[PD-01-014] GIS and Parallel Programming

Abstract

Programming is a sought after skill in GIS, but traditional programming (also called serial programming) only uses one processing core. Modern desktop computers, laptops, and even cellphones now have multiple processing cores, which can be used simultaneously to increase processing capabilities for a range of GIS applications. Parallel programming is a type of programming that involves using multiple processing cores simultaneously to solve a problem, which enables GIS applications to leverage more of the processing power on modern computing architectures ranging from desktop computers to supercomputers. Advanced parallel programming can leverage hundreds and thousands of cores on high-performance computing resources to process big spatial datasets or run complex spatial models.

Parallel programming is both a science and an art. While there are methods and principles that apply to parallel programming--when, how, and why certain methods are applied over others in a specific GIS application remains more of an art than a science. The following sections introduce the concept of parallel programming and discuss how to parallelize a spatial problem and measure parallel performance.

Keywords: Graphics Processing Units, High Performance Computing, Message Passing Interface (MPI) for GIS Applications

Author & citation

Shook, E. (2019). GIS and Parallel Programming. The Geographic Information Science & Technology Body of Knowledge (1st Quarter 2019 Edition), John P. Wilson (Ed.).
DOI: 10.22224/gistbok/2019.1.1

Explanation

1. Definitions
2. Introduction
3. How to Parallelize a Geospatial Problem
4. How to Coordinate Date and Execution in Parallel
5. How to Program in Parallel
6. How to Measure Parallel Performance

1. Definitions

- **Parallel programming**: A type of programming that involves using multiple processing cores simultaneously to solve a problem.
- **Core**: An independent processing unit embedded within a computer processor.
- **Parallel application**: An application programed to use multiple processing cores simultaneously to solve a problem or complete a computing task.
- **MPI: Message Passing Interface**. A de facto standard interface for distributed parallel programming.

2. Introduction

Parallel programming enables applications to leverage more processing power, which can have several benefits. A key benefit is faster execution time to complete a computing task. This can be important for large computing tasks that can take hours or even days. More processing power can also allow applications to solve problems that were previously impossible due to memory/computation constraints, thus opening up new ways to solve complex spatial problems (Armstrong, 2000). Alternatively, more processing power can enable a GIS scholar or practitioner to conduct more comprehensive analyses or run more simulations in the same amount of time as a non-parallel application. Parallel applications can leverage a range of different parallel processing architectures including multi-core processors, Graphics Processing Units (GPUs), or accelerators. As spatial data continue to grow in size and multi-core architectures grow in abundance, parallel programming will become increasingly important to process and analyze all these data.

3. How to Parallelize a Geospatial Problem

GIS developers write programs to help solve geospatial problems. There are two types of parallel programming commonly used in GIS applications: data parallelism and functional parallelism. Each type has advantages and disadvantages, which are discussed below. Both types can be used simultaneously to increase parallelism, which is referred to as hybrid parallelism.

