise algorithm faster?

2. Analyse the following operations and derive the BSP cost for a parallel algorithm. Let $\mathbf{x}$ be the input vector (of size n) of the operation and $\mathbf{y}$ the output vector. Assume that these vectors are block distributed over p processors,

with $p \leq n$. Furthermore, k is an integer with $1 \leq k \leq n$. The operations are:

(a) Minimum finding: determine the index j of the component with the minimum value and subtract this value from every component: $y_i = x_i - x_j$, for all i.
(b) Shifting (to the right): assign $y_{(i+k) \bmod n} = x_i$.
(c) Smoothing: replace each component by a moving average $y_i = 1/(k+1) \sum_{j=i-k/2}^{i+k/2} x_j$, where k is even.
(d) Partial summing: compute $y_i = \sum_{j=0}^{i} x_j$, for all i. (This problem is an instance of the **parallel prefix** problem.)
(e) Sorting by counting: sort $\mathbf{x}$ by increasing value and place the result in $\mathbf{y}$. Each component x_i is an integer in the range 0–k, where $k \ll n$.

3. Get acquainted with your parallel computer before you use it.

(a) Run the program `bspbench` on your parallel computer. Measure the values of g and l for various numbers of processors. How does the performance of your machine scale with p?
(b) Modify `bspbench` to measure `bsp_get`s instead of `bsp_put`s. Run the modified program for various p. Compare the results with those of the original program.

4. Since their invention, computers have been used as tools in cryptanalytic attacks on secret messages; parallel computers are no exception. Assume a plain text has been encrypted by the classic method of monoalphabetic substitution, where each letter from the alphabet is replaced by another one and where blanks and punctuation characters are deleted. For such a simple encryption scheme, we can apply statistical methods to uncover the message. See Bauer [14] for more details and also for a fascinating history of cryptology.

(a) Let $\mathbf{t} = (t_0, \ldots, t_{n-1})^{\mathrm{T}}$ be a cryptotext of n letters and $\mathbf{t}'$ another cryptotext, of the same length, language, and encryption alphabet. With a bit of luck, we can determine the language of the texts by computing Friedman's Kappa value, also called the index of coincidence,

$$\kappa(\mathbf{t}, \mathbf{t}') = \frac{1}{n} \sum_{i=0}^{n-1} \delta(t_i, t_i').$$

Here, $\delta(x, y) = 1$ if $x = y$, and $\delta(x, y) = 0$ otherwise. The value of κ tells us how likely it is that two letters in the same position of the texts are identical. Write a parallel program that reads an encrypted text, splits it into two parts $\mathbf{t}$ and $\mathbf{t}'$ of equal size (dropping the last letter if necessary), and computes $\kappa(\mathbf{t}, \mathbf{t}')$. Motivate your choice of data distribution.

(b) Find a suitable cryptotext as input and compute its κ. Guess its language by comparing the result with the κ-values found by Kullback (reproduced in [14]): Russian 5.29%, English 6.61%, German 7.62%, French 7.78%.
(c) Find out whether Dutch is closer to English or German.
(d) Extend your program to compute all letter frequencies in the input text. In English, the 'e' is the most frequent letter; its frequency is about 12.5%.
(e) Run your program on some large plain texts in the language just determined to obtain a frequency profile of that language. Run your program on the cryptotext and establish its letter frequences. Now break the code.
(f) Is parallelization worthwhile in this case? When would it be?

5. (∗) Data compression is widely used to reduce the size of data files, for instance texts or pictures to be transferred over the Internet. The LZ77 algorithm by Ziv and Lempel [193] passes through a text and uses the most recently accessed portion as a reference dictionary to shorten the text, replacing repeated character strings by pointers to their first occurrence. The popular compression programs PKZIP and gzip are based on LZ77.

Consider the text

'yabbadabbadoo'

(Fred Flintstone, Stone Age). Assume we arrive at the second occurrence of the string 'abbad'. By going back 5 characters, we find a matching string of length 5. We can code this as the triple of decimal numbers (5,5,111), where the first number in the triple is the number of characters we have to go back and the second number the length of the matching string. The number 111 is the ASCII code for the lower-case 'o', which is the next character after the second 'abbad'. (The lower-case characters 'a'–'z' are numbered 97–122 in the ASCII set.) Giving the next character ensures progress, even if no match was found. The output for this example is: (0,0,121), (0,0,97), (0,0,98), (1,1,97), (0,0,100), (5,5,111), (1,1,−1). The '−1' means end of input. If more matches are possible, the longest one is taken. For longer texts, the search for a match is limited to the last m characters before the current character (the search window); the string to be matched is limited to the first n characters starting at the current character (the look-ahead window).

(a) Write a sequential function that takes as input a character sequence and writes as output an LZ77 sequence of triples (o, l, c), where o is the offset, that is, number of characters to be moved back, l the length of the matching substring, and c is the code for the next character. Use suitable data types for o, l, and c to save space. Take $m = n = 512$.

