Parallel Computing

Xing Cai

September 24, 2013

1 Introduction

Parallel computing can be understood as solving a computational problem through collaborative use of multiple resources that belong to a parallel computer system. Here, a parallel system can be anything between a single multiprocessor machine and an internet-connected cluster that is made up of hybrid compute nodes. There are two main motivations for adopting parallel computations. The first motivation is about reducing the computational time, because employing more computational units for solving a same problem usually results in lower wall-time usage. The second—and perhaps more important—motivation is the wish of obtaining more details, which can arise from higher temporal and spatial resolutions, more advanced mathematical and numerical models, and more realizations, etc. In this latter situation, parallel computing enables us to handle a larger amount of computation under the same amount of wall-time. Very often, it also gives us access to more computer memory, which is essential for many large computational problems.

The most important issues for understanding parallel computing are finding parallelism, implementing parallel code, and evaluating the performance. These will be briefly explained in the following text, with simple supporting examples.

2 Identifying parallelism

Parallelism roughly means that some work of a computational problem can be divided into a number of simultaneously computable pieces. The applicability of parallel computing to a computational problem relies on the existence of inherent parallelism in some form. Otherwise, a parallel computer will not help at all.

Let us take for example the standard *axpy* operation, which updates a vector $\mathbf{y}$ by adding it to another vector $\mathbf{x}$ as follows:

$$\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y},$$

where α is a scalar constant. If we look at the entries of the $\mathbf{y}$ vector: $y_1, y_2, \ldots, y_n$, we notice that computing y_i is totally independent of y_j, thus making each entry of $\mathbf{y}$ a simultaneously computable piece. For instance, we can employ n workers, each calculating a single entry of $\mathbf{y}$.

The above example is extremely simple, because the n pieces of computation are completely independent of each other. Such a computational problem is often termed *embarrassingly parallel*. For other

problems, however, parallelism may be in disguise. This can be exemplified by the dot product between two vectors **x** and **y**:

$$d = \mathbf{x} \cdot \mathbf{y} := \sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \ldots + x_n y_n.$$

At a first glance, parallelism is not obvious within a dot product. However, if an intermediate vector **d** is introduced, such that $d_i = x_i y_i$, then parallelism immediately becomes evident because the n entries of the **d** vector can be computed simultaneously. Nevertheless, the remaining computational task:

$$d = 0, \quad d \leftarrow d + d_i \quad \text{for } i = 1, 2, \ldots, n$$

requires collaboration and coordination among n workers to extract parallelism. The idea is as follows. First, each worker with an odd-number ID adds its value with the value from the neighboring worker with an ID of one higher. Thereafter, all the even-numbered workers retire and the remaining workers repeat the same process until there is only one worker left. The solely surviving worker possesses the desired final value of d. Actually, this is how a parallel *reduction* operation is typically implemented. We can also see that the parallelized summation has $\lceil \log_2 n \rceil$ stages, each involving simultaneous additions between two and two workers. Although it may seem that the parallel version should be dramatically faster than the original serial version of summation, which has n stages, we have to remember that each stage in the parallel counterpart requires data transfer between two and two workers, causing so-called *communication* overhead.

What is more intriguing is that parallelism can exist on different levels. Let us revisit the example of summing up the **d** vector, but assume now that the number of workers, m, is smaller than the vector length n. In such a case, each worker becomes responsible for several entries of the **d** vector, and here are several issues that require our attention:

1. The n entries of the **d** vector should be divided among the m workers as evenly as possible. This is called *load balancing*. For this particular example, even when n is not a multiple of m, a fair work division makes the heaviest and lightest loaded workers only differ by one entry.
2. Suppose each worker prefers a contiguous segment of **d**, then worker k, $1 \leq k \leq m$, should be responsible for entry indices from $((k-1) * n)/m + 1$ until $(k * n)/m$. Here, we let / denote the conventional integer division in computer science.
3. The local summations by the m workers over their assigned entries can be done simultaneously, and each worker stores its local summation result in a temporary scalar value d_k^s.
4. Finally, the m local summation results d_k^s, $1 \leq k \leq m$, can be added up using a parallel reduction operation as described above.

The above examples are only meant for illustration. Parallelism in practical computational problems exist in many more different forms. An incomplete list of frequently encountered types of parallelizable computations involve dense linear-algebra operations, sparse linear-algebra operations, explicit and implicit computations associated with regular meshes, implicit computations associated with irregular meshes, fast Fourier transforms, and many-body computations.