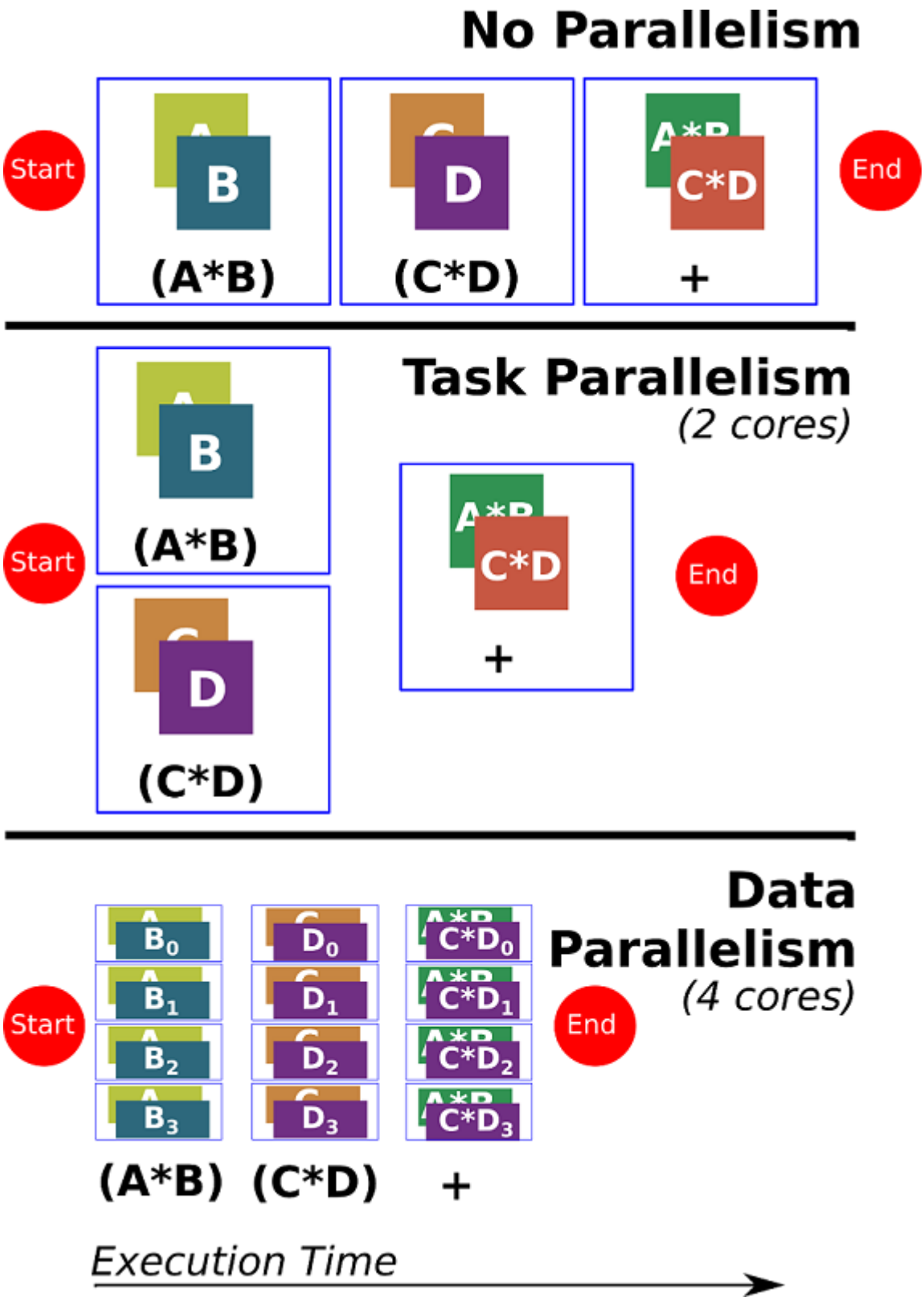


Figure 1. Comparison between parallel programming types to solve (A*B)+(C*D) for a raster calculator: no parallelism (top), functional parallelism (middle), and data parallelism (bottom).

Data parallelism involves partitioning spatial data (e.g., a set of vector features or raster pixels) into multiple smaller datasets. The partitioned data are distributed to multiple cores to be processed in parallel. This process is called spatial domain decomposition. The larger the datasets, the bigger the benefit of data parallelism. In cases where data are so large

that no single computer is capable of processing them, then data parallelism can be used to enable multiple computers to divide the data and process it in parallel. There are a multitude of ways that spatial data can be decomposed. Common decomposition strategies include row decomposition (oftentimes the easiest and most common), column decomposition, and grid decomposition (similar to a chessboard). Recursive partitioning strategies such as quadtree and recursive bi-section will divide spatially irregular data such as points, lines, and polygons into smaller and smaller partitions, careful to balance the amount of data in each subpartition. For further details readers are encouraged to look at (Ding & Densham, 1996).

Functional parallelism (sometimes called task parallelism) partitions a series of tasks and assigns tasks that can be completed simultaneously. This form of parallelism is most effective when there are many tasks. Examples include processing thousands of image tiles in remote sensing, running dozens of spatial models, or running hundreds of simulations in Monte Carlo-based spatial statistics. In each of these cases, there is little to no interaction between tasks so it is easy to partition the tasks across a number of processing cores. Functional parallelism can be easier to implement compared to data parallelism, because the data remain unchanged. However, it requires enough tasks to partition and distribute to multiple cores.

