Daniel Silbermann, s090116
Distribuerede og parallelle systemer, Spring 2021

Lectures

tl;dr

Lost 1-4 cause I overwrote the file by mistake
He doesn't really say anything of interest, so mostly skip this section, only the list of reading material is worth anything.

Week 01

Week 02
Cache (parallel hardware)
ch 1, 2 – 2.3 (inclusive)
– Read 2.3.3 briefly, not part of learning objectives

Week 03
Parallel programming
Pacheco: ch 2.4 – 2.10 (inclusive)

Week 04
MPI - Message Passing Interface
Pacheco: ch 3.1 – 3.3 (inclusive)

Week 05
MPI continued
• Pacheco: ch 3.4 – 3.8 (inclusive)

Week 06
Shared memory programming
• Pacheco: ch 4.1, 5.1-5.5 (inclusive)
• Pacheco: ch 4.2-4.8 (recap from prior courses)

Week 07
OpenMP continued
• Pacheco: ch 5.5 – 5.8 (inclusive)

Week 08
OpenMP part 3 (hybrid models)
• Pacheco: ch 6 – 6.2.8 (inclusive)

Week 09
Computing
• Maarten & Tanenbaum: Pages 967-986
Ignore formulas on page 980-981

Week 10
Maarten van Steen &
Andrew S. Tanenbaum:A brief introduction to distributed systems, Computing (2016) vol. 98. https://findit.dtu.dk/en/catalog/2335112463

Week 11
"L. Barroso, U. Hölzle, P. Ranganathan,
The Datacenter as a Computer: Designing Warehouse-Scale Machines, ch 1-2"

Week 12
Balouek-Thomert D, Renart EG, Zamani AR, Simonet A, Parashar M. Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows. The International Journal of High Performance Computing Applications. 2019;33(6):1159-1174. doi:10.1177/1094342019877383. You can find it at FindIT: https://findit.dtu.dk/

Week 13

Week 01

Intro, nothing of interest for the involvement of learning was said.

Practical info though.
Strict rules about plagiarism
Sven Karlsson
svea@dtu.dk
Speaks very bad Danish (he's from Sweeden)

House Rules:
• The course staff show up on time and stay the module
• The course staff is always respectful
• Slides uploaded after the activities
• This course is interactive
– You are expected to interact
– Answer questions
– Speak up
– You are called on to present and discuss your work
• We respect each other and the learning space
• Do interrupt

Something about 'learning by doing' type of course.

Sharing is good, right?
• Usually, yes!
• But when working on courses you are not allowed to share anything that leads to an hand-in, for example

–Text
–Code
• You are of course allowed to exchange ideas and thoughts

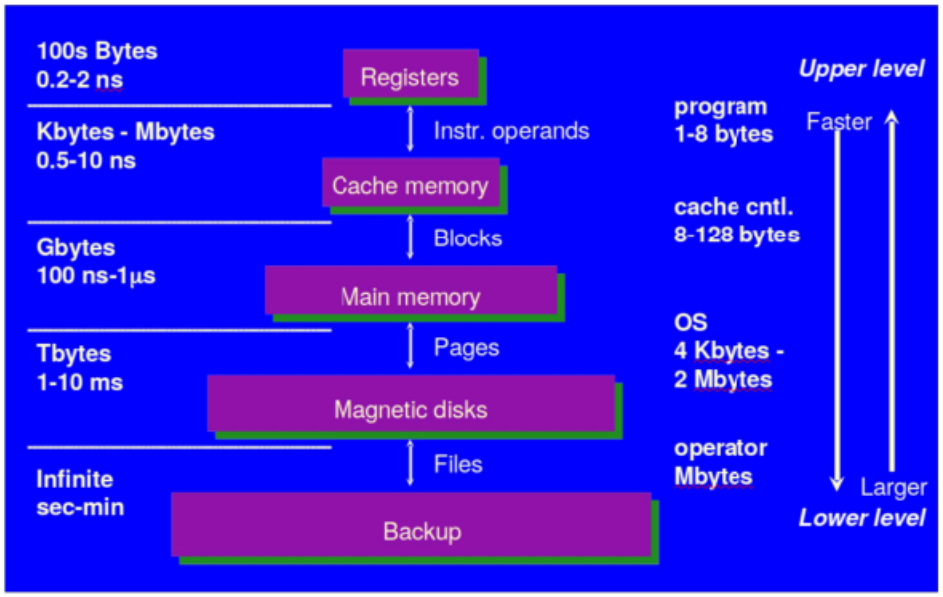
Mailing list setup: dtu-02346@lists.fenixforge.com

Week 02

Cache.

A few pieces of useful information

Memory Hierarchy



Principle of Locality

A program access a relatively small portion of the address space at any instant of time

• Two different types of locality:
  - Temporal locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon.
  - Spatial locality (Locality in space): If an item is referenced, items whose addresses are close, tend to be referenced soon.

• Locality is a property of programs. Different programs have different locality. Different parts of programs have different locality.
• Hardware takes opportunity of locality

Pipelining vs ILP

• To really see the difference between pipelining and ILP

one needs to look at the steady-state performance
• Pipelining typically stops at one new operation per clock cycle
• ILP improves the performance per clock cycle by allowing the steady-state performance to go over one operation per clock cycle

Week 03

Such a huge amount of unneeded crap about the format of the lecture rather than teaching us stuff.

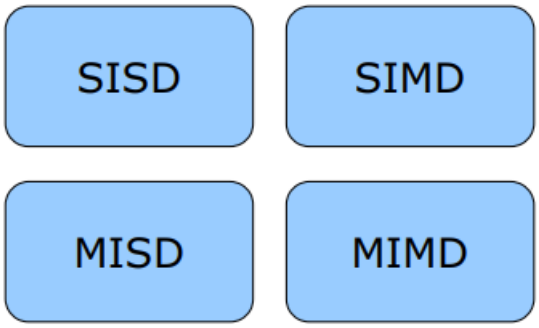
Parallel Computer

A parallel computer system uses a number of processors that can cooperate to solve a single problem faster.

• Processors must have a means to cooperate
• Parallelism is used to solve a large problem faster
• New programming models are needed
• Before we go parallel we need to be efficient on one processor

Flynn's Taxonomy

Flynn's taxonomy is a classification of computer architectures



Single Instruction stream, Single Data stream
Single Instruction stream, Multiple Data stream
Multiple Instruction stream, Single Data stream
Multiple Instruction stream, Multiple Data stream

Week 04

MPI - Message Passing Interface
Pacheco: ch 3.1 – 3.3 (inclusive)

Screw the lecture, he didn't say anything usefull at all.

Parallel programs are commonly defined by nonnegative ranks.
0, 1, 2, 3,

'module load mpi'

Week 05

Missed part of the lecture due to another meeting

Comment: Using MPI.reduce, can we parse null for *sendbuf

Comment: When we are asked to change some code in the assignment - can we choose to either upload the changed code as an extra file or paste it into the report ass we like.

Think-share: why do we want to use collective operations suah as MPI_bcast?

Comment: Because it relieves the programmers the stress of optimising an algorithm till death claims him
MPI_Bcast in particular is good for distributing data... but i would prefer MPI_Scatter in most situations

Comment: *To make it easier to distribute data to other processes for them to work on. It simplifies the communication pattern. Also MIP_Scatter helps with dividing data to decrease memory usage for each process as they will only receive the data they need to do the work*

Comment: *Because then the optimization can be done by the developers working on MPI instead of the developers for the individual applications.*
Furthermore it makes it much easier for the developer for the applications to "just get something working"

He hints at 2-3 other reasons that's not related to performance.

Broadcast operations can be done with a MPI_Bcast.

hint: is MPI_Bcast easeir to read than a for-loop
Yes
By having a one-liner Bcast, rather than multiple lines, makes the programming 'intend' easier to distinguish.

Example: using a collect operation such as mpi_help we can gather a lot of info that's easier to consume

hint 2: code sustainability (making it easier to maintain and keep code updated)

Gather Operation

Gather operation from all MPI processes into one local array

Scatter

Send information from one array

MPI Deadlock

Dead

(extremely rare in parallel programming, deadlock is rarely a reasonably source of potential issues)

Example of databar

Week 06

Important information: Look after each other

Stressfull situation

Stressfull situation
Support the lonely
Take care of yourself
Eat haelthy, Drink water, Sleep, Exercise in a safe way

Information tidbits

First report is eing graded
- Remember: Limited writted feedback, more elaborate via meetings
- Hopefully feedback will be out Saturday

Second Mandatory report due on March 14th, 23: 59
- Covers databar exercise 4, week 5.

- Need to hand in ALL reports.

Additional TA resources available over at:
dtu-02346@lists.fenixforge.com

House Rules

boring, copy paste, same as always.

New detail.
When presenting stuff, they will shut down the chat.

Sequence of sessions

- Recap of programming models
- Multiprocessor architecture ..
- ....
- ...

Learning objectives

Definitions or explanations of the following

What is a parallel computer
A parallel computer system uses a number of processors that can cooperate

Programming Models

Shared memory

Message passing

We're focusing on open MP, which is far more high level than pthreads.

Good things about shared memory machines, it can simplify a lot of operations

Slide 17 (picture)

IC = Interrupt Controller
TLB = Translation Look Aside Buffer, link: https://www.geeksforgeeks.org/translation-lookaside-buffer-tlb-in-paging/
$ = cache
M = Main Memory
NI = Network Interface

The NI allows us to connect multiple things together.

