Quantitative Big Imaging - Scaling up

Anders Kaestner

May 15, 2025

CONTENTS

This is the lecture notes for the 10th lecture of the Quantitative big imaging class given during the spring semester 2021 at ETH Zurich, Switzerland.

0.1 Scaling Up and Big Data

```
%load_ext autoreload
%autoreload 2
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

local_cluster = False
if local_cluster:
    from dask.distributed import Client, LocalCluster
    cluster = LocalCluster(n_workers=2, threads_per_worker=2)
    client = Client(cluster)
```

0.1.1 Literature / Useful References

Big Data

- Google's Presentation on Distributed Computing
  - Slides
- J. Dean et al. 2008, MapReduce: Simplified Data Processing on Large Clusters
- Scalable Systems Course
- Big Data for Engineers @ ETH
- Intro to Data Science @UCB

Cluster Computing

- Altintas, I. (2013). Workflow-driven programming paradigms for distributed analysis of biological big data. In 2013 IEEE 3rd International Conference on Computational Advances in Bio and medical Sciences (ICCABS)
- Condor High-throughput Computing
- Condor Setup at ITET (only on ETHZ network)
- Sun (now Oracle) Grid Engine
- Introduction to DASK

Databases

- Ollion, J., Cochennec, J., Loll, F., Escud, C., & Boudier, T. (2013). TANGO: a generic tool for high-throughput 3D image analysis for studying nuclear organization. Bioinformatics (Oxford, England), 29(14)

Cloud Computing

- Amazon AWS
- Sitaram, D., & Manjunath, G., 2012. Moving To The Cloud. null (Vol. null). Elsevier.
- Duan, P., Wang, W., Zhang, W., Gong, F., Zhang, P., & Rao, Y. (2013). Food Image Recognition Using Pervasive Cloud Computing. In 2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing (pp. 1631-1637). IEEE.

0.1.2 Outline

- Computer Science Principles
  - Parallelism
  - Distributed Computing
  - Programming models
- Organization
  - Queue Systems / Cluster Computing
  - Parameterization
  - Databases
- Big Data
  - MapReduce
  - Spark
  - Streaming
- Cloud Computing
- Beyond / The future

0.1.3 Motivation

There are three different types of problems that we will run into.

- Really big data sets
- Many data set
- Explorative studies

Really big data sets

- Several copies of the dataset need to be in memory for processing
  - Computers with more 256GB are expensive and difficult to find
  - Even they have 16 cores so still 16GB per CPU
  - Drive speed / network file access becomes a limiting factor
- If it crashes you **lose** everything
  - or you have to manually write a bunch of messy check-pointing code

Many datasets - some examples

- For genome-scale studies 1000s of samples need to be analyzed identically
- Dynamic experiments can have hundreds of measurements
- Animal phenotyping can have many huge data-sets (1000s of 328GB datasets)
- Radiologists in Switzerland alone make 1 Petabyte of scans per year

Exploratory Studies

Not sure what we are looking for

You need it to be

- Easy to develop new analyses
- Quick to test hypotheses

0.1.4 Example Projects

Zebra fish Full Animal Phenotyping

Full adult animal at cellular resolution 1000's of samples of full adult animals

- Imaged at 0.74 μm resolution:
- Images 11500 x 2800 x 628
- 20-40GVx / sample!

Objectives

- Identification of single cells (no down-sampling)
- Cell networks and connectivity
- Classification of cell type
- Registration with histology

0.1.5 Brain Project

Whole brain with cellular resolution
1 cm^3 scanned at 1 μm resolution
Images $\longrightarrow$ 1000 GVx / sample

- Registration separate scans together
- Blood vessel structure and networks
- Registration with fMRI, histology

0.1.6 What is wrong with usual approaches?

Inital workflow

Normally when problems are approached they are solved for a single task as quickly as possible

- I need to filter my image with a median filter with a neighborhood of 5 x 5 and a square kernel
- then make a threshold of 10
- label the components
- then count how many voxels are in each component
- save it to a file

```
im_in     = imread('test.jpg');
im_filter = medfilt2(im_in,[5,5]);
cl_img    = bwlabel(im_filter>10);
cl_count  = hist(cl_img,1:100);
dlmwrite(cl_count,'out.txt')
```

You want changes in the workflow

What if want to …

- compare Gaussian and Median?
- look at 3D instead of 2D images?
- run the same analysis for a folder of images?

You have to rewrite everything, everytime

If you start with a bad approach, it is very difficult to fix,

- big data and
- reproducibility

must be considered from the beginning

Preparing an image processing workflow for flexible upscaling to big data

The upscaling requires attention to scalability, modularity, and parallelizability.

- Design for Chunked and Streaming Processing
- Do not load entire datasets into memory.
- Use *lazy loading* and *chunked processing*
- Structure Code for Pipeline Reusability

Break your code into *small, stateless processing functions*.

- Use Parallel or Distributed Execution
- Data Format Considerations
- Use scalable formats that allow loading of chunks
- Ensure metadata is stored alongside the data for reproducibility.
- Containerization & Environment Reproducibility
- Use **Conda**, **Pixi**, or **Docker** to encapsulate dependencies.
- Track versions of packages and Python.
- Logging, Monitoring, and Fail-safety
- Log each pipeline step (`logging` or `prefect` for orchestration).
- Fail-safe processing: catch and log per-image errors.
- Use checkpointing in long pipelines.

0.1.7 Computer Science Principles

- Parallelism
- Distributed Computing
- Resource Contention
- Shared-Memory
- Race Conditions
- Synchronization
- Dead lock
- Imperative
- Declarative

Disclosure :

There are entire courses / PhD thesis's / Companies about this, so this is just a quick introduction

What is parallelism?

Parallelism is when you can:

- divide a task into separate pieces
- which can then be worked on at the same time.

An example

If you have to walk 5 minutes and talk on the phone for 5 minutes

Fig. 1: Walking and talking as serial and parallel tasks.

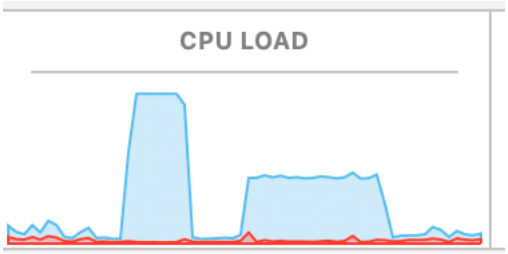
- you can perform the tasks serially which then takes 10 minutes
- you can perform the tasks in parallel which then takes 5 minutes

Some tasks are easy to parallelize while others are very difficult.

Rather than focusing on programming, real-life examples are good indicators of difficultly.

Accumulated CPU time

The accumulated CPU time is the sum of the time used by each CPU



- It can be done in shorter real-time with multiple CPUs.
- How much faster depends on the implementation.
  - *Advanced topic Scott Meyers: Cpu Caches and Why You Care*

In a parallel computing system, the time integral of CPU percentage, especially across multiple processors, is often referred to as “accumulated CPU time” or “total CPU time.” This metric represents the sum of the CPU time used by all processors or cores over a specific period. It gives a comprehensive view of the total computational effort expended by the system for a given task or during a specific time frame. This is useful for understanding overall system performance, load balancing, and resource utilization in parallel environments.

A more parallel median

Assume you have limited number of items like coins.

You want to know how many of each coin:

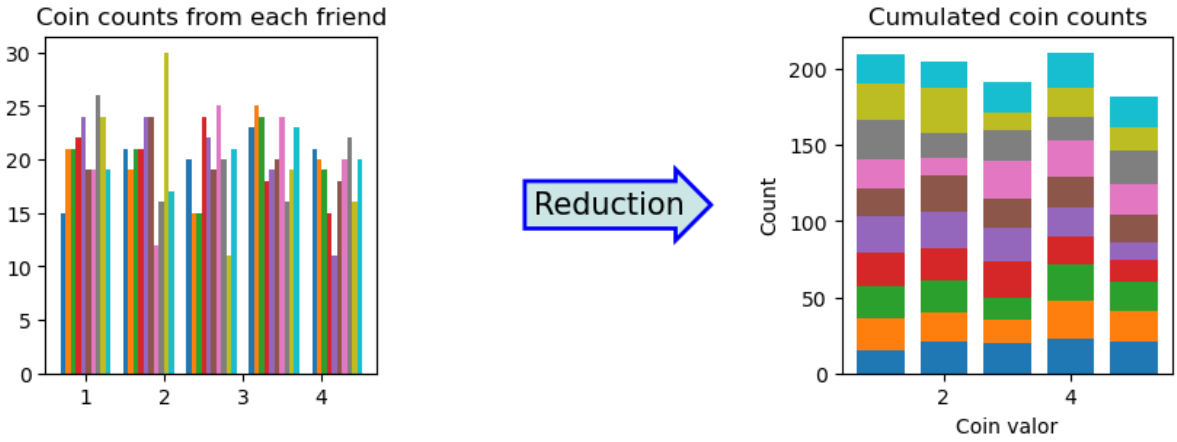
- Each friend makes a histogram of their coins
- Merge the histograms from each friend to get the median.

```
N=5
fig,ax=plt.subplots(1,3,figsize=(10,3))
friends_wallets = np.ceil(np.random.uniform(size=(100,10))*N)
h = ax[0].hist(friends_wallets,bins=5,align='left');
ax[0].set_title('Coin counts from each friend')
width = 0.75
lbls = np.unique(friends_wallets)

bottom = np.zeros(N)
ax[1].axis('off')
bbox_props = dict(boxstyle="rarrow", fc=(0.8, 0.9, 0.9), ec="b", lw=2)

t = ax[1].text(0.5,0.5, "Reduction", ha="center", va="center", rotation=0,
            size=15,
            bbox=bbox_props)
for count in h[0]:
    p = ax[2].bar(lbls,count, width, bottom=bottom)
    bottom += count

ax[2].set(title='Cumulated coin counts', xlabel='Coin valor', ylabel='Count');
```



0.1.8 Resource Contention

The largest issue with parallel / distributed tasks is the need to access the same resources at the same time

- memory / files
- pieces of information
- network resources

Challenges in parallel processing

1. Coordination

Parallel computing requires a significant of coordinating between computers for non-easily parallelizable tasks.

