3 Parallel Performance Analysis

Programs need to be correct. Programs also need to be fast. In order to write efficient programs, one surely must know how to evaluate efficiency. One might take recourse to our prior understanding of efficiency in the sequential context and compare observed parallel performance to observed sequential performance. Or, we can define parallel efficiency independent of sequential performance. We may yet draw inspiration from the way efficiency is evaluated in a sequential context. Into that scheme, we would need to incorporate the impact of an increasing number of processors deployed to solve the given problem.

Question: How do you reason about how long an algorithm or program takes?

Efficiency has two metrics. The first is in an abstract setting, *e. g.*, asymptotic analysis[1] of the underlying algorithm. The second is concrete – how well does the algorithm's implementation behave in practice on the available hardware and on data sizes of interest. Both are important.

[1] The notion of asymptotic complexity is not described here. Readers not aware of this tool should refer to a book.
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 1990

There is no substitute for measuring the performance of the real implementation on real data. On the other hand, developing and testing iteratively on large parallel systems is prohibitively expensive. Most development occurs on a small scale: using only few processors, p, on small input of size n. The extrapolation of these tests to a much larger scale is deceptively hard, and we often must resort to simplified models and analysis tools.

Asymptotic analysis on simple models is sometimes criticized because it over-simplifies several complex dynamics (like cache behavior, out of order execution on multiple execution engines, instruction dependencies, etc.) and conceals constant multipliers. Nonetheless, with large input sizes that are common in parallel applications, asymptotic measures do have value. They can be computed somewhat easily, in a standardized setting and without requiring iterations on large supercomputers. And, concealing constants is a choice to some degree. Useful constants can and should be retained. Nonetheless, the abstract part of our analysis will employ the big-O notation to describe the number of steps an algorithm takes. It is a function of the input size n and the number of processors p.

Asymptotic notation or not, the time $t(n, p)$ to solve a problem in parallel is a function of n and p. For this purpose, we will generally count in p the number of sequential processors – they complete their program instructions in sequence. Naturally, we want both n and p to be variable to allow a wider choice of computing platforms. $t(n, p)$ is the number of steps taken by the slowest of the p processors deployed. Like we expect a program to run on varying input sizes, we also must design programs that run well with varying p. In reality, t is also a function of the core structure, network topology, cache sizes, etc., but taking a cue from the sequential analysis style, we will use a simplified model of a parallel system.

3.1 *Simple Parallel Model*

We need a simple model of computation steps to be able to evaluate performance. Random Access Machine (RAM) model[2] is a sequential such model, where simple arithmetic operations and memory operations take a unit time-step. Given that, one may evaluate the total number of time-steps taken by an algorithm in the worst case, or on average.

[2] Stephen A. Cook and Robert A. Reckhow. Time-bounded random access machines. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72, page 73–80, New York, NY, USA, 1972. Association for Computing Machinery. ISBN 9781450374576

We seek a similar simple model to capture parallelism.

1. A parallel system consists of p sequential processors, p is a variable and may be chosen to be a function of n. It may be fixed for the entire duration of the algorithm or could be allowed to vary from step to step[3]

[3] Varying p may seem odd at first, considering that most computing systems have a fixed size. Nonetheless, we do not generally design algorithms and programs for one specific machine. They must be flexible and support variable p. See Section 3.5 for a more detailed explanation.

2. Each processor has access to an unbounded number of constant-sized local memory locations, which are not accessible to other processors.

3. Each processor can read from or write to any local memory location in unit time.

4. Communicating a constant-sized message from processor i to processor j takes unit time.

5. Each processor takes unit time to perform simple arithmetic and logical operations.

This model is simple and more useful than it may first seem. Its major shortcoming is that the time taken by the network in message transmission is not modeled. The cost of synchronization is also ignored. Instead, it assumes that if a message addressed to processor i is sent by some other processor, it arrives instantaneously and processor i spends 1 time-unit reading it. In effect, processor i may receive a message at any time, and only the unit time spent in receiving is counted. This model works reasonably well in practice for programs based on the distributed-memory model. A more precise model accounts for the message transmission delay as well as the synchronization overhead.

3.2 Bulk-Synchronous Parallel Model

The Bulk-synchronous parallel model (*i.e.*, *BSP* model[4]) addresses those two shortcomings. At the same time, it avoids modeling synchronizations in too great a detail. The BSP model limits synchronization to defined points after every few local steps. Thus recognizing that synchronization is an occasional requirement, it groups instructions into *super-steps*. A super-step consists of any number of local arithmetic or memory steps, followed by one synchronization step. Just as in the simple model, an arbitrary number of processors is available per super-step. We continue to denote their count by p. Each processor has access to an arbitrary number of local memory locations.

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

1. Super-steps proceed in synchrony: all processors complete step s before any starts step $s+1$.

2. Super-step consists of local steps, followed by a synchronization step. The synchronization is a global event – all processors take this step, and its end indicates that all processors have reached the synchronization step. After the synchronization completes, the next super-step may begin. The time taken to synchronize is a function of p, the number of processors.

3. The time taken by a super-step includes the local computation times, which is the maximum time taken by any processor: L_s for super-step s. This can vary from super-step to super-step. L_s may depend on the input size n and processor count p.

4. In super-step s, processor i sends h_{s_i} point-to-point messages to other processors. The total number of messages sent in super-step s is $\sum h_{s_i} = h_s$. h_s may be a function of n and p.

5. The messages are all received in the synchronization step. Thus the received data is available only in the next super-step. This clearly defines the send-receive synchronization point.

Figure 3.1 depicts the super-steps in a BSP model. All processors perform local computation interspersed with sends. The messages go into the network, which delivers them to their destinations. At

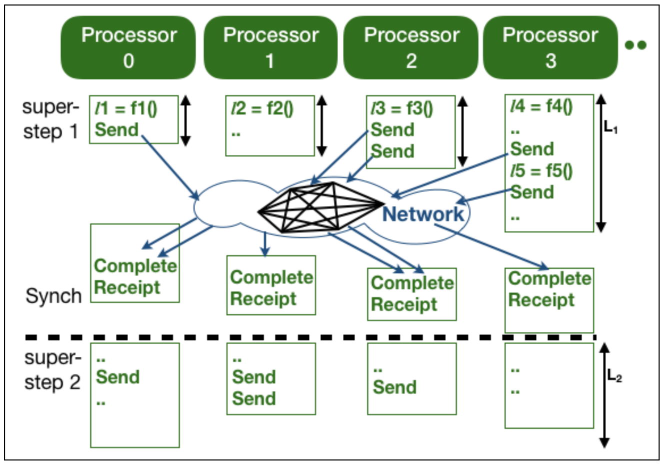

Figure 3.1: BSP computation model

the completion of these local steps, each processor proceeds to the synchronization *barrier*. Formally, no processor may cross this barrier until all processors have reached it and they have all received their messages.

BSP Computation Time

The time taken by a BSP algorithm is the sum of times taken by each super-step. The time taken by a super-step varies from step to step. This time includes computation time as well as data transmission and synchronization time. The computation time for super-step s is L_s, the maximum time taken by any processor. We assume in this analysis that the processors all execute at the same rate, meaning that a unit time is the same for all processors. The number of steps taken by different processors can then be compared to each other. It's a safe assumption for asymptotic analysis because the clock rates, in reality, do not differ by more than a constant factor. In fact, rates may not differ by much at all in practice, and even with the assumption of a common rate, the performance analysis is meaningful.

The data transmission time is proportional to h_s. For example, we might say that the network has the capacity to deliver $\frac{1}{t}$ messages per unit time. The time taken to deliver h_s message then is th_s. One might consider t to be a function of p, or let it be a constant for simpler analysis. Finally the synchronization time S_s is a function only of p.

Thus the super-step time is $L_s + th_s + S_s$. The total execution time =

$$\sum_{\forall parallel \text{ super-step } s} (L_s + th_s + S_s).$$

Each component is potentially a function if p. L_s and h_s may also depend on n. If p is constant for all steps, t and S_s can be taken as constants as well. In this case, $\sum S_s$ is proportional to the number of super-steps and $\sum(th_s)$ is proportional to the total number of messages sent by the algorithm and finally $\sum(L_s)$ measures the computation time across the steps.