Breakout:
What does this tiered system mean for programming?
What do you need to consider?

Petersons algorithm does not work on processors any longer, because it

Week 07

Information tidbits.
3rd mandatory report deadline
For help during databars, go to dtu learn.
- Course
- Discussion
- Databar waiting list
- Create a post

Recap

• Recap of OpenMP introduction
• OpenMP work-sharing constructs
• OpenMP synchronization
• OpenMP and memory data
• Examples

Learning objectives

pointless to include here

What is OpenMP

openMP is a programming standard for programming multithreaded programming

....

Parallel regions, something about setting environment 'setenv'

Week 08

Something about deadlines for feedback.
3rd report deadline April 4th.

Mail list dtu-02346@list.fenixforge.com

Overview:
OpenMP

what is learning

Concepts or questions which need attention today:

* Something about portability of programs written to become parallel
* Something about performance estimation

Synchronization

Used to protect against race conditions:

- Mutual Exclusion
  - Directive
  - Run-time functions
- Atomic operations
  - Directive

  (very useful when doing operations that happen very rarely)
- Barrier synchronization
  - Directive

Mutual exclusion is expensive
Atomic operations are useful for combining data from threads
Barrier synchronization, to prevent weird stuff from happening.

Storage

Breakout What storage attributes/specifiers are needed?

How do we need to implement threads
How do we determine what data is shared and how.

Book:

Chapter 6 Parallel Program development

Week 09

Exam on may 18th

We're now branching out into distributed systems.
A little bit more theoretical direction, and more general concepts such as concurrency, distribution and so on.

Learning objectives related to distributed systems:'
• You can list the main technical challenges in the distributed system
• You can describe some problem areas where distributed systems are needed
• ...

What is a parallel computer
A parallel computer system uses a number of processors that can cooperate ....

Why distributed systems
* Purpose:
 - Provide services/resources
 - Integrate existing systems
 - Improve performance: But not through parallelism
* Challenges
 - Heteroginity
 - Distribution transparency: Location, relocation, migration
 - Interoperabiity, extensibility
   Declare clear interfaces, so we can add new modules
 - Scalability: Load, resources
 - Security
 - Reliability

Reading material

A brief introduction to distributed systems

Maarten van Steen, & Andrew S. Tanenbaum
Total 43 pages.

Introduction to Parallel Programming

Peter Pacheco

Chapter 2

Foster's methodology
Reference to 'Ian Foster' who wrote a book called 'Designing and Building Parallel Programs"
He provides an outline of steps, which is sometimes called "**Foster's methodology**"

1. **Partitioning**. Divide the computation to be performed and the data operated on by the computation into small tasks. The focus here should be on identifying tasks that can be executed in parallel.
2. **Communication**. Determine what communication needs to be carried out among the tasks identified in the previous step.
3. **Agglomeration** or aggregation. Combine tasks and communications identified in the first step into larger tasks. For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.
4. **Mapping**. Assign the composite tasks identified in the previous step to processes/threads. This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work

Introduction to distributed systems

Maarten van Steen, A.S. Tanenbaum
Pages 968-

Overview

History

Over time:
- Processing speed increases
- Memory size increase
- Cost decreases

Concept: Multi core processors.

We've started the reach the limit with single core processors, to make the next 'quantum jump' we're progressing to multi core processors to increase efficiency.

This obviously require new programming strategies to create programs who can utilize multi core processors.

New concept: High-speed computer networks.
Local-area networks (***LAN***) allows thousands of machines within a building or campus can be conneccted in such a way that small amounts of information can be transferred in a few microseconds or so.
Larger amounts of data can be moved between machines at rates of billions of bits per second (bps).

Wide-area networks (***WAN***) allow hundreds of millions of machines all over the earth to be connected at speeds varying from tens of thousands to hundreds of millions bps, and sometimes even faster.

Distributed system, characteristics, organization

Definition: ***Distributed system.***
A distributed system is a collection of autonomous computing elements that appears to its users as a single coherrent system.

2 characteristic features of a distributed system.
- ***Collection of autonomous computing elements***, each being able to behave independently of each other (referred to as node)
- ***Single coherrent system***, the users (people or applications) believe they are dealing with a single system

Characteristic 1: *Collection of autonomous computing elements*

Group membership (managing admission control)
- Open group, any node is allowed to join the distributed system, effectively meaning that
  it can send messages to any other node in the system.
- Closed group, only members of that group can communicate with each other and a
  separate mechanism is needed to let a node join or leave the group

Problems with group membership
- Authentification mechanism, to 'certify' and authentify nodes, can cause a scalability
  bottleneck with authentification
- If confidentiality is an issue while comm unication within the distributed system, we may be facing trust issues

Organization: **Distributed system** is often organized as an overlay network, meaning that between any two nodse there is always a communication path.

2 basic types of overlay networks
- Structured overly, each node has a well-defined set of neighbors with whom it can communicate. For example, the nodes are organized in a tree or logical ring
- Unstructured overlay, each node has a number of references to randomly selected other nodes

Peer-to-peer (**P2P**) networks, is a well known class of overlays.

The organization of these nodes requires special effort and that it is sometimes one of the more intricate part of distributed-systems management.

Characteristic 2: **Single coherrent system**

tl;dr both users and people utilizing the distributed system, may be under the impression they're using a single system.

A user may not know:
- To which exact node he's currently connected
- If a task has been sent off to another node.

Shouldn't matter where data is stored, or if the system replicate data to enhance performance.

Distribution tranparency
When the internals of a distributed system, is hidden from the end user.

Inevitably a node (a small part of the distributed system) will fail at some point. Quote "Leslie Lamport" describe a distributed system as "[...] one in which the failure of a computer you did not even know existed can render your own computer unusable."
This is a reality that's impossible to disguise and needs to be addressed.

...

The Datacenter as a computer: Designing Warehouse-

Scale Machines

"L. Barroso, U. Hölzle, P. Ranganathan,
The Datacenter as a Computer: Designing Warehouse-Scale Machines

The book describes warehouse-scale computers (WSCs), the computing platforms that power cloud computing and the web services we use every day.

Chapter 1 introduction

More and more services are shifting to a cloud based service.
This is because memory is rather limited but networking capabilites are increasing, i.e. mobile applications.

Advantage:

- Ease of management (no configuration or backups needed)
- Ubiquity of ccess
- Advantages to vendors
- Faster application development
- Unified update, updating 1 system rather than millions of end clients (lots of different hardware and software configurations)
- Data centers allow low cost per user
- Better utilization of resources (Only need to connect to active users)

In Warehouse-Scale computing, the program is an internet service, which may consist of tens or more individual programs that interact to implement complex end-user servies such as email, search, or maps.

WSCs are literally contained within massive buildings that resemble warehouses because of their massive size.

Do not mistake it for a data center, a WSC can be considered a type of data center.
A traditional data center can host a multitude of different computing systems, that are de-coupled and protecte from other systems in the same facility.
Both hardware, software and maintenance of units within a data center can vary, and if that is the case, those systems tend not to communicate with each other at all.

For WSCs, they're often built in-house and maintained by a single corporation, with homogenous software and hardware.
Single-organization control

WSCs initially designed for online data-intensive web workloads, they now also power public clouds.
Public clouds do run many small applications, lke a regular data center.

1.2 Cost efficiency at Scale

Cost examples of WSCs:
• Hosting facility capital
• Operational expenses (including power provisioning and energy costs)
• Hardware
• Software
• Management personel
• Repairs
• Downtime

Costefficiency is a primary metric in the design of WSCs.
And the cost needs to be weighed against the benefits.

1.3 Not just a collection of servers

tl;dr One big (humongous) system, which adds increased complexity for WSCs

Additional complexity and larger scale of the paplication domain manifests itself as:
• Deeper and less homogeneous storage hiearchy
• Higher fault rates
• Higher perfoamnce variability
• Greater emphasis on microsend latency tolerance

1.4 One data center vs several

Multiple data center, is when you to run a query across multiple servers located far apart.
An example would be a Content Distribution Network (CDN).
For the book, we consider 'multi-data center' as more analogous to a network of computers.

Massive difference between intra- and inter-data center communications, so software
development environment and if it evolves a lot, we may need to adjust our choice of
'machine boundaries'.

1.5 Why WSCs might matter to you

Amount of hardware threads on a single rack, can now a days exceed 4k, which is more than
a whole WSCs from the previous decade.

Explosive growth and popularity of Infrastructure-as-a-Service (IaaS) cloud computing
offerings have now made WSCs available to anyone with a credit card.

1.6 Architectural overview of WSCs

While the architectures of WSCs might change significantly between (and even inside

companies) we're only focusing on a high level abtraction, since the general layout,
is roughly the same.

1.6.1 Servers

See pictures

Interesting part is about 'rack units' to measure height of servers.

1.6.2 Storage

WSCs require massive storage, so tradeoffs must be made on how to implement it.
Directly Attached Storage (**DAS**) when the drives are plugged directly
into the compute servers
Network Attached Storage (**NAS**) when the drives are disaggregated.