2. Mutability

The second major issue is mutability, if you have two cores / computers trying to write the same information at the same it is no longer deterministic (not good)

3. Blocking

The simple act of taking turns and waiting for every independent process to take its turn can completely negate the benefits of parallel computing

Dead-lock

Dead-lock is a phenomenon that may appear in parallel execution with systems of limited resources that require unique access to prevent collision and undefined behaviour. It means that two processes mutually prevent each others progress by locking a resource the other needs. A dead-lock is often illustrated using the dining philosophers problem which is outlined below.

Dining Philopher's Problem

- 5 philosophers at the table
- 5 forks
- Everyone needs two forks to eat
- Each philospher takes the fork on his left…

0.1.9 Parallel speedup and slowdown

Depend on:

- Available resources
- Implementation
- Scheduling
- Overhead

The speedup should ideally be linear with the number computing units.

```
nCPU      = np.linspace(1,8,8)
speedup   = np.array([1,2,3,4,5,6,7,8])
realspeedup = 0.9 * speedup + 0.1
realspeedup[5:] = np.array([5.2, 5.5, 5])

fig, ax1 = plt.subplots(figsize=(9,5))

ax1.set_xlabel('# CPUs')
```

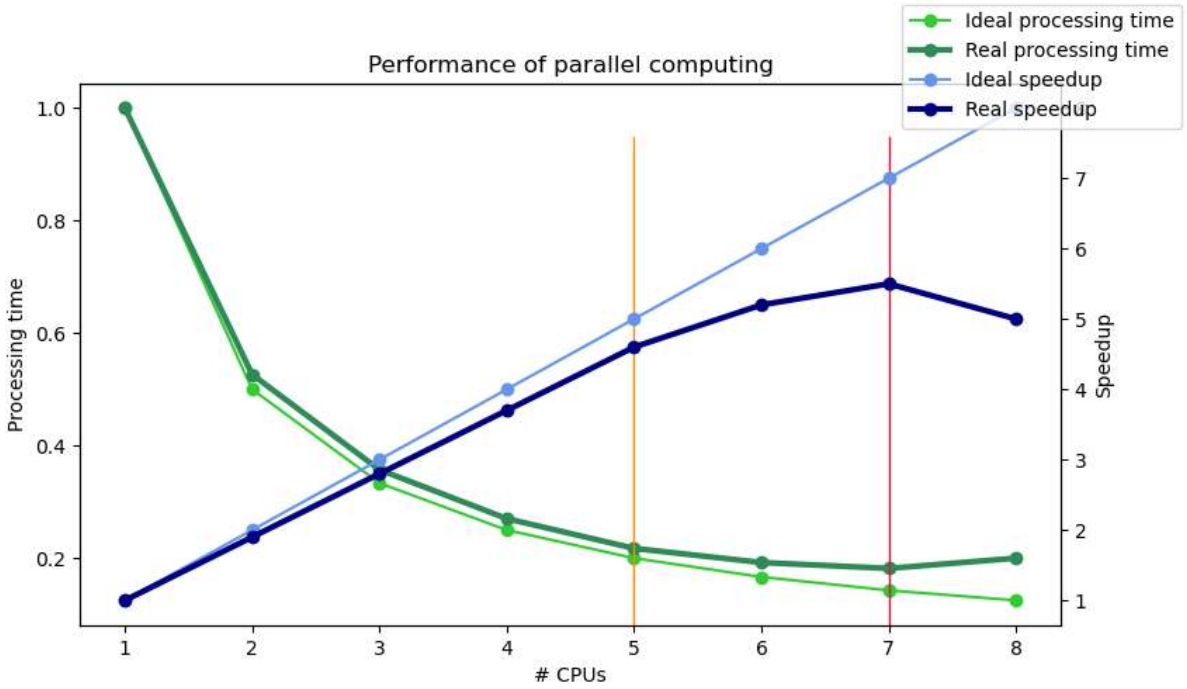
(continues on next page)

(continued from previous page)

```
ax1.set_ylabel('Processing time')
ax1.plot(nCPU, 1/speedup,marker="o", color="limegreen",label = 'Ideal processing time
 ↪' )
ax1.plot(nCPU, 1/realspeedup,marker="o",linewidth=3,color="seagreen", label = 'Real␣
 ↪processing time' )

ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis

ax2.set_ylabel('Speedup')  # we already handled the x-label with ax1
ax2.plot(nCPU, speedup, marker="o",color="cornflowerblue", label = 'Ideal speedup')
ax2.plot(nCPU, realspeedup, marker="o", linewidth=3,color="navy",label = 'Real speedup
 ↪')
ax1.axvline(x=5,linewidth=1, color='darkorange', ymax=0.9)
ax1.axvline(x=7,linewidth=1, color='crimson',ymax=0.9)
fig.legend();
ax1.set_title("Performance of parallel computing");
```



Real performance test run - Spotting hotspots

A time consuming component

Introduced parellel code

The two graphs show the execution time of a tomographic reconstruction as a system task including data IO and preprocessing. Splitting the timing per task in this way makes it possible to identify which part of the processing that consumes much processing time. In this example it appears that the single threaded implementation of the ring cleaning algorithm is takes a dominant fraction of the total processing time. This is therefore the target of tuning. There is a remarkable speed-up by distibuting the task on the available cores.

This example, there are components that are easy parallelize by altering the algorithm while others are harder. The yellow block labelled “other” mainly represents the time needed to read input data and to store the results. This brings us to take a closer look at Amdahls law.

Theoretical speed-up - Amdahl's law

The code can be divided into two parts:

- sequential
- parallelizable

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

p - time consumed by paralleizable code
s - number of cores

0.1.10 What is distributed computing?

Distributed computing is very similar to parallel computing, but a bit more particular.

- Parallel means you process many tasks at the same time,
- Distributed means you are no longer on the same CPU, process, or even on the same machine.

Implications of distributed computing

The distributed has some important implications since once you are no longer on the same machine the number of variables like

- Network delay,
- File system issues,
- and other users on the system

becomes a major problem.

Distributed Computing Examples

1. You have 10 friends who collectively know all the capital cities of the world.

- To find the capital of a single country you just yell the country and wait for someone to respond (+++)
- To find who knows the most countries, each, in turn, yells out how many countries they know and you select the highest (++)

2. Each friend has some money with them

- To find the total amount of money you tell each person to tell you how much money they have and you add it together (+)
- To find the **median** coin value, you ask each friend to tell you all the coins they have and you make one master list and then find the median coin (-)

0.1.11 Challenges in distributed processing

Inherits all of the problems of parallel programming with a whole variety of new issues.

Sending Instructions / Data Afar

Fault Tolerance

If you have 1000 computers working on solving a problem and one fails, you do not want your whole job to crash

Data Storage

How can you access and process data from many different computers quickly without very expensive infrastructure

0.2 Programming paradigms

0.2.1 Imperative Programming

Directly coordinating tasks on a computer.

- Languages like C, C++, Java, Matlab
- Exact orders are given (implicit time ordering)
- Data management is manually controlled
- Job and task scheduling is manual
- Potential to tweak and optimize performance

Making a soup (from lecture 1)

1. Buy vegetables at market
2. *then* Buy meat at butcher
3. *then* Chop carrots into pieces
4. *then* Chop potatos into pieces
5. *then* Heat water
6. *then* Wait until boiling then add chopped vegetables
7. *then* Wait 5 minutes and add meat

0.2.2 Declarative

- Languages like SQL, Erlang, Haskell, Scala, Python, R can be declarative
- Goals are stated rather than specific details
- Data is automatically managed and copied
- Scheduling is automatic but not always *efficient*

Making a soup (from lecture 1)

- Buy vegetables at market $\rightarrow shop_{veggies}$
- Buy meat at butcher $\rightarrow shop_{meat}$
- Wait for $shop_{veggies}$: Chop carrots into pieces $\rightarrow chopped_{carrots}$
- Wait for $shop_{veggies}$: Chop potatos into pieces $\rightarrow chopped_{potatos}$
- Heat water
- Wait for $boilingwater$,$chopped_{carrots}$,$chopped_{potatos}$: Add chopped vegetables
- Wait 5 minutes and add meat

0.2.3 Comparison

They look fairly similar, so what is the difference?

Imperative soup

Declarative soup

1. Buy {carrots, peas, tomatoes} at market
2. *then* Buy meat at butcher
3. *then* Chop carrots into pieces
4. *then* Chop potatos into pieces
5. *then* Heat water
6. *then* Wait until boiling then add chopped vegetables
7. *then* Wait 5 minutes and add meat

- Buy {carrots, peas, tomatoes} at market $\rightarrow shop_{veggies}$
- Buy meat at butcher $\rightarrow shop_{meat}$
- Wait for $shop_{veggies}$: Chop carrots into pieces $\rightarrow chopped_{carrots}$
- Wait for $shop_{veggies}$: Chop potatos into pieces $\rightarrow chopped_{potatos}$
- Heat water
- Wait for $boiling_{water}$,$chopped_{carrots}$,$chopped_{potatos}$: Add chopped vegetables
- Wait 5 minutes and ,$shop_{meat}$: add meat

The second is needlessly complicated for one person, but what if you have a team:

- how can several people make an imperative soup faster (chopping vegetables together?)
- How can many people make a declarative soup faster? Give everyone a different task (not completely efficient since some tasks have to wait on others)

0.2.4 Results

Imperative

- optimize specific tasks (chopping vegetables, mixing) so that many people can do it faster
  - Matlab/Python do this with fast-fourier-transforms (automatically uses many cores to compute faster)
- make many soups at the same time (independent)
  - This leads us to cluster-based computing

Declarative

- run everything at once
- each core (computer) takes a task and runs it
- execution order does not matter
  - wait for portions to be available (dependency)

An alternative - Lazy Evaluation

- do not run anything at all
- until something needs to be exported or saved
- run only the tasks that are needed for the final result
  - never buy tomatoes since they are not in the final soup

0.3 Organization

One of the major challenges of scaling up experiments and analysis is keeping all of the results organized in a clear manner.

