AMS 148: Chapter 1: What is Parallelization

1 Introduction of Parallelism

Traditional codes and algorithms, much of what you, the student, have experienced previously has been computation in a **sequential** setting, by one **processor**. That is, each instruction is computed in series, one after the other. Take for example, an algorithm to compute the linear combination

$$\mathbf{y} = c\mathbf{x} + \mathbf{b}$$

where c is a constant, $\mathbf{y}, \mathbf{x}$, and $\mathbf{b}$ are vectors of size n.

Algorithm 1: A simple sequential add/multiply

Data: $\mathbf{x}, \mathbf{b}, c$
Result: $\mathbf{y}$
1 **for** $i = 0 \to n - 1$ **do**
2 | temp $= cx_i$;
3 | $y_i =$ temp $+ b_i$;
4 **end**

In algorithm 1, the computer reads in the vectors $\mathbf{x}, \mathbf{b}$ and the constant c. Then, as the index i is run from 0 to $n - 1$, a temporary variable used to store in multiplication of the ith entry of $\mathbf{x}$ multiplied to c. After this, the temporary variable is added to the ith entry of $\mathbf{b}$ and stored in the ith entry of $\mathbf{y}$. This is purely sequential algorithm, and does n mulitplications and n additions.

Most modern computers have processors with multiple **cores** in them for the running of applications. For example, my computer has an Intel i7-4790K, which is a Quad-Core processor. This means that the processor can execute as if it were four different processors. We could run four programs in unison or use the cores for one application. In this class we will be doing the latter.

Some computers or compute systems have multiple processors in them, that could have multiple cores on the processor. This type of architecture is common on traditional supercomputers. We will discuss the difference between a core and processor in a bit more detail in future chapters. Now, we can cast algorithm 1 in **parallel**.

Algorithm 2: A simple parallel add/multiply

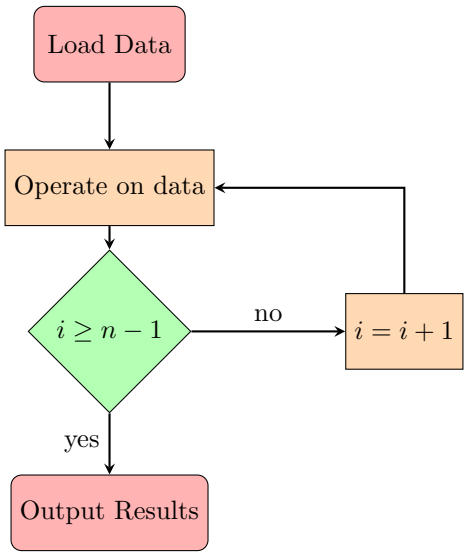
Data: i, x_i, b_i, c
Result: y_i
1 Thread i reads x_i, b_i, c;
2 **for** each thread **do**
3 | temp $= cx_i$;
4 | $y_i =$ temp $+ b_i$;
5 **end**
6 Thread i exports y_i;

Here, a **thread** is a sequence of programmed instructions handled by a processor or core, and depending on the architecture of the processor, a core may be able to handle one or a couple of threads simultaneously. Algorithm 2 is very similar in nature to algorithm 1, only instead of 1 thread doing the n multiplications and additions, we have n threads doing 1 multiplication and addition simultaneously. This is possible because

the operation is **embarassingly parallel**, i.e. each step is independent computationally and data-wise from the others. With an embarassingly parallel operation, the speed of the operation is essentially limited by the number of cores at the user's disposal. Not all algorithms will be able to parallelized in this way, as will be illustrated in the following sections. For now, we will treat, cores, processors, and threads the same, as computational devices to perform operations. However, later we will see that they are technically different.

We can examine sequence diagrams for a general sequential algorithm versus a general parallel algorithm. Figure 1 shows a generalized sequential loop that operates on data, then decides whether or not to stop based on an index counter. Algorithm 1 is a specific example of this type of process.

