DSC204A: Scaleable Data Systems Spring 2025

14: Parallelism

Instructor: Umesh Bellur *Scribe: Aryan Bansal*

1 Task Parallelism

Task parallelism is when we have the same piece of data, and multiple functions are applied to it in parallel.

1.1 Difference Between Parallelism and Concurrency

Concurrency is a property of the program, while parallelism is the execution method – concurrency refers to potential parallelism. Concurrency is driven (and limited) by ordering and sequence sharing. Parallelism is driven by the availability of resources.

1.2 Calculating Task Parallelism Speed-up

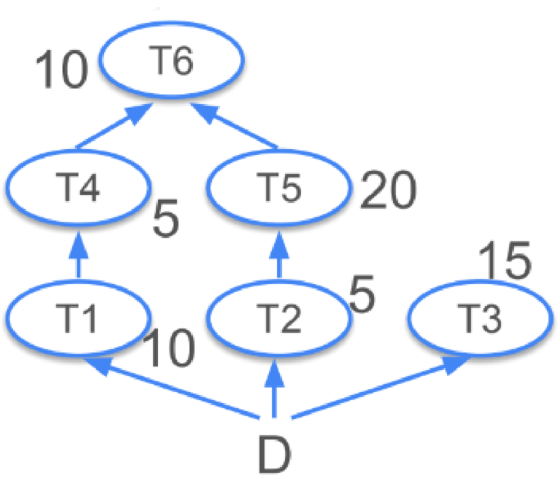


Figure 1: Enter Caption

Let's assume the above flow of work, given that we have 3 workers available. The completion time with 1 worker (all work done sequentially would be as follows):

$$\begin{aligned} T_{\text{sequential}} &= T_1 + T_2 + T_3 + T_4 + T_5 + T_6 \\ &= 10 + 5 + 15 + 5 + 20 + 10 \\ &= 65 \end{aligned}$$

Now, with task parallelism, we can schedule at most 3 different tasks in parallel, as long as those tasks don't have any dependencies. So, we start with T1, T2, and T3. When T1 finishes, we schedule T4. When T2 finishes, we schedule T5. We use the outputs of T4 and T5 to schedule T6. In parallel, the completion time would be the time taken by the longest path.

Here, we have 3 different path with lengths 25, 35, and 15. So,

$$T_{\text{parallel}} = 35$$

Now, speed-up is calculated as

$$\text{speed-up} = \frac{T_{\text{sequential}}}{T_{\text{parallel}}} = \frac{65}{35} = 1.9\times$$

Ideally, we should receive a speed-up of $3\times$, since we have thrice the number of workers. However, we see that not all workers are used at all times (worker 3 remains idle after finishing T3 because T4 and T5 require the outputs of T1 and T2, and similarly, T6 requires T4 and T5).

Every program has an inherently sequential part. The limiting factor to speed-ups is the longest sequential task.

1.3 Amdahl's Law

Amdahl's law places an upper bound on the speed-up achievable, given n workers. It assumes that a program has two parts: one that benefits from multi-core parallelism, and one that does not. The non-parallel portion could also be used for coordination.

The bound is given by:

$$\text{Speed-up} \leq \frac{1}{S + \frac{1-S}{n}}$$

where, S is the fraction of the program that is sequential
n is the number of workers used

If a part of the program is parallelizable, then we can apply Amdahl's law. If there are multiple possible parallel sequences, there is a divergence in the theoretical upper bound and the practically observed value. Amdahl's law is a conservative estimate.

In the problem shown in Section 2, the achieved speed-up is about $1.9\times$. The critical path takes 35 units of time to finish. So, $S = 35/65 = 0.538$. Substituting this, we get a speed-up bound closer to $1.45\times$ using Amdahl's law.

As shown in Fig. 2, the number of resources increase, the percentage of work done in the parallel region reduces, and performance is dominated by the sequential portion of the work. If we say that the green part is parallelizable, while the red part is not, the computation on the first core is dominated by the sequential part when we use 4 cores.

