Basic concepts: Measuring performance of a parallel system

1 Basic notation

We will use the following notation in describing the attributes of a problem and its algorithms.

Let n denote the input size. Sometimes we will refer to this as the "problem size". Let ω denote the sequential work that is performed by a *best* sequential[1] algorithm for the underlying problem. Note that ω need not be the same as n. For example, the work involved in sorting n numbers is $O(n \log n)$ using merge sort. Both n and ω describe the attributes of the underlying problem.

The following notation will be used with respect to describing the attributes of a parallel algorithm. We will use:

p to denote the number of processors used by an execution of a parallel algorithm. Sometimes we will refer to this as also the "processor size".

$T(n, p)$ to denote the runtime that the algorithm took to complete on a problem size of n and processor size of p. Therefore, $T(n, 1)$ is the runtime taken by the parallel algorithm to run on a single processor. Note that $T(n, 1)$ need not be necessary the same as ω. In fact, it should be easy to see that $T(n, 1) = \Omega(\omega)$.

2 Performance measures

2.1 Speedup

Speedup is defined as the ratio between the sequential time to the parallel runtime. In other words, it is the factor improvement in runtime achieved by the parallel code. We typically use S to denote speedup. By default, we will use the *best* sequential time as the reference. This is also sometimes called the "*real speedup*". Real speedup on p processors is given by:

$$\text{Real speedup } S \quad = \quad \frac{\omega}{T(n,p)} \tag{1}$$

Alternatively, we can also compute a weaker model of speedup called the "*relative speedup*", which is given by:

$$\text{Relative speedup } S \quad = \quad \frac{T(n,1)}{T(n,p)} \tag{2}$$

The notion of relative speedup is considered weaker than real speedup. For most practical applications, it is desirable to report the real speedup as it is more meaningful in conveying the gains yielded by parallelization against the best competing serial alternative. However, in some

[1] We will use the terms "sequential" and "serial" interchangeably.

practical cases use for relative speedup can be justified — for instance, in cases where there is no existing serial implementation that could scale to a large problem instance being studied, or if the goal is to diagnose scaling bottlenecks of a parallel code.

Henceforth, when we say "speedup" we imply real speedup (unless otherwise explicitly stated).

Typical speedup values are in the range of $[1, p]$ for parallel codes, with a $O(p)$ speedup curve representing *linear speedup*. However, as p increases it may become harder to maintain linearity of speedup due to parallel overheads (Section 2.2) and the presence of inherently sequential parts in the parallel program (Section 3), unless the problem size is allowed to scale up (Section 3.1). For these reasons, sub-linear speedups become the norm for large-scale applications, and maintaining the speedup near linear becomes one of the primary challenges in parallel algorithm design.

Alternatively, there could be applications where it is possible to achieve a *super-linear speedup*. Consider the example of a search operation (i.e., a specific search problem instance) where the best serial algorithm takes the worst-case time ($O(n)$) while the parallel implementation finds the solution in $O(1)$ time. Another common reason for observing super-linear speedup could be the effect of caching. A large problem instance when run on say, a single core on a multi-core system may make heavy use of the main memory during computation. However, as more cores are used, the problem size per core shrinks and the local cache lines are more effectively utilized. This could result in drastic reductions in run-time.

Notice the subtle difference between comparing a parallel algorithm's performance to a serial algorithm vs. comparing the performance of different serial algorithms. In the latter, it is typical to use their worst-case performances across inputs to compare; whereas in the former, we generally keep the input instance fixed and compare the parallel code's performance on that input to the best serial code's performance on the same input. As a result, when we study a parallel code's performance, it is common practice to report the speedup curve and other performance curves for each of the inputs tested separately.

2.2 Parallel overhead

A parallel code typically incurs overheads that are not part of a serial code. This could include mundane tasks such as initializing a communicating group of processes (or threads) or tasks that relate to inter-task communication, synchronization, etc. To measure the parallel overhead, it is necessary to quantify the net *work*[2] done by the parallel code and compare it to the work performed by the sequential code. The *net parallel work* is given by $p \times T(n, p)$. Therefore, the parallel overhead incurred by a parallel algorithm on p processors, denoted by $T_o(n, p)$, is given by:

$$\text{Parallel Overhead } T_o(n,p) \quad = \quad p \times T(n,p) - \omega \tag{3}$$

We note here that the parallel overhead is a function that typically increases linearly (if not superlinearly) with the processor size. One of the primary design goals for a parallel algorithm is to keep this overhead to a minimum.

[2]Think of "work" as the volume of computation (as opposed to the length in time) that is performed by a code.

2.3 Parallel efficiency (or simply, efficiency)

Parallel efficiency (E) is a measure of how well the processors in the system are utilized. It is given by the ratio between the work performed by the best sequential algorithm (ω) and the net work performed by the parallel algorithm ($p \times T(n,p)$).

$$\text{Efficiency } E \;=\; \frac{\omega}{pT(n,p)} \;=\; \frac{S}{p} \tag{4}$$

Efficiency is a fraction and is often expressed as a percentage. For instance, when we observe a parallel code that achieves say 75% efficiency on a system with 100 processors, it tells us th at the code is effectively utilizing 75 processors to perform useful computation. One of the main design goals in designing parallel algorithms is to design methods that can maintain efficiency close to 100% for as large parallel systems as possible (i.e., as p increases).

Example: For the $O(\lg p)$ time parallel algorithm we discussed in class for summing up p numbers using p processors, the speedup and efficiency are as follows:

$$\begin{aligned} S &= O(\frac{p}{\lg p}) \\ E &= O(\frac{1}{\lg p}) \end{aligned}$$

Exercise: Plot these curves for increasing values of p and observe the trends. What happens to the speedup and efficiency as $p \to \infty$?

One of the key properties of efficiency is that it can be preserved when the number of processors is scaled down. This can be stated and shown by the following lemma.

Lemma 2.1. *Let $E(n,p)$ denote the efficiency of a parallel algorithm when run on a given input of size of n and on p processors. If the same algorithm is run on the same input and on p' processors, where $p' < p$, then $E(n,p') \geq (E(n,p))$.*

Proof. By definition of parallel efficiency (Eqn. 4):

$$E(n,p) \;=\; \frac{\omega}{p \times T(n,p)}$$

Therefore,

$$\begin{aligned} E(n,p') &= \frac{\omega}{p' \times T(n,p')} \\ &\geq \frac{\omega}{p' \times \left\lceil \frac{p}{p'} \right\rceil T(n,p)} \\ &= E(n,p) \end{aligned}$$

This is because each processor in the reduced pool of p' processors can be made to simulate a unique batch of $\left\lceil \frac{p}{p'} \right\rceil$ processors from the larger pool of p processors—each batch costing at most $\left\lceil \frac{p}{p'} \right\rceil T(n,p)$ runtime. For instance, if we halve the number of processors ($p' = \frac{p}{2}$), then there

should be a way to pair up the workloads from each original processor such that the new parallel run-time at most doubles with p' processors. □

Note that the above simulation based argument provides one way to simulate the pool of p processors using a scaled down pool of p' processors. In other words, it suggests that the execution on a lower number of processors can be done without increasing the fraction of time contributed by the parallel overheads. In practice, the contribution from parallel overheads actually tends to decrease when the system is scaled down, resulting in a higher efficiency. In other words, the efficiency curve tends to be a non-increasing (and more typically, decreasing) curve as more processors are added to the system in most real world applications.