Figure 1: Sequence Diagram for Sequential Algorithm



Now figure 2 illustrates the instructions for a parallelized algorithm, requiring n steps, with m cores or threads, where m may be smaller than n.

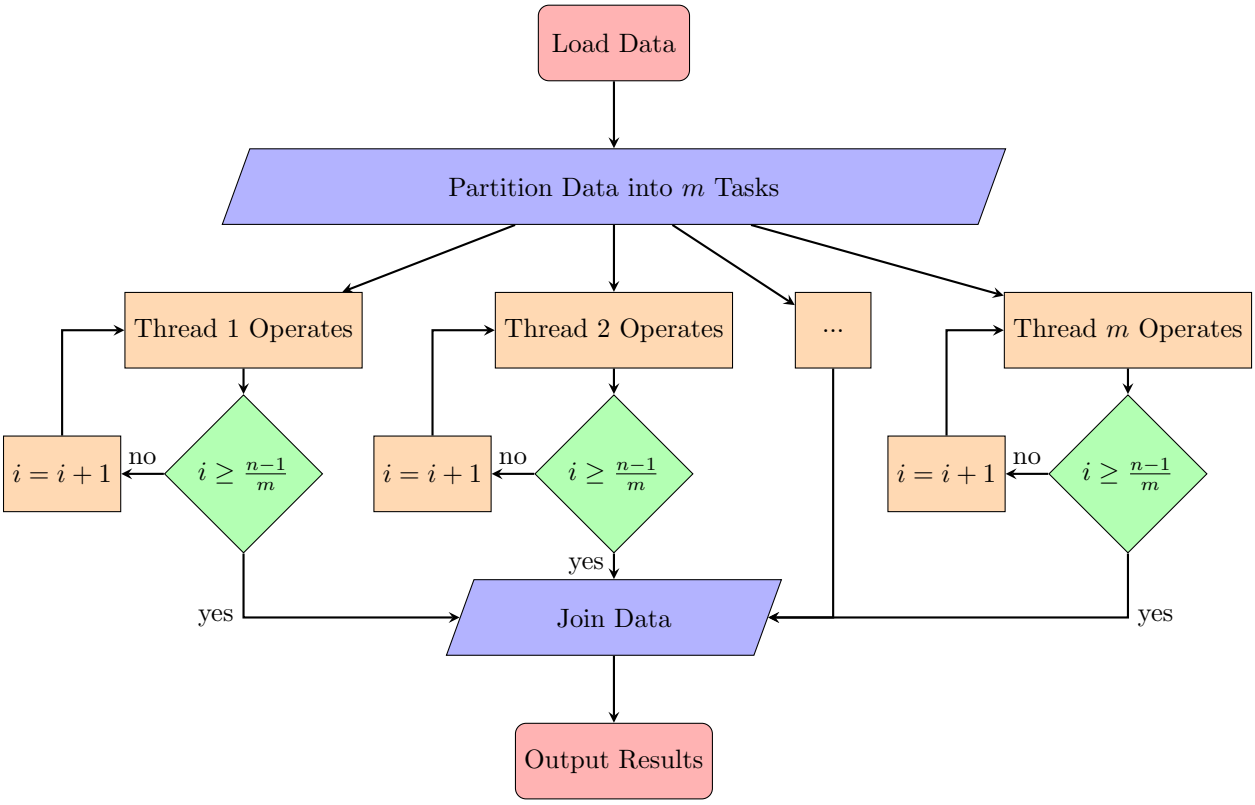


Figure 2: Sequence Diagram for Parallel Algorithm

Algorithm 2 is the simple case where the number of processors or cores match the ammount of data that needs to be processed. In that case each process is only doing one addition and multiplication. In most applications, industrial or academic, this will not be the case. Essentially in this model, the data is decomposed into m data sets, and each set is assigned to a processor or core. Then, each core executes serial style sequences until the all the data in its partion is processed. This type of parallelism is typical in large scale scientific computations, and is called **domain decomposition**. We'll come back to this concept later in the course.