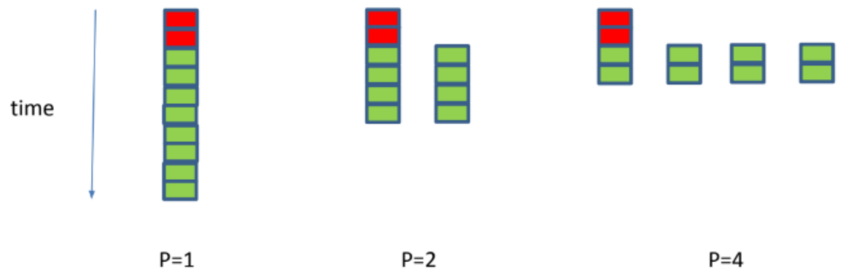


Figure 2: Additional Resources v/s Time Taken

If we compare the speed-ups for different number of processors, we see that the fraction of the program that can be parallelized makes a huge impact on the speed-up that we achieve. This helps us answer questions like given different amounts of resources, what is the maximum achievable speed-up.

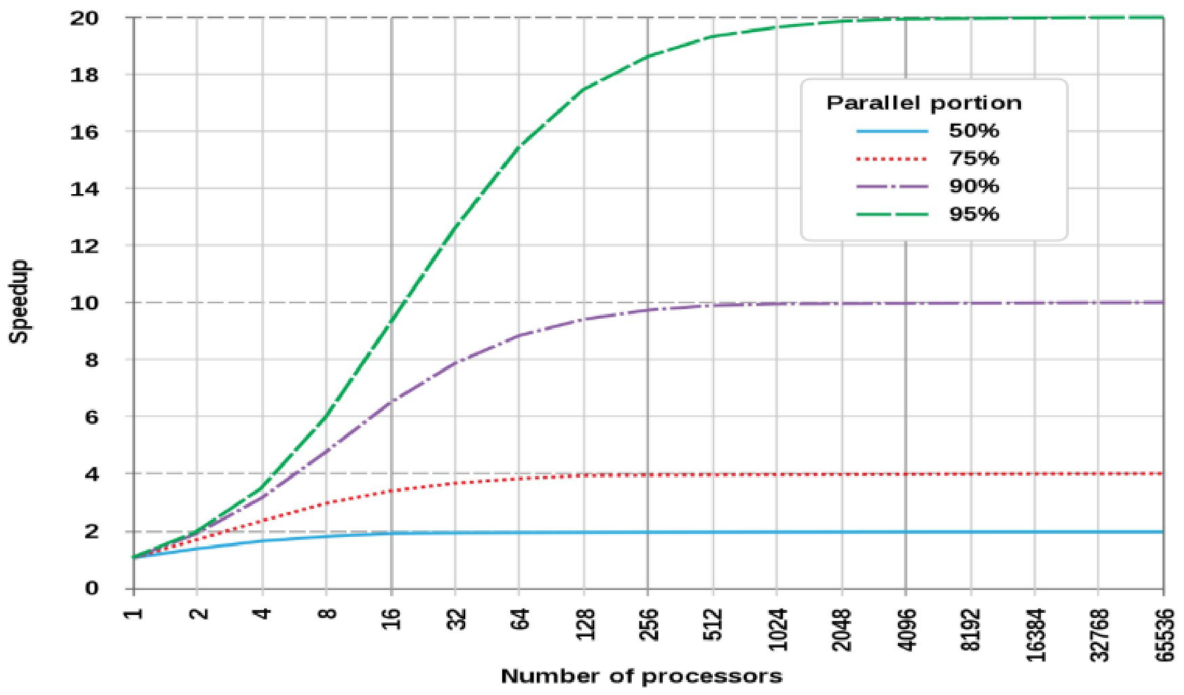


Figure 3: Number of Processors v/s Speed-up

1.3.1 Measuring Parallelism

A standard metric of parallelism is efficiency – it helps us measure how well-utilized the processors are while running the program in parallel.

$$\text{Efficiency} = \frac{\text{Speed-up(P)}}{P}$$

where, P is the number of processors used

Ideally, efficiency would be 100%, and speed-up scales linearly with each additional unit of resource. Idle times and sequential portions of the program reduce efficiency.

1.4 Task Parallelism with Ray

Ray is a framework for parallelizing tasks. Spark, Dask, Hadoop are other frameworks that can be used.