3 Parallelization

Finding parallelism in a computational problem is only the start. A formal approach to designing parallel algorithms is Foster's Methodology [2, 7], which is a four-step process. The first step is partitioning,

which cuts up the concurrent computational work and/or the accompanying data into as many small pieces as possible. The second step of Foster's Methodology is about finding out what data should be exchanged between which pieces. The third step is about agglomerating the many small pieces into a few larger tasks, to obtain an appropriate level of *granularity* with respect to the hardware resources on a target parallel computer. The last step of Foster's Methodology is about mapping the tasks to the actual hardware resources, so that load balance is achieved and that the resulting data communication cost is low. A rule-of-the-thumb regarding communication is that two consecutive data transfers, between the same sender and receiver, are more costly than one merged data transfer. This is because each data transfer typically incurs a constant start-up cost, termed *latency*, which is independent of the amount of data to be transferred.

To make a parallel algorithm run efficiently on a parallel computer, the underlying hardware architecture has to be considered. Although parallel hardware architectures can be categorized in many ways, the most widely adopted consideration is about whether the compute units of a parallel system share the same memory address space. If yes, the parallel system is termed *shared memory*, whereas the other scenario is called *distributed memory*. It should be mentioned that many parallel systems nowadays have a hybrid design with respect to the memory organization, having a distributed-memory layout on the top level, whereas each compute node is itself a small shared-memory system.

Luckily, the styles of parallel programming are less diverse than the different parallel architectures produced by different hardware vendors. The MPI programming standard [6, 3] is currently the most widely used. Although designed for distributed memory, MPI programming can also be applied on shared-memory systems. An MPI program operates a number of MPI processes, each with its own private memory. Computational data should be decomposed and distributed among the processes, and duplication of (global) data should be avoided. Necessary data transfers are enabled in MPI by invoking specific MPI functions at appropriate places of a parallel program. Data, which are called *messages* in MPI terminology, can either be passed from one sender process to a receiver process, or be exchanged collectively among a group of processes. Parallel reduction operations are namely implemented as collective communications in MPI.

OpenMP [1] is a main alternative programming standard to MPI. The advantage of OpenMP is its simplicity and minimally-intrusive programming style, whereas the performance of an OpenMP program is most often inferior to that of an equivalent MPI implementation. Moreover, OpenMP programs can only work on shared-memory systems.

With the advent of GPUs as main accelerators for CPUs, two new programming standards have emerged as well. The CUDA [4] hardware abstraction and programming language extension are tied to the hardware vendor NVIDIA, whereas the OpenCL framework [5] targets heterogeneous platforms that consist of both GPUs and CPUs and possibly other processors. In comparison with MPI/OpenMP programming, there are considerably more details involved with both CUDA and OpenCL. The programmer is responsible for host-device data transfers, mapping computational work to the numerous computational units—threads—of a GPU, plus implementing the computations to be executed by each thread. In order to use modern GPU-enhanced clusters, MPI programming is typically combined with CUDA or OpenCL.

4 Performance of parallel programs

It is common to check the quality of parallel programs by looking at their scalability, which is further divided as strong and weak scalability. The former investigates *speedup*, i.e., how quickly the wall-time

usage can be reduced when more compute units are used to solve a fixed-size computational problem. The latter focuses on whether the wall-time usage remains as constant when the problem size increases linearly proportional to the number of compute units used.

The blame for not achieving good scalability has traditionally been put too much on the non-parallelizable fraction of a computational problem, giving rise to the famous laws of Amdahl and Gustafson-Barsis. However, for large enough computational problems, the amount of inherently serial work is often negligible. The obstacle to perfect scalability thus lies with different forms of parallelization overhead.

In addition to the already mentioned overhead due to data transfers, there are other types of overhead that can be associated with parallel computations:

- Parallel algorithms may incur extra calculations that are not relevant for the original serial computational problems. Data decomposition, such as finding out the index range of a decomposed segment of a vector, typically requires such extra calculations.
- Synchronization is often needed between computational tasks. A simple example can be found in the parallel reduction operation, where all pairs of workers have to complete, before proceeding to the next stage.
- Sometimes, in order to avoid data transfers, duplicated computations may be adopted between neighbors.
- In case of load imbalance, either because the target computational problem is impossible to be decomposed evenly, or because an ideal decomposition is too costly to compute, some hardware units may from time to time stay idle while waiting for the others.

It should be mentioned that there are also factors that may be scalability friendly. First, many parallel systems have the capability of carrying out communications at the same time of computations. This gives the possibility of hiding the communication overhead. However, to enable communication-computation overlap can be a challenging programming task. Second, it sometimes happens that by using many nodes of a distributed-memory system, the subproblem per node falls beneath a certain threshold size, thus suddenly giving rise to a much better utilization of the local caches.

References

[1] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2007.

[2] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[3] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI*. MIT Press, 2nd edition, 1999.

[4] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[5] Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg. *OpenCL Programming Guide*. Addison-Wesley, 2011.

[6] Peter S. Pacheco. *Parallel Programming With MPI*. Morgan Kaufmann, 1997.

[7] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGrawHill, 2003.