2.4 Speedup vs. efficiency

Given that there are two ways to measure a parallel code's performance, viz. speedup and efficiency, which of the two measures should we use in practice to compare algorithms? First, observe that if the number of processors is kept *fixed*, then speedup and efficiency become equivalent (i.e., a higher speedup implies a better efficiency, and vice versa). Therefore, while comparing any two algorithms, we will need to observe their respective behaviors (i.e., measure speedup and efficiency) on varying number of processors, while keeping the input fixed. Consider an example where there are two different parallel algorithms A_1 and A_2 for a given problem such that:

- Algorithm A_1 yields the highest speedup and that occurs when the processor size is p_1;
- Algorithm A_2 yields the highest efficiency and that occurs when the processor size is p_2, where $p_2 < p_1$.

Scenario 1: Now, let us say we want to solve the same problem on p processors such that $p < p_2 < p_1$. Which algorithm should we pick — A_1 or A_2? The answer should be whichever algorithm solves the problem faster on p processors. But can we analytically determine which of the two algorithms would solve the problem quicker on p processors, i.e., without having to run the two codes explicitly? Yes, in fact it can be argued that the algorithm A_2 will be faster than A_1 on p processors. This can be shown by comparing their efficiencies on p processors. The argument is as follows:

Let E_{A_1} and E_{A_2} denote the efficiency functions of the algorithms A_1 and A_2 respectively. Similarly, we will use T_{A_1} and T_{A_2} to denote their respective runtime functions.

By Lemma 2.1, we know that $E_{A_2}(n,p) \approx E_{A_2}(n,p_2)$. Similarly, we also know that $E_{A_1}(n,p) \approx E_{A_1}(n,p_1)$. However, since the peak efficiency was observed for A_2, their expected behaviors on p processors is likely to be: $E_{A_2}(n,p) > E_{A_1}(n,p)$.

$$\begin{aligned} \Rightarrow \quad & \tfrac{\omega}{pT_{A_2}(n,p)} > \tfrac{\omega}{pT_{A_1}(n,p)} \\ \Rightarrow \quad & T_{A_1}(n,p) > T_{A_2}(n,p) \end{aligned}$$

In the above example, efficiency provides us a more direct basis using which we can select algorithms. Note that a similar argument can be made using the speedup function although such

an argument would still have to piggyback on the efficiency conservation result of Lemma 2.1. Directly comparing speedups would be helpful only when comparing algorithms on the same number of processors.

Scenario 2: Now, consider the case where prediction of system performance on a scaled *up* system becomes important. Let us say we are managing a multi-user, multi-task data center that has a supercomputer with a large number of processors, p_{max}, such that $p_{max} >> p_1$. Assume that the users are submitting multiple tasks that entail roughly identical work, and that each of those tasks is similar to the $O(n)$ input described under Scenario 1. For sake of simplicity, let us assume that all tasks have already been submitted and are available to the scheduling software, which is responsible of coming up with an execution plan for all the tasks.

Since the peak speedup was observed for algorithm A_1 on p_1 processors, it can be inferred that there is no point running that algorithm (or for that matter, A_2) on a larger processor size than p_1. Therefore, a more realistic use-case here for either of the two algorithms would be to run multiple instances of the algorithm concurrently, each task serving a different user submitted job. In other words, the goal becomes one of maximizing *throughput*, where throughput is defined as the rate at which user tasks are processed using as much of the p_{max} processors concurrently.

$$\text{Throughput} \quad = \quad \text{number of tasks completed per unit time} \tag{5}$$

If we decide to run each task using algorithm A_1 on p_1 processors, then the throughput is:

$$\text{Throughput}_{A_1} \quad = \quad \frac{\left\lceil \frac{p_{max}}{p_1} \right\rceil}{T_{A_1}(n, p_1)} \approx \frac{p_{max}}{p_1 \times T_{A_1}(n, p_1)}$$

Alternatively, if we decide to run each task using algorithm A_2 on p_2 processors, then the throughput is:

$$\text{Throughput}_{A_2} \quad = \quad \frac{\left\lceil \frac{p_{max}}{p_2} \right\rceil}{T_{A_2}(n, p_2)} \approx \frac{p_{max}}{p_2 \times T_{A_2}(n, p_2)}$$

However, since the efficiency of A_2 on p_2 processors is more than the efficiency of A_1 on p_1 processors, $p_2 \times T_{A_2}(n, p_2) < p_1 \times T_{A_1}(n, p_1)$.

$$\Rightarrow \quad \text{Throughput}_{A_2} > \text{Throughput}_{A_1}$$

Therefore, it would be more prudent to run multiple instances of algorithm A_2 on the parallel system as that would yield in a higher throughput. Intuitively, this argument makes sense because algorithm A_2 makes better utilization of the resources it is provided than A_1, at least on system sizes that contain p_2 processors or less. Therefore, running A_2 on p_2 processors will be more optimal from a throughput perspective.

Note that there is a caveat here in practice, that the user is willing to wait for $T_{A_2}(n, p_2)$ for completion of each task. If this time exceeds the budget then settling for a slightly larger number of processors than p_2 at the expense of ideal efficiency would perhaps make more sense. It is for this reason that maintaining a high efficiency for as large a processor size as possible becomes an important design goal during parallel algorithm design.

3 Amdahl's law

Speedups are important from the perspective of reducing the time-to-solution. A good parallel algorithm would provide increasing speedups as more processors are added to the system. However, this trend *cannot* continue indefinitely. This is shown by Amdahl's law, which observes that every parallel program contains within it an inherently sequential portion and a parallel portion. Let ω_s and ω_p denote the serial and parallel fractions of the total work performed by a parallel program on a specific input (i.e., $\omega_s + \omega_p = 1$). On p processors, the time is then divided up as follows:

$$T(n,p) \quad = \quad \omega_s + \frac{\omega_p}{p}$$

$$\Rightarrow \text{maximum achievable speedup on } p \text{ processors} \quad = \quad \frac{\omega_s + \omega_p}{\omega_s + \frac{\omega_p}{p}} = \frac{1}{\omega_s + \frac{1-\omega_s}{p}} \tag{6}$$

The above equation is called *Amdahl's law* and it provides a bound the maximum achievable speedup for any parallel program on p processors. Intuitively, it implies that as the sytem size is increased it is only the parallel portion that benefits, while the serial portion starts to dominate the overall run-time, thereby limiting the speedup that can be achieved.

Amdahl's law holds when the input size is fixed and the processor size is increased. The study of a parallel program on increasing processor size keeping the input size fixed is sometimes referred to as *strong scaling*.

Example: Consider a serial program that can be 90% parallelized — i.e., $\omega_p = 0.9$ and $\omega_s = 0.1$. And let the time to run the serial program (in serial) is 100 seconds (i.e., $\omega = 100$). Then by Amdahl's law, the maximum achievable speedup using p processors is $\frac{100}{10+\frac{90}{p}}$. The above suggests that there is little merit in using more than 10 processors to solve this problem as no practically no improvement in speedup can be achieved. Furthermore, even if the number of processors is doubled from 5 to 10, the resulting speedup only goes up marginally, from 3.57x to 5.26x. Based on this observation, one can argue that there is little merit in using more than even 5 processors for this problem.