As we have seen in the last lectures, many of the results produced many text files

- many files are difficult to organize
- Python, Matlab, and R are designed for in-memory computation
- Datasets can have many parameters and be complicated
- Transitioning from Excel to Python, Matlab, or R means rewriting everything

0.3.1 Queue Computing

Queue processing systems (like Sun Grid Engine, Oracle Grid Engine, Apple XGrid, Condor, etc) are used to manage

Resources

Computers, memory, storage

- a collection of processors (CPU and GPU)
- memory, local storage
- access to bandwidth or special resource like a printer
- for a given period of time

Jobs

Tasks to be run

- specific task to run
- necessary (minimal/maximal) resources to run with
- including execution time

Users

People who submit jobs

- accounts submitting jobs
- it can be undesirable for one user to dominate all of the resources all the time

Based on a set of rules for how to share the resources to the users to run tasks.

0.3.2 Structure of Cluster

Head node (master/login)

Manages the cluster and coordinates tasks. This is the main address when you access the cluster

It is responsible for tasks like

- Job scheduling
- User access
- File system coordination

It often runs the **scheduler**, e.g., SLURM, PBS, or a Dask scheduler.

Compute Nodes

These are the machines that actually perform the work.

Each node typically has:

- CPU(s), RAM, storage
- Sometimes GPU(s) or accelerators

May be homogeneous (identical) or heterogeneous (different roles/resources)

Scheduler

Allocates jobs and manages resources (CPUs, memory, GPUs).

The actual process that decides which jobs will run using which resources (worker nodes, memory, bandwidth) and at which time.

Networking

Interconnects all nodes for communication and data transfer.

Shared Storage

A common file system accessible to all nodes.

The shared storage

- is used for input/output data, software, and intermediate results
- sometimes nodes also have fast local SSDs for temporary data to reduce bottlenecks.

0.4 Databases

A database is a collection of data stored in the format of tables:

- a number of columns (data categories in the table)
- and rows (stored items)

```
from IPython.display import display, Markdown
import pandas as pd
display(Markdown('### Animals\nHere we have an table of the animals measured in an␣
↪experiment and their weight'))
display(pd.DataFrame(dict(id=(1, 2, 3),
                          Weight=(100, 40, 80)
                          )))

display(Markdown(
    '### Cells\nThe cells is then an analysis looking at the cellular structures'))

display(pd.DataFrame(dict(
    Animal=( 1, 2, 3),
    Type=("Cancer", "Healthy", "Cancer"),
    Anisotropy=(0.5, 1.0, 0.5),
    Volume=(1, 2, 0.95))))
```

Animals

Here we have an table of the animals measured in an experiment and their weight

```
   id  Weight
0   1     100
1   2      40
2   3      80
```

Cells

The cells is then an analysis looking at the cellular structures

```
   Animal     Type  Anisotropy  Volume
0       1   Cancer         0.5    1.00
1       2  Healthy         1.0    2.00
2       3   Cancer         0.5    0.95
```

0.4.1 SQL

SQL (pronounced Sequel) stands for **S** tructured **Q** uery **L** anguage and is *nearly* universal for both

- searching (called querying)
- and adding (called inserting) data into databases.

SQL is used in various forms from

- Firefox storing its preferences locally (using SQLite)
- to Facebook storing some of its user information (MySQL and Hive).

So refering to the two tables we defined in the last entry, we can use SQL to get information about the tables independently of how they are stored (a single machine, a supercomputer, or in the cloud)

SQL - Basic queries

- Get the volume of all cells

```
SELECT Volume FROM Cells
```

$$\rightarrow [1, 2, 0.95]$$

- Get the average volume of all cancer cells

```
SELECT AVG(Volume) FROM Cells WHERE Type = "Cancer"
```

$$\rightarrow 0.975$$

We could have done these easily without SQL using e.g., Excel or a Pandas data frame in python.

More Advanced SQL

- Get the volume of all cells in heavy mice

```
SELECT Volume FROM Cells WHERE Animal IN
  (SELECT id FROM Animal WHERE Weight>80)
```

- Get weight and average cell volume for all mice

```
SELECT ATable.Weight,CTable.Volume FROM Animals as ATable
  INNER JOIN Cells as CTable on (ATable.id=CTable.Animal)
```

$$\rightarrow [1, 0.95]$$

0.4.2 Beyond SQL: NoSQL

Basic networks can be entered and queries using SQL but relatively simple sounding requests can get complicated very quickly

Network Analysis

If we try to store cells and their connections in a SQL database, we can handle millions of cells and connections easily in a structured manner. However trying to perform analysis is trickier

- *How many cells are within two connections of each cell*

```
SELECT id,COUNT(*) AS connection_count FROM Cells as CellsA
  INNER JOIN Network as NetA ON Where (id=NetA.id1)
  INNER JOIN Network as NetB ON Where (NetA.id2=NetB.id1)
```

This is *still* readable but becomes very cumbersome quickly and difficult to manage

NoSQL (Not Only SQL)

A new generation of database software which extends the functionality of SQL to allow for

- more scalability (MongoDB)
- or specificity for problems like networks or graphs called generally **Graph Databases**

0.5 Big Data

0.5.1 Definition

Velocity, Volume, Variety

When a ton of heterogeneous is coming in fast. We need:

- **Performant**
- **Scalable**
- **Flexible**

You are ready when...

- **scaling isn't scary**
  10X, 100X, 1000X is the same amount of effort
- **you are starving for enough data**
  Director of AMPLab said their rate limiting factor is always enough interesting data
- **no 'clicks' per sample**
  Everything is automated, no human interaction required during processing

0.5.2 A brief oversimplified story

Google ran into 'big data' and its associated problems years ago:

- Peta- and exabytes of websites to collect and make sense of.
- Google uses an algorithm called PageRank(tm) for evaluating the quality of websites.

They could have probably used existing tools if page rank were some magic program that could read and determine the quality of a site

```
for every_site_on_internet
  current_site.rank=secret_pagerank_function(current_site)
end
```

Just divide all the websites into a bunch of groups and have each computer run a group, **easy!**

PageRank

While the actual internals of PageRank are not public, the general idea is that sites are ranked based on how many sites link to them

```
for current_site in every_site_on_internet
  current_pagerank = new SecretPageRankObj(current_site);
  for other_site in every_site_on_internet
    if current_site is_linked_to other_site
      current_pagerank.add_site(other_site);
    end
  end
  current_site.rank=current_pagerank.rank();
end
```

Complexity $O(N^2)$

How do you divide this task?

- Maybe try and divide the sites up: english_sites, chinese_sites, …
  - Run pagerank and run them separately.
  - What happens when a chinese_site links to an english_site?
- Buy a really big, really fast computer?
  - On the most-powerful computer in the world, one loop would take months!

It gets better

What happens if one computer / hard-drive crashes?

- Have a backup computer replace it (A backup computer for every single system)
- With a few computers ok, with hundreds of thousands of computers?
- What if there is an earthquake and all the computers go down?

PageRank doesn't just count

- Uses the old rankings for that page
- Run pagerank many times until the ranks converge

Google's Solution: MapReduce (part of it)

some people claim to have had the idea before, Google is certainly the first to do it at scale

Several engineers at Google recognized common elements in many of the tasks being performed. They then proceeded to divide all tasks into two classes **Map** and **Reduce**

Map

Map is where a function is applied to every element in the list and the function depends only on exactly that element

$$\vec{L} = [1,2,3,4,5]$$

$$f(x) = x^2$$

$$map(f \rightarrow \vec{L}) = [1,4,9,16,25]$$

Reduce

Reduce is more complicated and involves aggregating a number of different elements and summarizing them.

For example the Σ function can be written as a reduce function $\$\vec{L} = [1,2,3,4,5]g(a,b) = a+b\$$

Reduce then applies the function to the first two elements, and then to the result of the first two with the third and so on until all the elements are done

$$reduce(f \rightarrow \vec{L}) = g(g(g(g(1,2),3),4),5)$$

or $\$reduce(f \rightarrow \vec{L}) = g(g(g(1,2),g(3,4),5)\$$

0.5.3 MapReduce

Google designed a framework for handling distributing and running such of jobs on clusters.

So for each job a dataset ($\vec{L}$), Map-task (f), a grouping, and Reduce-task (g) are specified

1. Partition input data ($\vec{L}$) into chunks across all machines in the cluster
2. Apply **Map** (f) to each element
3. Shuffle and Repartition or Group Data
4. Apply **Reduce** (g) to each group
5. Collect all of the results and write to disk

All of the steps in between can be written once in a robust, safe manner and then used for every task which can be described using this MapReduce paradigm.

These tasks $\langle \vec{L}, f(x), g(a,b) \rangle$ is refered to as a job.

0.5.4 Key-Value Pairs / Grouping

The initial job was very basic, for more complicated jobs, a new notion of Key-value (KV) pairs must be introduced.

A KV pair is made up of a key and value:

- A key must be comparable / hashable (a number, string, immutable list of numbers, etc) and is used for grouping data.
- The value is the associated information to this key.

0.5.5 Counting Words

Using MapReduce on a folder full of text-documents: $\vec{L} = [\text{"Info}\cdots\text{"}, \text{"Expenses}\cdots\text{"}, \cdots]$

Map

is then a function f which takes in a long string and returns a list of all of the words (text seperated by spaces) as key-value pairs with the value being the number of times that word appeared

```
f(x) = [(word,1) for word in  x.split(" ")]
```

Grouping is then performed by keys (group all words together)

Reduce

adds up the values for each word

Example - The Map-Reduce Workflow

```
L = ["cat dog car",
     "dog car dog"]
```

$$\downarrow \textbf{ Map } : f(x)$$

```
[("cat",1),("dog",1),("car",1),("dog",1),("car",1),("dog",1)]
```

$$\downarrow \text{ Shuffle / Group}$$

```
 "dog": (1,1,1)
 "car": (1,1)```
$$ \downarrow \textbf{ Reduce } : g(a,b) $$
```[("cat",1),("dog",3),("car",2)]```
```

### Word Count Example

Here we make a word count example using all the lines of Shakespeare's "A midsommer-night's dream"

```
import os
shake_path = 'data/shakespeare.txt'
with open(shake_path, 'r') as f:
    all_lines = f.readlines()

print(all_lines[:5])
```

```
["A MIDSUMMER-NIGHT'S DREAM\n", '\n', 'Now , fair Hippolyta , our nuptial hour \n',
 ↪ 'Draws on apace : four happy days bring in \n', 'Another moon ; but O !␣
 ↪methinks how slow \n']
```

Imperative / Serial Execution

Here we run the code in an imperative fashion one line at a time.