1. Tasks
   `@ray.remote` is the decorator used for task parallelism. It is used to mark functions that can be executed remotely (in parallel) across different workers. It allows users to fork a task that can be run somewhere else. A remote function is called asynchronously, and that returns a future object handle that can be used to fetch the result later. Futures are returned because we don't know when the function is going to finish running.

2. Futures
   A future represents the result of a task that is being computed asynchronously. It allows you to continue working while the task is executing and retrieve the result once it completes.

   `ray.get` is a blocking call which actually gets the result of the computation from the futures.

3. Actors
   Actors are stateful workers in Ray and can run tasks in parallel. They allow you to model objects with methods that can run asynchronously.

2 Data Parallelism

Data parallelism is a paradigm when we partition the data into multiple chunks and run one function on all the chunks in parallel. The results are collated and processed for calculating the correct transformation. Ideally, we would run both task and data parallelism. Data parallelism is especially beneficial when the data is very big.

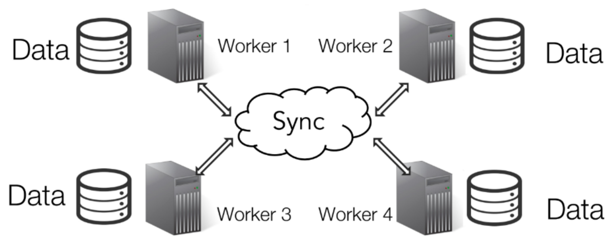


Figure 4: Data Parallelism

Modin uses Ray to parallelize the data.

Most computers today have multiple processors and multiple cores per processors, with a hierarchy of shared caches. So, parallelism essentially comes for free.

2.1 Replication v/s Partitioning

For a single machine, there is no sense in replicating the data. For distributed tasks, replication will make sense because accessing data over the network has significant costs. In case of partitions, only a few partitions are sent to each worker.

2.2 CPUs and GPUs

On a CPU with multiple cores, one can either run Single Program Multiple Data (SPMD), or Multiple Program Single Data (MPSD). In contrast, GPUs are superior when we have one program and very finely partitioned data, like images.

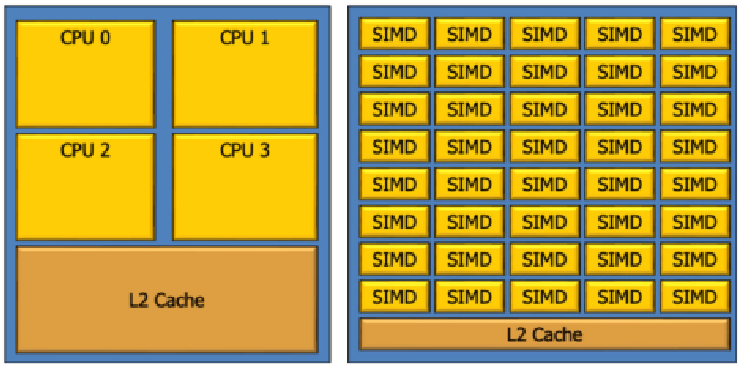


Figure 5: Typical CPU (left) v/s a typical GPU (right)

A CPU typically has 8-32 cores, with each core being a 3 GHz core. CPU cores' instruction sets are very powerful, and a single instruction can do a lot of work.

A GPU, like Nvidia's H100, has close to 15,000 CUDA cores. Each core is of 1 GHz, and the instruction set is much simpler – a CUDA core can do a few things, but it can do those things very well.

Processor performance is measured in Flops (floating point operations per second).

GPUs cannot execute complex instructions, and thus cannot be used everywhere.