This model does consider the synchronization overhead and idle times of processors. On the other hand, it ignores the complexities of network communication. For example, in real systems, multiple messages between two pairs could be batched, benefitting from a common setup time. The time need not be solely a function of the total number of messages. We have also seen in chapter 1 that not all pairs have equal latency or throughput. BSP model also ignores that messages may overlap local computation. A cleverly written program attempts to hide communication latency by performing other computation concurrently with the communication.

Assuming complete concurrency between computation and communication, we can account for the overlap by replacing $L_s + th_s$ with $\max(L_s, th_s)$. This would not impact asymptotic analysis as the big-O complexity remains the same. It is desirable for a computational model to abstract away many complexities – particularly ones that vary from system to system. The role of the model it to help with a gross analysis of the parallel algorithm. This algorithm may then be suitably adapted to the actual hardware architecture, at which point some of the abstracted details can be reconsidered.

BSP Example

Let us consider an illustrative example of performance analysis using the BSP model. Take the problem of computing the dot product of two vectors.

Assume that the n elements of vectors A and B are initially equally divided among p processors. The vector segments are in arrays referred to locally as lA and lB in all processors. The number of elements in each local array = $\frac{n}{p}$. Assume n is divisible by p and consider the following code:

Input: Array A and B with n integers each.

Output:

$$A \cdot B = \sum_{i=0}^{n-1} A[i] \times B[i]$$

Solution:

```
forall[5] processor i < p {// in parallel
  { // Super-step local computation:
    int lc = 0; // Local at each processor
    int lC[p]; // Only needed at processor 0. Used for Receipt.
```

[5] forall means that all indicated processors perform the loop in parallel. The range of forall index variable (i here), along with an optional condition indicates how many processors are used. The use of the index variable i in the enclosed body indicates what each processor does. We sometimes omit the keyword processor to emphasize the data-parallelism.

```
      for(int idx=0; idx<n/p; idx++)
        lc += lA[idx] * lB[idx];
      send lc to processor 0
    } { // Super-step synchronization:
      barrier; // The barrier is always there for BSP, listed or not.
      Receive any item from processor i into lC[i] // Only processor 0 has any
    }
  }
  //-- All receipts have now been completed into lC --
  forall processor i == 0 { // Indicates p == 0 now.
    { // Super-step local computation
      for(int idx=1; idx<p; idx++)
        lc += lC[idx];
      output lc;
    } {// The barrier is implicit.
    }
  }
```

We can now analyze the time complexity of this algorithm. The first super-step requires $k_1\frac{n}{p}$ local time, $k_2 p$ communication time, and $k_3 p$ synchronization time, assuming the network throughput to be a constant independent of p and the barrier to be a linear function of p. k_1, k_2, k_3 are constants. The second super-step takes time $k_4 p$. Thus the total time is $\Theta(\frac{n}{p} + p)$.

It is possible to make a different choice for the second super-step, whose goal is to add the p numbers at p processors. Consider the following alternative. Assume for simplicity that p is a power of 2.

```
forall processor i < p {// Assume p is a power of 2
  { // Super-step local computation
    int lc2; // Designate for receipt of 1 item
    int lc = 0;
    for(int idx=0; idx<n/p; idx++)
      lc += lA[idx] * lB[idx];
    if(i >= p/2)
      send lc to processor i - p/2 // 2nd half sends to 1st half
    p = p/2; // Halve the processor count for the next super-step
  }{ // Super-step synchronization
    barrier;
    receive any item into lc2
  }
}
while(p > 0) { // Super-step loop
  // Data sent in the previous step have now been received into lc2
  forall processor i < p { // Only those that remain active
    { // Super-step local computation
      lc += lc2; // Accumulate the received value
      if(i >= p/2)
        send lc to processor i - p/2 // 2nd half sends to 1st half
      p = p/2; // Halve the processors active at the next super-step
    }{ // Super-step synchronization
      barrier;
      receive any item into lc2 // to accumulate further in the next iteration
    }
  }
}
// -- Last super-step --
forall processor i == 0 {
   output lc; // Implicit barrier is after this local step
}
```

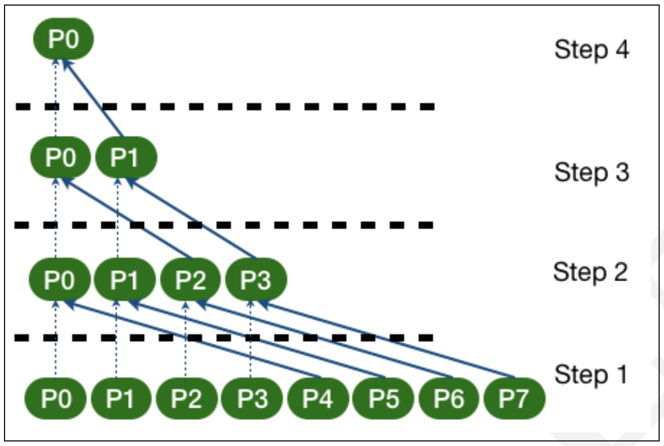

Figure 3.2: Binary tree like Computation tree

Now, there are more super-steps. The super-step loop has $\log p$ iterations. The structure of the computation is that of a binary tree, as shown in Figure 3.2.This process, where values in a vector are combined to produce a single scalar value, is called reduction[6]. In this variant of reduction, the number of processors employed in each super-step halves from that at the previous step, until it goes down to 1 in the final step. In this example, each active processor sends a single message in each iteration. Thus the total time is again $\Theta(\frac{n}{p} + p + \log p) = \Theta(\frac{n}{p} + p)$:

[6] ***Defined :*** Values in a vector are combined to produce a single scalar value. This is called reduction.

- The first super-step takes $k_1 \frac{n}{p}$ local time, $k_2 \frac{p}{2}$ communication time and $k_3 p$ synchronization time.
- The iterative super-step s takes $\Theta(1)$ local time and $\Theta(2^{(\log p - s)})$ communication and synchronization time. This sums to $\Theta(\log p + p)$ over the $\log p$ super-steps.
- The final super-step takes $\Theta(1)$ total time.

3.3 *PRAM Model*

The parallel RAM (*i.e.*, *PRAM*[7]) model mirrors the shared-memory

[7] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 114—118, New York, NY, USA, 1978. Association for Computing Machinery. ISBN 9781450374378

programming model. Like the BSP model, the PRAM model also assumes an arbitrary number of processors, p, each with an arbitrary number of constant-sized local memory locations. Further:

1. An arbitrary number of shared memory locations are accessible to all processors.

2. All processors proceed in complete synchrony: all complete step s before any starts step $s + 1$. Each step takes constant time. Thus, there is a barrier after each local step. While unrealistic in comparison to BSP, this leads to simpler analysis.

3. Each PRAM step is further divided into following three synchronous sub-steps, each taking a constant time:

   i) Each active processor i reads a constant sized value from any shared memory location r_i of its choosing.

   ii) Each active processor i performs a basic arithmetic or logical operation, or a local memory operation.

   iii) Each active processor i writes a constant sized value to any shared location w_i of its choosing.

The processors that are active at any step depends on the algorithm. Not all active processors are required to perform each sub-step. Some processors may remain idle in some sub-step.

The imposition of lock-step progress eliminates the need for explicit synchronization by the program, but it may yet result in conflicting writes by two processors to the same memory location in the same step. One solution is simply to disallow such shared reads and writes. This variant of the model is called *EREW* PRAM model: $r_i \neq r_j$ in any given step and similarly $w_i \neq w_j$, for $i \neq j$. Algorithms in this model must respect this restriction. Thus, each reader has exclusive access to its read location and each writer has exclusive access to its write location. Conflict is hence ruled out by the definition of the model. This restriction on the model (and hence the algorithms that assume this model) actually does not limit its generality. Algorithms designed for models that do not have these restrictions can be automatically translated into algorithms that do respect these restrictions. Only, the number of steps required by the resulting algorithm may be higher.