```
from tqdm.notebook import tqdm
from collections import defaultdict
import string
word_count = defaultdict(lambda: 0)  # default count is 0
for c_line in tqdm(all_lines):
    for c_word in c_line.lower().strip().split(' '):
        v_word = ''.join([c for c in c_word if c in string.ascii_lowercase])
        if len(v_word) > 0:
            word_count[v_word] += 1
```

```
  0%|          | 0/129107 [00:00<?, ?it/s]
```
```

Analysis results

```
print('Shakespeare used', len(word_count), 'different words')
print('Most frequent')
for w, count in sorted(word_count.items(), key=lambda x: -x[1])[:10]:
    print(w, '\t', count)

print('\nLeast frequent')
for w, count in sorted(word_count.items(), key=lambda x: x[1])[:10]:
    print(w, '\t', count)
```

```
Shakespeare used 26982 different words
Most frequent
the          26851
and          24077
i           20535
to           18561
of           16013
you           13856
a           13840
my           12282
that           10761
in           10537

Least frequent
midsummernights           1
wanes          1
newbent           1
solemnities           1
merriments          1
interchangd           1
lovetokens          1
prevailment           1
unhardend          1
filchd          1
```

0.5.6 MapReduce approach using Dask bags

Dask Bags implement operations like

- map,
- filter,
- fold,
- and groupby on collections of generic Python objects.

Execution on bags provide two benefits:

- Parallel:
  - data is split up,
  - allowing multiple cores or machines to execute in parallel
- Iterating:
  - data processes lazily,

- allowing smooth execution of larger-than-memory data, even on a single machine within a single partition

A support function for our Map-Reduce

Here we use the Map Reduce approach to divide the function up into Map and Reduce components

First we need a function to convert lines to words:

```
import doctest
import copy
import functools
# tests are very important for map reduce

def autotest(func):
    globs = copy.copy(globals())
    globs.update({func.__name__: func})
    doctest.run_docstring_examples(
        func, globs, verbose=True, name=func.__name__)
    return func
```

```
# map function
@autotest
def line_to_words(in_line):
    """
    Takes a single line and returns the words and counts
    >>> line_to_words("hi i am. bob .  ")
    ['hi', 'i', 'am', 'bob']
    """
    words = in_line.lower().strip().split(' ')
    v_words = [''.join([c for c in c_word if c in string.ascii_lowercase])
               for c_word in words]
    return [c_word for c_word in v_words if len(c_word) > 0]
```

```
Finding tests in line_to_words
Trying:
    line_to_words("hi i am. bob .  ")
Expecting:
    ['hi', 'i', 'am', 'bob']
ok
```

Build a bag for the word analysis

Building the bag is a declarative task where we only define the steps to run in the analysis. The analysis will only take place when the `compute`-function is called.

```
import dask.bag as dbag
line_bag = dbag.from_sequence(all_lines, partition_size=10000)
line_bag
```

```
dask.bag<from_sequence, npartitions=13>
```

```
map_output = line_bag.map(line_to_words).flatten()
map_output
```

```
dask.bag<flatten, npartitions=13>
```

```
# we cheat a bit for the reduce step
reduce_output = map_output.frequencies()
top10 = reduce_output.topk(10, lambda x: x[1])
bot10 = reduce_output.topk(10, lambda x:  -x[1])
```

Run the analysis

```
import dask.diagnostics as diag

workers = 10

with diag.ProgressBar(), diag.Profiler() as prof, diag.ResourceProfiler(0.5) as rprof:
    print('Top 10\n', top10.compute(num_workers=workers))
    print('Bottom 10\n', bot10.compute(num_workers=workers))
```

```
[                                        ] | 0% Completed | 137.04 us

[                                        ] | 0% Completed | 117.97 ms

[                                        ] | 0% Completed | 224.35 ms

[##                                      ] | 6% Completed | 329.84 ms

[##                                      ] | 6% Completed | 435.24 ms

[########################################] | 100% Completed | 540.82 ms

```

```
Top 10
 [('the', 26851), ('and', 24077), ('i', 20535), ('to', 18561), ('of', 16013), ('you
 ↪', 13856), ('a', 13840), ('my', 12282), ('that', 10761), ('in', 10537)]

[                                        ] | 0% Completed | 111.17 us

[                                        ] | 0% Completed | 146.11 ms

[                                        ] | 0% Completed | 251.11 ms

[##                                      ] | 6% Completed | 357.51 ms

[##################                      ] | 46% Completed | 463.19 ms

[########################################] | 100% Completed | 573.63 ms

```

```
Bottom 10
 [('midsummernights', 1), ('wanes', 1), ('newbent', 1), ('solemnities', 1), (
↪'merriments', 1), ('interchangd', 1), ('lovetokens', 1), ('prevailment', 1), (
↪'unhardend', 1), ('filchd', 1)]
```

Visualize parallel task distribution

```
diag.visualize([prof, rprof]);
```


```
diag.visualize([prof, rprof]);
```


0.6 Environments for distributed computing

0.6.1 Hadoop

Hadoop is the open-source version of MapReduce developed by Yahoo and released as an Apache project.

It provides underlying infrastructure and filesystem that handles storing and distributing data so each machine stores some of the data locally and processing jobs run where the data is stored.

- Non-local data is copied over the network.
- Storage is automatically expanded with processing power.
- It's how Amazon, Microsoft, Yahoo, Facebook, … deal with exabytes of data

0.6.2 Spark / Resilient Distributed Datasets

Technical Specifications

- Developed by the Algorithms, Machines, and People Lab at UC Berkeley in 2012
- General tool for all Directed Acyclical Graph (DAG) workflows
- Course-grained processing $\rightarrow$ simple operations applied to entire sets
  - Map, reduce, join, group by, fold, foreach, filter,…
- In-memory caching

Zaharia, M., et. al (2012). Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing

Practical Specification

- Distributed, parallel computing without **logistics**, libraries, or compiling
- Declarative rather than imperative
  - Apply operation f to each image / block
  - **NOT** tell computer 3 to wait for an image from computer 2 to and perform operation f and send it to computer 1
  - Even scheduling is handled automatically

- Results can be stored in memory, on disk, redundant or not

0.6.3 Dask

In the pure python ecosystem, there has been a recent development called Dask.

Dask aims to bring

- the fault-tolerant,
- robust distributed computing

to numerical python codes.

In particular the focus has been on taking libraries like numpy and scipy and making them run as easily as possible in a distributed setting.

We will use these for the examples but they can be applied equally well to Spark and Hadoop-like problems.

0.6.4 Directed acyclical graphs - DAGs

More general than the MapReduce structure is the idea of making directed acyclical graphs.

These are used in

- Spark,
- Dask for distributed computing
- and in Tensorflow and PyTorch

for massively parallel computing since it allows complex operations to be defined in a declarative way.

This allows them to be optimized later depending on the actual resources available (and re-executed if some of those resources crash).

Some DAG resources

- PyData Dask - https://dask.pydata.org/en/latest/
- Apache Spark - https://spark.apache.org/
- Spotify Luigi - https://github.com/spotify/luigi
- Airflow - https://airflow.apache.org/
- KNIME - https://www.knime.com/
- Google Tensorflow - https://www.tensorflow.org/
- Pytorch / Torch - http://pytorch.org/

0.6.5 Tensor Comprehensions

Facebook shows an example of why such representations are useful since they allow for the operations to be optimized later and massive performance improvements even for *fairly* basic operations.

0.6.6 DAG examples

A basic DAG

Create two 5x5 images and use a single chunk:

- image1 all elements = 0
- image2 all elements = 1

```
import dask.array as da
from dask.dot import dot_graph
image_1 = da.zeros((5, 5), chunks=(5, 5))
image_2 = da.ones((5, 5), chunks=(5, 5))
dot_graph(image_1.dask,rankdir="LR")
```

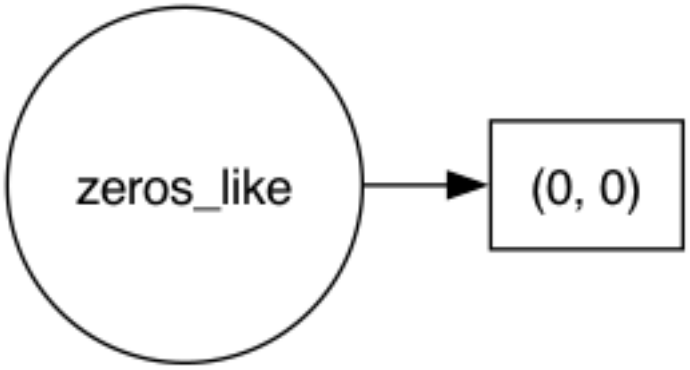
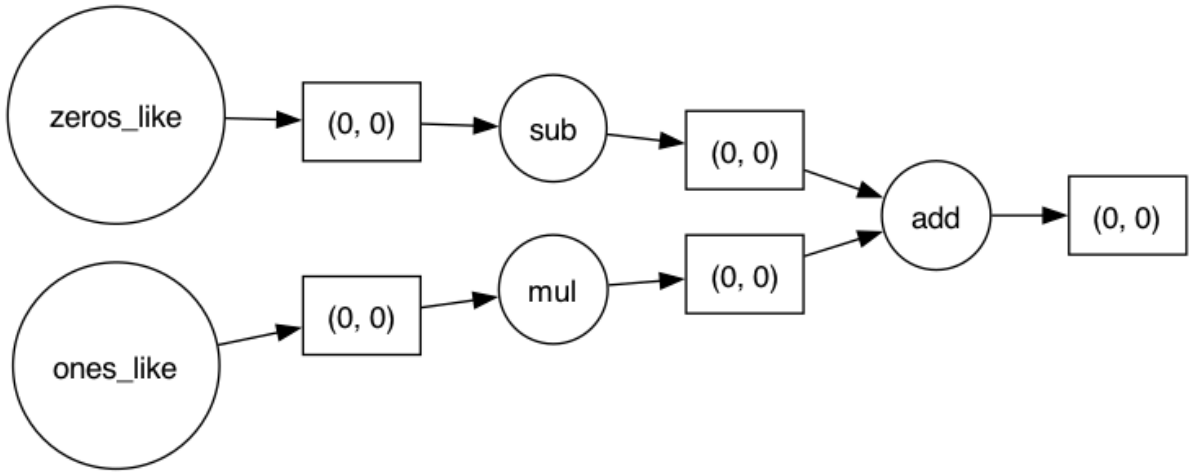


