# Quantitative Big Imaging - Scaling up

Anders Kaestner

May 20, 2021

# 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.

# CHAPTER ONE

# SCALING UP AND BIG DATA

```
%load_ext autoreload
%autoreload 2
import seaborn as sns
import matplotlib.pyplot as plt
from bokeh.io import output_notebook
from bokeh.resources import CDN
output_notebook(CDN, hide_banner=True)
local_cluster = False
if local_cluster:
    from dask.distributed import Client, LocalCluster
    cluster = LocalCluster(n_workers=2, threads_per_worker=2)
    client = Client(cluster)
plt.rcParams["figure.figsize"] = (8, 8)
plt.rcParams["figure.dpi"] = 72
plt.rcParams["font.size"] = 16
plt.rcParams["figure.constrained_layout.use"] = True
plt.rcParams['font.family'] = ['sans-serif']
plt.rcParams['font.sans-serif'] = ['DejaVu Sans']
plt.style.use('ggplot')
sns.set_style("whitegrid", {'axes.grid': False})
```

## 1.1 Literature / Useful References

### 1.1.1 Big Data

- Google's Presentation on Distributed Computing
- Slides
- MapReduce Paper: Jeffrey Dean, et al. (n.d.). MapReduce: Simplified Data Processing on Large Clusters.
- Scalable Systems Course
- Tutorial in Hadoop
- Intro to Data Science @UCB

### 1.1.2 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
- Sun (now Oracle) Grid Engine or here
- Walker, E. (2009), The Real Cost of a CPU Hour, IEEE Computer|

### 1.1.3 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) doi:10.1093/bioinformatics/btt276

### 1.1.4 Cloud Computing

- Amazon S3
- Sitaram, D., & Manjunath, G. (2012). Moving To The Cloud. null (Vol. null). Elsevier. doi:10.1016/B978-1-59749-725-1.00006-8
- 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. doi:10.1109/GreenCom-iThings-CPSCom.2013.296

## 1.2 Outline

- Motivation
- Computer Science Principles
- Parallelism
- Distributed Computing
- Imperative Programming
- Declarative Programming
- Organization
- Queue Systems / Cluster Computing
- Parameterization
- Databases
- Big Data
- MapReduce
- Spark
- Streaming

- Cloud Computing
- Beyond / The future

## 1.3 Motivation

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

### 1.3.1 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 mess check-pointing code

### 1.3.2 Many datasets

- 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)
- Radiologist in Switzerland alone make 1 Petabyte of scans per year

### 1.3.3 Exploratory Studies

- Not sure what we are looking for
- Easy to develop new analyses
- Quick to test hypotheses

CHAPTER

# TWO

# EXAMPLE PROJECTS

## 2.1 Zebra fish Full Animal Phenotyping

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

- Imaged at 0.74 $\mu m$ resolution:
- Images 11500 x 2800 x 628
- 20-40GVx / sample!

### 2.1.1 Objectives

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

## 2.2 Brain Project

**Whole brain with cellular resolution** 1 $cm^3$ scanned at 1 $\mu m$ resolution: Images $\longrightarrow$ 1000 GVx / sample

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

CHAPTER

THREE

# WHAT IS WRONG WITH USUAL APPROACHES?

## 3.1 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

Some python code to solve the task:

```
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')
```

## 3.2 You want changes in the workflow

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

**You have to rewrite everything, everytime**

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

- big data and
- reproducibility

must be considered from the beginning

CHAPTER

FOUR

# COMPUTER SCIENCE PRINCIPLES

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

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

## 4.1 What is parallelism?

Parallelism is when you can:

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

### 4.1.1 An example

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

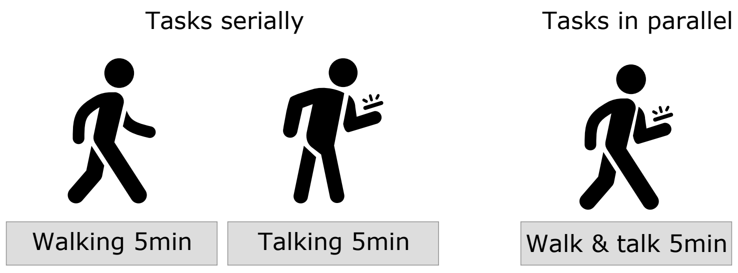


Fig. 4.1: Walking and talking as serial and parallel tasks.

- 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.