Key metrics for global optimization
- Bandwidth
- IOPS (Input/Output Operations Per Second)
- Capacity
- Tail latency
- Total Cost of Ownership (TCO)

Distributed storage systems not only manage storage devices, but also provide unstructured and structured APIs for application developers.
Example, some newer generations of structured storage systems (such as a Spanner) provide an SQL-like interface and strong consistency models.

Fast evolution and improvement of data center network have reated a large gap between
network and disk performance, to the point that WSC designs can be dramatically simplified
to not consider disk locality.

Designers need to build balanced systems with a hiearchy of memory and storage technologies holistically considering the cluster-level aggregate capacity, bandwidth and latency. (Details concerning balancíng this in chapter 3)

1.6.3 Networking fabric

tl;dr network fabric intra and inter connectivity for racks

Prize of network switches with high port counts cost a LOT more than commodity rack switches.
Meaning that we tend to go for a cheaper option to connect the rack to itself, and then
pony up (to an extent) for the network switching.

Assume that there is relatively clarce cluster-level bandwidth resource, which a programmer must be aware of.

Tradeoff should money be spent on more servers, or better connectivity.
For now, assume that intra-rack connectivity is cheaper than inter-rack connectivity.

1.6.4 Buildings and infrastructure

WSCs require power, power equals excess heat, excess heat requires cooling.
Copious amounts of cooling.

There's a direct corrolation between the amount of power used, compared to the excess heat that is generated.
This has to be included in the thought process when designing WSCs.

Lots of techniques mentioned, but the one I was interested in wasn't, namely that excess heat can be used to warm up water, which can be distributed to households.
And thus used more efficiently than simply dissipating the heat, using fans, cooling
towers, chiller units, heat exhcangers, liquid cooling for accelerators.

Point is that this will affect the overall cost, and that it is proportional to the amount of power delivered, as well as imparing the performance of the compute equipments.

1.6.5 Power usage

Figure out how power is used within the WSC which can vary significantly
Example of one generations of WSCsdeployed at Google in 2017 by
main component groups:

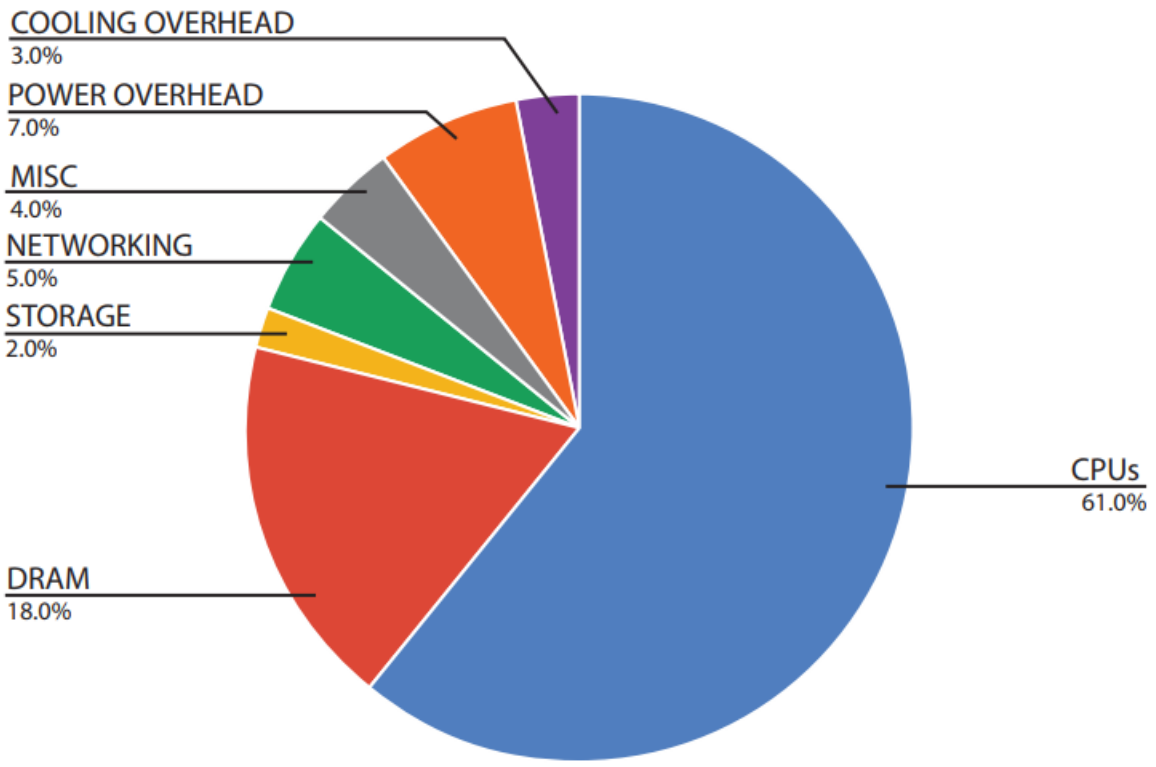


Figure 1.8: Approximate distribution of peak power usage by hardware subsystem in a
modern data center using late 2017 generation servers. The figure assumes two-socket
x86 servers and 12 DIMMs per server, and an average utilization of 80%.

Worth noting that technological improvements continue to optimize different areas,
as a result power requirements change accordingly, although CPU remained and continue to do so a dominant factor in power consumption.

Examples:
- *Sophosticated thermal management, means that CPU can run at higher power consumption, closer to their maximum power envelope.*
- *From power hungry FBDIMMs to DDR3 and DDR4 systems with better energy management.*
- *DRAM voltage has dropped from 1.8 V down to 1.2 V.*

1.6.6 Handling failures and repairs
tl;dr stuff breaks all the time.
Requires that internet services software tolerates relatively high component fault rates.
Disk drives can exhibit annualized failure rates higher than 4%.
Different deployments have reported between 1.2 and 16 average server-level

restarts per year.
More in chapter 2.

1.6.7 Overview of the book
tl;dr ignore or read the section in the book
Brief introduction to other chapters.

Chapter 2 Workloads and software infrastructure

2.1 Warehouse data center system stack

Chapter is about 'some' distinguishing characteristics of software that runs in large internet services and the system software and tools neeed for a complete computing platoform.

Some common terms
- Platform-level software: The common firmware, kernel, operating system distribution, and libraries expected to be present in all individual servers to abstract the hardware of a single machine and provide a basic machine abstraction layer.
- Cluster-level infrastructure: The collection of distributed systems software that manages resources and provides services at the cluster level. Ultimately, we consider these services as an operating system for a data center. Examples are distributed file systems, schedulers and remote procedure call (RPC) libraries, as well as programming models that simplify the usage of resources at the scale
  of data centers, such as MapReduce, Dryad, Hadoop, Sawzall, BigTable, Dynamo, Dremel, Spanner, and Chubby.
- Application-level software: Software that implements a specific service. It is often
  useful to further divide application-level software into online services and offline
  computations, since they tend to have different requirements. Examples of online
  services are Google Search, Gmail, and Google Maps. Offline computations are typically used in large-scale data analysis or as part of the pipeline that generates
  the data used in online services, for example, building an index of the web or processing satellite images to create map tiles for the online service

Figure 2.1 summarizes these layers as part of the overall software stack in WSCs

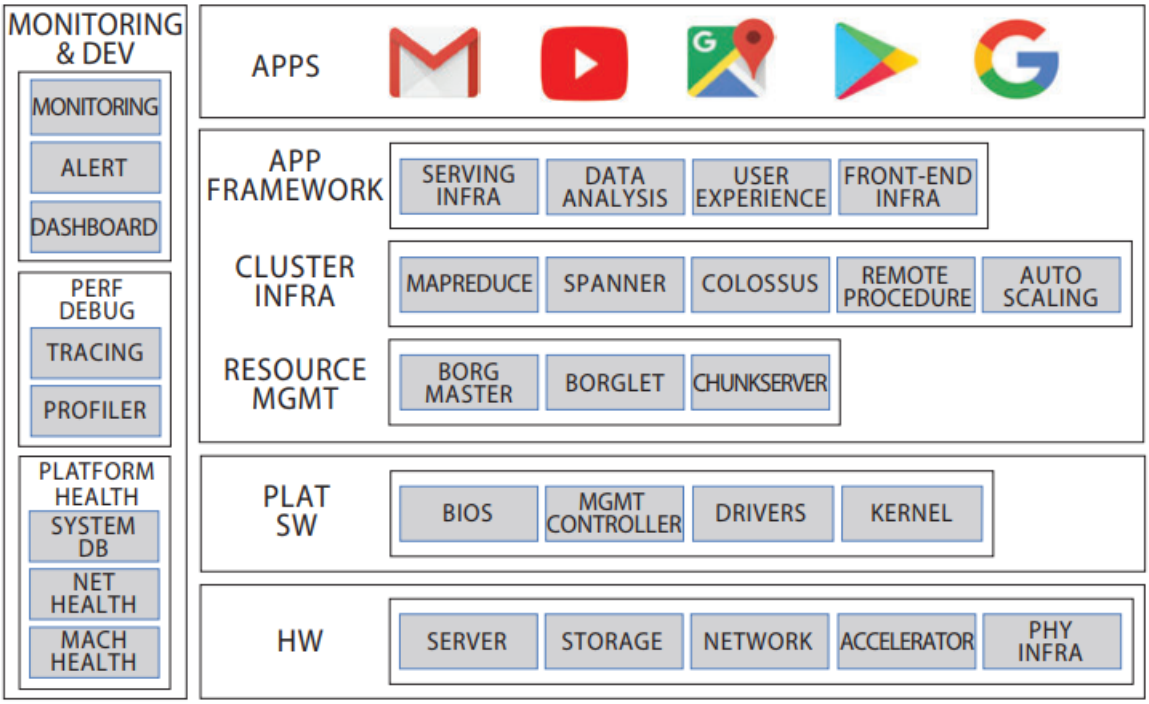


Figure 2.1: Overview of the Google software stack in warehouse-scale computers.