A more general variant is *CREW* PRAM, which allows two processors to read values from the same location in the same step. Writes remain exclusive. *CRCW* PRAM models, which allow conflicting writes as well, are also meaningful if the result of such conflicts are well defined. Several CRCW models have been proposed.[8,9] These

[8] Luděk Kučera. Parallel computation and conflicts in memory access. *Information Processing Letters*, 14(2):93 – 96, 1982. ISSN 0020-0190

[9] Yossi Shiloach and Uzi Vishkin. An o(logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57 – 67, 1982. ISSN 0196-6774

allow w_i to equal w_j for any number of different i, j pairs, but with certain restrictions. Some of these are:

- Common-CRCW: If $w_i = w_j$, both processors i and j must write the same value. So, there is no data conflict.
- Arbitrary-CRCW: If $w_i = w_j$, either of the conflicting values may be written. The other is discarded. If more than two processors conflicts, any one write may succeed. The algorithm's correctness must not depend on which value is actually written.
- Priority-CRCW: If $w_i = w_j$, the smaller of i and j succeeds. If more than two processors conflict, the smallest indexed processor among all conflicting processors has the priority and its value is written.

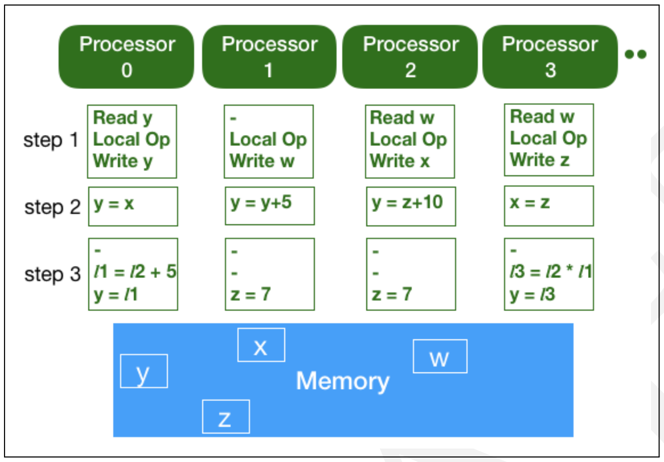

Figure 3.3: PRAM computation model

Figure 3.3 demonstrates the PRAM model. Step 1 shows that processors 2 and 3 read from the same location w. This would not be possible in an EREW PRAM. All the writes in step 1 are to different locations – they do not conflict. Hence this step would be allowed by a CREW PRAM. Note that processor 2 writing to location w and other processors reading from w in the read sub-step of the same step is not considered common or conflicting. The read fetches the older value.

Step 2 shows a succinct way to write instructions. Each processor reads from a shared memory location, optionally adds two value, and then writes to a shared memory location. Notice that processors 0–2 all write to location y. This is not possible in CREW or EREW PRAM. It is possible only in CRCW PRAM. Again note that the reading of x by processor 0 happens strictly before its update by processor 3 in the write sub-step of this step.

The third step shows that processors 1 and 2 have a common write to location z. Since the two values are the same, all three CRCW variants support this. Processors 0 and 3 must also have the same values in their respective local variables $l1$ and $l3$ for this program to be supported by Common CRCW. They may not. Both Priority-CRCW and Arbitrary-CRCW would support this program. In Priority-CRCW PRAM, the value in variable $l1$ of processor 0 is expected to be written by this program. In Arbitrary-CRCW PRAM, this program must produce the correct result irrespective of the value ($l1$ or $l3$) written into y at the end of this step.

All the listed PRAM variants are equally general, and an algorithm designed in any model can be translated into any other.[10,11] The difference is in their execution times and the simplicity of designing algorithms. Priority-CRCW is the most useful since any algorithm of other models can be executed in this model *as is* without any translation. We could choose this model for our design. However, in practice, this model is the furthest from practical hardware, and hides more cost than the others. Detecting and prioritizing conflicts of an arbitrary number of processors in constant time is not feasible. Comparatively, Common-CRCW and Arbitrary-CRCW are safer models to design algorithms with, being more representative of the hardware. However, the cost of supporting conflicting reads and writes can be non-trivial in a distributed-memory setting, where the EREW model may be more effective.

[10] Bogdan S. Chlebus, Krzysztof Diks, Torben Hagerup, and Tomasz Radzik. New simulations between crcw prams. In J. Csirik, J. Demetrovics, and F. Gécseg, editors, *Fundamentals of Computation Theory*, pages 95–104, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. ISBN 978-3-540-48180-5

[11] Joseph Jájá. *Introduction to Parallel Algorithms*. Pearson, 1992

Regardless, all models assume perfect synchrony, which is hard to achieve in hardware in constant time for a large number of processors. This means that that communication and synchronization costs are not accounted for in PRAM analysis.

PRAM Computation Time

Each step of PRAM takes a constant time-unit. The total time taken is then proportional to the number of PRAM steps.

There is a local step in PRAM, quite like BSP does. The communication step maps to reads and writes. Processors 'send' in the PRAM model by writing to a shared location. 'Recipients' read from there. In a sense, the (read, local step, write) triplet is analogous to BSP super-step, except each of the three sub-steps is synchronous in PRAM, whereas only the full super-step is synchronous in BSP. Also, the cost of synchronization is hidden in PRAM, while BSP accounts for synchronization and also allows arbitrary but local super-steps.

Thus, accounting is simpler in PRAM: assume each step takes unit time. One can equivalently say that each of the three sub-steps takes unit time, and the step takes '3' units. Asymptotically, they lead to the same analysis. Note that each processor is allowed local memory in PRAM, just like BSP. It may be tempting to allow the middle sub-step to include an arbitrary number of sub-steps, as BSP does. On the other hand, that is equivalent to having those local sub-steps be simply associated with 'NULL' read and write steps (*i. e.*, all processors remain inactive in those sub-steps). The difference effectively is that BSP accounts for the cost of shared reads and writes, and PRAM does not.

Let us analyze the same dot product example, now in the PRAM model. This time A and B are in shared memory accessible to all processors.

PRAM Example

Input: Array A and B with n integers each in shared memory.

Output:

$$A \cdot B = \sum_{i=0}^{n-1} A[i] \times B[i]$$

Solution:

```
int C[p]; // C is a shared int array of size p
forall processor i < p {
   C[i] = 0;
   for(int idx=0; idx<n/p; idx++)
      C[i] += A[i*n/p+idx] * B[i*n/p+idx];
}
forall processor i == 0 {
   for(int idx=1; idx<p; idx++)
      C[0] += C[idx];
   output C[0];
}
```

At each iteration of the first loop, processor i reads from A, B and C in three consecutive steps. The local computation of the product and sum as well as the write-back of $C[i]$ also takes place in the third step. Thus the processors all take $\Theta(\frac{n}{p})$ steps in the first loop. The second loop employs only a single processor, which takes $\Theta(p)$ time. Thus the total time complexity of this PRAM algorithm is $\Theta(\frac{n}{p} + p)$. This matches the complexity of the equivalent algorithm in the BSP model.

We can also do a tree-like reduction in the PRAM model, as we did in the BSP model, as follows:

```
int C[p]; // C is a shared int array of size p
forall processor i < p {
   C[i] = 0;
   for(int idx=0; idx<n/p; idx++) // Assume n is divisible by p
      C[i] += A[i*n/p+idx] * B[i*n/p+idx];
}
```

```
p = p/2; // Halve the number of processors used
while(p > 0) {
   forall processor i < p {
      C[i] += C[i+p/2];
      p = p/2;
   }
}

forall processor i == 0
   output C[0];
```

The first loop is unchanged from the previous version and takes time $\Theta(\frac{n}{p})$. The second loop takes $\Theta(1)$ time per iteration and $\log p$ iterations, taking total time $\Theta(\log p)$. The last step takes $\Theta(1)$ time by processor 0. Notice that the total time based on this analysis, *i. e.*, $\Theta(\frac{n}{p} + \log p)$, is different from the time taken by the analogous algorithm in the BSP model. This is because the extra messages passed in the reduction variant are exposed and counted in the BSP model. This count remains hidden in the PRAM model because more processors are able to perform more shared-memory accesses in parallel in the same time-step. In this aspect, PRAM is like the simple parallel model. In the case of shared-memory hardware, this unit time-step for shared-memory read is a reasonable assumption. Note that we sometimes allow p to be a suitable function of n for unified analysis. For example, if $p = \Theta(n)$ in the example above, the time complexity is $\Theta(\log n)$.