The differences between no parallelism, functional parallelism, and data parallelism are illustrated in a simple raster calculator example (Figure 1). In this example, four rasters are used to calculate (A*B) + (C*D). In the no parallel scenario, first the application calculates A*B, then it calculates C*D, and finally it adds the two resultant rasters to get the final answer. In the functional (or task) parallelism example, the two multiplications can be executed as separate tasks on two cores in parallel. The final addition task will be executed on a single core. Notice, since there are no more than two simultaneous tasks that only two cores can be used to parallelize the computation. In the data parallelism scenario, four cores (or even more cores) are used to process partitioned raster data (using row decomposition). Since raster data oftentimes has hundreds or thousands of rows that can be divided among dozens or even hundreds of cores, this scenario can use more than two cores. Each core processes a subdomain (a set of rows labeled 0-3 in Figure 1), first by calculating A*B, then C*D, and finally adding the resultant subdomain to get to the final answer which is merged into a single raster.

Some parallel programming languages and parallel strategies exist for GIS applications. For example, the Parallel Cartographic Modeling Language is a custom programming language that hides some of the complexities of parallel programming (Shook et al., 2016). Parallel strategies have been created to adapt parallel programs to different parallel computing platforms for GIS applications (Qin, Zhan, Zhu, & Zhou, 2014). Common platforms include not only multi-core processors, but also increasingly GPUs that are often programmed using the Compute Unified Device Architecture (CUDA), readers are referred to Graphics Processing Units.

4. How to Coordinate Data and Execution in Parallel

Once the problem is decomposed through data and/or task parallelism, a GIS application must coordinate sending, receiving, and processing data across all the cores in parallel.

Data are read from disk and stored in memory. There are two primary memory models for parallel computing: shared memory and distributed memory.

The shared memory model assumes that all processing cores have access to a single shared memory (Figure 2). The shared memory model provides an easy mechanism to share data, however multiple cores writing the same data can lead to race conditions, which result in inconsistent values. To resolve this challenge requires the use of locks and atomic operations. Locks are used to 'lock' a certain section of memory so that no other core can use it, and thus eliminate race conditions by insuring that another core cannot modify it while it is locked. Atomic operations ensure that the memory cannot be modified during the execution of the operation. In general, multi-threaded computing is the most common method to develop programs for shared memory model. In this case, a GIS application will launch multiple threads that all have access to a shared memory. For further details regarding shared memory, readers are referred to (Pacheco, 2011). Two widely used libraries for multi-threaded programming are the OpenMP (www.openmp.org) and OpenCV (www.opencv.org) packages.

Distributed memory is a different memory model in which each core has its own private memory that is not accessible to other cores (Figure 2). Data exchange in this model is explicit, because cores do not have access to each other's memory so applications need a way to exchange information. Message passing is the most common method in which cores send messages to each other containing data and/or coordinating information. The most common standard for message passing is the Message Passing Interface, which is available on every major supercomputer and cluster around the world due to its wide adoption (Pacheco, 2011).