CHAPTER

FIVE

# 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.

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.

## 5.1 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 (++)

1. 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 you all the coins they have and you make one master list and then find the median coin (-)

## 5.2 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

### 5.2.1 Dead-lock

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

## 5.3 Challenges in parallel processing

### 5.3.1 1. Coordination

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

### 5.3.2 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)

### 5.3.3 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

## 5.4 Parallel speedup and slowdown

Depend on:

- Available resources
- Implementation
- Scheduling

```
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=(10,6))

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

(continues on next page)

(continued from previous page)

```
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");
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-3cf42a4bb686> in <module>
----> 1 nCPU     = np.linspace(1,8,8)
      2 speedup  = np.array([1,2,3,4,5,6,7,8])
      3 realspeedup = 0.9 * speedup + 0.1
      4 realspeedup[5:] = np.array([5.2, 5.5, 5])
      5

NameError: name 'np' is not defined
```

## 5.5 Challenges in distributed processing

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

### 5.5.1 Sending Instructions / Data Afar

### 5.5.2 Fault Tolerance

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

### 5.5.3 Data Storage

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

CHAPTER

SIX

# PROGRAMMING PARADIGMS

## 6.1 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

### 6.1.1 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

## 6.2 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*

### 6.2.1 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

## 6.3 Comparison

They look fairly similar, so what is the difference?

### 6.3.1 Imperative soup

### 6.3.2 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)

## 6.4 Results

### 6.4.1 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

### 6.4.2 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)

### 6.4.3 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

CHAPTER

# SEVEN

# 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 a script means rewriting everything

## 7.1 Queue Computing

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

**Resources**

**Jobs**

**Users**

computers, memory, storage, network

tasks to be run

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

## 7.2 Structure of Cluster

### 7.2.1 Master (or Name) Node(s)

The node with which every other node communicates, the main address.

### 7.2.2 Worker Nodes

The nodes where the computation is performed.

### 7.2.3 Scheduler

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

CHAPTER

EIGHT

# 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))));
```

Non-consecutive header level increase; 0 to 3

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

Non-consecutive header level increase; 0 to 3

```
   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
```

## 8.1 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)

### 8.1.1 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 Python, Excel, Matlab or R

### 8.1.2 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)
```

$$\rightarrow [1, 0.95]$$

## 8.2 Beyond SQL: NoSQL

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

### 8.2.1 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

### 8.2.2 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**

CHAPTER

NINE

# BIG DATA

## 9.1 Definition

### 9.1.1 Velocity, Volume, Variety

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

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

### 9.1.2 When scaling isn't scary

10X, 100X, 1000X is the same amount of effort

### 9.1.3 When you are starving for enough data

Director of AMPLab said their rate limiting factor is always enough interesting data

### 9.1.4 No 'clicks' per sample

Everything is automated, no human interaction required during processing

## 9.2 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!**

### 9.2.1 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 $O(N^2)$

### 9.2.2 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

### 9.2.3 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

### 9.2.4 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 $\$\vec{L} = \left[1,2,3,4,5\right] f(x) = x^2 map(f \rightarrow \vec{L}) = \left[1,4,9,16,25\right]\$$

#### Reduce

Reduce is more complicated and involves aggregating a number of different elements and summarizing them. For example the $\Sigma$ function can be written as a reduce function $\$\vec{L} = \left[1,2,3,4,5\right] g(a,b) = a + bReducethenappliesthefunctiontothefirsttwoelements, andthentotheresultofthefirsttwowiththethirdandsoonuntilalltheee$
$\vec{L}) = g(g(g(g(1,2),3),4),5) or reduce(f \rightarrow \vec{L}) = g(g(1,2),g(3,4)),5)\$$

## 9.3 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.

## 9.4 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.

## 9.5 Counting Words

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

### 9.5.1 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)

### 9.5.2 Reduce

adds up the values for each word

#### 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)]```
~~~

### 9.5.3 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']
```

### 9.5.4 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
```

```
HBox(children=(HTML(value=''), FloatProgress(value=0.0, max=129107.0), HTML(value='
↪')))
```

#### 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
```

### 9.5.5 MapReduce Approach

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:

```python
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
```

```python
# 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
```

## 9.6 MapReduce approach using Dask bags

Dask Bag implements 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

### 9.6.1 Build a bag for the word analysis

```
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])
```

### 9.6.2 Run the analysis