2.2 Platform-level software

Basic software system image running in WSC server nodes is very similar to those found on a regular enterprise server platform, so no details needed supplied.

Firmware, device drivers or operating system modules in WSC servers can be simplified to a larger degree than in a general purpose enterprise server.
Due to higher degree of homogenety in the hardware configuration of WSC servers,
less testing on other configurations are needed, to achieve an optimized software before implementing it.
Example:
Possible to tune transport or messaging parameters(timeouts, window sizes, and so on)
for higher communication efficiency.

Virtualization has become increasingly popular for WSC, especially for Infrastructure-
as-a-Service (Iaas) cloud offerings (VMware).

VM upside:
Provides a concise and portable interface to manage both the security and performance isolation of a customer's application, and allows multiple guest operating systems to19
co-exist with limited additional complexity.

VM downside:
Performance, particularly for I/O-intensive workloads.

Now a days overheads are improving, and the benefits of the VM model outweigh their costs.

Simplicity of VM encapsulation makes it easier to implement live migration (moving the VM
to another server, without bringing down the VM instance).
Allowing hardware or software infrastructure to be upgraded or repaired without impacting a
users computation.

Containers are an alternate popular abstraction that allow for isolation across multiple
workloads on a single OS instance. Because each container shares the host OS kernel and
associated binaries and libraries, they are more lightweight compared to VMs, smaller in size
and much faster to start.

2.3 Cluster-level infrastructure for software

An Operating System is needed to manage resources and basic services for a single computer.
Cluster-level infrastructure is a layer of software that does the same for a system composed of thousands of computers, networking, and storage.

Three broad groups of infrastructure software make up this layer. (??? no info)

2.3.1 Resource management

This component controls the mapping of user tasks to hardware resources, enforces priorities and quotas, and provides basic task management services.

In its simplest form, it is an interface to manually (and statically) allocate groups of machines to a given user or job.

Some programs (like kubernetes - www.kubernetes.io a popular open source program) orchestrates these functions for a container-based workload.

Users define and make requests, and the program finds the resources.

Cluster schedulers should consider:
- Resource needs (primary concern)
- Power limitations
- Energy usage optimization
- Account for emergencies (i.e. cooling and equipmenta failure)
- Maximize usage of hte provisioned data center power budget

Recent inclusion:
- Correlated failure domains and fault tolerance when making scheduling decisions

2.3.2 Cluster infrastructure

Every (almost) large-scale distributed application, needs a small set of basic functionalities.
Examples:
- Reliable distributed storage
- Remote Procedure Calls
- Message Parsing
- Cluster-level synchronization

Tasks that are amenable to a maual process in a small deployment require a significant amount of infrastract for efficient operations in a large-scale system.

Example:
- Software image distribution and configuration management
- Monitoring service performance and quality
- Triaging alarms for operators in emergency situations.

Performance debugging and optimization in systems of this scale need specialized solutions as well.

2.3.3 Application framework

super complex for a programmer to work with
Some higher-level operations or subsets or problems are common enough in large-scale services that it pays off to build targeted programming frameworks that simplify the development of new products.

Flume, MapReduce, Spanner, BigTable, and Dynamo, are good examples of pieces of infrastructure software that greatly improve programmer productivity by:
- automatically handling data partitioning
- Distribution
- Fault tolerance

Within their respective domains.

Equivalent software for the cloud could be; Google Kubernetes Engine (GKE), CloudSQL, AppEngine, etc.

2.4 Application-level sofware

2.4.1 Workload diversity

The type of work required for a WSC can change over time, which is the nature of progression, it is therefore generally not a good idea to build a specialized hardware solution.
This is because by the time it has been designed and are ready to be

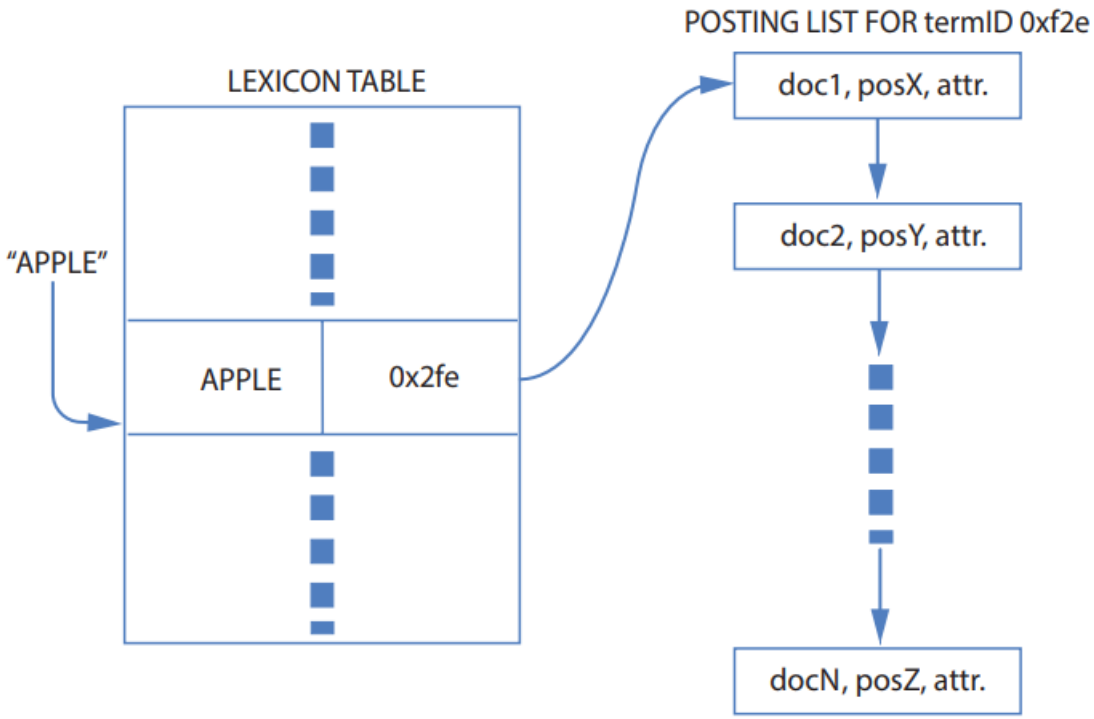
implemented, it may no longer be a good solution for the intended purpose.

There are cases where specialization has yelded significant gains.

2.4.2 Web search

'Needle in a haystack' problem.
Since the haystack (webpages) continue to expand at a rapid rate finding stuff becomes increasingly difficult.

An index has been created that takes input, sorts them into a lexicon talbe and from that table, generate a posting list for the termID.



Searches can contain different terms such as 'washington restaurants', the search algorithm must traverse the posting list for each term [washington, restaurants]

Too boring, lots of details about the specific process for web searches.

2.4.3 Video serving

IP video traffic represented 73% of the global internet in 2016, and was expected to grow to 83% in 2021.

Video processing is a HUGE market, and requirement on the internet.
Lots of ways to upload videos, do stuff with it, then be able to host and share it with people.

Three major cost componets:
• Compute costs due to video transcoding
• Storage costs for the video catalog (both original and transcoded versions)
• The network egress costs for sending trancoded videos to end users.

Improvements in video comppression codecs improve the strorage and egress cost, at the expense of higher compute costs.

Youtube use an algorithm to determine whether to use additional resources on a smaller video size of same perceptual quality.
This use of additional resources is made up for, by the large amount of views to watch the videos.

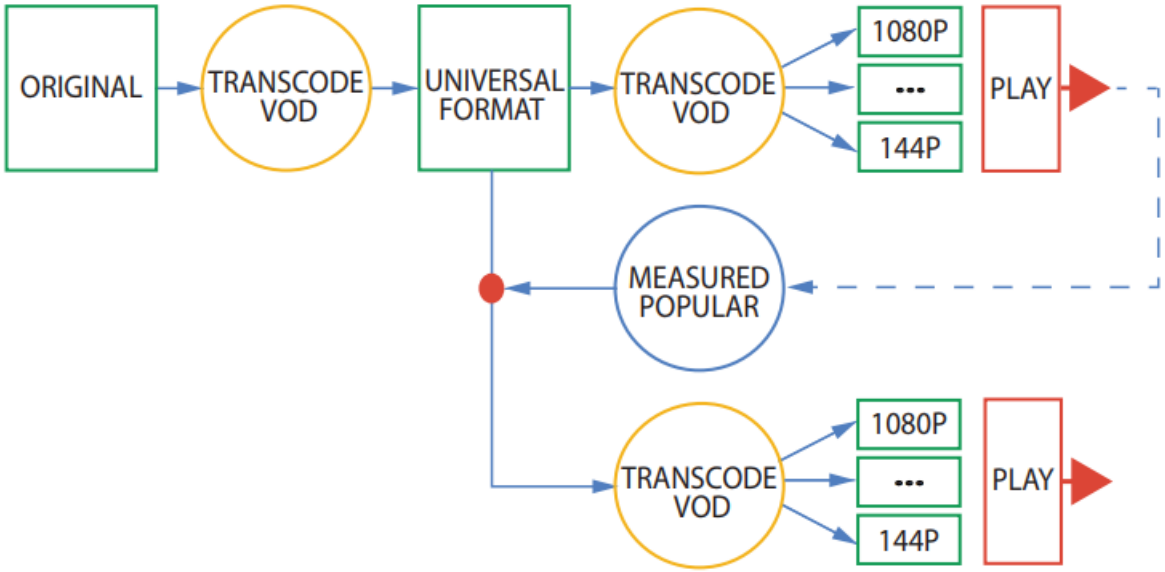


Figure 2.5: The YouTube video processing pipeline. Videos are transcoded multiple times depending
on their popularity. VOD = video on demand

2.4.4 Scholarly article similarity

If you want to search for scholarly articles, and want to find simmilarities with the article of your choice, there are a number of parameters to check.
One in particular is the citation of references, where if there is a high number of identical references (citations)between 2 scholarly articles, they must have a lot in common.