3.1 Gustafson's law

In 1988, John Gustafson observed that Amdahl's law could potentially pose a mental barrier among parallel programmers as it can be used as an excuse for not scaling up to large processor sizes [1]. In his paper, he argues that it is indeed possible to achieve near linear speedups on large-scale systems (i.e., with thousands of processors). The key is to scale up the problem/input size along with the processor size so as to keep the time to solution as flat as possible (e.g., double n while doubling p). This would result in the scaling up of local computation relative to the overheads introduced due to increased parallelism.

$$\Rightarrow \text{maximum achievable } \textit{scaled} \text{ speedup on } p \text{ processors} \quad = \quad \frac{\omega_s + p \times \omega_p}{\omega_s + \omega_p} \tag{7}$$

This is a simple and yet important result and reflects the general way we use parallel systems in the real world. Please refer to the paper for further details. The study of parallel systems by increasing both processor and problem sizes in tandem is referred to as *weak scaling*.

4 Isoefficiency

While conducting weak scaling studies, it is important to estimate how much the serial work (ω) should be increased to, as we increase the number of processors by a constant factor. The goal of weak scaling studies is to maintain efficiency by solving a larger problem in roughly the same alloted run-time. To facilitate calculating the rate at which the input work should increase, the *isoefficiency* function was introduced, which can be derived as follows:

Recall from Eqn 3 the parallel overhead can be measured as follows:

$$\begin{aligned} & T_o(n,p) = p \times T(n,p) - \omega \\ \Rightarrow \quad & T(n,p) = \frac{T_o(n,p)+\omega}{p} \\ \Rightarrow \quad & \text{Speedup } S = \frac{p\omega}{T_o(n,p)+\omega} \\ \Rightarrow \quad & \text{Efficiency } E = \frac{\omega}{T_o(n,p)+\omega} \\ \Rightarrow \quad & E = \frac{1}{1+\frac{T_o(n,p)}{\omega}} \end{aligned} \tag{8}$$

Eqn 8 indicates the following: if ω is fixed and p is increased, E degrades. Alternatively, if p is fixed and w is increased, E could be improved. A "scalable" parallel system is one that maintains efficiency as the system size is increased. This can be achieved by increasing the input work ω. However, at what rate should ω be increased to maintain efficiency? To keep efficiency fixed in Eqn 8, we need to keep the ratio between $T_o(n,p)$ and ω fixed. This can be seen by rearranging Eqn 8 as follows:

$$\begin{aligned} & E = \frac{1}{1+\frac{T_o(n,p)}{\omega}} \\ \Rightarrow \quad & \omega + T_o(n,p) = \frac{\omega}{E} \\ \Rightarrow \quad & T_o(n,p) = \frac{\omega(1-E)}{E} \\ \Rightarrow \quad & \omega = \frac{E}{1-E} T_o(n,p) \\ \Rightarrow \quad & \omega = K \times T_o(n,p) \end{aligned} \tag{9}$$

where K is the isoefficiency constant. Eqn 9 is called the *IsoEfficiency* function. The function can be used to calculate the appropriate value of ω that will help maintain efficiency at a target (predetermined) value E. Note that $T_o(n,p)$ can be pre-determined as well using the parallel and serial time complexities.

Example: Consider the problem of summing p numbers on p processors (i.e., $n = p$); The serial work for this problem is $\omega = \Theta(p)$. As discussed in class, we can compute the sum in $\Theta(\lg p)$ time in parallel. Let us now derive the isoefficiency function for this parallel algorithm.

$$\begin{aligned} T_o(n,p) &= p(\lg p - 1) \\ \Rightarrow \quad \omega &= K \times p(\lg p - 1) \end{aligned}$$

where $K = \frac{E}{1-E}$ is the isoefficiency constant. Now, let us say we want to double the number of processors. By what factor should the serial work increase by in order to maintain efficiency? If we use the isoefficiency function, then it should be easy to see that the new serial work assigned for computation on $2p$ processors should grow by a factor of $\frac{2\lg p}{\lg p - 1}$ in order to maintain efficiency. (Proof left as exercise.)
What this implies in practice is as follows. For example, if the number of processors is increased from 8 to 16, then the above equation implies that the serial work (in this case, equal to the input array size) should grow by a factor of 3x. On the other hand, if the number of processors is increased from 1024 to 2048, then growing the serial work by a factor of 2.2x would suffice.

Exercise: Provide the parallel speedup, efficiency and isoefficiency functions for the parallel algorithm used to sum n numbers on p processors (assuming $n >> p$, n is divisible by p, and p is a power of 2).

[1] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988. 01225.

Communication Patterns and MPI Primitives

1 Models for Analyzing Communication Cost

There are more than one model for analyzing communication costs over a network intraconnect — the Hockney model [3], LogP model [2], LogGP model [1] and PLogP model [4]. All these models are described indifferent to the topological details of the underlying network intraconnect.

Hockney model: The Hockney model [3] assumes that the time to send a message of size m bytes from one node to another over the network intraconnect is $\tau + \mu \times m$, where τ is a fixed latency (aka. setup cost some times), and μ is the reciprocal of the network bandwidth[1]. Network congestion is *not* captured by this model.

LogP and related models: The LogP model and their derivatives differ slightly but they all assume a latency (L) to set up internode transfer, overhead (o) to read and write from/to the network buffer, and a minimum gap in time (g) between two consecutive sends from a sender[2]. More specifically, the LogP model assumes a message sent between two nodes is of a *fixed* size. Under this assumption, the transfer time is $L + 2 \times o$, where L is the latency and o is a measure of the additional overhead to account for the time required to place the message on the network buffer by the sender and the time to read the message from the network card by the receiver. The LogGP model extends the LogP model by allowing for variable sized messages. The time to communicate a message of size m between two nodes is equal to $L + 2o + (m - 1)G$, where G is the minimum enforced time gap between two successive sends. The PLogP model is a further extension to the above models in which the sender and receiver overheads and the gap per message are all functions of the message size.

For a review of these competing models and their relative performance, please refer to [6]. While there are more such models, the most commonly used model by the message passing community is the Hockney model. For the same reason, we will use the Hockney model for analyzing communication complexity.

In what follows, we describe some of the basic point to point and collective communication primitives supported by the Message Passing Interface (MPI). Although there are multiple open source implementations of MPI (MPICH, OpenMPI, etc.), the discussion in the text does *not per se* cater to any particular implementation. Instead we will focus on at least one efficient way to structure the algorithm for these primitives and analyze their respective communication complexities. In practice, MPI implementations are configured to take advantage of the underlying network topologies and interfaces.

[1] Network bandwidth is the speed of the network and is given by the number of transferred bytes per sec.

[2] The "L" in LogP stands for latency, o for overlap and g for gap

2 MPI Point to Point Communication Primitives

2.1 Blocking communication

MPI functions: *MPI_Send, MPI_Recv, MPI_Ssend*

A *blocking* communication is one that forces the process that issues the call to wait until the communication is "complete". The notion of "completeness" here is different between the sender and receiver versions.

Any call to *MPI_Recv* waits until the entire message is received by the receiver node and is copied in its entirety into the local application buffer. If a receiver posts its receive call prior to the corresponding send by the sender, then measuring the time for the *MPI_Recv* call to return effectively measures the entire time for that communication.