Image arithmetics

We want to compute: $$image_4 = (image_1 - 10) + (image_2 * 50)$$

```
image_4 = (image_1-10) + (image_2*50)
dot_graph(image_4.dask,rankdir="LR")
```



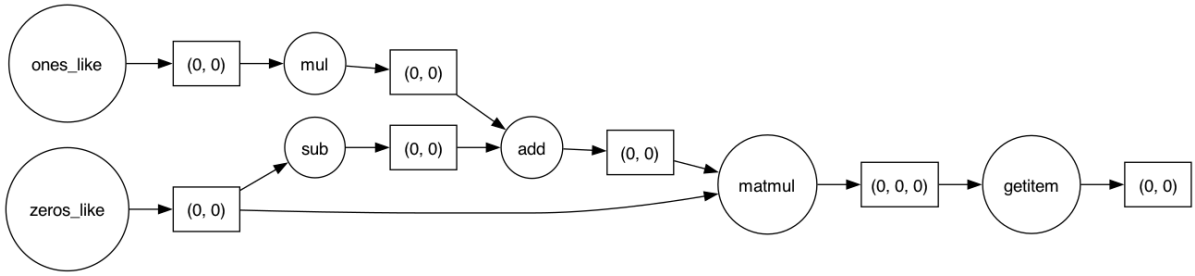
More calculations

$$image_5 = image_1 * image_4$$

Remember

$$image_4 = (image_1 - 10) + (image_2 * 50)$$

```
image_5 = da.matmul(image_1, image_4)
dot_graph(image_5.dask,rankdir="LR")
```



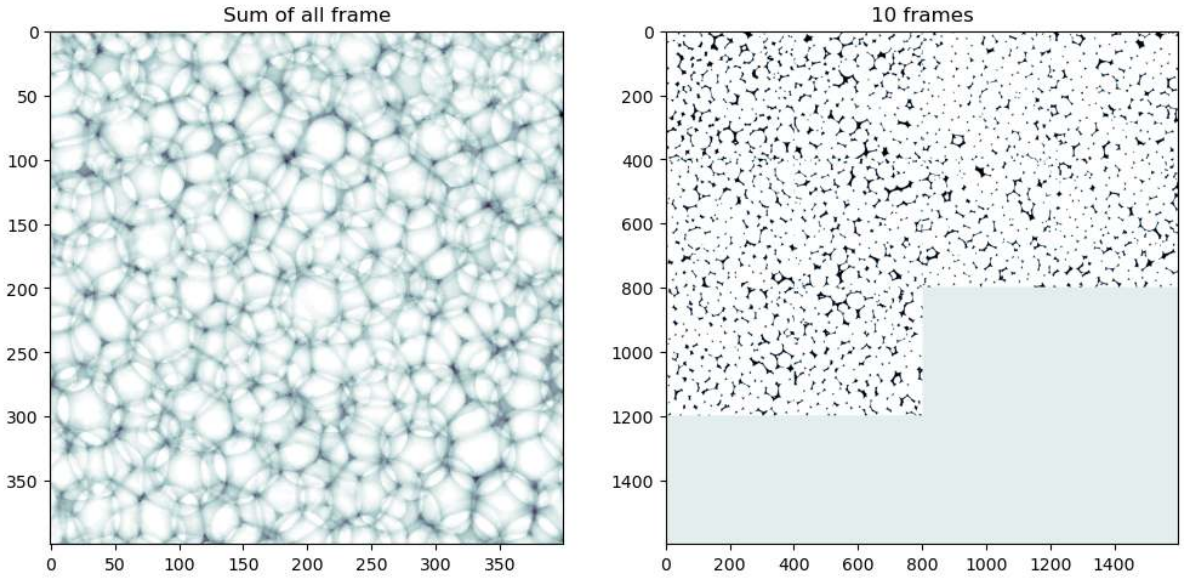
0.6.7 Image Processing using DAGs

the initial examples were shown on very simple image problems.

Here we can see how it looks for real imaging issues.

```
%matplotlib inline
import matplotlib.pyplot as plt
import dask.array     as da
from   dask.dot       import dot_graph
import numpy          as np
from skimage.io       import imread
from   skimage.util   import montage as montage2d
# for showing results
import dask.diagnostics as diag

foam_stack = imread('data/plateau_border.tif')[:-54, 52:-52, 52:-52]
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12,6))
ax1.imshow(np.sum(foam_stack, 0), cmap='bone_r'), ax1.set_title('Sum of all frame')
ax2.imshow(montage2d(foam_stack[::10]), cmap='bone_r'), ax2.set_title('10 frames');
```



Visualizing the data as 3D rendering

```
foam_stack.dtype
```

```
dtype('uint8')
```

```
# only seems to work with numpy==1.23.1
# pip install "pyvista[all]"
import pyvista as pv
img = foam_stack.astype(dtype=np.int32)
grid = pv.ImageData()

grid.dimensions = np.array(img.shape)
grid.origin  = (0, 0, 0)   # The bottom left corner of the data set
```

(continues on next page)

(continued from previous page)

```
grid.spacing = (1, 1, 1)  # These are the cell sizes along each axis, i.e., the pixel␣
↪size

grid.point_data["values"] = img.flatten(order="F")  # Flatten the array!
slices = grid.slice_orthogonal(x=img.shape[0]//2, z=img.shape[2]//2)
cpos = [
    (540.9115516905358, -617.1912234499737, 180.5084853429126),
    (128.31920055083387, 126.4977720785509, 111.77682599082095),
    (-0.1065160140819035, 0.032750075477590124, 0.9937714884722322),
]
dargs = dict(cmap='plasma')

p = pv.Plotter()
p.add_mesh(slices, **dargs)

p.show()
```

```
Widget(value='<iframe src="http://localhost:51783/index.html?ui=P_0x14caa0040_0&
↪reconnect=auto" class="pyvista…
```

```
import pyvista as pv

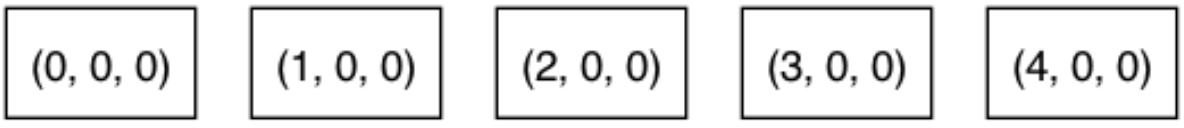
p1 = pv.Plotter()
va = p1.add_volume(foam_stack,cmap='viridis')
f = lambda val: va.GetProperty().SetScalarOpacityUnitDistance(val)
p1.add_slider_widget(f, [-0.1, 1], title="Opacity Distance")
p1.show()
```

```
Widget(value='<iframe src="http://localhost:51783/index.html?ui=P_0x14cf2aeb0_1&
↪reconnect=auto" class="pyvista…
```

0.6.8 Create a DAG with the foam image

- The image is 100x400x400
- We want 20 slices per chunk
- Scale intensities to from [0,255] to [0,1]

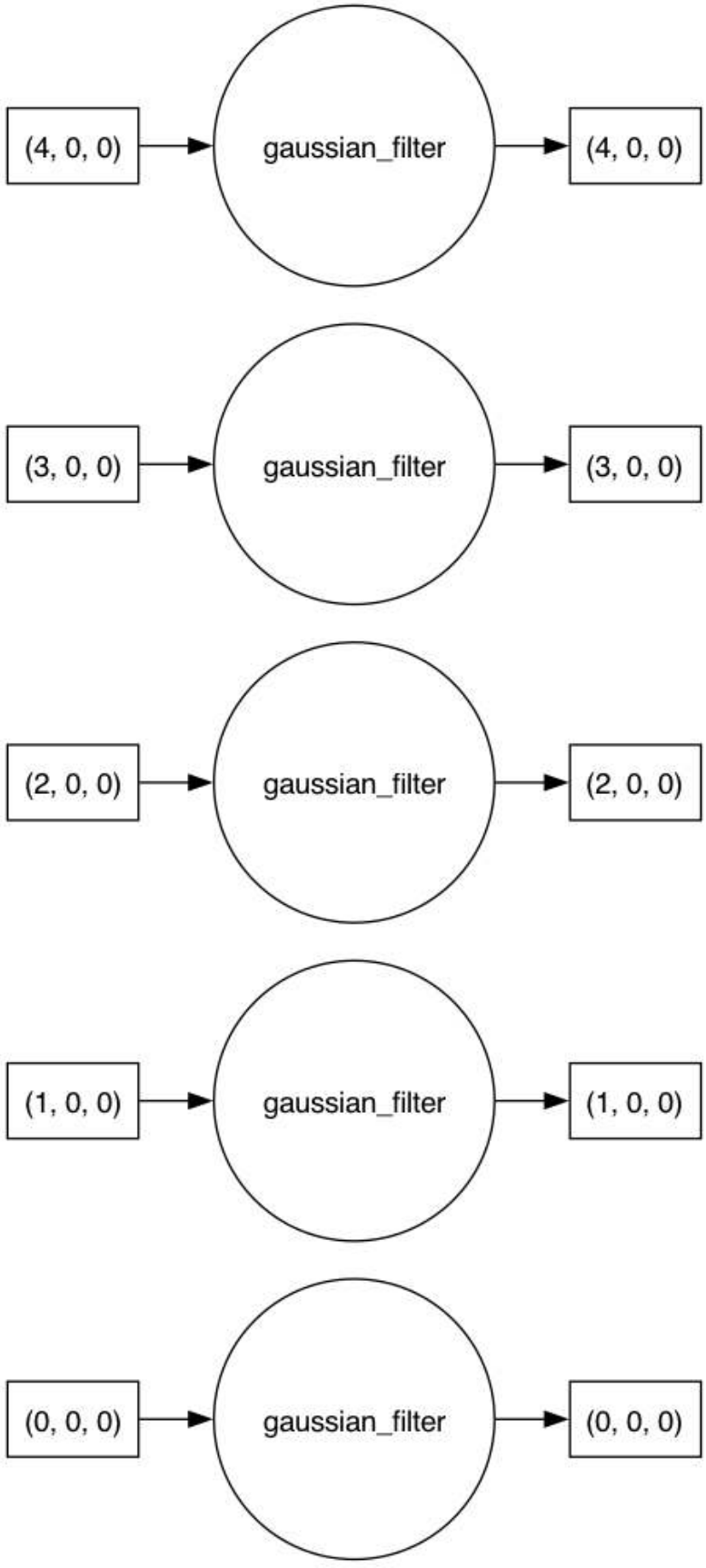
```
da_foam = da.from_array(
    foam_stack/255.0, chunks=(20, 400, 400), name='FoamImage')
dot_graph(da_foam.dask)
```