****

Page 27 continued from here.

2.4.5 Machine learning

Three kinds of Neural Networks (NNs) are popular today.

1. Multi-Layer Perceptrons (MLP): Each new layer is a set of nonlinear functions of
   weighted sum of all outputs (fully connected) from a prior one.
2. Convolutional Neural Networks (CNN): Each ensuing layer is a set of nonlinear
   functions of weighted sums of spatially nearby subsets of outputs from the prior layer,
   which also reuses the weights.
3. Recurrent Neural Networks (RNN): Each subsequent layer is a collection of nonlinear
   functions of weighted sums of outputs and the previous state. The most popular RNN
   is Long Short-Term Memory (LSTM). The art of the LSTM is in deciding what to
   forget and what to pass on as state to the next layer. The weights are reused across
   time steps.

2.5 Monitoring infrastructure

2.5.1 Service-level dashboards

2.5.2 Performance debugging tools

2.5.3 Platform-level health monitoring

2.6 WSC software tradeoffs

2.6.1 Data center vs desktop

2.6.2 Perfomance and availability toolbox

2.6.3 Buy vs. build

2.6.4 Tail-tolerance

2.6.5 Latency numbers that engineers should know

2.7 Cloud computing

2.7.1 WSC for public cloud services vs. internal workloads

2.7.2 Cloud native software

2.8 Information security at warehouse scale

Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows.*

Preparation questions

Week 02

Question 1
Today, what are the fundamental limits to the execution speed of a single CPU?

We've begun to reach the physical limitation for the:
- Size (transistor size on CPU)
- Cooling (a certain amount of power is converted to heat, and despite new technologies which greatly optimize it, we're still reaching the limit)

Question 2
Explain how caches can reduce the average memory access time.

Accessing the memory takes a very long time, but the bus length to transfer data can be rather large in comparison, so by taking a LOT of data and dumping in the cache, allows for faster access on average, significantly.
There are different policies about what to include, and when to change it,

Common examples for 'Cache Replacement policies':
First in First out (FIFO)
Last in First out (LIFO)
Least-frequently Used (LFU)
Most recently used (MRU)
Source list: https://en.wikipedia.org/wiki/Cache_replacement_policies

Question 3

How can instruction level parallelism improve execution speed? What are the main
approaches for implementing instruction level parallelism?

You do more in less time.
The concept is that you're running multiple processes/nodes at the same time, working tasks that can be handled in parallel, thus improving the overall execution speed.

Source: page 25 in the pacheco book
2.2.5 Instruction-level parallelism
Instruction-level parallelism, or ILP, attempts to improve processor performance by having multiple processor components or functional units simultaneously executing instructions. There are two main approaches to ILP: pipelining, in which
functional units are arranged in stages, and multiple issue, in which multiple instructions can be simultaneously initiated. Both approaches are used in virtually all
modern CPUs.

Question 4

Give one example of each type of machines in Flynn's taxonomy.

SISD (LC-3)
SIMD (Parallel computing such as multithreading)
MISD (Multiple computers having to agree with the same data, example space shuttle flights)
MIMD (Distributed systems (i.e. networking))

Week 03

This is the second set of preparation questions. Prepare class so that you can make a serious attempt at solving these questions.

Before attempting to answer the questions, read chapters 2.4 - 2.10 in Pacheco's book. Many of these preparation questions could be, and in

some cases, have been, used as exam problems.

---
* Question 1
[Repetition from 62588] What is a process? What is a thread? What are the main differences and similarities between the two.

Process: Executable program run on a processor
Thread: Can run as part of a program, as a thread on a core

---
* Question 2
What is a race condition? How can you remove a race condition from your parallel program?

When multiple processes/threads access the same data at the same time, which can cause issues.
Such as the variable 'derp' (= 10) is accessed from 20 different sources, at the same time, each of them adding 1, and then saving the result.
At the end, the final value in 'derp' is 11.
Solved by utilizing: Mutex & Semaphores

---
* Question 3
List some problems which may occur when doing input/output in parallel programs.

Readers/write
Deadlock
Race condition

---
* Question 4
What is the speedup of a parallel program?

Seriel / parallel
S = (T_serial / T_parallel)

So time in serial over parallel.

---
* Question 5
What is the main message of Amdahl's law?

Use it to find the theoretical max speedup, of a combined serial/parallel program.

---
* Question 6
[This is a typical exam question. It would correspond to 10% of an exam.]

You have one implementation of a program with a running time of six (6) time units and another with a running time of four (4) time units if both programs have the same input. What is the speedup of the faster implementation? Present all your calculations.

Solution:
6/4 = 1½

---
* Question 7
What are the main steps in writing a parallel program, according to Foster?

1. **Partition** the problem solution into tasks.
2. **Identify** the communication channels between the tasks.
3. **Aggregate** the tasks into composite tasks. (combine stuff)
4. **Map** the composite tasks to core

---

* Question 8
Measuring execution time can be challenging. List some problems you may face when measuring time in a program.

- Scheduling policies (OS)
- Communication between nodes
- Faulty clock cycle
- Speed can be different on different computers (i.e. hardware specs, processor and RAM)

Week 04
This is the third set of preparation questions. Prepare class so that you can make a serious attempt at solving these questions.

Before attempting to answer the questions, read chapters 3.1 - 3.3 in Pacheco's book.

---

* Question 1

Why do we need to start an MPI program with MPI_Init and end it with MPI_Finalize?

The init and finalize is to create and close a communication method between the nodes.

---

* Question 2

In MPI, what is a communicator?

Holds a group of processes that can communicate with each other. All message passing calls in MPI must have a specific
communicator to use the call with. An example of a communicator handle is MPI_COMM_WORLD. MPI_COMM_WORLD
is the default communicator that contains all processes available for use.

Week 05

This is the fourth set of preparation questions. Prepare class so that you can make a serious attempt at solving these questions.

Before attempting to answer the questions, read chapters 3.4 - 3.8 in Pacheco's book.

* Question 1

What is the point of collective communication in MPI? Can't all possible communication patterns be implemented with point-to-point communication primitives?

Technically yes, it's possible.
But it's highly inefficient, so tl;dr Superflous steps, & 'Efficiency'

---

* Question 2

In MPI, there is an MPI_Barrier primitive? What does it do? Isn't it odd for MPI to have such a primitive? In what situations is it useful?

Barrier is a function, that ensures that all nodes reach the same point and stop.
It can be useful to prevent one or more nodes progressing beyond a point where the other nodes haven't processed the data to be shared.

Week 06

This is the fifth set of preparation questions. Prepare class so that you can make a serious attempt at solving these questions.

Before attempting to answer the questions, read chapters 5.1 - 5.4 in Pacheco's book.

* Question 1
What does it mean to parallelize code with OpenMP?

OpenMP is a 'shared memory' parallel programming model.
Meaning that you can use 'OpenMP' to turn parts of a serial code into a parallel version.
'OpenMP' is a specific library which can implement this function with a higher abstraction level.

---

* Question 2
Why are #pragma directives needed in OpenMP?

They're used to communicate between the different process threads
Such as running specific code chunks in parallel, or even atomic operations.
Concept is about having a higher abstraction level, for multiple threads at the same time

---

Week 07

This is the seventh set of preparation questions. Prepare class so that you can make a serious attempt at solving these questions.

Before attempting to answer the questions, read chapters 5.1 - 5.8 in Pacheco's book.

---

* Question 1

Why can you not break out of a parallel for loop in OpenMP using return or a goto?

tl;dr for-loops in OpenMP are rather restrictive, because they're working on same data, so even if one of the threads break/return, the others won't (even if they're supposed to).

In a parallel for-loop any data dependency in C will be disregarded

page: 226 in the Pacheco book
The variables and expressions in this template are subject to some fairly obvious restrictions:
- The variable index must have integer or pointer type (e.g., it can't be a float).
- The expressions start, end, and incr must have a compatible type. For example,
  if index is a pointer, then incr must have integer type.
- The expressions start, end, and incr must not change during execution of the loop.
- During execution of the loop, the variable index can only be modified by the "increment expression" in the for statement.

---

* Question 2

Why are there different types of OpenMP scheduling?