1.1 Types of Parallelism

There are many types of parallelization: bit-level, instruction-level,and data-level parallelism. In this class we will be examining and practicing data-level parallelism.

1.1.1 Bit-Level

A computer processor has a limit to the size of word, i.e. the amount of information the processor can manipulate per cycle. Increasing the word size allowed for fewer instructions when manipulating a parcel of data whose bit size is larger than what the processor can handle. So increasing the word size of a processor allowed more data to be handled simultaneously by a processor.

1.1.2 Instruction-Level

A computer program is, in essence, a stream of instructions executed by a processor. Without instruction-level parallelism, a processor can only issue less than one instruction per clock cycle (IPC$<$ 1). These processors are known as subscalar processors. These instructions can be re-ordered and combined into groups which are then executed in parallel without changing the result of the program. This is known

as instruction-level parallelism. Advances in instruction-level parallelism dominated computer architecture from the mid-1980s until the mid-1990s.

All modern processors have multi-stage instruction pipelines. Each stage in the pipeline corresponds to a different action the processor performs on that instruction in that stage; a processor with an N-stage pipeline can have up to N different instructions at different stages of completion and thus can issue one instruction per clock cycle (IPC= 1). These processors are known as scalar processors. The canonical example of a pipelined processor is a RISC processor, with five stages: instruction fetch , instruction decode , execute , memory access , and register write back. The Pentium 4 processor had a 35-stage pipeline.

A canonical five-stage pipelined superscalar processor. In the best case scenario, it takes one clock cycle to complete two instructions and thus the processor can issue superscalar performance (IPC= $2 > 1$). Most modern processors also have multiple execution units. They usually combine this feature with pipelining and thus can issue more than one instruction per clock cycle (IPC> 1). These processors are known as superscalar processors. Instructions can be grouped together only if there is no data dependency between them.

1.2 Data-Level and Task-Level Parallism

Data-level parallelism involves implementing the same task on different partions or sets of data. Algorithm 2 is a good example of data-level parallelism.

Task-level parallelism is to deconstruct a program into tasks, and assign the tasks to different threads, cores, or processors. For example:

Algorithm 3: Task Parallel Program

```
1 if A TRUE then
2 |  Processor 1 Operates;
3 end
4 if B TRUE then
5 |  Processor 2 Operates;
6 end
```

The majority of this class will be exposing data parallelism for scientific, mathematical, and image processing purposes. Note that the skills in this class can apply to many other fields.

2 Parallelizing Serial Algorithms

In scientific computation, parallelizing algorithms is a very important skill. The first step is to look for **concurrency**. Concurrency is the attribute where many operations within an algorithm are data independent, and can be performed at the same time, by different processors or cores.

2.1 Example 1: Vector Addition

Vector addition with a scalar multiplication is a simple example to expose concurrency, since vector addition is done element by element. In algorithm 1, we do the addition and multiplication on each element within a loop that was executed sequentially. However, since the each result does not rely on the previous (or next), the algorithm can be distributed. This results in algorithm 2. The next example is far less trivial.

2.2 Example 2: The "Modified" Gram-Schmidt Algorithm

Let us examine the Gram-Schmidt Algorithm for creating an orthonormal basis from a set of linearly independent set of vectors. Given a matrix $\mathbf{A}$ whose columns form a basis for the vector space $\mathbb{R}^N$. The Graham-Schmidt algorithm, takes the columns of $\mathbf{A}$ and creates a set of orthonormal vectors $\{\mathbf{q}_i\}$, which can be stored in a matrix $\mathbf{Q}$.

