Parallelism / Concurrency 6.S080 Lec 15 10/28/2019

What is the objective:

- Exploit multiple processors to perform computation
  faster by dividing it into subtasks
- Do something else while you wait for a slow task to complete

Overview:

This lecture is mostly about the first, but we'll talk briefly about the 2nd too.

Hardware perspective:

What does a modern machine look like inside, how many processors, etc.

Network architecture, bandwidth numbers, etc.

Ping test

Software perspective:

Single machine:

Multiple processes

Multiple threads

Python:

multithreading vs processing

Global interpreter lock lock

Although multi-threaded programs technically share memory, it's almost always a good idea to coordinate their activity through some kind of shared data structure. Typically this takes the form of a queue, or a lock

Example threading.py, threading_lock.py

<u>What is our objective</u>

Performance metric: speed up vs scale up

Scale up = N times larger task, N times more processors

goal: want runtime to remain the same

Speed Up = same size task, N times more processors

goal: want N times faster (show slide)

Impediments to scaling:

- Global interpreter lock -- switch to multiprocessing
  Show API slide
- Fixed costs -- takes some time to set up / tear down
- Task grain
  - tasks are too small, startup / coordination cost dominates
- Lack of scalability - some tasks cannot be parallelized, e..g, due to contention or lack of a parallel algorithm
- Skew - some processors have more work to do than others

test_queue.py (show slide with walkthrough of approach, then show code)

Show speedup slide

Why isn't it as good as we would like? (Not enough processors!)

Ok, so we can write parallel code like this ourselves, but these kinds of patterns are pretty common. This is a lot of work to rewrite a program to be parallel.

Fortunately, there are lots of good tools out there that can help you write efficient parallel data processing programs. We'll look at one popular one, called dask, and mention a few others.

Overall, our basic approach to parallelism is going to be to split a given data set split into N partitions, and use M processors to process this data in parallel.

Show parallel data flow slide

Going to introduce a few of them, but first let's talk about frequent operations you might want to do with data:

- Filter
- Project
- Element-wise or row-wise transform
- Join

- Repartition vs broadcast
- Aggregate
- Sort
- Train an ML model?
- User defined functions "UDF"s

Most of these are straightforward, i.e., just assign a processor to each partition -- no further coordination needed

"Embarrassingly parallel" -- Filter, project, element wise operations

Others still paralellizable, but have to do some extra work -- e.g., aggregation example above. We'll look at join in a bit. Lots of work on parallel sorting, parallel model building, etc -- not going to focus on in this class.

Some general challenges:

What if N >> M ?
- fine as long as each partition isn't too small (which would waste work)
- task grain issue

What if M >> N?
- can split some partitions

How to assign workers to tasks / stages
- if each task happens one at a time, straightforward, but trickier if we have multiple branches that can execute in parallel, or are pipelining data from one task to the next

Types of data partitioning
- randomly (or in whatever way it is given)

vs

- attribute based (e.g., on a range of values, or a date)

Attribute based can be good, because it can allow you to skip some partitions if you filter on that attribute, and can also help in joins, as we will see.

Parallel joins

- Show slide sequence

Parallel aggregations can be done similarly to shuffle -- each worker scans one

partition, computes partial aggregate, and then merge aggregates at the end

Common SQL aggregtes -- sum, count, etc -- are commutative and associative, so easy to merge

Postgres example?

Dask

Spark

Summary