For distributed-memory setting, PRAM is simpler, but BSP may be better suited. Particularly so for algorithms that are communication-heavy. Other more elaborate computational models exist, but they also increase the complexity of algorithm analysis without necessarily providing significantly more realistic prediction of hardware performance. We discuss practical performance metrics next, which focus on measured running times of programs.

3.4 *Parallel Performance Evaluation*

Question: What are the different aspects of measuring a parallel program's performance?

If we describe a parallel program using the simple parallel model or one of the other models, we can compute the time it takes in the context of that model. We may next translate such a description to actual program implementation and measure the time it takes on real hardware. Either way, we can compare the speeds of two programs, given an input size n and a processor count p. We can also chart the speed of one program with increasing processor counts. How are these varying speeds to be evaluated? This behavior or performance with increasing processor count is a critical ingredient of parallel programming and is called *scalability*. Scalability is important because it predicts performance on large input and on large systems (that may not be immediately available).

The following definitions are useful to study various aspects of parallel performance evaluation. These may be measured by executing implemented programs. They can be equally well defined in terms of algorithms and computational models we have just studied. In the context of programs, we may use measured wall-clock times, and for algorithms, we talk of the number of notional steps as described above.

Latency and Throughput

The time taken to complete one program, call it job execution, since the time it began is also called the elapsed time or job *latency*. Often, many jobs are executed on a parallel system. They may be processed one at a time from a queue, or several could execute concurrently on a large parallel system. These could be unrelated programs, related programs, or different executions of the same program. In all cases, the number of jobs retired per unit time is known as the job *throughput*. Job throughput is related to average job latency. If jobs take less time on average, more jobs are processed per unit time. However, the latency of different jobs may vary wildly from job to job, without impacting the throughput. The worst-case latency, *i.e.*, the longest latency of any job, is an important metric.

Speed-up

The *speed-up* $\mathcal{S}$ of a program $\mathcal{P}$ taking time $t(n,p)$ with respect to another program $\mathcal{P}_1$ taking time $t_1(n_1,p_1)$ is the ratio of their speeds, which is the inverse of their execution times :

$$\mathcal{S} = \frac{t_1(n_1,p_1)}{t(n,p)} \tag{3.1}$$

Like before, n is the size of the input and p is the number of processors deployed by an algorithm. So are n_1 and p_1, respectively. Although not explicit in the notation, $\mathcal{S}$ is clearly a function of $\mathcal{P}$, $\mathcal{P}_1$, n, n_1, p, and p_1. We will keep this notation for brevity; it should be clear from the context. We often consider *parallel speed-up*, the special case of the speed-up with respect to the sequential execution of a parallel program, *i.e.*, $p_1 = 1$ and $n_1 = n$:

$$\mathcal{S}_{par} = \frac{t(n,1)}{t(n,p)} \tag{3.2}$$

Similarly, *maximum speed-up* may be defined as the maximum speed with respect to the 'best known' sequential program (let us say that is $\mathcal{P}_1$).

$$\mathcal{S}_{max} = \frac{t_1(n,1)}{t(n,p)} \tag{3.3}$$

$\mathcal{S}_{par} \geq 1$ and $\mathcal{S}_{max} \geq 1$ in principle, because a parallel program may simply choose to inactivate $p-1$ processors and degenerate to a sequential version. Thus, a parallel program should always be able to beat the sequential version. In fact, the speed-up of parallel program $\mathcal{P}$ using p processors with respect to it using p_1 processors, $p_1 < p$ for the same input size should be greater than 1. (In reality, however, early learners often find this hard to achieve at first. It does get better in due course.)

Cost

Speed-up can increase with increasing p. On the other hand, deploying more processors is costly. We define the cost $\mathcal{C}$ of a parallel program as the product of its time and the processor count:

$$\mathcal{C} = t(n,p) \times p \tag{3.4}$$

A parallel program is *cost-optimal* if $\mathcal{C} = t_1(n,1)$, the cost of the best sequential program. Cost-optimality means the speed-up gained by deploying a large p is commensurate with their increased cost. For example, doubling the number of available processors doubles the speed, *i.e.*, halves the execution time.

Often, we do not know $t_1(n,1)$ precisely, but only in an asumptotic sense. In such situation a definition of asymptotic optimality is useful. A parallel program (or algorithm) is *asymptotically cost-optimal* if $\mathcal{C} = O(t_1(n,1)$.

Efficiency

Another way to express the 'quality' of speed-up is efficiency. Expected speed-up over a sequential program is higher for a higher value of p. The quality of this speed-up, or the speed-up efficiency $\mathcal{E}$, is the maximum speed-up per deployed processor:

$$\mathcal{E} = \frac{\mathcal{S}_{max}}{p} \tag{3.5}$$

$\mathcal{E} \leq 1$, because any speed-up larger than p implies the discovery of a better sequential algorithm than the best known sequential algorithm (making the newly discovered algorithm the new best). After all, any flexible parallel algorithm can be executed sequentially by setting $p = 1$. $\mathcal{E} = 1$ implies the program is cost-optimal, and the speed-up is proportional to the number of processors used.

In practice, it is quite possible to observe values of efficiency greater than 1. This occurs because the underlying system on which the executions of the sequential program and the parallel program are measured are necessarily different. For example, with larger p may come larger caches, improving data access times. Recall that data access latency is significantly higher than arithmetic operation latency. Hence, the performance of a program with many memory operations can depend heavily on this latency. Consequently, even small improvements in memory access latency can improve the program's performance. There can also be other scenarios, *e.g.*, a parallel "multi-pronged" search may serendipitously converge to a solution quicker. The tools we develop next are designed in a more idealized setting and ignore these real effects. Regardless, they are meaningful and may generally be used even in the presence of these effects.

Scalability

Scalability is related to efficiency and measures the ability to increase the speed-up linearly with p. In particular, if the efficiency of program $\mathcal{P}$ remains 1 with increasing processor count p, we say it scales perfectly with the size of the computing system. Most problems cannot be solved this efficiently, and those that can are often said to be embarrassingly parallel. Indeed, the program may begin to slow-down for larger values of p, as shown in Figure 3.4 for $p = 17$ and $n = 10^4$. This can happen due to several reasons. For example, communication may increase, or more processors remain idle. Of course, the efficiency may also depend on the size of the input, n. For example, an $\Theta(n)$ sequential program, on parallelization, might not get faster for $p > n$. It is often the case that performance scales better for larger values of n. For example, Figure 3.4 shows higher speed-up for $n = 10^6$. In some cases, however, the speed-up may even reduce for larger n, *e.g.*, because caches become less effective.

When efficiency remains high with increasing p, regardless of n, we say the program exhibits *strong scaling*. On the other hand, if efficiency for higher values of p remains high only if n is also increased, we call it *weak scaling*. If efficiency is low regardless, we say the program does not scale. But how high is high? For the efficiency to remain 1 is unrealistic, and such definition would hardly be useful. One might instead say, if the speed-up for a higher value of p is lower than that for a lower value of p, the efficiency is low, and scaling is poor. This seems too low a bar. A slightly tighter definition says

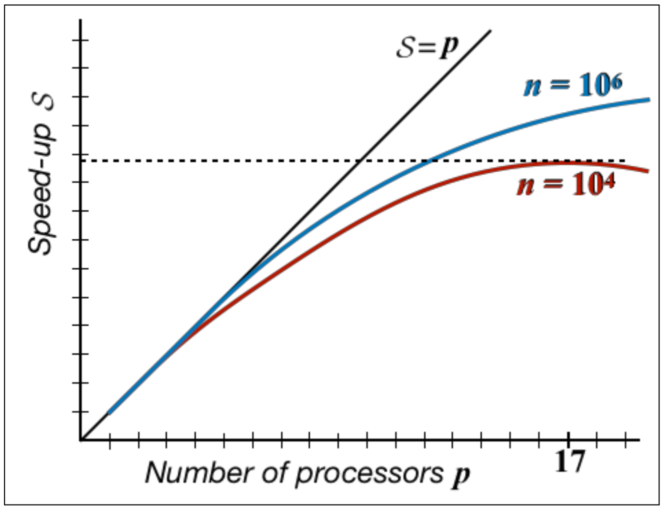

Figure 3.4: Efficiency curve: speed-up vs. processor count

that the efficiency $\mathcal{E}$ does not reduce with increasing p – it remains constant. This means the efficiency curve remains linear, even if its slope may be somewhat less than 1. We refine this quantitative measure of scalability next.