Add filter operation

- We want to use the Dask version of scipy.ndfilter.gaussian_filter

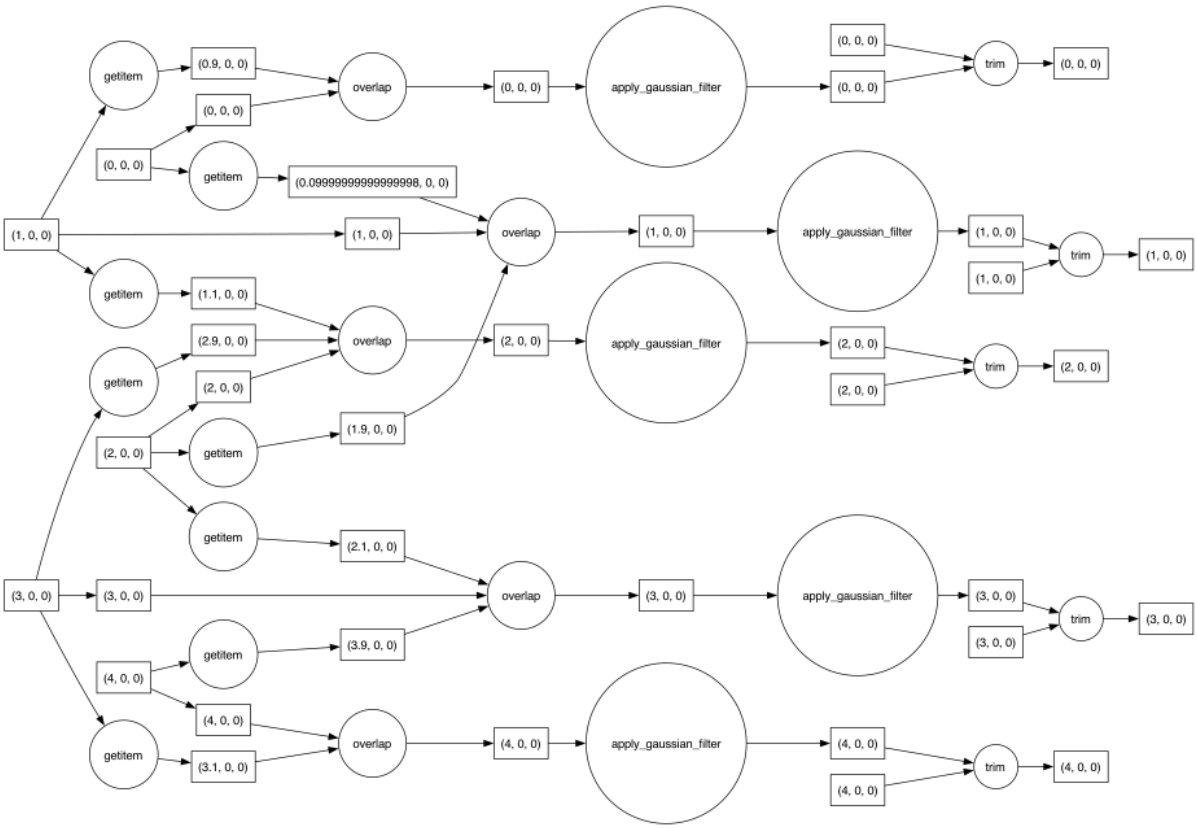
```
from scipy.ndimage import gaussian_filter
image_filt = da_foam.map_blocks(gaussian_filter, sigma=[3,6,6])
image_filt.visualize(rankdir="LR")
```


(4, 0, 0)
gaussian_filter
(4, 0, 0)
(3, 0, 0)
gaussian_filter
(3, 0, 0)
(2, 0, 0)
gaussian_filter
(2, 0, 0)
(1, 0, 0)
gaussian_filter
(1, 0, 0)
(0, 0, 0)
gaussian_filter
(0, 0, 0)

A better filter

```
def apply_gaussian_filter(block):
    return gaussian_filter(1-block, sigma=3)

# Apply the Gaussian filter to the Dask array
image_filt2 = da_foam.map_overlap(apply_gaussian_filter, depth=int(3*3), boundary=
↪'none')
image_filt2.visualize(rankdir="LR")
```



Why so complicated?

The filter needs to process the boundaries correctly

This requires an exchange of boundary data between the blocks.

Add segment and erode to the DAG

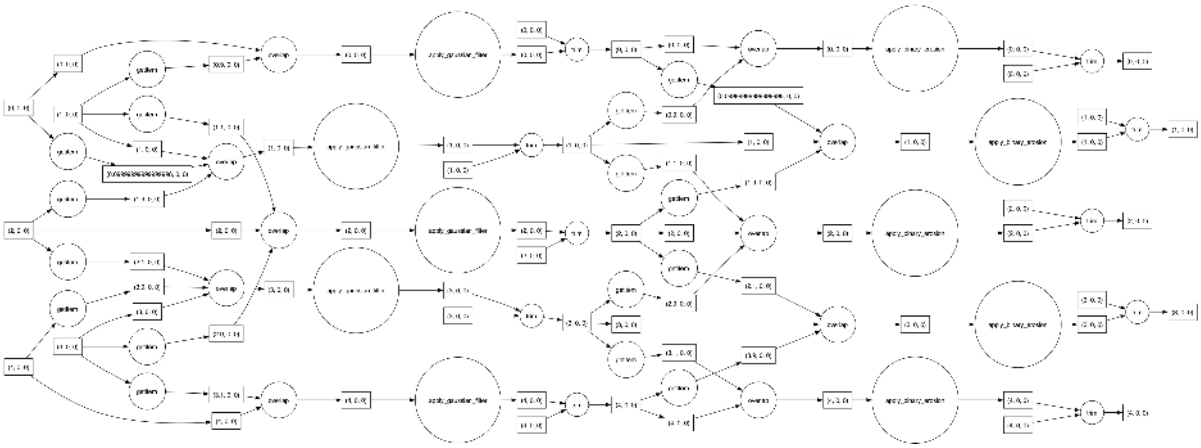
- Apply a threshold at 0.9
- Erode with a ball structure element with radius 12

```
import scipy.ndimage as ndimage
from skimage.morphology import ball
def apply_binary_erosion(block):
    return ndimage.binary_erosion(block>0.9, ball(12))
```

(continues on next page)

(continued from previous page)

```
# Apply the binary erosion to the Dask array
erode_foam = image_filt2.map_overlap(apply_binary_erosion, depth=int(3*3), boundary=
↪'none')
erode_foam.visualize(rankdir="LR")
```



Label the items in the image

- Create a label function
- Labels must be globally unique

```
from scipy.ndimage import label

def block_label(in_block, block_id=None):
    slice_no = block_id[0]
    offset = (np.prod(in_block.shape)*slice_no).astype(np.int64)
    label_img = label(in_block)[0].astype(np.int64)
    label_img[label_img > 0] += offset
    return label_img

lab_bubbles = erode_foam.map_blocks(block_label, dtype='int64')
```

Run the labelling DAG

The labelling is executed with a profiler to check the timing

```
with diag.ProgressBar(), diag.Profiler() as prof, diag.ResourceProfiler(0.5) as rprof:
    processed_stack = lab_bubbles.compute(num_workers=8)
#     processed_stack = erode_foam.compute(num_workers=8)
```

```
[                                        ] | 0% Completed | 138.75 us
```

```
[###############                         ] | 39% Completed | 106.83 ms
```

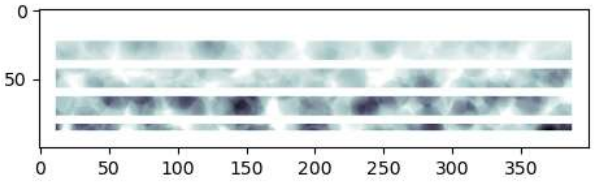
```
[###################################     ] | 89% Completed | 1.19 s
```