Different scheduling types have different runtimes, so it's important to choose the type that would fit your scenario the best.

- Static
- Dynamic
- auto/guided
- runtime

If you don't define it within the program it merely use a 'default' scheme (uncertain of specific formatting)

---

* Question 3

How do you define a critical section in OpenMP?

For sections that require preventions of race conditions, use 'critical' directive.
pragma omp critical

This forces the threads to run in serial, rather than parallel, for that specific section.

Week 08 *

This is the preparation questions for the eight week. Prepare class so that you can make a serious attempt at solving these questions.
Before attempting to answer the questions, read chapters 6 - 6.2.8 in Pacheco's book.

* Question 1
Solve problem 6.1 in Pacheco.
6.1 In each iteration of the serial n-body solver, we first compute the total force on each particle, and then we compute the position and velocity of each particle.
Would it be possible to reorganize the calculations so that in each iteration we did all of the calculations for each particle before proceeding to the next particle? That is, could we use the following pseudocode?

```
for each timestep
   for each particle {
      Compute total force on particle;
      Find position and velocity of particle;
      Print position and velocity of particle;
   }
```

If so, what other modifications would we need to make to the solver?
If not, why not?

If there is data dependency then no, else yes.
I can't provide a proper answer based on that piece of pseudo code.

---

* Question 2
Solve problem 6.6 in Pacheco.
6.6 In our discussion of the OpenMP implementation of the basic n-body solver, we observed that the implied barrier after the output statement wasn't necessary. We could therefore modify the single directive with a nowait clause. It's possible to also eliminate the implied barriers at the ends of the two for each particle q loops by modifying for directives with nowait clauses. Would doing this cause any problems?
Explain your answer.

Week 09

Prepare class so that you can make a serious attempt at solving these questions.

Before attempting to answer the questions, read Maarten van Steen & Andrew S. Tanenbaum:A brief introduction to distributed systems, Computing (2016) vol. 98. https://findit.dtu.dk/en/catalog/2335112463

For this week read pages 967 - 986, up to section 4 but not into section 4. Ignore the formulas on page 980-981.

* Question 1
Compare a parallel and distributed system. What are the differences in purpose between the two types of systems.

Parallel system is mainly created in such a way that multiple processes/threads can actively worn on the same problem in parallel, thus increasing efficiency.

For distributed systems, each 'node' can work independently of one another, on multiple different problems.
Obviously it can also work as a 'parallel' program, but it would require some additional communication between the nodes to achieve this

---
---

* Question 2

What is the role of the middleware?

To assist the development of distributed applications, distributed systems are often organized to have a separate layer of software that is logically placed on top of the respective operating systems of the computers that are part of the system. In a sense, middleware is the same to a distributed system as what an operating system is to a computer: a manager of resources offering its applications to efficiently share and deploy those resources across a network. Next to resource management, it offers services that can also be found in most operating systems, including:
– Facilities for interapplication communication.

– Security services.
– Accounting services.
– Masking of and recovery from failures.
The main difference with their operating-system equivalents, is that middleware services are offered in a networked environment. Note also that most services are useful to many applications. In this sense, middleware can also be viewed as a container of commonly used components and functions that now no longer have to be implemented by applications separately.

Week 10

Prepare class so that you can make a serious attempt at solving these questions.

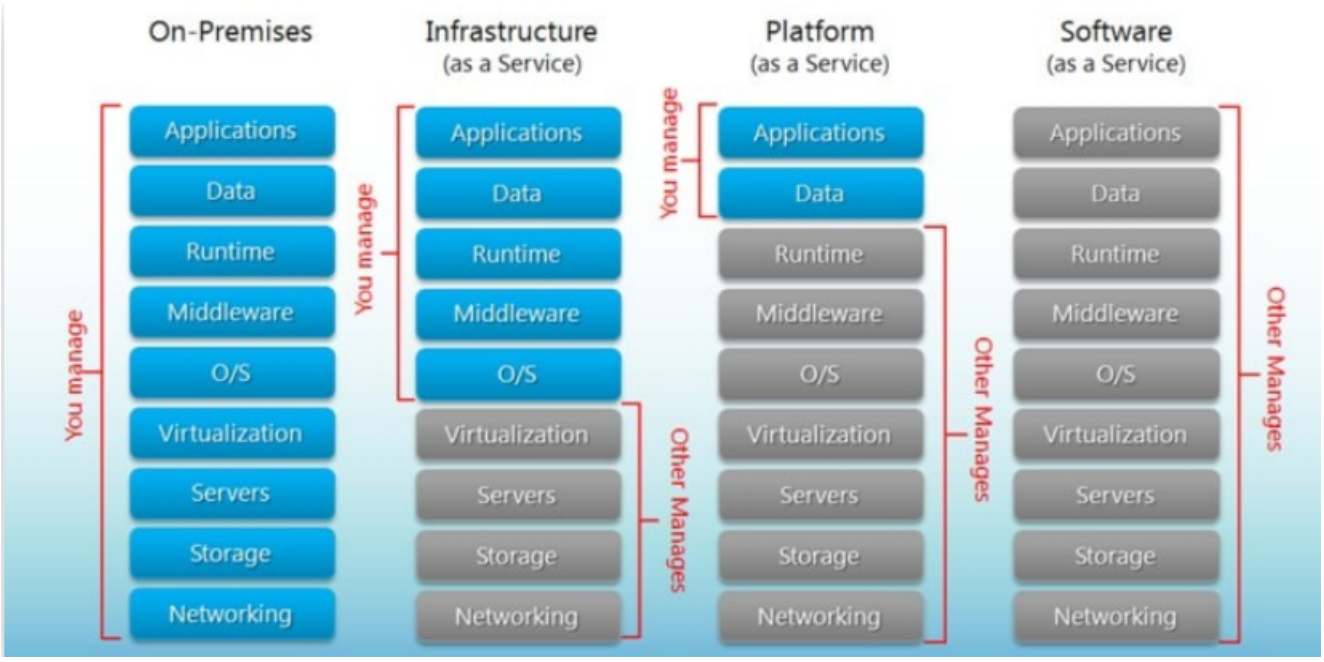
Before attempting to answer the questions, read Maarten van Steen & Andrew S. Tanenbaum:A brief introduction to distributed systems, Computing (2016) vol. 98. https://findit.dtu.dk/en/catalog/2335112463

For this week read all of the handout. Ignore the formulas on page 980-981.

* Question 1
What are the pros and cons of IaaS, Paas, and SaaS respectively?

3 Main types of cloud computing.
Iaas - Infrastructure as a Service
PaaS: Platform as a Service
SaaS: Software as a Service

IaaS: cloud-based services, pay-as-you-go for services such as storage, networking, and virtualization.
PaaS: hardware and software tools available over the internet.
SaaS: software that's available via a third-party over the internet.
On-premise: software that's installed in the same building as your business



---

* Question 2

Explain the ACID properties and how they can be maintained.

This all-or-nothing property of transactions is one of the four characteristic properties
that transactions have.
More specifically, transactions adhere to the so-called ACID properties:

– Atomic To the outside world, the transaction happens indivisibly
– Consistent The transaction does not violate system invariants
– Isolated Concurrent transactions do not interfere with each other
– Durable Once a transaction commits, the changes are permanent
Source: Page 993 (aka. 27) of 'A brief introduction to distributed systems'

This allows a way to validate a transaction from the time of data input to the final output, which is stored in a database

Make use of middleware to 'maintain' and support the functions.

In distributed systems, transactions are often constructed as a number of sub transactions, jointly forming a nested transaction as shown in Fig. 9. The top-level transaction may fork off children that run in parallel with one another, on different machines, to gain performance or simplify programming.

This allows a way to validate a transaction from the time of data input to the final output, which is stored in a database Make use of middleware to 'maintain' and support the functions.

Week 11

Prepare class so that you can make a serious attempt at solving these questions.

Before attempting to answer the questions, read "L. Barroso, U. Hölzle, P. Ranganathan,
The Datacenter as a Computer: Designing Warehouse-Scale Machines, ch 1-2"

* Question 1

What are the high-level building blocks for a warehouse-scale computer?

- Specific hardware (chipsets, storage)
- Networking infrastructure (communication between racks, aka network switches)
- Cooling capacity and associated hardware
- Power usage

---
* Question 2
What is churn?

"Workload churn"
Most common term is the 'rate of members of a group leaving the group'
Which in this case refers to how many members are leaving the service, thus negatively
impacting the amount of work required

Week 12

Prepare class so that you can make a serious attempt at solving these questions.

Before attempting to answer the questions, read "Balouek-Thomert D, Renart EG, Zamani AR, Simonet A, Parashar M. Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows. The International Journal of High Performance Computing Applications. 2019;33(6):1159-1174. doi:10.1177/1094342019877383."

* Question 1

Communication is costly in many systems, including IoT system. How can the computing continuum reduce communication and its cost?

By running lots of processes along the edge

---
* Question 2

What type of systems can be part of a computing continuum?

Databar

tl;dr

Databar1
Cachetest, how does cache work

Databar2
Dotprod, explain concept of moving from serial to parallel

Databar3
MPI (Message Passing Interface)

Databar 1

Instructions for completing the databar exercise.