```
import dask.diagnostics as diag

workers = 4

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))
```

```
[########################################] | 100% Completed |  0.6s
Top 10
 [('the', 26851), ('and', 24077), ('i', 20535), ('to', 18561), ('of', 16013), ('you',␣
↪13856), ('a', 13840), ('my', 12282), ('that', 10761), ('in', 10537)]
[########################################] | 100% Completed |  0.5s
Bottom 10
 [('midsummernights', 1), ('wanes', 1), ('newbent', 1), ('solemnities', 1), (
↪'merriments', 1), ('interchangd', 1), ('lovetokens', 1), ('prevailment', 1), (
↪'unhardend', 1), ('filchd', 1)]
```

### 9.6.3 Visualize parallel task distribution

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

CHAPTER

TEN

# ENVIRONMENTS FOR DISTRIBUTED COMPUTING

## 10.1 Hadoop

Hadoop is the opensource 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

## 10.2 Spark / Resilient Distributed Datasets

### 10.2.1 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

### 10.2.2 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

## 10.3 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.

## 10.4 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).

### 10.4.1 Some 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/

## 10.5 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.

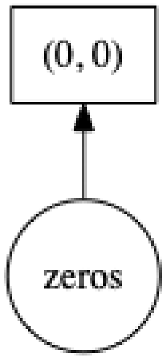
## 10.6 DAG examples

### 10.6.1 A basic DAG

Create two 5x5 images and use a single chunk:

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

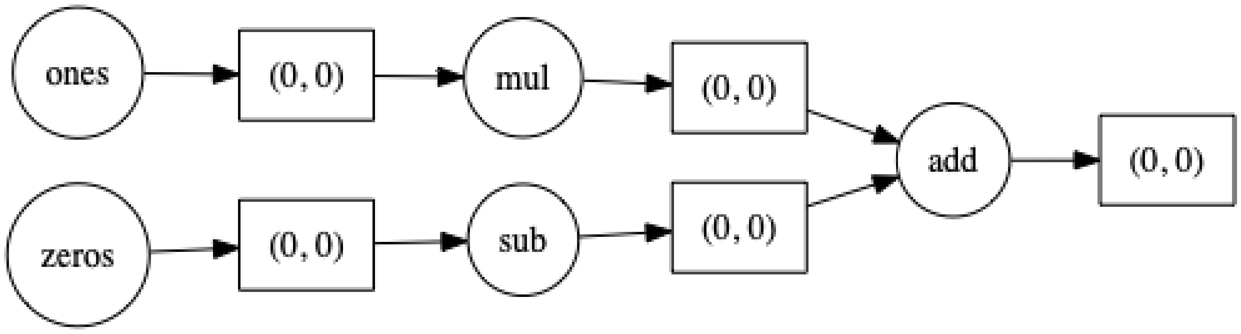
```
import dask.array as da
from dask.dot import dot_graph
image_1 = da.zeros((5, 5), chunks=(5, 5))
image_2 = da.ones((5, 5), chunks=(5, 5))
dot_graph(image_1.dask)
```



### 10.6.2 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")
```