Iso-efficiency

The *Iso-efficiency* of a scalable program indicates how (and if) the problem size must grow to maintain efficiency on increasingly larger computing systems. Iso-efficiency is, in reality, a restating of the sequential execution time as a function of p, the processor count. Recall from Eq 3.3 and 3.5:

$$t_1(n, 1) = \mathcal{E}(n, p)\, t(n, p)\, p \tag{3.6}$$

$t_1(n, 1)$ is a measure of the problem's size and complexity. Given p and the time-function for a parallel program $t(n, p)$, we want to derive that t_1, which would ensure a constant efficiency $\mathcal{E}$. t_1 changes because n changes. Thus, deriving t_1 really amounts to finding the appropriate problem size n that takes time t_1. To emphasize that we seek to find the problem size for a given p, we use the notation $\mathcal{I}(p)$ for problem size in place of t_1. $\mathcal{I}(p)$ is called the iso-efficiency function. The parameterization with p signifies that we adapt the problem size to p. A rapid growth in $\mathcal{I}$ with increasing p means that only much larger problems can be efficiently solved on larger machines. This is poor scalability.

We can relate $\mathcal{I}$ to the overhead of parallelization $\bar{o}(n, p)$: the computation that is not required in the sequential solution. In other words, $\bar{o}(n, p)$ is the 'extra' time collectively spent by the parallel processors compared to the best sequential program. This may include idle processors, communication time, etc. Hence,

$$\bar{o}(n,p) = t(n,p)\,p - t_1(n,1) \tag{3.7}$$

and

$$\mathcal{I}(p) = t_1(n,1) = t(n,p)\,p - \bar{o}(n,p) \tag{3.8}$$

Substituting $\mathcal{E}$ from Equation 3.5 and 3.3.

$$\Rightarrow \mathcal{I}(p) = \left[\frac{\mathcal{E}(n,p)}{1-\mathcal{E}(n,p)}\right]\bar{o}(n,p) \tag{3.9}$$

This means that if $\mathcal{I}$ increases proportionally to the overhead $\bar{o}$, the term within [] above – call it K – remains constant, *i.e.*, the efficiency remains constant. In other words, if the overhead grows rapidly with increasing p, the problem size also must grow as rapidly to maintain the same efficiency. That indicates poor iso-efficiency.

For illustration, consider the BSP example of parallel reduction in Section 3.2: $t(n,p) = \Theta(\frac{n}{p} + p)$. We know the optimal sequential algorithm is linear in n: $t_1(n,1) = \Theta(n)$. This means:

$$\bar{o}(n,p) = \Omega(p^2)$$

$$\Rightarrow \mathcal{I}(p) = K\Omega(p^2)$$

This means that the problem size must grow at least quadratically with increasing p to maintain constant efficiency. Check that in the PRAM model, $\mathcal{I}$ is bounded sub-quadratically (see Exercise 3.11) in p.

Note that by Equations 3.6 and 3.7, for embarrassingly parallel problems, $\bar{o}$ remains 0, and $\mathcal{E}$ remains 1 because $t_1(n,1) = t(n,p)\,p$. The problem size apparently does not need to grow to keep $\mathcal{E}$ constant. However, there is a limit. If $p > t_1(n,1)$, there is not enough work to go around. Hence, the problem size must eventually grow at least as fast as p, *i.e.*, asymptotically $\mathcal{I}(p) = \Omega(p)$. Practically speaking also, the overhead usually grows at least in proportion to p, and often faster. In other words, we expect that the input size n needs to grow at least as fast as the processor count p to maintain efficiency. Similarly, if $\bar{o}(n,p) = O(t_1(n,1))$, Equation 3.7 indicates that $t(n,p)\,p = O(t_1(n,1))$ meaning that the solution is asymptotically cost-opimal.

Note that p is bounded in practice. Surely, there is not an unlimited supply of processors. Nonetheless, scalability with increasing p is a useful measure. Of course, it indicates the possibility of speed-up with increasing system size. It is also often the case that better scaling programs – and better scaling algorithms – tend to perform better on a wider variety of systems and system architecture.

3.5 Parallel Work

The final metric we will study is called *parallel work*. This is the total sum of work done by processors actually employed at different steps of an algorithm. Recall, the cost is the time taken by an algorithm multiplied by the maximum number of processors available for use at any step. Work is a more thorough accounting of the processors actually used. In other words, parallel work required for input of size n,

$$W(n) = \sum_{s=1}^{t(n,p)} p_s(n), \tag{3.10}$$

where $p_s(n)$ processors are active at step s. Note that we allow the number of active processors to be a function of input size n. Each processor takes unit time per step, and the algorithm takes $t(n, p)$ steps. Note also that in $t(n, p)$, p varies at each step. We leave this intricacy out of the notation for p. The value of p at each step is specified for algorithms, however.

As an example, the initial number of processors assumed in the binary tree reduction algorithm is $\frac{n}{2}$. The algorithm requires $\log n$ steps, but the number of active processors halves at each step. For instance, in the first step of the PRAM algorithm $\frac{n}{2}$ processors each perform unit work (a single addition in this example). $\frac{n}{4}$ processors are used in the second step and so on. Thus the total work, W(n) is:

$$\sum_{s=0}^{\log n-1} 2^s = n$$

The total parallel work performed in the reduction algorithm is $\Theta(n)$, but the cost is $\Theta(n \log n)$. One may question the logic of using work as a performance metric. If n processors were available and not used in step two, that seems like a wasted opportunity. Maybe, it is not so because the unused processors are available to a different job. However, there is a more fundamental reason this work complexity is important.

It allows us to design highly scalable algorithms that allow an arbitrarily large value for p_s, sometimes even equal to or greater than n. An implementation would, of course, have a limited number P_r of real processors available. We then map each step of the algorithm to P_r processors simply by each real processor performing the work of $\frac{p_s}{P_r}$ assumed processors in a loop. What can we say about the expected time taken by such an execution then? This is given by Brent's work-time scheduling principle.

Brent's Work-time Scheduling Principle

Let us assume the PRAM model to take a specific example, but other models are equally compliant. Step s of the original algorithm takes $\Theta(1)$ using p_s processors. In its execution, step s is scheduled on P_r processors taking $\left\lceil \frac{p_s}{P_r} \right\rceil$ steps. The total number of steps are:

$$\sum_{s=1}^{t(n,p)} \left\lceil \frac{p_s}{P_r} \right\rceil \leq \sum_{s=1}^{t(n,p)} (\frac{p_s}{P_r} + 1) = \sum_{s=1}^{t(n,p)} \frac{p_s}{P_r} + \sum_{s=1}^{t(n,p)} 1 = \frac{W(n)}{P_r} + t(n,p) \tag{3.11}$$

The work and time both impact the actual performance. For many algorithms $t(n,p) = O(W(n))$, and hence the work is the main determinant of the execution time. Another useful way to think about this is that with P_r processors, the algorithm takes time $O(\frac{W(n)}{P_r})$, for $P_r \leq W(n)/t(n)$.

We can also now define the notion of work optimality. A parallel algorithm is called *work-optimal*, if $W(n) = O(t_1(n,1))$. Further, a work-optimal algorithm for which $t(n,p)$ is a lower bound on the running time and cannot be further reduced is called *work-time optimal*.

3.6 *Amdahl's Law*

Question: Is this the best performance achievable?