For example, suppose we have two linearly independent vectors $\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^2$, and we want to form an orthonormal basis using them. Then the process would be

$$\mathbf{u}_1 = \mathbf{v}_1$$

$$\mathbf{u}_2 = \mathbf{v}_2 - \frac{\langle \mathbf{u}_1, \mathbf{v}_2 \rangle}{\langle \mathbf{u}_1, \mathbf{u}_1 \rangle} \mathbf{u}_1$$

$$\mathbf{e}_1 = \frac{\mathbf{u}_1}{||\mathbf{u}_1||}; \qquad \mathbf{e}_2 = \frac{\mathbf{u}_2}{||\mathbf{u}_2||}$$

where $\langle \cdot, \cdot \rangle$ is the Euclidean inner product (dot product for real vectors), and $||\mathbf{v}|| = \langle \mathbf{v}, \mathbf{v} \rangle^{1/2}$.

Intuitively, the Gram-Schmidt process takes a set of vectors and subtracts out the essence of the other vectors from them. Literally, the next vector in the sequence has the projections onto the previous vectors subtracted out of it.

Figure 3 gives a good illustration of this example.

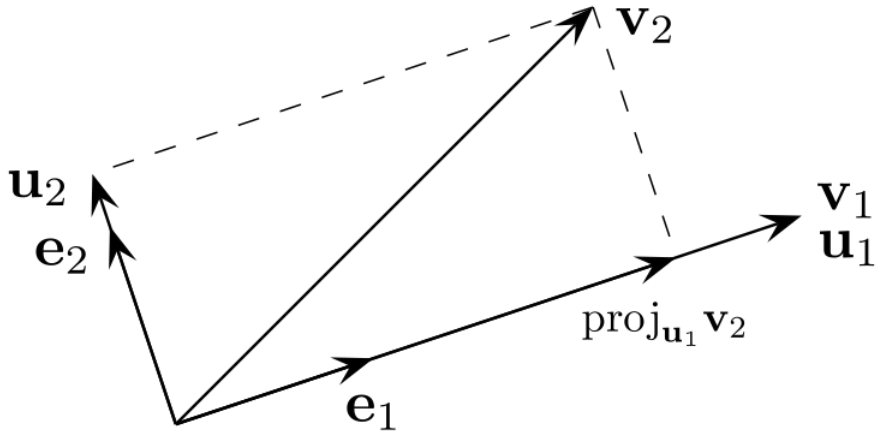


Figure 3: 2D Gram-Schmidt, Wikipedia

Using the definition of inner product, and norm it is straight foreward to write down a numerical algorithm for the Gram-Schmidt process. Algorithm 4 is the sequential version. Notice that this algorithm is already set up to generalize to do QR factorization, which is the main application of Gram-Schmidt. Also for stability purposes, the normalization takes place before the orthogonalization. For more details on why this is, consider taking AMS 213a, numerical linear algebra.

Algorithm 4: Sequential "Modified" Gram-Schmidt Algorithm

```
Data: A
Result: Q
for i = 0 → N − 1 do
    q_i = a_i;
    r_ii = ||q_i||;
    u_i = q_i / r_ii;
    for k = 0 → i do
        r_{i,k} = u_i^T q_k;
        q_k = q_k − r_{i,k} u_i;
    end
end
```

On the surface, this algorithm does not look to be easily paralizable. With a little bit of work we can illustrate the concurrency. From Linear Algebra we know that the Gram-Schmidt can be written as the

subtraction of the vector $\mathbf{a}_i$ and its projection on the vectors proceding it. That is,

$$\mathbf{q}_i = \mathbf{a}_i - \sum_{j=0}^{i-1} \text{proj}_{\mathbf{q}_k}(\mathbf{a}_i)$$

$$\mathbf{v}_i = \frac{\mathbf{q}_i}{||\mathbf{q}_i||}.$$

Figure 4 is an illustration of this process.