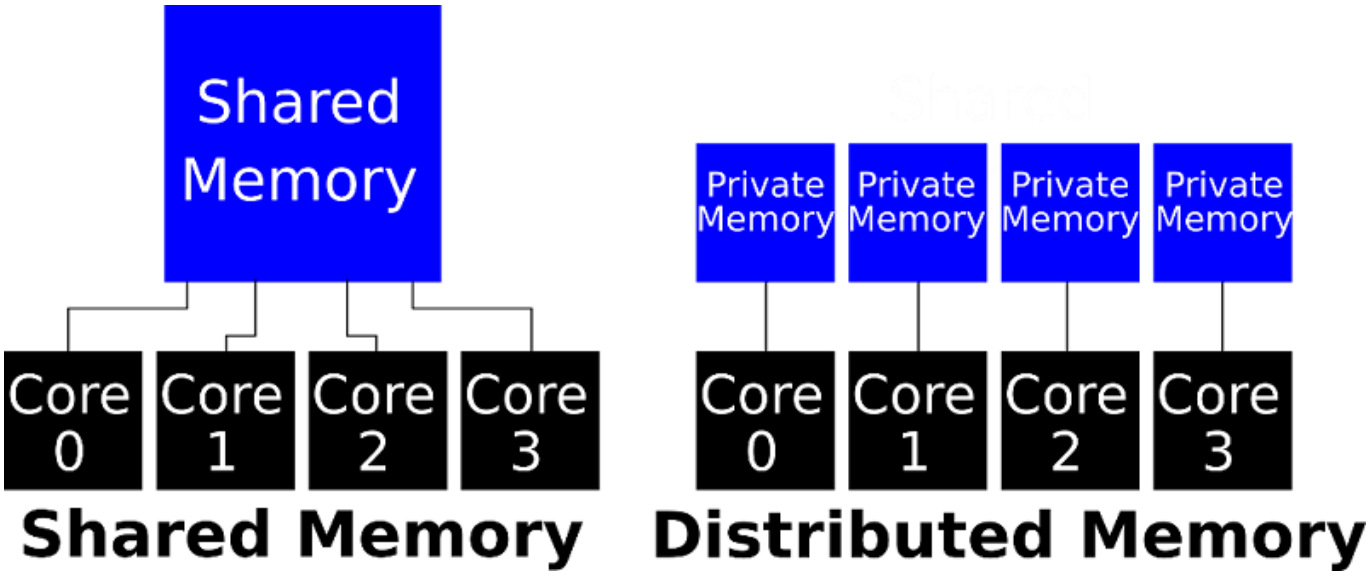


Figure 2. Comparison between two primary memory models: shared memory (left) and distributed memory (right).

5. How to Program in Parallel

There are two common ways to parallelize a program: master/slave and fully distributed. The master/slave paradigm is the most common way to parallelize a program, especially for beginners, because it is easy to conceptualize and implement. In this paradigm, a GIS application is the master application, and will launch a number of threads or processes, which act as slaves. The master process gives work to the slave processes and the results are returned to the master. Under this paradigm it is rare that slave processes communication to other slave processes, but rather all communication is channeled through the master process. This simplifies communication (called inter-processor communication), but can create a chokepoint (called a computational bottleneck), because one master can be overwhelmed by many slave processes. There are two key ways that slaves are assigned work: pulling from a shared queue or being assigned work from the master. If a shared queue of 'work' is available to the master and all slave processes, then when a slave is ready to complete a unit of work, they pull one unit of work off the queue and begin the computation. When it is complete they pull another unit of work. If the queue is empty then the slave quits, because there is no more work to do. The master is then responsible for filling up the work queue for all the slaves. The second scenario involves the slave making a 'work request' to the master, which then assigns a unit of work to the slave. In this scenario there is no shared information between the slave processes, but rather everything is communicated explicitly through the master process. These advantageous features (e.g., explicit communication and automatic workload balancing) make master/slave parallelization a good choice for all programmers from beginner to advanced.

Fully distributed paradigms are more common once a problem is sufficiently large and developers have gained parallel programming experience. In this paradigm there is no master, which can become a bottleneck when there are too many slaves (i.e., cores). Especially in cases where hundreds or thousands of processing cores are used in parallel, a single master can struggle to continue feeding tasks and data to all these cores. To resolve this problem, developers can create fully distributed codes in which all cores coordinate their activities in parallel through inter-processor communication (Shook, Wang, & Tang, 2013). Since there is no single master there is no single bottleneck. Careful thought is required to ensure that all the tasks are completed following the intended procedure and that the procedure itself is well designed otherwise issues such as data-overwrites and deadlock can occur. Such concurrency problems have been studied extensively in the area of Computer Science. A prime example is the Dining Philosopher's problem in which five philosophers sit in front of five bowls of spaghetti with a total of five forks, one between each philosopher. Philosophers can only eat their spaghetti when they hold both forks and they cannot speak to each other, the problem must be solved without allowing a philosopher to starve. This seemingly simple problem requires careful thought to solve and creating parallel programs that ensure no process (or philosopher) starves or becomes deadlocked can be challenging. One approach to simplifying this problem is to use synchronization in which processes wait for others to finish at a synchronization point, which can alleviate some of the problems of concurrency. Further, hunting down and fixing bugs (i.e., debugging parallel programs) can be difficult and time consuming, which is one reason why master/slave is preferred for beginning and advanced parallel programmers alike.

6. How to Measure Parallel Performance

Once you have a parallel program, how do you know whether it is performing well? Faster execution time is a good indicator, but is difficult to determine if your program is performing optimally using execution time alone. There are two primary ways to measure performance gains provided through parallelizing GIS applications: speedup and efficiency.