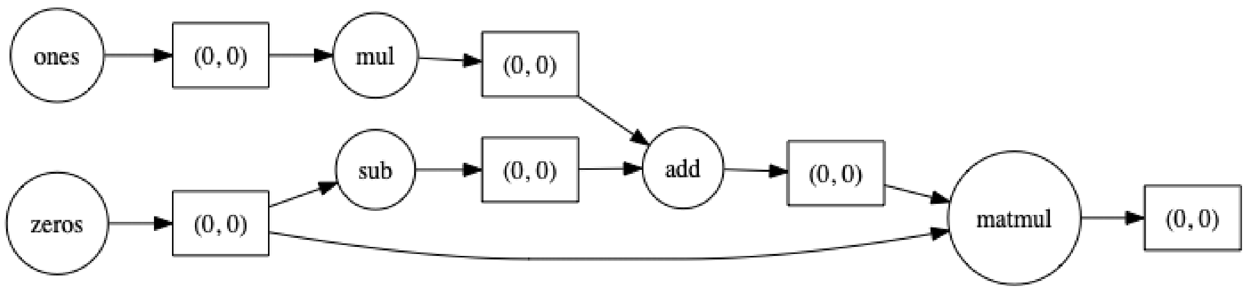
### 10.6.3 More calculations

$$image_5 = image_1 * image_4$$

#### Remember

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

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

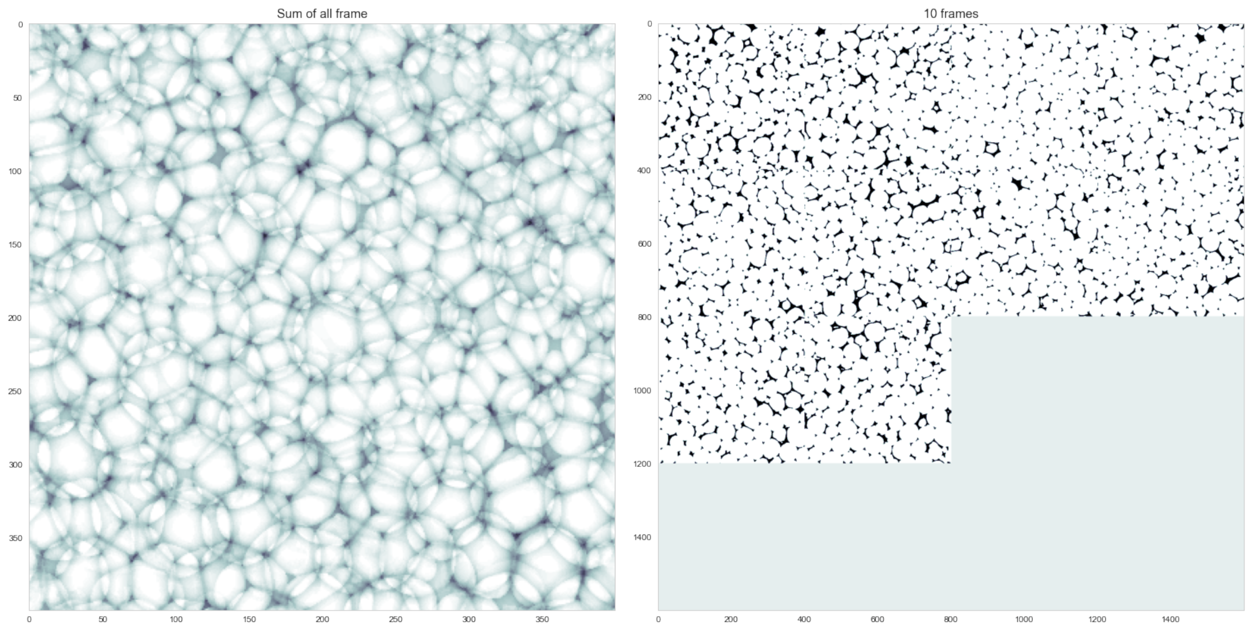


## 10.7 Image Processing using DAGs

the initial examples were shown on very simple image problems.

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

```
%matplotlib inline
import dask.array as da
from dask.dot import dot_graph
import numpy as np
from skimage.io import imread
import matplotlib.pyplot as plt
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=(20, 10))
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');
```



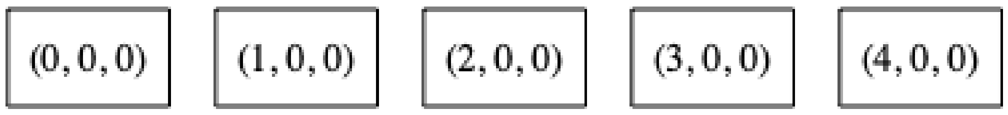
```
from itkwidgets import view
from IPython.display import display
import itk

if True:
    viewer = view(itk.GetImageFromArray(foam_stack.copy()))
    display(viewer)
```

```
Viewer(geometries=[], gradient_opacity=0.22, point_sets=[], rendered_image=<itk.
↪itkImagePython.itkImageUC3; pr...
```