Contrary to the receive function, MPI provides two different blocking versions for sending — viz. *MPI_Send* and *MPI_Ssend*. According to the MPI standard, *MPI_Send* is only required to wait until the message has left the local application (send) buffer (i.e., the latter can be safely modified by the application). In other words, this *could* potentially mean that an MPI_Send call returns when the message transfer is still in progress, or even possibly that a corresponding receive has *not* yet been posted by the receiver. For small message sizes this is probably safe, as the local network buffer is typically well equipped to temporarily cache the message before it is dispatched to the received. However, for large message sizes, this poses the risk of packet loss.

As a safer alternative, MPI offers the *MPI_Ssend* (or the synchronous send) function. A call to *MPI_Ssend* is guaranteed to return only after the receiver has posted a receive call. Even if the message size is small enough to be buffered at the sender's local network buffer, the call waits until a matching receive call is posted by the receiver. Evidently, *MPI_Ssend* calls require a handshake and are hence more expensive and also safer than *MPI_Send*. The additional cost is however warranted in applications where message sizes could be arbitrarily large and/or multiple senders could send to the same receiver concurrently.

2.2 Nonblocking communication

MPI functions: *MPI_Irecv, MPI_Test, MPI_Wait, MPI_Isend*

The time spent waiting for a communication to complete contributes directly to parallel overhead. A popular technique that is used to reduce this overhead is to *overlap communication with computation* — the idea is to perform a nonblocking call to initiate the communication and instead of waiting for the communication to complete, perform local computations that are not dependent on the communication to finish. This often requires the algorithm designer to carefully prefetch data prior to the phase in computation when they are needed, so that the computation for which data is already locally available can be allowed to continue and used to effectively mask the communication. A key to allow this overlap is the support for nonblocking calls.

MPI supports the nonblocking *MPI_Irecv* call. This call returns immediately after posting the receive so that the receiver process can perform local computation based on available/prefetched data. Subsequently, to test if the message has been received, the receiver uses either the nonblocking *MPI_Test* function or the blocking *MPI_Wait* function. A typical usage to the nonblocking receive follows one of the following two patterns:

S1) Post *MPI_Irecv*;
S2) Perform partial computation on the next batch of available data; If no more computation is possible with available data then go to step S4;
S3) Perform *MPI_Test*. If the receive status is complete, then loop back to step S1; or else, perform loop back to step S2;
S4) Perform *MPI_Wait*; when the call returns loop back to step S1.

Alternatively,

S1) Post *MPI_Irecv*;
S2) Perform computation on all available data;
S3) Perform *MPI_Wait*; when the call returns loop back to step S1.

For symmetry, MPI also supports a nonblocking send *MPI_Isend* although it is typically identical to *MPI_Send*.

MPI also supports *one-sided* communication calls such as *MPI_Get* and *MPI_Put*. As the name implies, these calls are expected to involve only one of the two processes — the sender for *MPI_Put* or the receiver for *MPI_Get* — but not both. MPI implementations vary on the level of support for these one-sided calls. For more details about these calls the reader is referred to [7].

3 Permutations

In MPI applications, multiple pairwise interprocess communications can be arranged systematically such that all processes participate within each step of communication. Such communication patterns are referred to as *permutations*. Some of the common permutations are as follows. Recall that p denotes the number of processes.

Shift permutations: Interprocess communications are arranged such that rank i sends a message to rank $i-1$ and receives from rank $i+1$ (with the exceptions of rank 0 which only receives from rank 1, and rank $p-1$ which only sends to $p-2$. This would be a *left* shift permutation. A mirror variant is the *right* shift permutation. Other variants are possible — e.g., an interleaved shift permutation where the processes communicating are separated by a fixed number of ranks apart. In shift permutations, processes are logical ordered as a bus. If indeed the physical ordering also is consistent with this logical ordering, then the performance expectations will be met in practice. Otherwise, there will be additional delays imposed by the physical layer.

Care must be taken while implementing shift permutations. A poor implementation could destroy parallelism in communcation. Can you identify such as a case?

Ring permutations: Ring permutations are identical to shift permutations with the exception that ranks 0 and $p-1$ are allowed to communicate with one another, so as to form a logical ring of processes. Ring permutations map well to network intraconnects that have an embedded ring topology.

Care must be taken while implementing ring permutations. A poor implementation could lead to deadlocks. Can you identify such as a case?

Even though the shift and ring permutations are simple, they are adequate for many real world use-cases especially in applications relating to box simulations where the next state of a cell is affected by the states of the cells in its neighborhood. The Conway's Game of Life is an apt example.

Hypercubic permutation: A hypercubic permutation is one in which all pairwise interprocess communications happen between ranks i and j, such that the bit representation of j is given by toggling the k^{th} least significant bit (alternatively, most significant bit) in the bit representation of i, for a fixed $k \in [0, \lg p - 1]$. The term hypercube comes from the network topology of the same name where each physical node contains $\lg p$ links (or neighbors). If a hypercubic permutation is carried out on a hypercubic network there will be *no* contention.

4 MPI Collective Communication Primitives

A *collective communication* is one in which *all* the processes in a communicating world participate. Examples of collective operations are: broadcast, reducing a sum, gathering data from all processes, scattering/distributing data among processes, etc. For a collective operation to happen, all processes in a communicating group (e.g., *MPI_COMM_WORLD*) should issue a call to the corresponding communication primitive passing the same values to the communicator argument and the "root" argument (if any). Note that this also implies that a collective operation can be performed for any smaller communicating groups of processes that are subsets of the *MPI_COMM_WORLD*. A "root" is a special process identified by some (but not all) collective communicators as either the originator or the final receiver for the collective operation (e.g., a broadcast root, or a reducer's root where the reduced value will be stored).

Caveat about collective operations and MPI implementations: While it is typical to expect that all participating processes in a collective operation at least post a call before the operation starts, this is not enforced for most of the collective primitives by the latest standard for MPI, MPI-3.0 [5]. The standard also does *not* enforce that the calls should return only after the communication is completed in all participating processes. In other words, for many of the collective primitives, it is possible that a caller may return soon after its contribution to the collective operation is completed while some other processes may be still working on the communication.

4.1 Barrier

MPI functions: *MPI_Barrier*

The *MPI_Barrier* function is used to ensure that *all* processes have reached a certain stage in computation before proceeding. The barrier function is invoked on a specific communicating group (e.g., *MPI_COMM_WORLD*) and each process within that group blocks on the call until *all* other processes of that group have invoked the barrier function as well. At that point the call returns.

Note that there is no application level data that is exchanged during barrier. In contrast, all the remaining collective primitives described below involve exchanging application data transfers. For purpose of uniformity, we will use the notation m to denote the size of the message (in bytes) at any process prior to the collective operation. We will also use the notation n to denote the sum of the number of elements (of any particular data type) in all the processes, and p to denote the number of processes.

4.2 Reduce

MPI functions: *MPI_Reduce, MPI_Allreduce*

Precondition: Each process has $O(\frac{n}{p})$ elements as input. Let the input in rank i be represented as an array $A_i[1 \ldots \frac{n}{p}]$. Also given is a binary associative operator denoted as $\otimes$ (e.g., addition, logical or/and, max, min, etc.).

Postcondition: Output array $B[1 \ldots \frac{n}{p}]$ s.t., $B[k] = A_0[k] \otimes A_2[k] \otimes \ldots A_{p-1}[k]$. If the function is *MPI_Reduce*, then the array B is made available in the process identified as the root. If the function is *MPI_Allreduce*, then array B is made available in all the processes.