Also write a sequential function that decompresses the LZ77 sequence. Which is fastest, compression or decompression?

(b) Design and implement a parallel LZ77 compression algorithm. You may adapt the original algorithm if needed for parallelization as long as the output can be read by the sequential LZ77 program. Hint: make sure that each processor has all input data it needs, before it starts compressing.

(c) Now design a parallel algorithm that produces exactly the same output sequence as the sequential algorithm. You may need several passes through the data.

(d) Compare the compression factor of your compression programs with that of `gzip`. How could you improve the performance?

(e) Is it worthwhile to parallelize the decompression?

6. (∗) A **random number generator** (RNG) produces a sequence of real numbers, in most cases uniformly distributed over the interval [0,1], that are uncorrelated and at least appear to be random. (In fact, the sequence is usually generated by a computer in a completely deterministic manner.) A simple and widely used type of RNG is the **linear congruential generator** based on the integer sequence defined by

$$x_{k+1} = (ax_k + b) \bmod m,$$

where a, b, and m are suitable constants. The starting value x_0 is called the **seed**. The choice of constants is critical for the quality of the generated sequence. The integers x_k, which are between 0 and $m-1$, are converted to real values $r_k \in [0,1)$ by $r_k = x_k/m$.

(a) Express x_{k+p} in terms of x_k and some constants. Use this expression to design a parallel RNG, that is, an algorithm, which generates a different, hopefully uncorrelated, sequence for each of the p processors of a parallel computer. The local sequence of a processor should be a subsequence of the x_k.

(b) Implement your algorithm. Use the constants proposed by Lewis, Goodman, and Miller [129]: $a = 7^5$, $b = 0$, $m = 2^{31} - 1$. This is a simple multiplicative generator. Do not be tempted to use zero as a seed!

(c) For the statistically sophisticated. Design a statistical test to check the randomness of the local sequence of a processor, for example, based on the χ^2 test. Also design a statistical test to check the independence of the different local sequences, for example, for the case $p = 2$. Does the parallel RNG pass these tests?

(d) Use the sequential RNG to simulate a random walk on the two-dimensional integer lattice $\mathbf{Z}^2$, where the walker starts at (0,0), and at

each step moves north, east, south, or west with equal probability 1/4. What is the expected distance to the origin after 100 steps? Create a large number of walks to obtain a good estimate. Use the parallel RNG to accelerate your simulation.

(e) Improve the quality of your parallel RNG by adding a local shuffle to break up short distance correlations. The numbers generated are written to a buffer array of length 64 instead of to the output. The buffer is filled at startup; after that, each time a random number is needed, one of the array values is selected at random, written to the output, and replaced by a new number x_k. The random selection of the buffer element is done based on the last output number. The shuffle is due to Bays and Durham [16]. Check whether this improves the quality of the RNG. Warning: the resulting RNG has limited applicability, because m is relatively small. Better parallel RNGs exist, see for instance the SPRNG package [131], and in serious work such RNGs must be used.

7. (∗) The sieve of Eratosthenes (276–194 BC) is a method for generating all prime numbers up to a certain bound n. It works as follows. Start with the integers from 2 to n. The number 2 is a prime; cross out all larger multiples of 2. The smallest remaining number, 3, is a prime; cross out all larger multiples of 3. The smallest remaining number, 5, is a prime, etc.

(a) When can we stop?

(b) Write a sequential sieve program. Represent the integers by a suitable array.

(c) Analyse the cost of the sequential algorithm. Hint: the probability of an arbitrary integer $x \geq 2$ to be prime is about $1/\log x$, where $\log = \log_e$ denotes the natural logarithm. Estimate the total number of cross-out operations and use some calculus to obtain a simple formula. Add operation counters to your program to check the accuracy of your formula.

(d) Design a parallel sieve algorithm. Would you distribute the array over the processors by blocks, cyclically, or in some other fashion?

(e) Write a parallel sieve program `bspsieve` and measure its execution time for $n = 1000,\ 10\,000,\ 100\,000,\ 1\,000\,000$ and $p = 1, 2, 4, 8$, or use as many processors as you can lay your hands on.

(f) Estimate the BSP cost of the parallel algorithm and use this estimate to explain your time measurements.

(g) Can you reduce the cost further? Hints: for the prime q, do you need to start crossing out at $2q$? Does every processor cross out the same number of integers? Is all communication really necessary?

(h) Modify your program to generate twin primes, that is, pairs of primes that differ by two, such as (5, 7). (It is unknown whether there are infinitely many twin primes.)

(i) Extend your program to check the Goldbach conjecture: every even $k > 2$ is the sum of two primes. Choose a suitable range of integers to check. Try to keep the number of operations low. (The conjecture has been an open question since 1742.)