## 10.8 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]

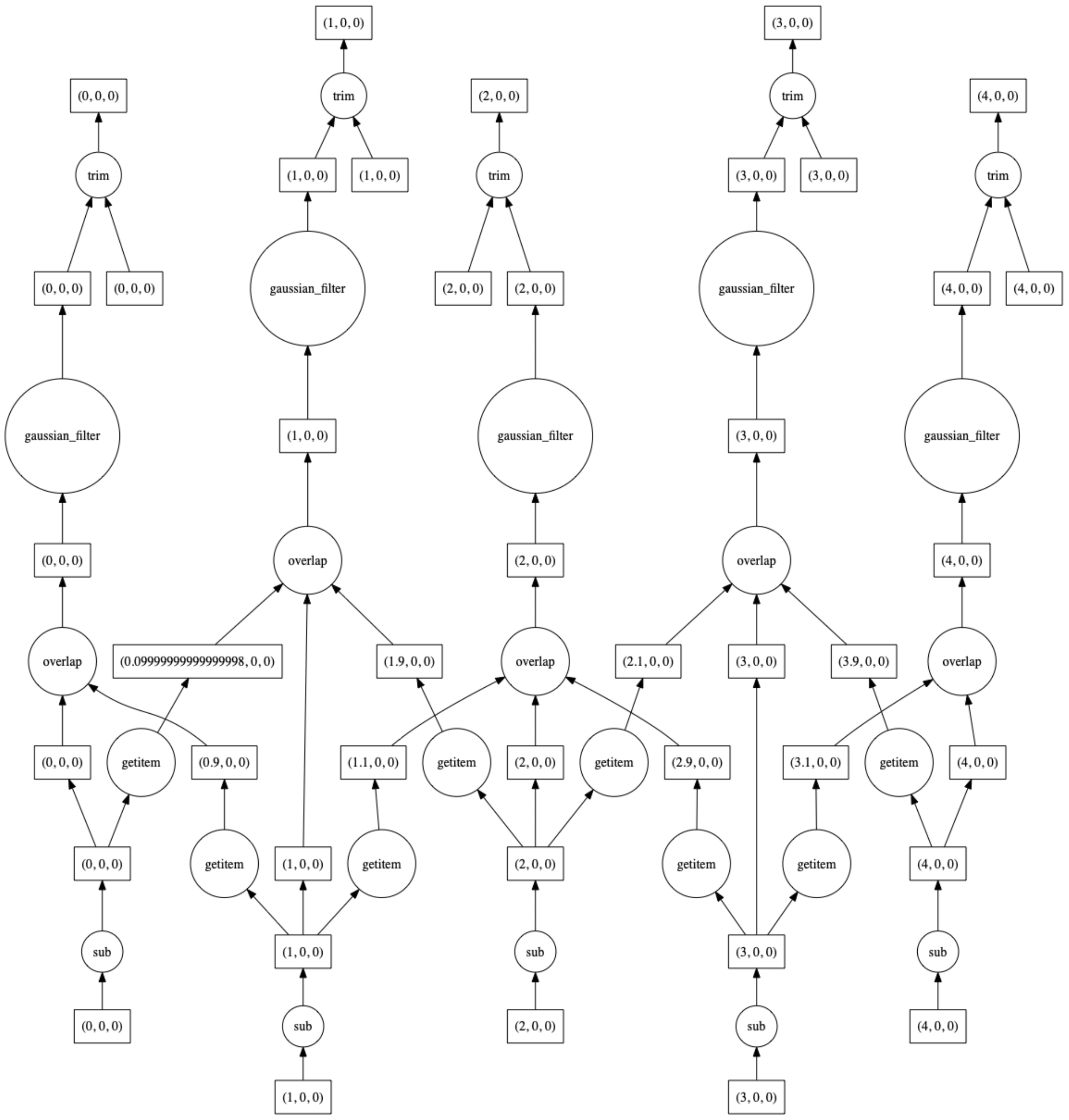
```
da_foam = da.from_array(
    foam_stack/255.0, chunks=(20, 400, 400), name='FoamImage')
dot_graph(da_foam.dask)
```