Assuming that each application of the binary operator $\otimes$ takes constant time, reduction can be achieved in $O(\lg p)$ stages of hypercubic permutations. Within each permutation, $\frac{n}{p}$ elements are exchanged between any two processes to compute the intermediate B arrays. The time complexity for this communication is therefore $O((\tau + \mu\frac{n}{p}) \times \lg p) = O((\tau + \mu m) \times \lg p)$.

4.3 Broadcast

MPI functions: *MPI_Bcast*

Precondition: The root process has a message of m bytes.

Postcondition: The message at the root process is available at all the other processes.

It should be easy to see that the broadcast can be implemented by reversing the communication stages of the reduce operation, implying that it can also be achieved in $O(\lg p)$ stages of hypercubic permutations. The time complexity for broadcast is therefore $O((\tau + \mu m) \times \lg p)$.

4.4 Gather

MPI functions: *MPI_Gather, MPI_Allgather*

Precondition: Each process has $O(\frac{n}{p})$ elements. Let the input in rank i be represented as an array $A_i[1 \ldots \frac{n}{p}]$.

Postcondition: Output array $B[1 \ldots n]$ which is given by concatenating all the individual rank arrays A_i in that order of ranks. If the function is *MPI_Gather*, then the array B is made available in the process identified as the root. If the function is *MPI_Allgather*, then array B is made available in all the processes.

The gather operation can also be completed in $O(\lg p)$ hypercubic permutations as follows. Let $O(m_k)$ be the message size at each process at the beginning of the k^{th} hypercubic permutation stage. During the k^{th} stage, all communicating pairs of processes perform a gather of their respective $O(m_k)$ messages such that at the end of the stage, both processes of a pair have the same message of size $O(2 \times m_k)$. Consequently, the last stage results in the gather of $O(m \times p)$ size message at the root process (alternatively, at all processes for *MPI_Allgather*). The communication time complexity is bounded by $O(\tau \times \lg p + \mu \times m \times p)$.

4.5 Scatter

MPI functions: *MPI_Scatter*

Precondition: The root process has $O(n)$ elements. Let this input be $A[1 \ldots n]$ (of size m).

Postcondition: Scatter array $A[1 \ldots n]$ to all processes such that process rank i contains elements $A[i \times \frac{n}{p} \ldots (i+1) \times \frac{n}{p} - 1]$.

The scatter operation can also be completed by reversing $O(\lg p)$ hypercubic permutations of the gather operation. Consequently, the communication time complexity is $O(\tau \times \lg p + \mu \times m))$.

4.6 Parallel Prefix

MPI functions: *MPI_Scan*

(p element case)

Precondition: An array of A of p elements is kept distributed such that rank i holds element $A[i]$. We will refer to the value of $A[i]$ as x_i. We are also specified a binary associative operator which we will denote by $\otimes$.

Postcondition: An output array B of size p available in a distributed manner such that rank i holds element $B[i]$. We will refer to the value of $B[i]$ as y_i. The value of y_i is given by:

$$y_i \ = \ x_0 \otimes x_1 \otimes \ldots x_{i-1} \otimes x_i$$

The algorithm for parallel prefix can be accomplished in $O(\lg p)$ communication steps using hypercubic permutations, similar to the reduce and broadcast primitives. For details on the parallel prefix algorithm and its applications, please refer to the lecture notes by Prof. Aluru.

(n element case)

An easy extension of the $O(\lg p)$ approach lends itself to solving the more generic case of parallel prefix where the input array A is of size n, where $n >> p$ (i.e., each processor initially holds $O(\frac{n}{p})$ entries). Here the complexity will be:
Computation complexity: $O(\frac{n}{p})$
Communication complexity: $O((\tau + \mu) \lg p)$.

4.7 Transportation Primitives

MPI functions: *MPI_Alltoall, MPI_Alltoallv*

MPI_Alltoall:

Precondition: Each processor p_i has a message size of m to send to every processor (including to itself). We will denote the set of p messages in rank i has $m_{i,0}$, $m_{i,1}$, ..., $m_{i,p-1}$.

Postcondition: Processor i holds all its messages from every processor in order — i.e., at output, rank i holds an array which is given by $m_{0,i} \cdot m_{1,i} \cdot \ldots \cdot m_{p-1,i}$.

There are a number of ways to implement the Alltoall primitive on different parallel architectures. A simple lowerbound to these implementations, however, is imposed by the maximum between the sizes of the messages that needs to sent and received at any processor. If this value is bound by $O(m)$ for a communication that happens between any two processor, then the overall complexity of the Alltoall communication is $\Omega(m \times p)$. Therefore, a simple approach to implement Alltoall is to have every processor send to every other processor in a round-robin manner (or any other deterministic way). A simple pseudocode at rank i could be as follows:

S1) For $j \leftarrow 1$ to p do:
S2) destination $k \leftarrow (i + j) \mod p$
S3) Send $m_{i,k}$ to k.

The communication complexity for this algorithm is $O(\tau p + \mu m p)$.

Note that the Alltoall operation is same as performing the equivalent of Allscatter from every processor. It is for this reason that MPI does *not* explicitly support Allscatter.

MPI_Alltoallv:

The MPI_Alltoallv is a generalization of the MPI_Alltoall primitive in that the Alltoallv version allows for variable message sizes to be sent from each processor to any other processor. The assumption here is that on any processor's local memory the values that need to be sent to the same destination processor are made available contiguously. Prior to calling the Alltoallv primitive, each rank has to specify the locations (i.e., pointers) on its local memory to each of the p send buffers and the corresponding send counts (i.e., how many values of what type going to each destination processor). Using this model, Alltoallv can be used to send arbitrary sized messages from any processor to any other processor.

In many applications that need variable size messages to be sent between processors, it is possible to guarantee that the sum of the sizes of the outgoing messages and the sum of the sizes of the incoming messages at any processor satisfies a bound $O(m)$ — i.e., at rank i, $\Sigma_{j=1}^{p} m_{i,j} = O(m)$. As an example, consider the sorting operation. The following example shows a three processor system each holding 6 elements marked for different destinations as shown below:

p_0: 0 0 1 1 1 2 | p_1: 0 1 1 2 2 2 | p_2: 0 0 0 1 2 2

Here, rank 0 has different number of values to send to each of the p processors — i.e., 2 to p_0, 3 to p_1 and 1 to p_2. However the sum of the outgoing messages and the sum of the incoming messages at any processor is bounded by 6. For these scenarios, where there is variability in the size distribution of messages but the net inbound/outbound message volume can be bounded in every processor, one can implement Alltoallv using two Alltoall communication steps so that the entire communication cost can be bounded by the cost of Alltoall — i.e., $O(\tau p + \mu m)$.

The algorithm to implement Alltoallv using two calls to Alltoall is as follows:

S1) Each processor i splits *each* of its p messages into p equal parts (generating a total of p^2 parts per processor).
S2) Rearrange the p^2 parts such that part k is marked for destination rank $(k \mod p)$.
S3) Perform an Alltoall using the send buffer generated in the above step. Note that it becomes possible to use Alltoall here because each processor is guaranteed to have the same message size ($\frac{m}{p}$) outbound to every other processor, as a result of the division in step S1.