There are certain limits to the speed-up and scalability of algorithms. Sometimes the problem itself is limited by its definition. Such limits may exist, e.g., because there may be dependencies that reduce or preclude concurrency. Recall that concurrency is a prerequisite for parallelism. Reconsider, for example, the problem of moving vans. Boxes can be transported in vans, but before they can be moved, they must be loaded, say, at the warehouse. There is no way to perform the task of loading a van at the warehouse and driving it to its destination in parallel with each other. Driving must happen after loading sequentially and is dependent on it.

Sometimes, the dependency is imposed by the algorithm. For example, in hopes of better packing, the loaders may load large boxes first and small ones later. Possibly, the large boxes may be loaded in parallel by multiple loaders. However, the small boxes' loading may only begin after a certain minimum number of large boxes are loaded.

Here is a more 'computational' example, called the prefix sum problem.

Input: Array A with n integers

Output: Array B with n integers such that

$$B[i] = \sum_{j=0}^{i} A[i]$$

Solution:

```
B[0] = A[0];
for(int i=1; i<n; i++)
   B[i] = A[i] + B[i-1];
```

This solution has each iteration `i` depend on the value of B[i-1] computed in the previous iteration. Thus, different entries of B cannot be filled in parallel; rather, the entire loop is sequential. We will later see that this is a shortcoming of the chosen algorithm and not a limitation of the problem itself. There do exist parallel solutions to this problem.

Amdahl's law[12] is an idealization of such sequential constraints. Suppose fraction f of a program is sequential. That may be because of inherent limits to parallelization or because that fraction was simply not parallelized. The fraction is in terms of the problem size (*i. e.*, the fraction of time taken by the sequential program). This implies that fraction f would take time at least $t_1(n,1)f$. Assuming that the rest is perfectly parallelizable, it can be speeded up by factor up to p. This means that time $t(n,p)$ taken by a parallel program can be no lower than $t_1(n,1)f + \frac{t_1(n,1)}{p}(1-f)$. This implies a maximum speed-up of:

[12] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery. ISBN 9781450378956

$$\mathcal{S}_{max} = \frac{t_1(n,1)}{t_1(n,1)f + \frac{t_1(n,1)}{p}(1-f)} = \frac{1}{f + \frac{1-f}{p}} \tag{3.12}$$

No matter how many processors we apply (say, $p \to \infty$), a speed-up greater than $\frac{1}{f}$ could never be achieved. Even that is possible only if the solution scales strongly with an efficiency of 1 for an unlimited number of processors. This equation may seem hardly surprising, but looking at the actual value of such limits can be eye-opening.

The graph in Figure 3.5 plots the maximum speed-up that is theoretically possible for a varying number of processors. The different plots are for different values of f. Notice how much limit even small values of f can place. If the sequential fraction is only 10%, the parallel speed-up could never be more than 10. It would seem that there is little benefit of using, say, more than a hundred processors, which yield a speed-up greater than 9. This is rarely true in practice. First, the formula assumes an efficiency of 1. If the efficiency is less, even the speed-up of 9 likely requires many more than a hundred processors. Second, for weakly scaling solutions, larger problems could be

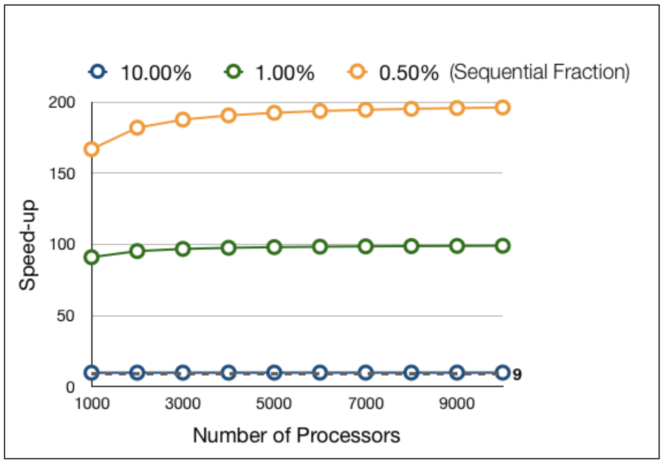

Figure 3.5: Maximum speed-up possible with different processor counts (in idealized setting)

solved efficiently on larger machines, even if the small problem does not scale beyond a hundred processors. Gustafson's law accounts for precisely that.

3.7 *Gustafson's Law*

Gustafson[13] redirects Amdahl's equation to bring the size of the problem into the mix. Suppose that the time spent in sequential components by a parallel program is $t(n,p)f$, and the time spent in the parallel part is $t(n,p)(1-f)$. The fraction is now in terms of the execution time of the parallel program (and can vary with p). The sequential fraction also includes all overheads, meaning the $(1-f)$ fraction of the time is spent in fully parallel computation keeping all processors busy.

Given that breakup, any sequential implementation must take $t(n,p)f + pt(n,p)(1-f)$ time. After all, the single processor must perform the work of each of the p processors, one at a time. This means that the speed-up of the parallel implementation over the sequential one is:

$$\frac{t(n,p)f + t(n,p)(1-f)p}{t(n,p)} = f + p(1-f) \qquad (3.13)$$

Note that the fractions f used by Amdahl and Gustafson are different In Amdahl's treatment, f represents the fraction of a sequential program that is not parallelized, and f does not vary with p, whether the problem size n grows or not. In Gustafson's treatment, f accounts for the overheads of parallel computation. This fraction relative to the parallel execution time remains constant even as n and p change. This effectively means that the time spent in the sequential part reduces in proportion to that spent in the parallel part. In Amdahl's

[13] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5): 532–533, May 1988. ISSN 0001-0782

treatment, the sequential time remains constant even as the parallel time reduces with more processors.

If Gustafson's fraction remains constant as p increases, the obtained speed-up $\mathcal{S}$ grows linearly with p, as Figure 3.6 shows. Remember that n grows along with p, but that is not highlighted in the graph.

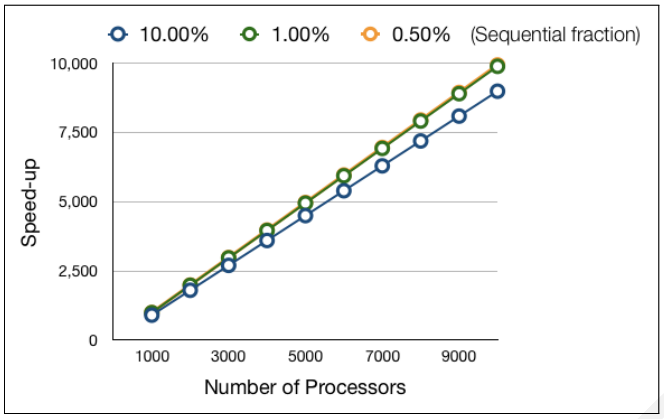

Figure 3.6: Maximum speed-up possible by scaling problem size with processor count (in idealized setting)

In practice, it is possible that Gustafson's f does not remain constant but grows more slowly than envisaged by Amdahl. This would lead to a sub-linear growth of speed-up with increasing p, but possibly not as slow as Amdahl envisages. In any case, neither law accounts for the higher overhead with more processors. This overhead has a major impact on real program execution times, and causes the efficiency to decrease with increasing p.

3.8 *Karp-Flatt Metric*

Karp-Flatt Metric[14] turns the discussion around and seeks to estimate the unparallelized part f in a program, given the measured speed-up over the sequential execution $\mathcal{S}$:

[14] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Commun. ACM*, 33(5):539–543, May 1990. ISSN 0001-0782

$$f = \frac{\frac{1}{\mathcal{S}} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{3.14}$$

It is not hard to verify that this metric is consistent with Amdahl's law. Just reorganize equation 3.12 to bring f to the left-hand side. According to this equation, if the speed-up obtained by a program using 100 processors is 10, the sequential part takes approximately 9.1% of the execution time. How this fraction varies with p can now be computed by running the experiment with different processor counts.

Again, it is possible that the actual sequential part is lower than the value of f so computed. This means that the observed speed-up is less than the maximum possible. That can happen due to the overheads of parallelization. In that sense, f may thus generically represent the overhead $\bar{o}$.