2.2.1 Processing Flow

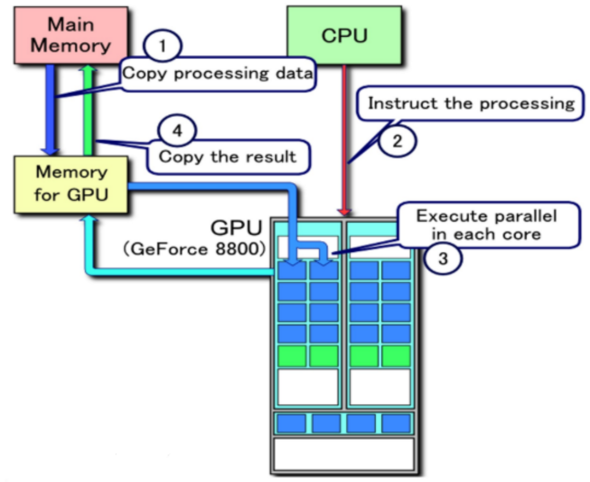


Figure 6: Flow of processing data on a GPU

When we run a program, it has to be started on the CPU, and is not started on GPU directly. The main entrypoint always starts on the CPU. The host performs some CPU computation, and then kicks off a CUDA kernel. All the data is copied into the GPU's memory (which is separate from RAM), from the CPU memory. Newer advancements allow GPUs to read directly from the disk. The GPU then executes the kernel to compute some results, which are then copied back to CPU memory.

For example, we could write a loop for vector addition on a CPU. For execution on a GPU, we would follow the steps below:

1. Define the kernel (function to compute the sum).

2. Initialize the input vectors and then allocate memory on the GPU. Multiple programs can be using the GPU simultaneously. In addition to allocating memory for the input vectors, we also reserve some memory for the result, so as not to overwrite the input vectors.

3. Configure the number of threads or cores to be used for the computation. If the size of the data is greater than the number of cores available, the GPU will "batch" the computation. For example, if the GPU has 15,000 cores, but the dimension of the vectors is 1M, then the addition will be done in batches of 15,000 dimensions at a time.

The loop gets eliminated as long as the number of cores available is greater than the dimension of the data. The trade-off here is the overhead in copying data from the CPU to the GPU, and back.

If we are doing two operations on an image (say, sharpening and denoising an image). The overhead could be minimized by not copying the data back to the CPU after the first result.

2.3 GPU Organization

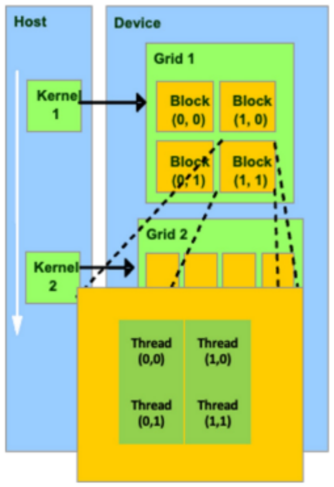


Figure 7: GPU Organization

GPUs are organized broadly into grids, with each grid having a 2-D array of blocks, and each block having multiple threads. A thread operates on a part of the data.

2.3.1 Matrix Multiplication on a GPU

Let's assume that we have two matrices, M and N, and that P is the matrix obtained by multiplying M and N. So, $P = M * N$.

Each cell of the resultant matrix is computed using one row of M and one column of N. Each thread computes one element of the resultant matrix, and this is where the real power of a GPU can be seen. Every thread loads a row of matrix M and a column of N, and then performs one multiple and addition for each pair of M and N elements.

The size of the matrix is limited by the number of threads allowed in a thread block, but this is configurable.

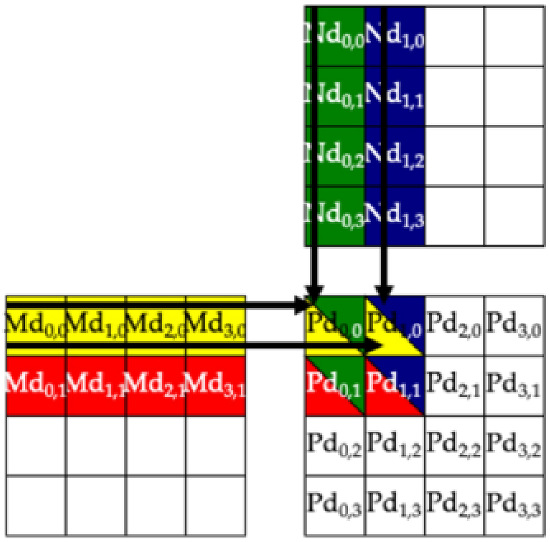
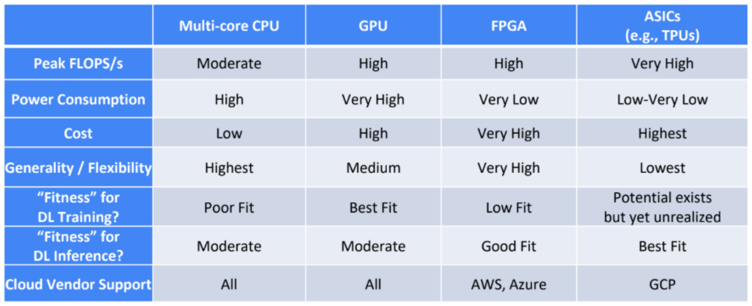