S4) Upon Alltoall completion, every processor rearranges the p^2 parts that it received from all p processors such that part k is marked for destination rank $(k \mod p)$.
S5) Perform another Alltoall using the send buffer created by the above step.
S6) Upon Alltoall completion, each processor should have only the parts that are destinted for that processor. However, the parts will be shuffled in a cyclic permutation. Therefore, a shuffle operation is performed locally to gather parts that originated at the same processor consecutively.

For an illustrative example of the process, please refer to Figure 1.

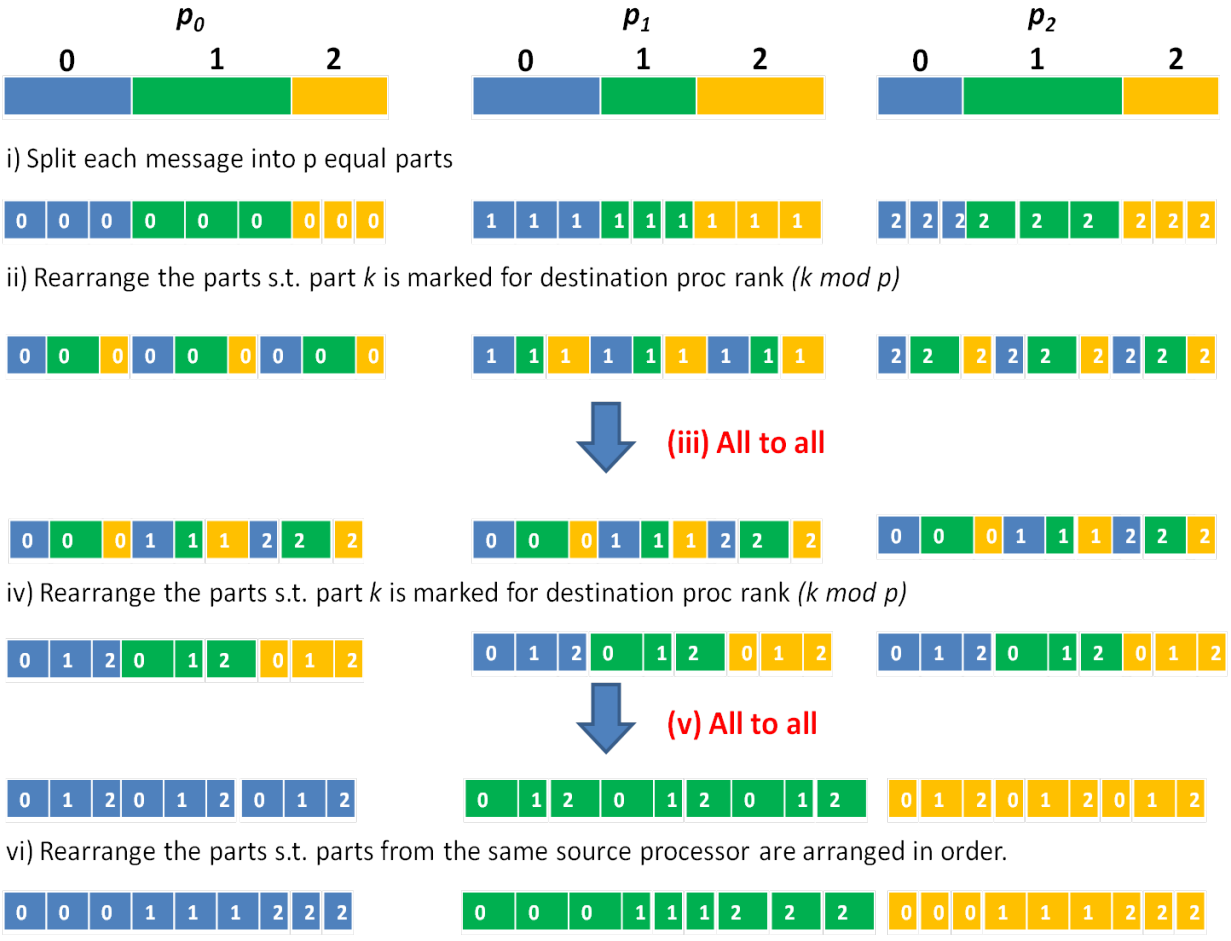

Figure 1: An example showing the Alltoallv transportation primitive algorithm using two Alltoallv communication steps.

5 Error Handling

MPI functions: *MPI_Abort*

MPI supports an abort function which can be invoked from any processor rank in the event an exception needs to be raised. A call to MPI_Abort will result in aborting the processes at all ranks.

MPI_Abort is typically useful for generating exceptions and also during debugging.

[1] Albert Alexandrov, Mihai F Ionescu, Klaus E Schauser, and Chris Scheiman. LogGP: incorporating long messages into the LogP modelone step closer towards a realistic model for parallel computation. pages 95–105. ACM, 1995.

[2] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. *LogP: Towards a realistic model of parallel computation*, volume 28. ACM, 1993.

[3] Roger W Hockney. The communication challenge for MPP: Intel paragon and meiko CS-2. *Parallel computing*, 20(3):389–398, 1994.

[4] Thilo Kielmann, Henri E Bal, and Kees Verstoep. Fast measurement of LogP parameters for message passing platforms. In *Parallel and Distributed Processing*, pages 1176–1183. Springer, 2000.

[5] MPIForum. The MPI 3.0 standard: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, 2012.

[6] Jelena Pjeivac-Grbovi, Thara Angskun, George Bosilca, Graham E Fagg, Edgar Gabriel, and Jack J Dongarra. Performance analysis of MPI collective operations. *Cluster Computing*, 10(2):127–143, 2007.

[7] Rajeev Thakur, William D Gropp, and Brian Toonen. Minimizing synchronization overhead in the implementation of MPI one-sided communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 57–67. Springer, 2004.

Matrix Computations

1 Overview

In this chapter we will visit two basic matrix computations from a parallel processing perspective — viz. matrix-vector multiplication and matrix-matrix multiplication. For simplicity, we will focus on dense matrices where the problem is regular and easier to parallelize than for sparse matrices. We will also assume the distributed memory model for parallelization. However it should become easy to see that these parallelization strategies can be adapted to the shared memory model as well.

The reader is encouraged to look at other lecture resources mentioned in the course webpage for a more detail discussion.

2 Basic Notation

We will denote a row vector $\mathbf{v}$ of length n as $\mathbf{v}[1, n]$, and a column vector $\mathbf{v}$ of length n as $\mathbf{v}[n, 1]$. To refer to the i^{th} element of a vector (row or column), we simply use $\mathbf{v}[i]$.

We will denote a matrix $\mathbf{M}$ of size $m \times n$ as $\mathbf{M}[m, n]$, where m and n denote the number of rows and columns respectively. $\mathbf{M}[i][j]$ denotes the element in cell (i, j) of the matrix.

Let us consider a p processor system in a distributed memory parallel computer. We will denote M_{local} to denote the memory that is available locally to any given processor on the distributed memory machine. The sum of all the local memory is sometimes referred to as the *aggregate memory* of the distributed memory machine.

3 Data Partitioning Approaches for Distributed Memory

The process of dividing an input among a group of p processors is called *input partitioning*. Partitioning schemes are used to assign the computation task associated with a data part to the processor space. This assignment can either be performance statically (i.e., in a pre-determined fashion) or dynamically (particularly useful if there is variability in the time to process each data part).