3.9 Summary

An understanding of performance issues is fundamental to the exercise of designing parallel algorithms and writing programs. Measuring actual execution time is useful, but one must design programs that perform well on all n and p, or at least many n and p. It is not practical to measure the performance on all instances. Rather, one must argue about the performance on n and p that are anticipated.

Hence, modeling and analyzing performance are pre-requisites for writing efficient parallel programs. This chapter discusses a few abstract models of computation, which can be used to express and analyze parallel algorithms. It also introduces practical metrics to evaluate parallel programs' design and performance in comparison to, say, sequential programs, and as it relates to the number of processors used. Theses lessons include:

- The PRAM model relies on an arbitrary number of synchronous processors. Each has local memory, and they together share global memory. Simple computation and memory operations take a single time-unit each. Since the processors proceed in lock-step and share memory, there is no synchronization or explicit communication. As a result, such overheads are ignored in the analysis.

- Variants of the PRAM model control the possibility of different processors read from or writing to the same memory location or address in a single time-step. Either common addresses are supported (*e.g.*, EREW, CREW), or the addresses must be exclusive (*e.g.*, CREW, CRCW). Such support is set separately for reading and writing operations.

- For CRCW PRAM, different semantics are possible. In Common-CRCW PRAM, if multiple processors write to a common address at the same time-step, they must all present the same value to write. Alternatively, in Arbitrary-CRCW PRAM, if multiple processors write to a common address, any of their values may be written. The algorithm's correctness must not depend on which value is written. In the Priority-CRCW PRAM, each processor is accorded a distinct priority. If multiple processors write to a common address, the value presented by the one with the highest priority is always written.

- All variants of the PRAM model are functionally equivalent, for each can simulate the behavior of others. However, such simulation may not take constant time. Priority-CRCW model, for example, can simulate the steps of every other model in constant time each. Other models cannot simulate Priority-CRCW steps in constant time each. In that sense, the Priority-CRCW is more powerful than others.

- The BSP model maintains the synchronizing characteristic of PRAM, but it does not require complete lock-step progress of processors. Instead, processors may take an arbitrary number of local steps before synchronizing. Further, data is exchanged by the processors explicitly – there is no shared memory. BSP counts the number of messages communicated. The lack of per-step synchrony does not make algorithms much more complicated than in the PRAM model, but the communication overhead is counted. BSP does not consider the size or batching of messages.

- Work is an important metric to measure parallel performance. We start by exposing the entire parallelism inherent in an algorithm by assuming as many processors as the number of independent steps. Recall that two steps are independent if there is no order required between them, and they can be taken simultaneously. At different time-steps, different numbers of independent steps may be possible. This means that the number of processors used at each time-step varies. The total of all processor-steps in this manner is called the work. Work complexity on its own is not sufficient to indicate the level of parallelism. After all, a sequential algorithm has a low work complexity. Our goal is to keep work complexity similar to that of the sequential solution while minimizing the time complexity.

- Brent's scheduling principle shows how work translates to the real execution time on a specific machine with p processors. If the number of sequential time-steps is t and the number of processor-steps (*i.e.*, work) are w, a p-processor machine takes time $\frac{w}{p} + t$.

- Speed-up measures the ratio of the speed of one algorithm or implementation with another. When comparing algorithms in a PRAM or BSP setting, asymptotic speed-up is usually of concern. With measured execution times of implementations, the actual speed-up value on specific computing systems becomes possible.

- Although absolute speed-up on specific computing systems is the primary statistic for the user of an application, the efficiency with which it is obtained is a more meaningful metric for the program designer and developer: the speed-up per processor used to obtain it. The same speed-up obtained on a smaller machine points to a more efficient than when more processors are required.

- The cost of an execution is related to its efficiency. Cost is the product of the time taken by a program and the number of processors used. The cost does not require a comparison to the speed of another program. Low-cost implementations take low time or use very few processors. In other words, efficient programs are likely to be cost-effective because the speed per processor is high.

- The speed of a program or algorithm relative to the number of processors used is important. However, some programs are efficient only if a small number of processors are used. As the number of processors grows, so do the overheads of synchronizing them, exchanging data, or simply waiting for certain action by other processors. This overhead can be detrimental to both efficiency and cost. More the number of processors, more such overhead. In fact, the overhead from using too many processors can outweigh the entire benefit of the extra execution engines. Scalable programs limit such overheads. As a result, they continue to get faster with more processors. Some even continue to maintain the speed-up per processor, i.e., they continue to remain efficient, for large values of p.

- A strongly scaling program gets faster if more processors are available. A weakly scaling program roughly maintains speed with more processors if the problem size grows as well. The same program may scale strongly for smaller p, scale weakly for medium p, and stop scaling altogether for larger p.

- The notion of iso-efficiency formalizes scalability. The Iso-efficiency of an algorithm or program measures the growth required in the problem size as a function of the number of processors to maintain constant efficiency. Iso-efficiency combines the impact of n and p on scalability, and a slow growing iso-efficiency function works well for larger problem cases than a fast growing one.

- There are limits to scaling in most situations. Amdahl's law states one fundamental limit: the limit to the parallel speed-up of problems (or their solution) if they contain strictly sequential components. Such sequential components must be processed on a single processor, while all other processors wait for it to finish. Amdahl's law assumes that the problem of a certain size is solved using increasingly more processors. In this case, the sequential components remain a fixed fraction of the entire problem and do not get faster with more processors. On the other hand, the parallel components do get faster. Consequently, the sequential components start to dominate the total execution time, limiting total speed-up.

- Gustafson's law instead considers the case when the sequential components are a fixed fraction of the parallel execution time. Thus, as more processors are employed to solve larger problems, the sequential components' execution time keeps pace with the parallel components'. Linear scaling of speed-up is possible in this scenario.
- Instead of debating the components' sizes, the Karp-Flatt metric estimates them. Rather, it estimates the entire parallelization overhead by observing the speed-up with an increasing number of processors. Growth of this overhead with an increasing number of processors while keeping the problem size constant indicates that the overhead is significant. This suggest that attempts to reduce overhead may be useful.

The abstract computation models that this chapter focuses on are the BSP model and the PRAM model. Historically, the PRAM model was proposed first by Fortune and Wylie[15]. Valiant later proposed the BSP model as a 'bridge' between the abstract model and practical architecture. These two are popular, but others that account for more overheads and parameters have also been proposed. For example, block-transfer and communication latency have been considered.[16,17]. Mehlhorn and Vishkin propose an extension: the *module parallel computer*[18] (MPC). In MPC, the shared memory is divided into modules (*i.e.*, banks) and only one word may be accessed from each module in one time-step. Limitations of perfect synchrony have also been addressed.[19,20]

The BSP model also addresses both the synchrony and communication shortcomings of the PRAM model. The BSPRAM model[21] attempts to combine the PRAM and BSP models. Others like the LogP model (Culler et al., 1993) account the message cost more realistically by considering detailed parameters like the communication bandwidth and overhead and message delay. Barrier is still supported but not required. Others have also focussed on removing the synchronous barrier by supporting higher-level communication primitives *e.g.*, the *coarse grained multi-computer* model (Dehne et al., 1993).

Other than shared-memory and message-passing style architectures, purely task-graph based models have also been used (Ullman and Papadimitriou, 1984; Papadimitriou and Yannakakis, 1988) using parameters like task time, message complexity, and communication

[15] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 114—118, New York, NY, USA, 1978. Association for Computing Machinery. ISBN 9781450374378

[16] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Hierarchical memory with block transfer. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, SFCS '87, page 204–216, USA, 1987. IEEE Computer Society. ISBN 0818608072

[17] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Communication complexity of prams. *Theoretical Computer Science*, 71(1):3 – 28, 1990. ISSN 0304-3975

[18] Kurt Mehlhorn and Uzi Vishkin. Randomized and deterministic simulations of prams by parallel machines with restricted granularity of parallel memories. *Acta Inf.*, 21(4):339–374, November 1984. ISSN 0001-5903

[19] P. B. Gibbons. A more practical pram model. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '89, page 158–168, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 089791323X

[20] R. Cole and O. Zajicek. The expected advantage of asynchrony. *J. Comput. Syst. Sci.*, 51(2):286–300, October 1995. ISSN 0022-0000

[21] Alexandre Tiskin. The bulk-synchronous parallel random access machine. *Theoretical Computer Science*, 196(1):109 – 130, 1998. ISSN 0304-3975

delay. All these models can simulate each other and are equivalent in that sense. That may be the reason why the simplest models like PRAM and BSP have gained prevalence. However, the models do differ in their performance analysis. A case can be made that a more realistic model discourages algorithms from taking steps that are costly on real machines by making such cost explicit in the model. More importantly, though, it is the awareness of the differences between the model and the target hardware that drives good algorithm design.