Make sure you have a linux terminal on your computer (bash, ubuntu, ...)
Open thinlinc, navigate to learn.inside.dtu.dk and download 'cachetest.c' and place it in a folder you'll remember.
Close thinlinc.

Need to access the LSF 10 Cluster, which could be done via thinlinc, but easier with a linux terminal.
Follow instructions here: http://www.cc.dtu.dk/?page_id=
2501

Open the linux terminal, and type in:

$ ssh userid@login1.gbar.dtu.dk

Remember to change 'userid' to your student id,and then type in your password, if you screw up, hit ctrl+c to abort.

Type in 'linuxsh' to access another node.
Use 'cd' and 'ls -l' to navigate to the correct folder.

Compile the cachetest.c with this command.
gcc -O0 cachetest.c
(or just gcc -o a.out cachetest.c)

Now type in
./a.out
Hit enter, and let it complete.
You should've now completed the databar exercise.

Databar 2

Instructions for completing the databar exercise.

Make sure you have a linux terminal on your computer (bash, ubuntu, ...)
Open thinlinc, navigate to learn.inside.dtu.dk and download 'dotprod.c' and place it in a folder you'll remember.
Close thinlinc.

Need to access the LSF 10 Cluster, which could be done via thinlinc, but easier with a linux terminal.
Follow instructions here: http://www.cc.dtu.dk/?page_id=
2501

Open the linux terminal, and type in:
http://www.cc.dtu.dk/?page_id=
2501
ssh userid@login1.gbar.dtu.dk
Remember to change 'userid' to your student id,and then type in your password, if you screw up, hit ctrl+c to abort.

Type in 'linuxsh' to access another node.
Use 'cd' and 'ls -l' to navigate to the correct folder.
Compile the dotprod.c with this command.
gcc -o dotprod dotprod.c

Now type in
./dotprod
Hit enter, and let it complete.
You should've now completed the databar exercise.

Size (bytes): 4096, Stride (bytes): 4, read+write: -2.14 ns
Size (bytes): 4096, Stride (bytes): 8, read+write: -2.43 ns
Size (bytes): 4096, Stride (bytes): 16, read+write: -2.47 ns
Size (bytes): 4096, Stride (bytes): 32, read+write: -2.31 ns
Size (bytes): 4096, Stride (bytes): 64, read+write: -2.21 ns

Size (bytes): 4096, Stride (bytes): 128, read+write: -2.55 ns
Size (bytes): 4096, Stride (bytes): 256, read+write: -2.42 ns
Size (bytes): 4096, Stride (bytes): 512, read+write: -2.32 ns
Size (bytes): 8192, Stride (bytes):   4, read+write: -2.76 ns
Size (bytes): 8192, Stride (bytes):   8, read+write: -2.28 ns
Size (bytes): 8192, Stride (bytes): 16, read+write: -2.51 ns
Size (bytes): 8192, Stride (bytes): 32, read+write: -2.47 ns
Size (bytes): 8192, Stride (bytes): 64, read+write: -2.39 ns
Size (bytes): 8192, Stride (bytes): 128, read+write: -2.24 ns
Size (bytes): 8192, Stride (bytes): 256, read+write: -2.82 ns
Size (bytes): 8192, Stride (bytes): 512, read+write: -2.85 ns
Size (bytes): 16384, Stride (bytes):   4, read+write: -2.68 ns
Size (bytes): 16384, Stride (bytes):   8, read+write: -2.56 ns
Size (bytes): 16384, Stride (bytes): 16, read+write: -2.49 ns
Size (bytes): 16384, Stride (bytes): 32, read+write: -2.58 ns
Size (bytes): 16384, Stride (bytes): 64, read+write: -2.42 ns
Size (bytes): 16384, Stride (bytes): 128, read+write: -2.53 ns
Size (bytes): 16384, Stride (bytes): 256, read+write: -2.34 ns
Size (bytes): 16384, Stride (bytes): 512, read+write: -2.68 ns
Size (bytes): 32768, Stride (bytes):   4, read+write: -2.66 ns
Size (bytes): 32768, Stride (bytes):   8, read+write: -2.65 ns
Size (bytes): 32768, Stride (bytes): 16, read+write: -2.58 ns
Size (bytes): 32768, Stride (bytes): 32, read+write: -2.56 ns
Size (bytes): 32768, Stride (bytes): 64, read+write: -2.42 ns
Size (bytes): 32768, Stride (bytes): 128, read+write: -2.49 ns
Size (bytes): 32768, Stride (bytes): 256, read+write: -2.43 ns
Size (bytes): 32768, Stride (bytes): 512, read+write: -2.36 ns
Size (bytes): 65536, Stride (bytes):   4, read+write: -2.59 ns
Size (bytes): 65536, Stride (bytes):   8, read+write: -2.62 ns
Size (bytes): 65536, Stride (bytes): 16, read+write: -2.56 ns
Size (bytes): 65536, Stride (bytes): 32, read+write: -2.09 ns
Size (bytes): 65536, Stride (bytes): 64, read+write: -0.18 ns
Size (bytes): 65536, Stride (bytes): 128, read+write: -0.06 ns
Size (bytes): 65536, Stride (bytes): 256, read+write: -0.06 ns
Size (bytes): 65536, Stride (bytes): 512, read+write: -0.06 ns
Size (bytes): 131072, Stride (bytes):   4, read+write: -2.71 ns
Size (bytes): 131072, Stride (bytes):   8, read+write: -2.66 ns
Size (bytes): 131072, Stride (bytes): 16, read+write: -2.59 ns
Size (bytes): 131072, Stride (bytes): 32, read+write: -2.04 ns
Size (bytes): 131072, Stride (bytes): 64, read+write: -0.18 ns
Size (bytes): 131072, Stride (bytes): 128, read+write: -0.18 ns
Size (bytes): 131072, Stride (bytes): 256, read+write: -0.12 ns
Size (bytes): 131072, Stride (bytes): 512, read+write: 0.06 ns
Size (bytes): 262144, Stride (bytes):   4, read+write: -2.75 ns
Size (bytes): 262144, Stride (bytes):   8, read+write: -2.77 ns
Size (bytes): 262144, Stride (bytes): 16, read+write: -2.44 ns
Size (bytes): 262144, Stride (bytes): 32, read+write: -1.72 ns
Size (bytes): 262144, Stride (bytes): 64, read+write: 0.50 ns

Size (bytes):   262144, Stride (bytes): 128, read+write: 0.52 ns
Size (bytes):   262144, Stride (bytes): 256, read+write: 0.66 ns
Size (bytes):   262144, Stride (bytes): 512, read+write: 0.33 ns
Size (bytes):   524288, Stride (bytes):   4, read+write: -2.68 ns
Size (bytes):   524288, Stride (bytes):   8, read+write: -2.74 ns
Size (bytes):   524288, Stride (bytes): 16, read+write: -2.39 ns
Size (bytes):   524288, Stride (bytes): 32, read+write: -1.42 ns
Size (bytes):   524288, Stride (bytes): 64, read+write: 1.12 ns
Size (bytes):   524288, Stride (bytes): 128, read+write: 1.63 ns
Size (bytes):   524288, Stride (bytes): 256, read+write: 2.01 ns
Size (bytes):   524288, Stride (bytes): 512, read+write: 1.17 ns
Size (bytes): 1048576, Stride (bytes):   4, read+write: -2.71 ns
Size (bytes): 1048576, Stride (bytes):   8, read+write: -2.70 ns
Size (bytes): 1048576, Stride (bytes): 16, read+write: -2.41 ns
Size (bytes): 1048576, Stride (bytes): 32, read+write: -1.47 ns
Size (bytes): 1048576, Stride (bytes): 64, read+write: 1.12 ns
Size (bytes): 1048576, Stride (bytes): 128, read+write: 1.57 ns
Size (bytes): 1048576, Stride (bytes): 256, read+write: 2.05 ns
Size (bytes): 1048576, Stride (bytes): 512, read+write: 1.17 ns
Size (bytes): 2097152, Stride (bytes):   4, read+write: -2.73 ns
Size (bytes): 2097152, Stride (bytes):   8, read+write: -2.67 ns
Size (bytes): 2097152, Stride (bytes): 16, read+write: -2.30 ns
Size (bytes): 2097152, Stride (bytes): 32, read+write: -1.52 ns
Size (bytes): 2097152, Stride (bytes): 64, read+write: 1.03 ns
Size (bytes): 2097152, Stride (bytes): 128, read+write: 1.54 ns
Size (bytes): 2097152, Stride (bytes): 256, read+write: 2.09 ns
Size (bytes): 2097152, Stride (bytes): 512, read+write: 1.14 ns
Size (bytes): 4194304, Stride (bytes):   4, read+write: -2.76 ns
Size (bytes): 4194304, Stride (bytes):   8, read+write: -2.70 ns
Size (bytes): 4194304, Stride (bytes): 16, read+write: -2.27 ns
Size (bytes): 4194304, Stride (bytes): 32, read+write: -1.41 ns
Size (bytes): 4194304, Stride (bytes): 64, read+write: 1.00 ns
Size (bytes): 4194304, Stride (bytes): 128, read+write: 1.40 ns
Size (bytes): 4194304, Stride (bytes): 256, read+write: 1.99 ns
Size (bytes): 4194304, Stride (bytes): 512, read+write: 0.95 ns
Size (bytes): 8388608, Stride (bytes):   4, read+write: -2.66 ns
Size (bytes): 8388608, Stride (bytes):   8, read+write: -2.73 ns
Size (bytes): 8388608, Stride (bytes): 16, read+write: -2.38 ns
Size (bytes): 8388608, Stride (bytes): 32, read+write: -1.38 ns
Size (bytes): 8388608, Stride (bytes): 64, read+write: 1.03 ns
Size (bytes): 8388608, Stride (bytes): 128, read+write: 1.50 ns
Size (bytes): 8388608, Stride (bytes): 256, read+write: 1.89 ns
Size (bytes): 8388608, Stride (bytes): 512, read+write: 0.79 ns
Size (bytes): 16777216, Stride (bytes):   4, read+write: -2.66 ns
Size (bytes): 16777216, Stride (bytes):   8, read+write: -2.60 ns
Size (bytes): 16777216, Stride (bytes): 16, read+write: -2.27 ns
Size (bytes): 16777216, Stride (bytes): 32, read+write: -1.39 ns
Size (bytes): 16777216, Stride (bytes): 64, read+write: 1.19 ns