Let us consider a matrix $\mathbf{M}$ of size $m \times n$. Here we will assume, without loss of generality, that $O(\frac{mn}{p})$ will fit in M_{local}. We say, without loss of generality here because given unlimited parallel resources, we always have the option of scaling up the number of processors until the space lower bound $O(\frac{mn}{p})$ can be met.

Given the above assumption, there are different ways to partition a matrix of size $m \times n$ among p processors.

Block partitioning: In block partitioning, consecutive chunks (or "blocks") of either $O(\frac{m}{p})$ rows or $O(\frac{n}{p})$ columns are distributed among the p processors. For instance, in row-wise partitioning, rank i will hold rows $i \times \frac{m}{p}$ to $(i + 1) \times \frac{m}{p} - 1$ of the matrix (assuming m is divisible by p).

Cyclic partitioning: In a row-wise cyclic partitioning, a given row i will be given to rank $(i \mod p)$. Similarly, in a column-wise cyclic partitioning, column j will be given to rank $(j \mod p)$. It should be easy to see that under this model too, each processor holds only $O(\frac{mn}{p})$ of the input.

Block-Cyclic partitioning: A row-wise (alternatively, column-wise) block-cyclic partitioning combines both block and cyclic approaches. More specifically, consider a row-wise block-cyclic distribution of a matrix with m rows. Here, the matrix rows is first divided into fixed size blocks of size k such that $k << p$ (i.e., k rows per block; for a total of $\frac{m}{k}$ blocks). Subsequently, block i is given to rank $(i \mod p)$.

Extension to arbitrary dimensions: The aforementioned partitioning schemes either operate on the set of rows or columns of a two dimensional matrix. In other words, the partitioning schemes apply to a given dimension of the input matrix. Consequently, we can extend the schemes to arbitrary dimensions. For instance, consider a 2-D matrix containing m rows and n columns. One can apply both row- and column-wise block partitioning on a logical network of $\sqrt{p} \times \sqrt{p}$ processors — s.t., each processor holds a distinct $O(\frac{mn}{p})$ sub-matrix. More specifically, rank i holds the sub-matrix defined by $\mathbf{M}[k'\frac{m}{\sqrt{p}} \ldots (k'+1)\frac{m}{\sqrt{p}} - 1, k\frac{n}{\sqrt{p}} \ldots k\frac{n}{\sqrt{p}} - 1]$, where $k' = \lfloor \frac{i}{\sqrt{p}} \rfloor$ and $k = i \mod \sqrt{p}$. Such a partitioning would correspond to a *2-D block partitioning* of the input matrix.

4 Matrix Vector Multiplication: MxV

Input:

- $\mathbf{M}[m, n]$: an $m \times n$ matrix
- $\mathbf{x}[n, 1]$: a column vector with n values

Output:

- $\mathbf{y}[m, 1]$: a column vector with m values, such that: $\mathbf{M} \times \mathbf{x} = \mathbf{y}$

The serial algorithm for the matrix vector multiplication is given in Algorithm 1. The algorithm takes $\Theta(m \times n)$ run-time and $\Theta(m \times n)$ space.

```
Data: M[m, n], x[n, 1]
Result: y[m, 1]
for i: 0 to m − 1 do
    y[i] = 0;
    for j: 0 to n − 1 do
        y[i]+ = M[i][j] × x[j];
    end
end
```

Algorithm 1: Serial algorithm for Matrix Vector multiplication.

4.1 Parallelization

Consider the case where the local memory per processor is sufficient to accommodate $O(n)$ input. Under this assumption, the matrix-vector multiplication can be parallelized as shown in Algorithm 2.

Data: $\mathbf{M}[m, n]$, $\mathbf{x}[n, 1]$; p processors
Result: $\mathbf{y}[m, 1]$
Pre-condition:
$\mathbf{x}$ resides on each processor;
The rows of $\mathbf{M}$ are distributed evenly among p processors (i.e., $O(\frac{m}{p})$ rows per proc.). This is same as row-wise block partitioning. We will refer to the local copy of the matrix as $\mathbf{M_{local}}$.
Post-condition:
Rank i outputs $\mathbf{y}[i \times \frac{m}{p}, (i+1) \times \frac{m}{p} - 1]$. We will refer to the local copy of the output vector as $\mathbf{y_{local}}$.
Parallel algorithm:
for i: 0 *to* $\frac{m}{p} - 1$ **do**
  $\mathbf{y_{local}}[i] = 0$;
  for j: 0 *to* $n - 1$ **do**
    $\mathbf{y_{local}}[i] += \mathbf{M_{local}}[i][j] \times \mathbf{x}[j]$;
  end
end
Output $\mathbf{y_{local}}$.

Algorithm 2: Parallel algorithm for Matrix Vector multiplication. The pseudocode assumes that the local memory on each processor is $\Omega(n)$).

Analysis: The run-time complexity of Algorithm 2, assuming the input is already made available as specified in the pre-condition, is as follows:
Computation complexity = $O(\frac{mn}{p})$
Communication complexity = $O(1)$ (as there is no communication).

If the local memory available to each processor is *not sufficient* to accommodate $O(n)$ input[1], which can be expected to occur for large values of n, then the parallel algorithm 2 can be modified such that each processor loads only a part of a given row of $\mathbf{M}$ and the vector $\mathbf{x}$. More specifically:
Initialization/Partitioning:

1) Use the 2-D block partitioning of the matrix (as described in Section 3). At the end of this step, each processor holds a unique $\frac{m}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ submatrix of the input.
2) Use a modified version of the block-cyclic distribution of the the vector $\mathbf{x}$ such that $\sqrt{p}$ copies of $\mathbf{x}$ are stored among the p processors. More specifically, the vector is split into $\sqrt{p}$ blocks such that rank i holds $\mathbf{x}[k\frac{n}{\sqrt{p}} \ldots (k+1)\frac{n}{\sqrt{p}-1}]$, where $k = i \mod \sqrt{p}$. At the end of this step, each processor holds a unique $\frac{n}{\sqrt{p}}$ part of the input vector $\mathbf{x}$ which corresponds to the same set of columns that this processor stores (after step 1).

[1] This can happen for large, rectangular matrices, where $n >> m$.

Once the input has been partitioned in the above manner, each processsor is responsible to perform an independent matrix-vector product for the sub-matrix and vector parts that it holds. Each processor will output a vector of $\frac{n}{\sqrt{p}}$ values. To compute the final output vector $\mathbf{y}$, there needs to be a subsequent step where there are $\sqrt{p}$ separate parallel reductions, each involving $\sqrt{p}$ processors. More specifically, all ranks i that have the same $(i \mod \sqrt{p})$ value participate in a single reduction that outputs a distinct $\frac{n}{\sqrt{p}}$ part of the vector $\mathbf{y}$. The verification of the algorithm and a precise pseudocode is left as an exercise.

Analysis: The run-time complexity for this revised parallell algorithm is as follows:
Computation complexity = $O(\frac{mn}{p})$
Communication complexity = $O(\tau \lg \sqrt{p} + \mu \frac{n}{\sqrt{p}} \lg \sqrt{p})$

5 Matrix Matrix Multiplication: MxM

Please refer to the lecture notes by Prof. Aluru for a detailed discussion on Cannon's algorithm.