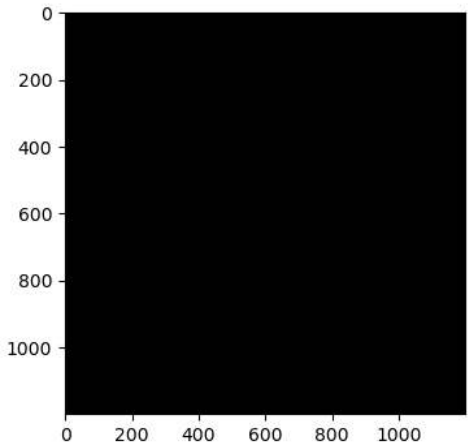
```
[####################################     ] | 89% Completed | 4.43 s
[#####################################    ] | 93% Completed | 5.64 s
[#####################################    ] | 93% Completed | 5.74 s
[#####################################    ] | 93% Completed | 5.85 s
[#####################################    ] | 93% Completed | 5.95 s
[#####################################    ] | 93% Completed | 6.06 s
[#####################################    ] | 93% Completed | 6.17 s
[#####################################    ] | 93% Completed | 6.27 s
[#####################################    ] | 93% Completed | 6.38 s
[#####################################    ] | 93% Completed | 6.48 s
[#####################################    ] | 93% Completed | 6.59 s
[#####################################    ] | 93% Completed | 6.69 s
[#####################################    ] | 93% Completed | 6.80 s
[#####################################    ] | 93% Completed | 6.90 s
[######################################   ] | 95% Completed | 7.01 s
[######################################   ] | 95% Completed | 7.11 s
[######################################   ] | 95% Completed | 7.22 s
[######################################   ] | 95% Completed | 7.32 s
[######################################   ] | 95% Completed | 7.43 s
[######################################   ] | 95% Completed | 7.54 s
[######################################   ] | 95% Completed | 7.64 s
[######################################   ] | 95% Completed | 7.75 s
[######################################   ] | 95% Completed | 7.85 s
[######################################   ] | 95% Completed | 7.96 s
```

```
[########################################  ] | 95% Completed | 8.06 s
[########################################  ] | 95% Completed | 8.17 s
[########################################  ] | 95% Completed | 8.27 s
[########################################  ] | 95% Completed | 8.38 s
[########################################## ] | 97% Completed | 8.48 s
[########################################## ] | 97% Completed | 8.59 s
[########################################## ] | 97% Completed | 8.69 s
[########################################### | 100% Completed | 8.80 s
```

Results of running the DAG

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.imshow(np.sum(processed_stack, 1), cmap='bone_r')
ax2.imshow(montage2d(processed_stack[::20]), cmap='nipy_spectral');
```





Profiler output

```
diag.visualize([prof, rprof]);
```

0.6.9 The importance of operation order

Select a slice and filter

Filter and select a slice

Implementation of execution order example

```
foam_slices_da = da.from_array(
    foam_stack/255.0, chunks=(10, 500, 500), name='FoamSlices')

def apply_gaussian_filter2(block):
    return gaussian_filter(1-block, sigma=[1,9,9])

filt_slices   = da_foam.map_overlap(apply_gaussian_filter2, depth=int(3*3), boundary=
 ↪'none')
single_slice   = filt_slices[50]

# single_slice.visualize(filename="singleslice.svg", optimize_graph=True);
# filt_slices.visualize(filename="filtslices.svg"  , optimize_graph=True);
single_slice.visualize(filename="singleslice.png", optimize_graph=True);
filt_slices.visualize(filename="filtslices.png"  , optimize_graph=True);
```

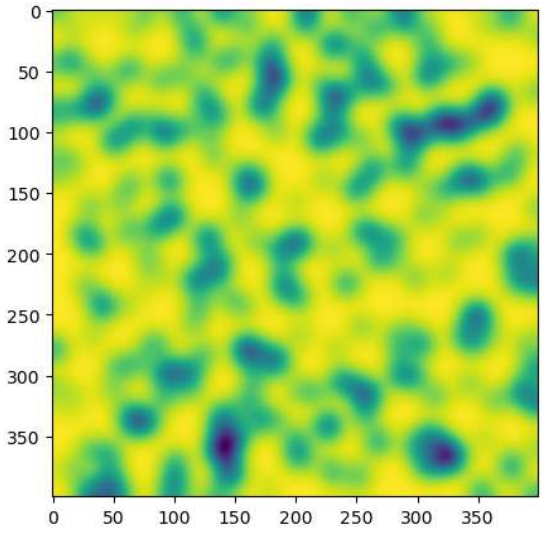
Compare performance

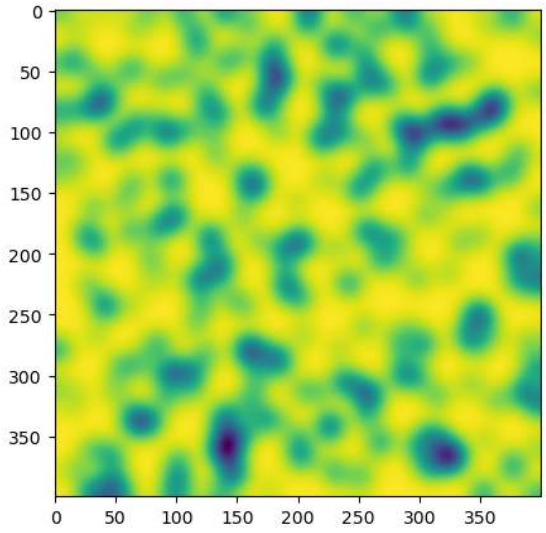
```
fig, (ax1,ax2) = plt.subplots(1,2,figsize=(12,5))
num_workers = 1
with diag.ProgressBar():
    ax1.imshow(single_slice.compute( num_workers = num_workers))
    ax2.imshow(filt_slices.compute(  num_workers = num_workers)[50])
```

```
[                                        ] | 0% Completed | 111.50 us
[#########################               ] | 66% Completed | 105.96 ms
[#########################               ] | 66% Completed | 211.42 ms
[########################################] | 100% Completed | 316.88 ms

[                                        ] | 0% Completed | 100.83 us
[######                                  ] | 17% Completed | 106.39 ms
[######                                  ] | 17% Completed | 211.74 ms
[###############                         ] | 39% Completed | 317.10 ms
[###############                         ] | 39% Completed | 422.54 ms
```