### 10.8.1 Add filter operation

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

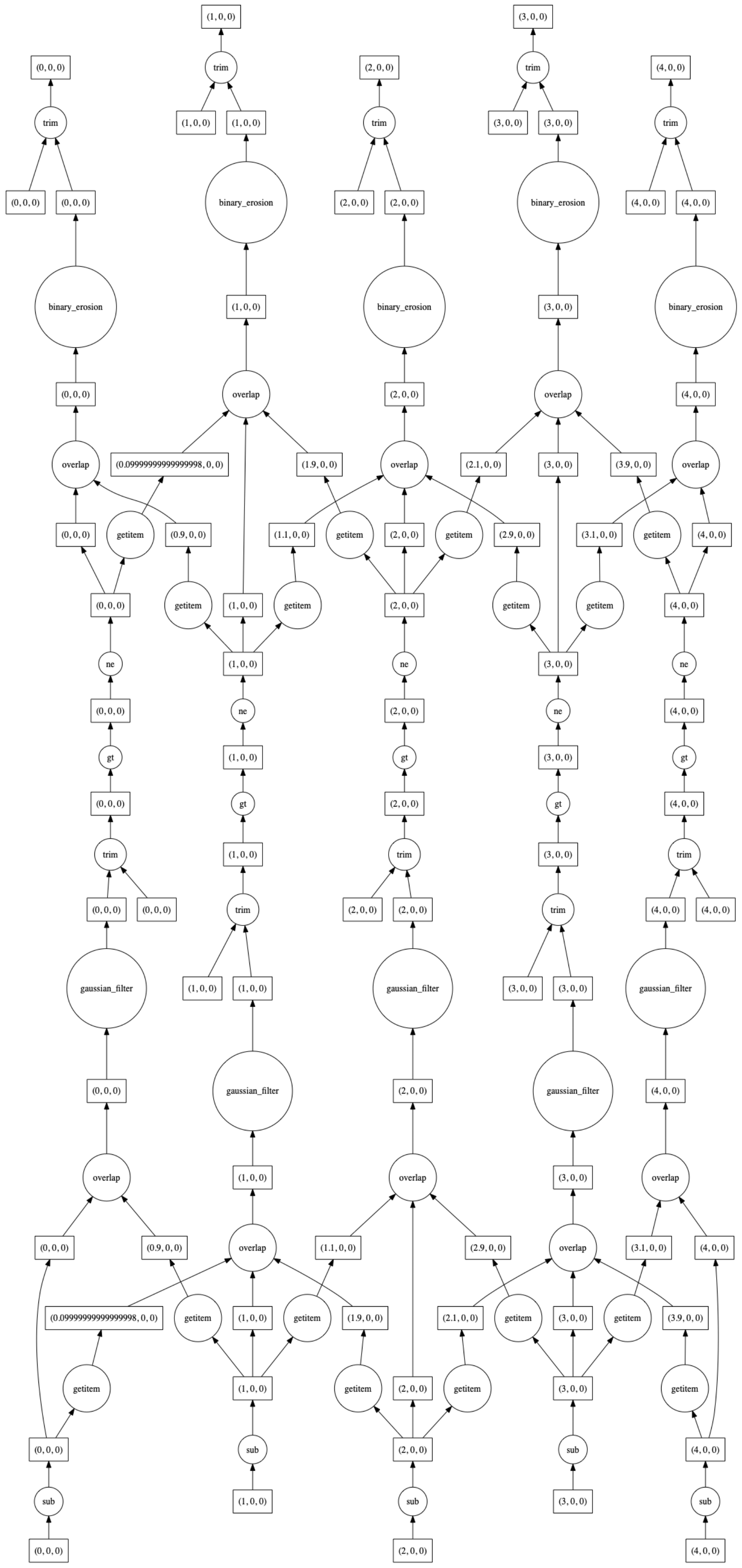
```
import dask_ndfilters as da_ndfilt
image_filt = da_ndfilt.gaussian_filter(1-da_foam, sigma=(3, 6, 6))
dot_graph(image_filt.dask)
```



### 10.8.2 Add segment and erode to the DAG

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

```
import dask_ndmorph as ndmorph
from skimage.morphology import ball
erode_foam = ndmorph.binary_erosion(image_filt > 0.9, ball(12))
dot_graph(erode_foam.dask)
```



### 10.8.3 Label the items in the image

- Create a label function
- Labels must be globally unique

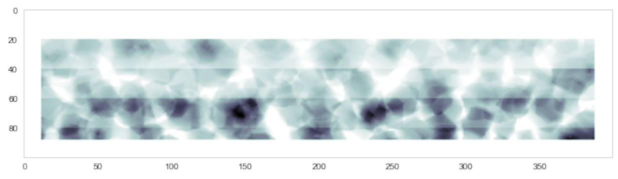
```
from scipy.ndimage import label


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')
```