Besides designing efficient algorithms suitable for specific hardware and software architecture, one must also select the number of the processors before execution begins. Large supercomputers may be available, but they are generally partitioned among many applications. It is important for applications not to oversubscribe to processors. As many processors should be used as provide the best speed-up and efficiency trade-off. Sometimes speed-up can reduce with large p. At other times speed-up increases, but the efficiency reduces rapidly beyond a certain value of p. In many applications, the size of the problem, n, can also be configured. Further, the memory reserved for an application, m, may also be configured. Optimally choosing $\mathcal{S}, \mathcal{E}, p, n$, and m is hard. A study of time and memory constrained scaling[22,23] is useful in this regard. In particular, Sun-Ni law[24] extends Amdahl's and Gustafson's laws to study limits on scaling due to memory limits.

Multiple studies[25,26,27] have shown the utility of optimizing the product of efficiency and speed-up: $\mathcal{ES}$. Several of these conclude that there exists a maximum value of p beyond which the speed-up inevitably plateaus or decreases for a given problem. In general, seeking to obtain an efficiency of 0.5 provides a good trade-off between speed-up and efficiency.[28,29]

[22] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4):609–638, 1988

[23] Patrick H. Worley. The effect of time constraints on scaled speedup. *SIAM Journal on Scientific and Statistical Computing*, 11(5):838–858, 1990

[24] X.H. Sun and L.M. Ni. Scalable problems and memory-bounded speedup. *Journal of Parallel and Distributed Computing*, 19(1):27 – 37, 1993. ISSN 0743-7315

[25] David J. Kuck. Parallel processing of ordinary programs. In Morris Rubinoff and Marshall C. Yovits, editors, *Advances in Computers*, volume 15, pages 119 – 179. Elsevier, 1976

[26] D. L. Eager, J. Zahorjan, and E. D. Lozowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Comput.*, 38 (3):408–423, March 1989. ISSN 0018-9340

[27] Horace P Flatt and Ken Kennedy. Performance of parallel processors. *Parallel Computing*, 12(1):1 – 20, 1989. ISSN 0167-8191

[28] D. L. Eager, J. Zahorjan, and E. D. Lozowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Comput.*, 38 (3):408–423, March 1989. ISSN 0018-9340

[29] Horace P Flatt and Ken Kennedy. Performance of parallel processors. *Parallel Computing*, 12(1):1 – 20, 1989. ISSN 0167-8191

Exercise

3.1. Consider the following steps in a 3-processor PRAM. Explain the effect of each instruction for each of the following models. Note that some instruction may be illegal under certain models; indicate so. All variables are in shared memory.

(a) EREW PRAM

(b) CREW PRAM

(c) Common-CRCW PRAM

(d) Aribitray-CRCW PRAM

(e) Priority-CRCW PRAM (assume priority diminishes from left to right).

P0	**P1**	**P2**
x = 5;	x = 5;	x = z;
y = z;	y = z;	y = z;

3.2. Show that each step of p-processor Common-CRCW PRAM is also valid for p-processor Arbitrary-CRCW PRAM.

3.3. Show that each step of p-processor Arbitrary-CRCW PRAM is also valid for p-processor Priority-CRCW PRAM.

3.4. Show that each step of a p-processor Priority-CRCW PRAM can be completed in up to $O(\log p)$ steps of p-processor EREW PRAM.

3.5. Show that every BSP algorithm can be converted to a PRAM algorithm.

3.6. Write pseudo-code to multiply two $n \times n$ matrices A and B, assuming the PRAM model. Analyze its time and work complexity. Assume that the input matrices A and B are stored in the shared memory in row-major order. Assume as many processors as you need.

3.7. Write pseudo-code to multiply two $n \times n$ matrices A and B, assuming the BSP model. Analyze its time complexity. The entire input matrices A and B initially reside in the processor 0. Assume as many processors as you need.

3.8. Consider the following BSP algorithm to distribute n items equally among p processors (assume n is divisible by p).

Input: Array B_0 with n integers in the memory of processor 0

Output: Array B_i in the memory of each processor i such that $B_i = A[i * b..(i+1) * b - 1]$, where $b = \frac{n}{p}$

Algorithm:

```
for(step i=0; step<logn; step++)
  forall processor i {
    {
      len = $2^{step}$
      if i < len
        send $B_i$[n/(2*len)] .. $B_i$[n/len-1] to processor i + len
    } {
      Barrier
      if len <= i < 2*len
        receive n/(2*len) items into $B_i$[0..n/(2*len)-1]
    }
  }
```

This is called a scatter operation. Analyze its time complexity. You may assume that p is a power of 2.

3.9. Devise an EREW PRAM algorithm for the problem in Exercise 3.8. Analyze its time complexity.

3.10. Consider a parallel sorting algorithm *psort* with PRAM work complexity $O(\log^2 n)$ and time complexity $O(\log n)$. Assume a PRAM limited to p processors. Compute $t(n, p)$ in the asymptotic sense. What is the efficiency compared to the best sequential sorting algorithm of $O(\log n)$?

3.11. Show the iso-efficiency function $\mathcal{I}(p)$ for the PRAM reduction algorithm in Section 3.3 is $\Omega(p \log p)$.

3.12. The following table lists execution times of two different solutions (Program 1 and Program 2) to a problem. The executions times were recorded with varying number of processors p and varying input size n. This table applies to many following questions.

Input size n	Processor count	Time $t(n,p)$ (minutes)	
(million)	p	Program 1	Program 2
1	1	12	12
	10	3.5	5.28
	50	3.2	17.0
	100	3.0	26.5
	500	3.1	126.6
10	1	22	22
	10	8.6	10.5
	50	7.1	11.9
	100	7.0	31.5
	500	7.2	126.2
50	1	263	263
	10	63.2	64.9
	50	43.2	57.9
	100	40.6	59.9
	500	40.3	158.6
100	1	1021	1021
	10	189	191
	50	110.5	125
	100	100.7	118.5
	500	94.5	211.9

Find the cost of the implementation for each n and p.

3.13. Referring to the table in Exercise 3.12, what is the latency of Program 1 for $n = 10$ million and $p = 10$?

3.14. Referring to the table in Exercise 3.12, what is the minimum latency of Program 1 execution for $n = 10$ million.

3.15. Refer to the table in Exercise 3.12. Consider a computing system with 50 total processors. What is the maximum throughput of Program 1 for $n = 10$ million?

3.16. Referring to the table in Exercise 3.12, find the maximum Speed-up $\mathcal{S}$ of Program 2 over the sequential implementation for each given value of n.

3.17. Referring to the table in Exercise 3.12, find the maximum Speed-up $\mathcal{S}$ of Program 1 over Program 2 for $n = 10$ million.

3.18. Referring to the table in Exercise 3.12, find the efficiency $\mathcal{E}$ of Program 1 and Program 2 for $n = 10$ million and $p = 100$.

3.19. Referring to the table in Exercise 3.12, estimate the Iso-efficiency function $\mathcal{I}$ for Program 1 and Program 2.

3.20. Analyze the scalability of Program 1 and Program 2 in the table in Exercise 3.12 solution. (Discuss strong *vs.* weak scalability and the iso-efficiency function.)

3.21. Discuss how well Amdahl's law and Gustafson's law hold for Programs 1 and 2 for the table in Exercise 3.12. Do they accurately estimate the bounds on the speed-up?

3.22. Refer to the table in Exercise 3.12. Using the Karp-Flatt metric, estimate the overhead (including any sequential components) in Program 2 for each value of p and $n = 10$ million. Discuss how the overhead grows with p.