$Nd_{0,0}$ $Nd_{1,0}$
$Nd_{0,1}$ $Nd_{1,1}$
$Nd_{0,2}$ $Nd_{1,2}$
$Nd_{0,3}$ $Nd_{1,3}$
$Md_{0,0}$ $Md_{1,0}$ $Md_{2,0}$ $Md_{3,0}$
$Md_{0,1}$ $Md_{1,1}$ $Md_{2,1}$ $Md_{3,1}$
$Pd_{0,0}$ $Pd_{1,0}$ $Pd_{2,0}$ $Pd_{3,0}$
$Pd_{0,1}$ $Pd_{1,1}$ $Pd_{2,1}$ $Pd_{3,1}$
$Pd_{0,2}$ $Pd_{1,2}$ $Pd_{2,2}$ $Pd_{3,2}$
$Pd_{0,3}$ $Pd_{1,3}$ $Pd_{2,3}$ $Pd_{3,3}$

Figure 8: Matrix multiplication on a GPU

2.4 Alternatives to CPUs and GPUs



	Multi-core CPU	GPU	FPGA	ASICs (e.g., TPUs)
Peak FLOPS/s	Moderate	High	High	Very High
Power Consumption	High	Very High	Very Low	Low-Very Low
Cost	Low	High	Very High	Highest
Generality / Flexibility	Highest	Medium	Very High	Lowest
"Fitness" for DL Training?	Poor Fit	Best Fit	Low Fit	Potential exists but yet unrealized
"Fitness" for DL Inference?	Moderate	Moderate	Good Fit	Best Fit
Cloud Vendor Support	All	All	AWS, Azure	GCP

Figure 9: Comparing Modern Parallel Hardware

Tensor Processing Units (TPUs) are extremely efficient at tensor math. In the figure above, as we move from left to right, the hardware becomes more and more specialized for one particular application.

Running a sequential program on a GPU would be imprudent. FPGAs are custom hardware chips that are reprogrammable. ASICs on the other hand, cannot be reprogrammed.

Inferencing is not as highly parallelizable as training, since training is done with a very large volume of data. During training, we divide the data into batches, and in the extreme case, every piece of data can be trained on in parallel. This is why GPUs are better suited for training, while TPUs are better suited to inferencing.

2.5 Data Parallelism with Modin

Modin is a parallel DataFrame library built to speed up pandas by utilizing multiple cores.

Modin supports **automatic data partitioning**: it automatically partitions data when you create a Modin DataFrame. The data is split into smaller chunks (based on the available CPU cores), and operations are applied to these chunks in parallel. This allows you to work with large datasets while Modin automatically handles parallel execution under the hood, leveraging multiple cores for operations like filtering, aggregations, and joins.

Modin spawns Ray tasks while chunking the data to parallelize function calls.

2.6 How do Ray and Modin work together?

2.6.1 Ray for Task Parallelism

Ray handles task parallelism by distributing tasks across multiple workers. You can define remote functions using the `@ray.remote` decorator and execute them asynchronously using futures.

2.6.2 Modin for Data Parallelism

Modin handles data parallelism by automatically partitioning data and applying operations across available CPU cores. It performs operations on chunks of data in parallel, increasing the speed of computation.

2.6.3 Combining Moding and Ray

Moding itself uses Ray (or Dask) as the underlying parallel execution engine. So, when you run Modin operations, they are parallelized using Ray's task scheduling and worker management system.

You can also combine Ray's task parallelism with Modin's data parallelism to handle very large datasets efficiently and apply complex parallel operations across distributed resources.