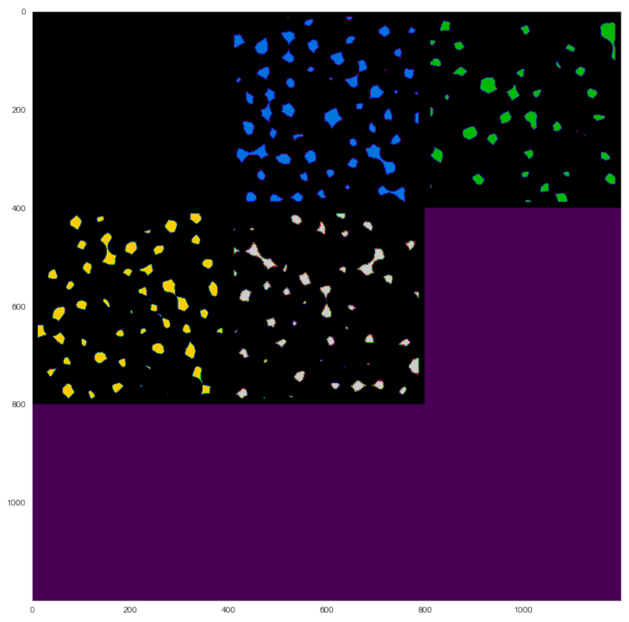
```
with diag.ProgressBar(), diag.Profiler() as prof, diag.ResourceProfiler(0.5) as rprof:
    processed_stack = lab_bubbles.compute(num_workers=4)
```

```
[########################################] | 100% Completed | 11.1s
```

### 10.8.4 Results of running the DAG

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))
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]);
```

## 10.9 The importance of operation order

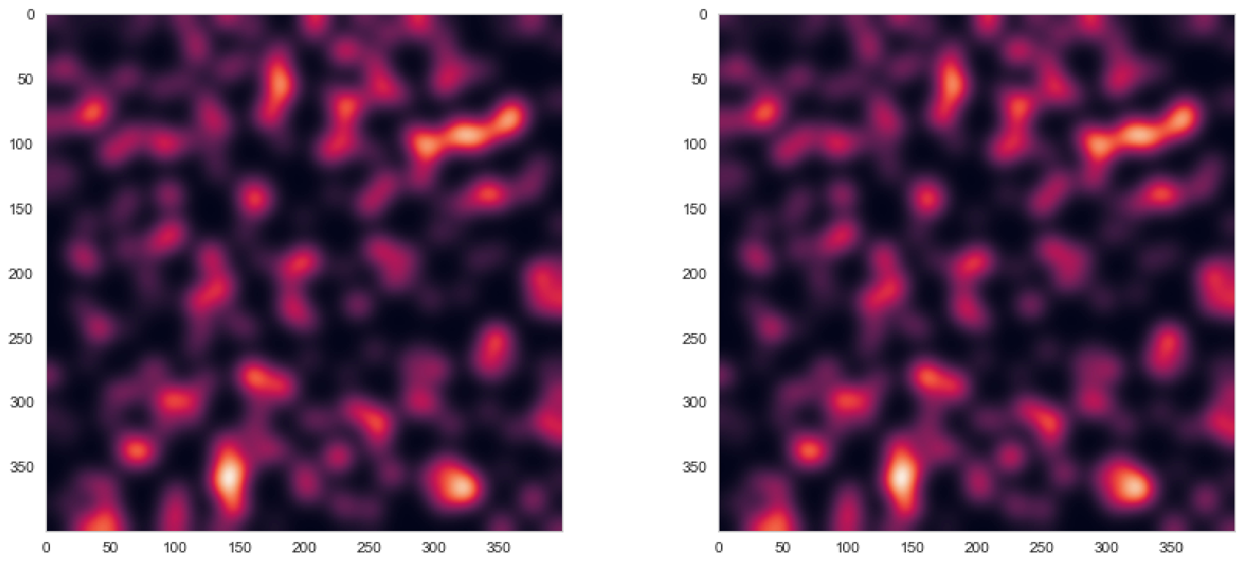
- Select a slice and filter
- Filter and select a slice

```
foam_slices_da = da.from_array(
    foam_stack/255.0, chunks=(10, 500, 500), name='FoamSlices')
filt_slices = da_ndfilt.gaussian_filter(foam_slices_da, sigma=(1.0, 9, 9))
single_slice = filt_slices[50]
single_slice.visualize(filename="singleslice.svg",optimize_graph=True);
filt_slices.visualize(filename="filtslices.svg", optimize_graph=True);
```

### 10.9.1 Compare performance

```
fig, (ax1,ax2) =plt.subplots(1,2,figsize=(12,5))
with diag.ProgressBar():
    ax1.imshow(single_slice.compute(num_workers=4))
    ax2.imshow(filt_slices.compute(num_workers=4)[50])
```