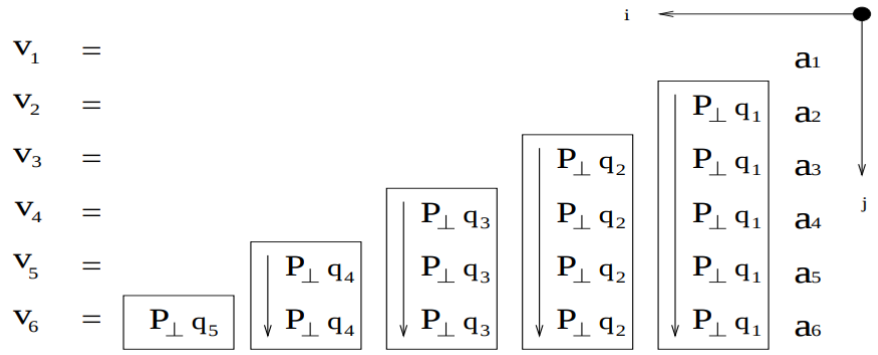


Figure 4: Visual Representation of Gram-Schmidt Orthogonalization

When the algorithm is constructed as in figure 4, concurrency is revealed. In the diagram $\mathbf{P}_\perp$ represents the projector operator. This is a good example of a parallizable algorithm that is not embarassingly parallel. We have exposed concurrency in the algorithm, but not in an a wide spread since. Each $\mathbf{q}_i$ needs to be computed in order, but we can still distribute the computation. Notice that each boxed column can be computed simultaneously, this is parallel portion of the algorithm. This translates to parallelizing the inner loop of of the algorithm, resulting in algorithm 5

Algorithm 5: Parallelized Gram-Schmidt Algorithm

Data: A
Result: Q
Using m threads:
for $i = 0 \rightarrow \text{int}((N-1)/m))$ **do**
  $id = processor_{id}\text{int}((N-1)/m)) + i$;
  $\mathbf{q}_{id} = \mathbf{a}_{id}$;
  $r_{ii} = ||\mathbf{q_i}||$;
  $\mathbf{u}_{id} = \mathbf{q}_{id}/r_{id,id}$;
  for $k = 0 \rightarrow id$ **do**
    $r_{id,k} = \mathbf{u}_{id}^T \mathbf{q}_k$;
    $\mathbf{q}_k = \mathbf{q}_k - r_{id,k}\mathbf{u}_{id}$;
  end
end

Notice that in algorithm 5 we a concept called **thread id**. This thread id is a number between 0 and $m-1$, and is used to separate the data that is controlled by each thread. This algorithm can be used to implement a parallel QR factorization for solution of $\mathbf{Ax} = \mathbf{b}$.

The Modified Gram-Schmidt algorithm is a good example of extracting parallelism from an application that appears to be inherently sequential.

2.3 Excercise List

These exercises are theoretical, no code should be turned in. However, you will be expected to write a psuedocode algorithm, much like what is in the notes.

2.3.1 Parallel Matrix-Vector Product

Show how to parallalize the matrix vector product $\mathbf{Ax}$, where $\mathbf{A}$ is an $M \times N$ matrix and $\mathbf{x}$ of size N. Assume you have k threads at your disposal, where $k \leq M$.

2.3.2 Parallel Matrix-Matrix Product

- Show how to parallelize matrix multiplication where both matrices $\mathbf{A}$ and $\mathbf{B}$ are square of size N. Also assume you have N threads to complete the computation. How many indices could be a parallelized index?

  Extra Credit: If you know a bit about threaded indexing, which index is the fastest for parallelization? Recall that in C/C++ memory transactions are row-major order.

- We know that the operation count for standard matrix multiplication is $\mathcal{O}(N^3)$. Suppose our N core processor has a clock speed of 1.4GHz and performs floating point operations (nevermind the precision at this time) at that speed. Estimate how long (an order of magnitude estimate) the parallel matrix multiplication above will take. *Hint*: the answer will be in terms of N.

  Suppose $N = 10000$, give an order of magnitude estimate for the speed of the parallel matrix multiplication. Realize that we are not taking into account data size, or how much data the processor can use each clock cycle. So this estimate will be faster than in reality.