Parallel Sorting

1 Overview

In this chapter we will visit the problem of parallel comparison-based sorting on distributed memory machines. The input is a distributed representation of an array $A[0 \ldots n-1]$ containing n elements and a comparison function to compare any two elements. The output is a distributed representation of the sorted array $B[0 \ldots n-1]$. If p denotes the number of processors, we will assume $n >> p$. For ease of exposition, we will assume n to be divisble by p.

Pre-condition: Every processor stores $O(\frac{n}{p})$ distinct elements of the unsorted array A initially. We will denote the array local at rank i by A_i.

Post-condition: The sorted array $B[0 \ldots n-1]$ is stored in a distributed manner, such that rank i stores the i^{th} $O(\frac{n}{p})$ segment of the sorted array. We will denote the portion of the sorted array output at rank i by B_i.

In this chapter, we will describe two parallel sorting algorithms.

1.1 Sample sort

Sample sort is an extension of the partitioning-based scheme used in conventional quick sort. The main idea is to identify pivots to partition the input array so that the individual parts can then be subsequently sorted locally within each processor. Finding pivots that can evenly partition the input array into roughly p equal parts can be done in many ways; however, size bounds (on the output array) are harder to guarantee. For instance one can collect statistics such as minimum and maximum to decode the range but a skewed input could cause potential imbalances among processor workloads.

Traditional sample sort-based methods adopt typically a two-step approach: of first generating a set of "local pivots" based on the local distribution and using those pivots to generate a set of "global pivots" to perform the final partitioning.

The algorithmic template for sample sort is as follows:

S1) Every processor sorts its local array A_i, and picks $p-1$ evenly spaced pivots from the local sorted array. These local pivots are also referred to as "*local splitters*". We will also refer to the sorted array as A_i^s; note that A_i^s could be derived directly from array A_i without additional space, if sorting is done inplace.

S2) The $p(p-1)$ splitters generated across all p processors are sorted, and subsequently a subset of $p-1$ evenly spaced pivots is selected. These pivots are also referred to as the "*global splitters*". The sorting in this step can be either performed in parallel (using another parallel sorting method such as bitonic sort) or in serial, depending on the size of p. For the purpose of analysis in this chapter, we will assume this step is performed using first an Allgather communication of the local splitters, followed by each processor locally sorting all $p(p-1)$ pivots to obtain the $p-1$ global splitters.

S3) Using the global splitters, each processor partitions its local array into p (possibly empty) parts, marking the i^{th} part to be sent to processor rank i.
S4) The arrays are redistributed using the Alltoallv transportation primitive.
S5) The set of elements received at every processor is then locally sorted. This can either be achieved using a local sort of all the elements achieved, or as a local merge operation of p sorted lists. Note that the individual parts received from each source processor in the Alltoallv step are already sorted internally.
At the end of this step, the concatentation of the local arrays (B_i) from rank 0 to rank $p-1$ in that order represents the final sorted output (B).

Let n_i be the number of elements in the sorted array output by rank i at the end of sample sort. In the above version of sample sort, this number (which denotes the output size within each processor) can be bounded.

Lemma 1.1. *The number of elements in the sorted array output by any rank i at the end of sample sort is $O(\frac{n}{p})$.*

Proof. In the sample sort algorithm let the sequence of $(p-1)$ global splitters be labelled $\{s_0, s_1, \ldots, s_{p-2}\}$. Processor p_i ($i > 0$) receives all elements x in the input array A s.t. $s_{i-1} < x \leq s_i$, while processor p_0 receives all $x \leq s_0$. Let us refer to the set of values received at a given processor p_i as $Range(i)$. Note that we are interested in determining an upper-bound for the size of this $Range(i)$. In what follows, we show that $|Range(i)| < 2 \times \frac{n}{p}$.

It should be easy to see that this set $Range(i)$ could include a subset of local splitters that are generated in different processors. In fact there has to be exactly $(p-1)$ local splitters included in the $Range(i)$. This is because the global splitters themselves were originally picked by selecting $(p-1)$ evenly spaced splitters from the sorted array of $p(p-1)$ local splitters (step S2).

These $(p-1)$ local splitters can originate from one or more processors. This gives raise to two sub-cases:
Case A) All $(p-1)$ local splitters originate from the same processor.
Case B) The $(p-1)$ local splitters originate from two or more processors.

Case A) Let the $(p-1)$ local splitters originate from processor p_j. This implies a possibility that all the $\frac{n}{p}$ elements of the local input array $A_j$could belong to $Range(i)$. Now, what can we say about the contribution of other processors p_k ($k \neq j$) to p_i? We show here that the net contribution from these remaining processors cannot exceed $\frac{n}{p}$. The proof is as follows: Since p_j contributed all the $(p-1)$ local splitters in $Range(i)$, other processors could have not contributed any local splitters. However there could still be elements in their respective local input arrays that belong to $Range(i)$. If there are such elements, they need to be occurring prior to the first local splitter or after the last local splitter within those processor's sorted arrays (i.e., A_k^s). This implies that the maximum contribution from any such p_k to $Range(i)$ is bounded by $\frac{n}{p^2}$. Therefore, the net contribution to $Range(i)$ from any p_k ($k \neq j$) can be bounded by $(p-1) \times \frac{n}{p^2} < \frac{n}{p}$. Therefore, $|Range(i)| < 2 \times \frac{n}{p}$.

Case B) Let k_j denote the number of local splitters contributed by processor p_j to $Range(i)$. If $k_j > 1$ then all the elements in A_j^s between those local splitters are also in $Range(i)$. Note that there are exactly $\frac{n}{p^2}$ elements between any two consecutive local splitters of A_j^s. In addition, it is possible that a subset of the $\frac{n}{p^2}$ elements preceding the first of those k_j splitters or following

the last of those k_j splitters (but not both) could also belong in $Range(i)$. This implies that the maximum contribution from p_j to $Range(i)$ is $(k_j + 1)\frac{n}{p^2}$. Therefore, the total contribution from all p processors to $Range(i)$ is given by:

$$\begin{aligned} |Range(i)| &\leq \Sigma_{j=0}^{p-1}(k_j+1)\frac{n}{p^2} \\ &= \frac{n}{p^2}(\Sigma_{j=0}^{p-1}(k_j) + \Sigma_{j=0}^{p-1}1) \\ &= \frac{n}{p^2}(p-1+p) \\ &< 2 \times \frac{n}{p} \end{aligned} \tag{1}$$

□

Analysis: The sample sort algorithm has the following complexity:

Step	Computation time	Communication time
S1	$\Theta(\frac{n}{p}\lg\frac{n}{p})$	
S2[1]	$\Theta(p^2 \lg p^2)$	$O(\tau \lg p + \mu p^2)$
S3	$O(\frac{n}{p})$	
S4		$O(\tau p + \mu\frac{n}{p})$ ($\because |Range(i)| < \frac{n}{p}$)
S5	$\Theta(\frac{n}{p}\lg\frac{n}{p})$	
Total:	$O(\frac{n}{p}\lg\frac{n}{p} + p^2 \lg p^2)$	$O(\tau p + \mu(p^2 + \frac{n}{p}))$

1.2 Bitonic sort

Please refer to the notes by Prof. Aluru for a detailed discussion on Bitonic sort.

[1]Note that the quadratic terms in step S2 can be brought down by performing that sort in parallel.