Speedup measures the execution time on N cores when compared to the time to run on a single core. Speedup is a better measure than execution time. While execution time generally goes down in well-designed parallel applications, it can be difficult to tell by execution time alone if the application is achieving excellent performance when given more and more processing cores. For example, a 38 second reduction in execution time when adding 16 cores may be considered excellent if the execution time on a single core was a few minutes, but may be considered poor if the execution time on a single core was several hours. This is largely due to the fact that shaving 38 seconds off of a two hour execution has limited impact on the overall performance especially when the application is given 16X more cores, but shaving 38 seconds off of a 43 second execution time does impact the overall performance. Speedup helps evaluate how much faster our application is running on a given number of cores (N).

Speedup = Execution time for serial program / Execution time for parallel program on N cores

To see how close an application is to theoretically optimal speedup we use the measure of efficiency, which takes speedup and divides it by N. It measures how close an application is to theoretically optimal speedup on a scale of 0-100%.

Efficiency = Speedup / N

Table 1 provides a hypothetical execution time of a parallel application and calculates speedup and efficiency. Notice how execution times are going down, which appears exciting, but there is a steep decrease in efficiency suggesting a reduction in parallel performance. It is tempting to think that adding X cores will achieve X speedup. However, this is never the case, which can be demonstrated using a classic parallel computing law.

Table 1. Hypothetical Execution Times, Speedup, and Efficiency for a Parallel Program

Number of Cores	Execution Time (seconds)	Speedup	Efficiency
1	100	1.0	100%
2	58	1.72	86%
4	36	2.77	69%
8	22	4.54	56%

Amdahl's Law calculates the theoretical speedup of a parallel application given a fixed problem size (Amdahl, 1967). It demonstrates why real applications can never truly achieve linear speedup, because parallel applications always have some portion of serial execution such as startup procedures, locks, and shutdown procedures. As parallelism increases, even the seemingly small portions of serial execution can become a larger portion of total execution time, which reduces overall parallel efficiency. Understanding the relationship between parallel portions and serial portions of a GIS application is crucial to estimating

actual speedup that can be achieved by a well-designed parallel program. For example, Amdahl's Law can help explain why master/slave paradigms can have scalability problems (the master process is a serial process that can become a significant portion of execution time) when compared to a fully distributed paradigm that has fewer serial portions.

With an understanding of speedup, efficiency, and the theoretical limits of parallelization through Amdahl's Law, how does one evaluate the parallel performance of a GIS application? There are two primary approaches: strong scaling and weak scaling. Strong scaling (illustrated in Table 1) evaluates the scalability of a parallel application for a fixed problem size. In other words, the problem size remains constant while the number of cores increase. Weak scaling evaluates the scalability of a parallel application for a problem size that grows with the number of cores. In other words, the problem size increases linearly as the number of cores increase. Strong scaling is generally the default evaluation criteria, because it demonstrates how fast a parallel application can solve a problem. But for massive problems (such as global satellite imagery analytics or streaming data problems) weak scaling can be a good evaluation criteria because it keeps the problem size per core constant, helping to evaluate how well a parallel application can scale to a massive spatial problem.

References

Amdahl, G. M. (1967). Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In Proceedings of the Spring Joint Computer Conference (pp. 483–485). New York, NY: ACM.

Armstrong, M. P. (2000). Geography and Computational Science. Annals of the Association of American Geographers, 90(1), 146–156.

Ding, Y., & Densham, P. J. (1996). Spatial strategies for parallel spatial modelling. International Journal of Geographical Information Systems, 10(6), 669-698.

Pacheco, P. (2011). An Introduction to Parallel Programming. Burlington, MA: Morgan Kaufmann.

Qin, C.-Z., Zhan, L.-J., Zhu, A.-X., & Zhou, & C.-H. (2014). A strategy for raster-based geocomputation under different parallel computing platforms. International Journal of Geographical Information Science, 28(11), 2127–2144.

Shook, E., Hodgson, M. E., Wang, S., Behzad, B., Soltani, K., Hiscox, A. and Ajayakumar, J. (2016). Parallel cartographic modeling: a methodology for parallelizing spatial data processing. International Journal of Geographical Information Science 30 (12):2355–2376.