```
[########################################] | 100% Completed |  0.3s
[########################################] | 100% Completed |  0.7s
```



**Same output but different timing!**

### 10.9.2 What happened?

**Select a slice and filter**

**Filter and select a slice**

- 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

CHAPTER

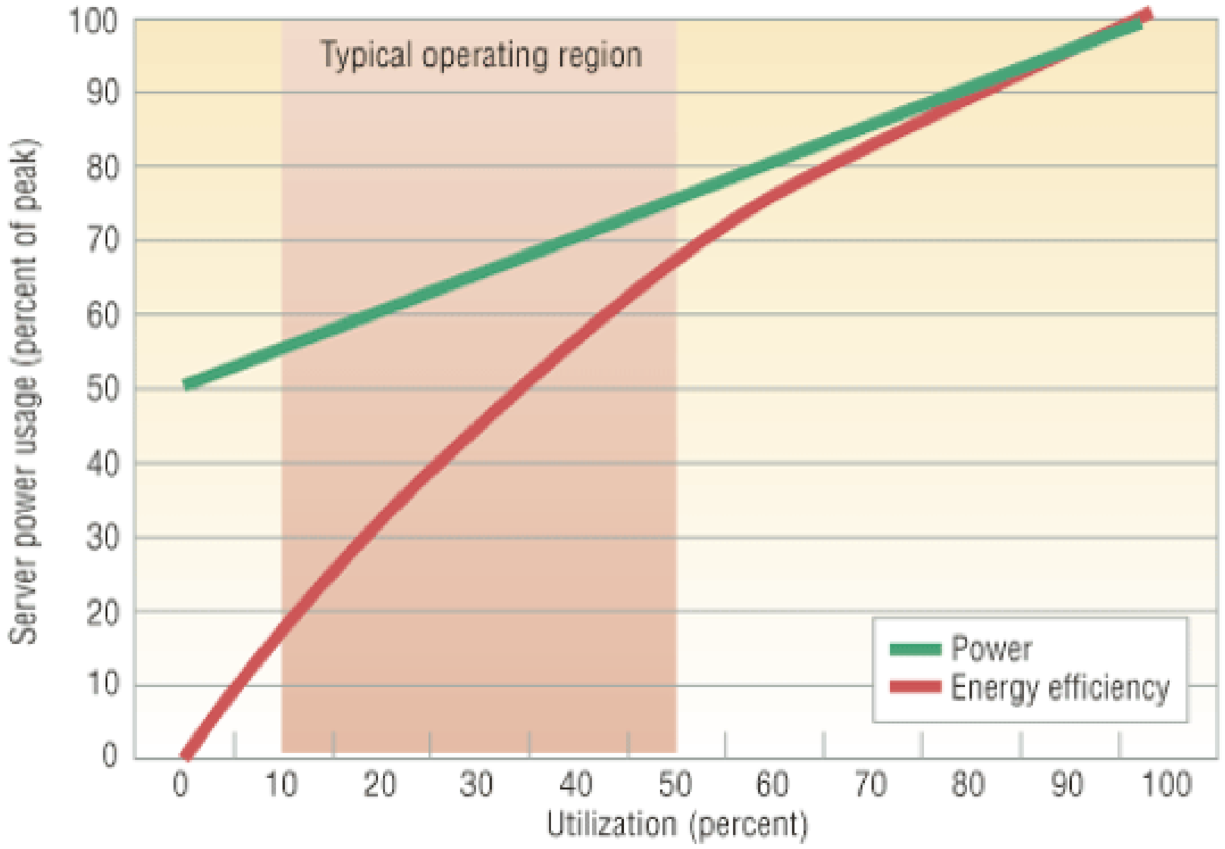
# ELEVEN

# CLOUD COMPUTING

## 11.1 Motivation for Cloud Computing

### 11.1.1 Local recources

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



### 11.1.2 Cloud Resources

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

## 11.2 Spark - A rich, heavily developed platform

### 11.2.1 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

### 11.2.2 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

### 11.2.3 Academic Support

- All source code is available on GitHub
- Elegant (20,000 lines vs my PhD of 75,000+)
- No patents or restrictions on usage
- Machine Learning Course in D-INFK next semester based on Spark

## 11.3 Beyond: Streaming

### 11.3.1 Post-processing goals

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

### 11.3.2 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

### 11.3.3 Scalability

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

## 11.4 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.

CHAPTER

TWELVE

# SUMMARY

- Motivation
- Computer Science Principles
- Organization
- Databases
- Big Data
- Cloud Computing
- Beyond / The future