```
[########################                ] | 60% Completed | 527.92 ms
[########################                ] | 60% Completed | 633.30 ms
[###############################         ] | 78% Completed | 738.67 ms
[#####################################   ] | 95% Completed | 839.38 ms
[#####################################   ] | 95% Completed | 944.79 ms
[########################################] | 100% Completed | 1.05 s
```





Same output but different timing!

What happened?

Select a slice and filter (313.33ms)

Filter and select a slice (422.06ms)

- A single slice only activates the chunks needed (lazy evaluation)
- 3 Chunks are processed
- The image is divded into 10 chunks
- All chunks are processed
- The slice is taken from the result

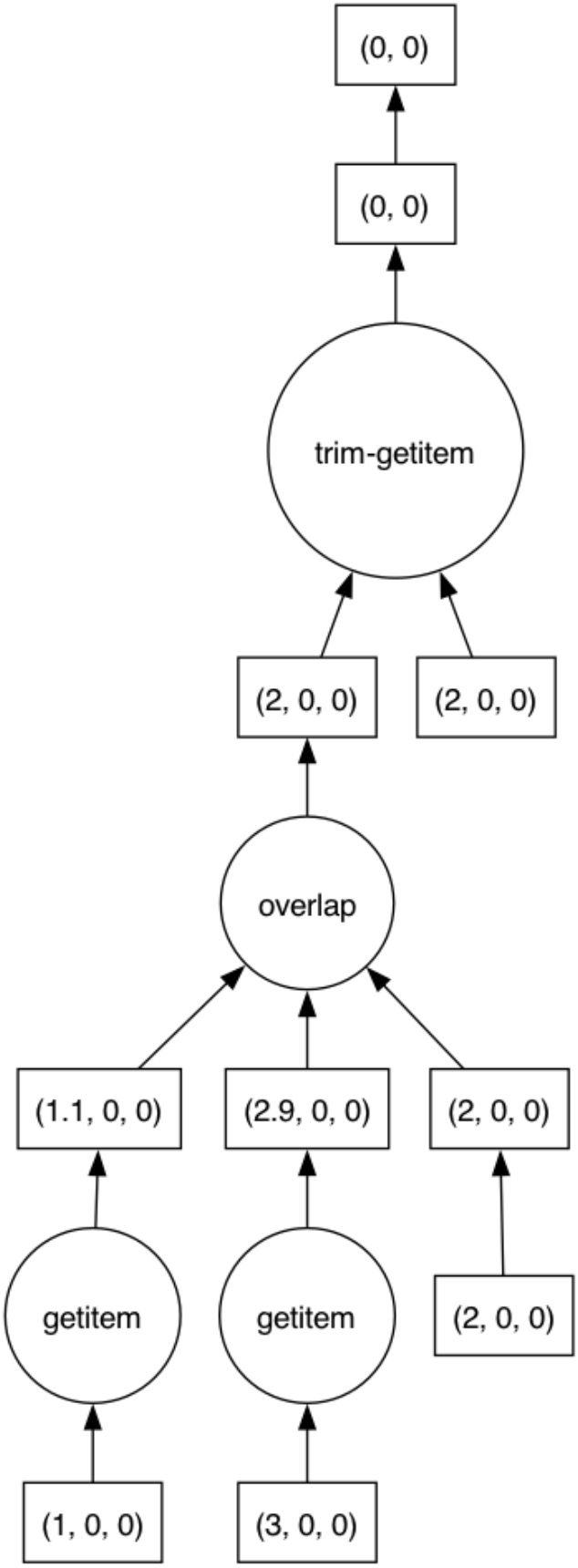


Fig. 1: The processing graph for a single slice.

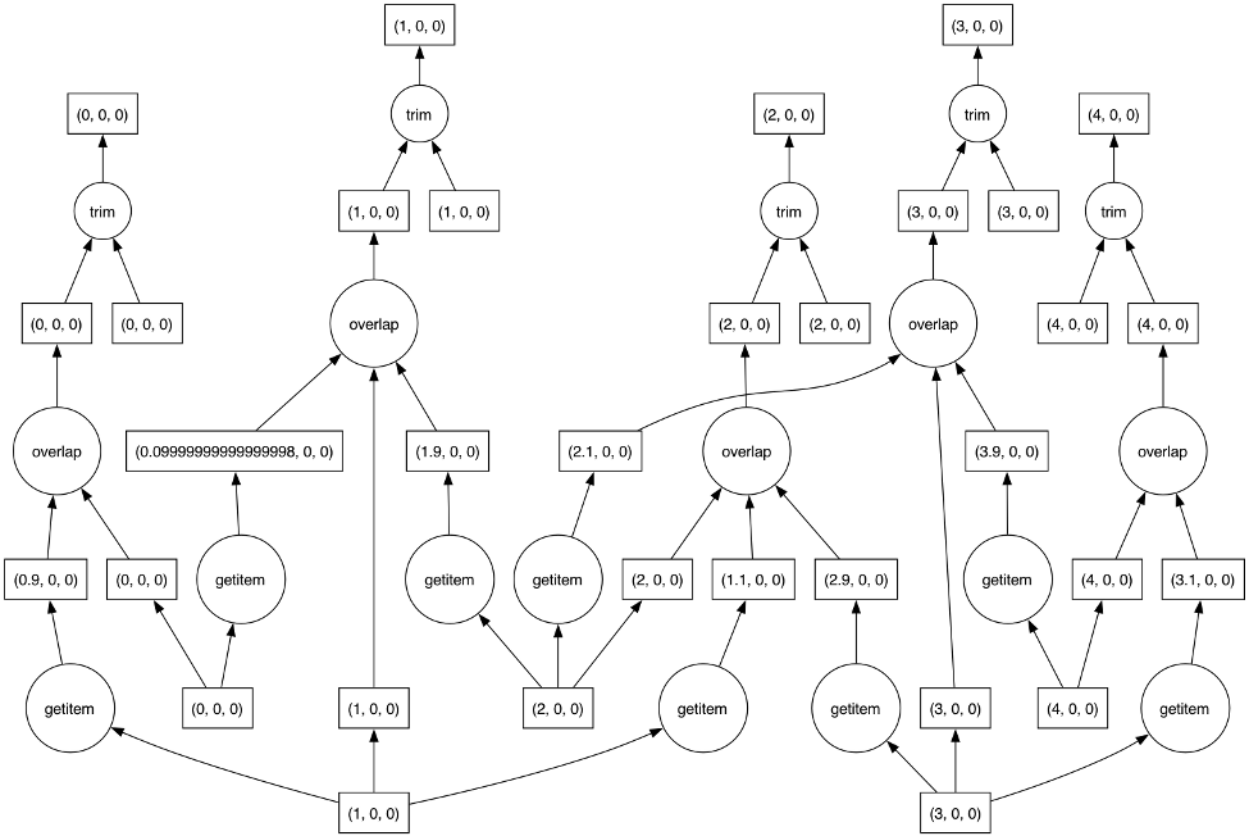


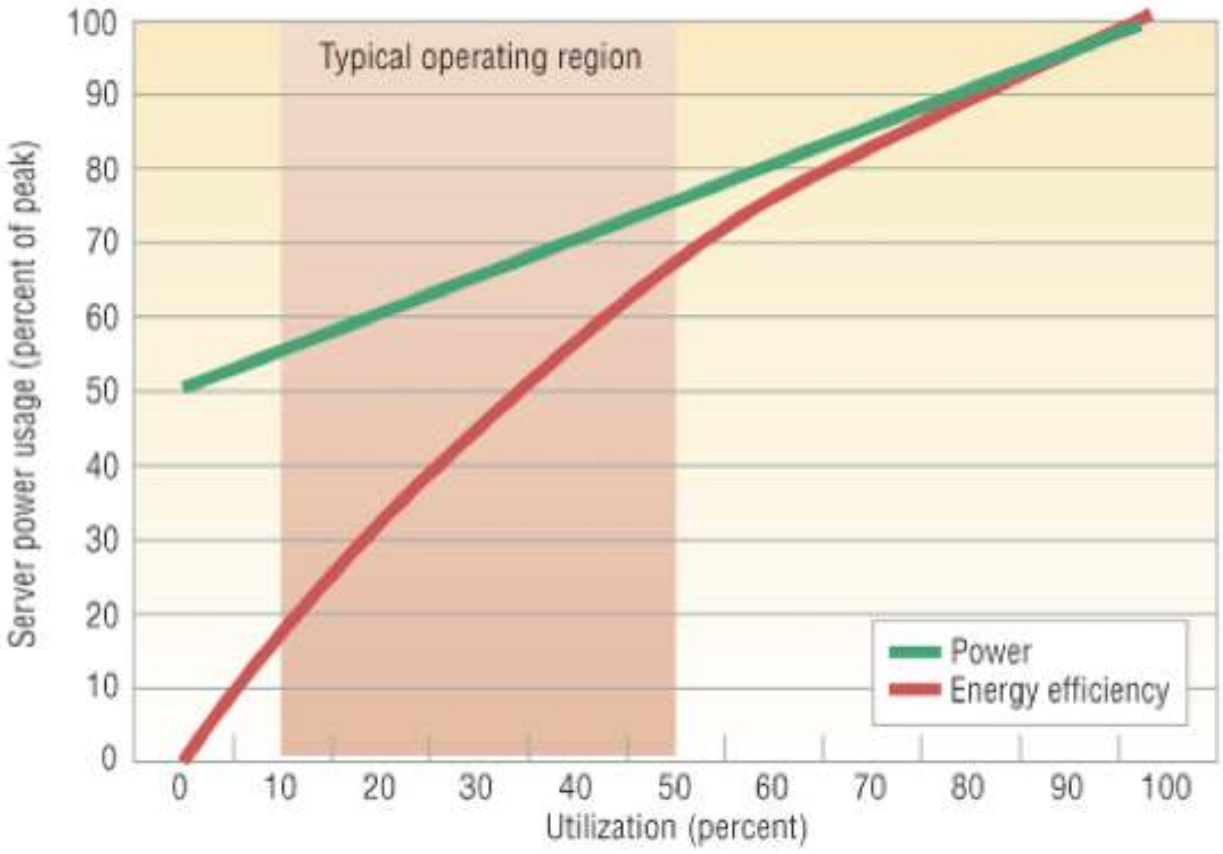
Fig. 2: The processing graph for filtering the entire volume and extracting a slice.

0.7 Cloud computing

0.7.1 Motivation for Cloud Computing

Local recources

- Local resources are expensive and underutilized
- Management and updates are expensive and require dedicated IT staff



Cloud Resources

- Automatically setup
- “Unlimited” potential capacity and storage
- Cluster management already setup
- Common tools with many people using the same

0.7.2 Spark - A rich, heavily developed platform

Available Tools

Tools built for table-like data data structures and much better adapted to it.

- K-Means,
- Matrix Factorization, Genomics, Graph Analytics, Machine Learning

Commercial Support

Dozens of major companies (Apple, Google, Facebook, Cisco, …) donate over $30M a year to development of Spark and the Berkeley Data Analytics Stack

- 2 startups in the last 6 months with seed-funding in excess of $15M each

Academic Support

- All source code is available on GitHub
- Elegant (20,000 lines vs some PhDs of 75,000+)
- No patents or restrictions on usage

0.7.3 Beyond: Streaming

Post-processing goals

- Analysis done in weeks instead of months
- Some real-time analysis and statistics

Streaming

- Can handle static data
- or live data coming in from a 'streaming' device like a camera to do real-time analysis.

The exact same code can be used for real-time analysis and static code

Scalability

- Connect more computers.
- Start workers on these computer.

0.7.4 Beyond: Approximate Results

Projects at AMPLab like Spark and BlinkDB are moving towards approximate results.

- Instead of `mean(volume)`
- `mean(volume).within_time(5)`
- `mean(volume).within_ci(0.95)`

For real-time image processing it might be the only feasible solution and could drastically reduce the amount of time spent on analysis.

0.8 Summary

- More can be done in parallel
- Limited resources, redundancy, race conditions
- Cloud computing
- DAGs

0.9 Wrapping up QBI

0.9.1 Why QBI?

We have experimental data

- Great amounts
- Limited time
- Limited manpower

We want to publish it

- Quantitative analysis
- Reliable
- Repeatable
- Efficent automated workflows

0.9.2 Objectives

We need to understand the components of QBI

- The data creation
- Image processing and analysis methods
- Working with data
- Some computer science/engineering
- How to present the results

0.9.3 The work flow

0.9.4 Lecture 1 - Image basics

- What is an image?
- Pixelwise operations
- The histogram

0.9.5 Lecture 2 - Data sets and validation

- Using prepared data
- What is a good data set
- Analysis validation workflow
- Ground truth

0.9.6 Lecture 3 - Image enhancement

- Noise
- Reducing noise
- Enhancing image features
- Orientation analysis
- Filter performance analysis

0.9.7 Lecture 4 - Basic segmentation

- How are images formed
- What is a change in contrast
- How to apply thresholds
  - Using the histogram

0.9.8 Lecture 5 - Advanced segmentation

- Methods to automatically find thresholds
- Unsupervised methods
  - Histogram based
  - Clustering
- Supervised methods
  - The need for ground truth
  - Training
  - Prediction

0.9.9 Lecture 6 - Shape analysis

- Component labelling
- Shape properties
- Textures

0.9.10 Lecture 7 - Complex shape

- Particle clouds
- Segmentation of touching objects
- Skeletons

0.9.11 Lecture 8 - Statistics

- Uncertainties in the analysis
- Probabilty
- Implications of statistical analysis
- Presenting the results in graphs

0.9.12 Lecture 9 - Dynamic experiments

- Different types of dynamic experiments
- Time series analysis
  - Tracking
  - Registration

0.9.13 Lecture 10 - Multiple modalities

Multi modality imaging

Can we gain more information with additional image sources

- Modality combinations
- Registration
- Data fusion

Software engineering

- Unit testing
- Working with repositories
- Continuous integration

0.9.14 Lecture 11 - Scaling up

- Parallel processing
- Distributed computing
- DAGs

0.9.15 Projects

Project Overview

- Scientifically driven project clear question
  - Not only count number of bubbles in system
  - Rather: Does temperature affect bubble size?
- Take images through appropriate processing steps
- Characterize parameter space and error sources
- Short presentation on **Thursday May 22nd**
  - 10 minutes depending on number of students who present
  - Problem
  - Solution
  - Key Findings

A list of projects

Project support

- There are no more QBI lectures!
- Make appointments with Daniel or Anders for consultation.