Size (bytes): 16777216, Stride (bytes): 128, read+write: 1.59 ns
Size (bytes): 16777216, Stride (bytes): 256, read+write: 1.99 ns
Size (bytes): 16777216, Stride (bytes): 512, read+write: 0.95 ns
Size (bytes): 33554432, Stride (bytes):   4, read+write: -2.60 ns
Size (bytes): 33554432, Stride (bytes):   8, read+write: -2.41 ns
Size (bytes): 33554432, Stride (bytes): 16, read+write: -1.43 ns
Size (bytes): 33554432, Stride (bytes): 32, read+write: 0.66 ns
Size (bytes): 33554432, Stride (bytes): 64, read+write: 5.56 ns
Size (bytes): 33554432, Stride (bytes): 128, read+write: 7.05 ns
Size (bytes): 33554432, Stride (bytes): 256, read+write: 7.95 ns
Size (bytes): 33554432, Stride (bytes): 512, read+write: 6.56 ns
Size (bytes): 67108864, Stride (bytes):   4, read+write: -2.54 ns
Size (bytes): 67108864, Stride (bytes):   8, read+write: -2.21 ns
Size (bytes): 67108864, Stride (bytes): 16, read+write: -1.09 ns
Size (bytes): 67108864, Stride (bytes): 32, read+write: 1.99 ns
Size (bytes): 67108864, Stride (bytes): 64, read+write: 7.45 ns
Size (bytes): 67108864, Stride (bytes): 128, read+write: 11.03 ns
Size (bytes): 67108864, Stride (bytes): 256, read+write: 14.21 ns
Size (bytes): 67108864, Stride (bytes): 512, read+write: 12.32 ns

Another attempt

Size (bytes):   4096, Stride (bytes):   4, read+write: -2.05 ns
Size (bytes):   4096, Stride (bytes):   8, read+write: -2.43 ns
Size (bytes):   4096, Stride (bytes): 16, read+write: -2.40 ns
Size (bytes):   4096, Stride (bytes): 32, read+write: -2.42 ns
Size (bytes):   4096, Stride (bytes): 64, read+write: -2.34 ns
Size (bytes):   4096, Stride (bytes): 128, read+write: -2.50 ns
Size (bytes):   4096, Stride (bytes): 256, read+write: -2.51 ns
Size (bytes):   4096, Stride (bytes): 512, read+write: -2.32 ns
Size (bytes):   8192, Stride (bytes):   4, read+write: -2.60 ns
Size (bytes):   8192, Stride (bytes):   8, read+write: -2.59 ns
Size (bytes):   8192, Stride (bytes): 16, read+write: -2.63 ns
Size (bytes):   8192, Stride (bytes): 32, read+write: -2.53 ns
Size (bytes):   8192, Stride (bytes): 64, read+write: -2.39 ns
Size (bytes):   8192, Stride (bytes): 128, read+write: -2.33 ns
Size (bytes):   8192, Stride (bytes): 256, read+write: -2.82 ns
Size (bytes):   8192, Stride (bytes): 512, read+write: -2.51 ns
Size (bytes):   16384, Stride (bytes):   4, read+write: -2.63 ns
Size (bytes):   16384, Stride (bytes):   8, read+write: -2.60 ns
Size (bytes):   16384, Stride (bytes): 16, read+write: -2.48 ns
Size (bytes):   16384, Stride (bytes): 32, read+write: -2.58 ns
Size (bytes):   16384, Stride (bytes): 64, read+write: -2.48 ns
Size (bytes):   16384, Stride (bytes): 128, read+write: -2.47 ns
Size (bytes):   16384, Stride (bytes): 256, read+write: -2.25 ns
Size (bytes):   16384, Stride (bytes): 512, read+write: -2.70 ns
Size (bytes):   32768, Stride (bytes):   4, read+write: -2.58 ns
Size (bytes):   32768, Stride (bytes):   8, read+write: -2.66 ns

Size (bytes):   32768, Stride (bytes): 16, read+write: -2.63 ns
Size (bytes):   32768, Stride (bytes): 32, read+write: -2.48 ns
Size (bytes):   32768, Stride (bytes): 64, read+write: -2.43 ns
Size (bytes):   32768, Stride (bytes): 128, read+write: -2.44 ns
Size (bytes):   32768, Stride (bytes): 256, read+write: -2.41 ns
Size (bytes):   32768, Stride (bytes): 512, read+write: -2.25 ns
Size (bytes):   65536, Stride (bytes):   4, read+write: -2.56 ns
Size (bytes):   65536, Stride (bytes):   8, read+write: -2.64 ns
Size (bytes):   65536, Stride (bytes): 16, read+write: -2.59 ns
Size (bytes):   65536, Stride (bytes): 32, read+write: -2.03 ns
Size (bytes):   65536, Stride (bytes): 64, read+write: -0.12 ns
Size (bytes):   65536, Stride (bytes): 128, read+write: -0.06 ns
Size (bytes):   65536, Stride (bytes): 256, read+write: 0.00 ns
Size (bytes):   65536, Stride (bytes): 512, read+write: -0.06 ns
Size (bytes):   131072, Stride (bytes):   4, read+write: -2.65 ns
Size (bytes):   131072, Stride (bytes):   8, read+write: -2.63 ns
Size (bytes):   131072, Stride (bytes): 16, read+write: -2.57 ns
Size (bytes):   131072, Stride (bytes): 32, read+write: -2.01 ns
Size (bytes):   131072, Stride (bytes): 64, read+write: -0.24 ns
Size (bytes):   131072, Stride (bytes): 128, read+write: -0.06 ns
Size (bytes):   131072, Stride (bytes): 256, read+write: -0.12 ns
Size (bytes):   131072, Stride (bytes): 512, read+write: 0.06 ns
Size (bytes):   262144, Stride (bytes):   4, read+write: -2.74 ns
Size (bytes):   262144, Stride (bytes):   8, read+write: -2.71 ns
Size (bytes):   262144, Stride (bytes): 16, read+write: -2.38 ns
Size (bytes):   262144, Stride (bytes): 32, read+write: -1.73 ns
Size (bytes):   262144, Stride (bytes): 64, read+write: 0.42 ns
Size (bytes):   262144, Stride (bytes): 128, read+write: 0.35 ns
Size (bytes):   262144, Stride (bytes): 256, read+write: 0.42 ns
Size (bytes):   262144, Stride (bytes): 512, read+write: 0.41 ns
Size (bytes):   524288, Stride (bytes):   4, read+write: -2.62 ns
Size (bytes):   524288, Stride (bytes):   8, read+write: -2.68 ns
Size (bytes):   524288, Stride (bytes): 16, read+write: -2.32 ns
Size (bytes):   524288, Stride (bytes): 32, read+write: -1.40 ns
Size (bytes):   524288, Stride (bytes): 64, read+write: 1.10 ns
Size (bytes):   524288, Stride (bytes): 128, read+write: 1.62 ns
Size (bytes):   524288, Stride (bytes): 256, read+write: 1.79 ns
Size (bytes):   524288, Stride (bytes): 512, read+write: 1.05 ns
Size (bytes): 1048576, Stride (bytes):   4, read+write: -2.59 ns
Size (bytes): 1048576, Stride (bytes):   8, read+write: -2.62 ns
Size (bytes): 1048576, Stride (bytes): 16, read+write: -2.32 ns
Size (bytes): 1048576, Stride (bytes): 32, read+write: -1.42 ns
Size (bytes): 1048576, Stride (bytes): 64, read+write: 1.22 ns
Size (bytes): 1048576, Stride (bytes): 128, read+write: 1.57 ns
Size (bytes): 1048576, Stride (bytes): 256, read+write: 1.93 ns
Size (bytes): 1048576, Stride (bytes): 512, read+write: 1.27 ns
Size (bytes): 2097152, Stride (bytes):   4, read+write: -2.56 ns
Size (bytes): 2097152, Stride (bytes):   8, read+write: -2.52 ns

Size (bytes): 2097152, Stride (bytes): 16, read+write: -2.20 ns
Size (bytes): 2097152, Stride (bytes): 32, read+write: -1.37 ns
Size (bytes): 2097152, Stride (bytes): 64, read+write: 1.29 ns
Size (bytes): 2097152, Stride (bytes): 128, read+write: 1.68 ns
Size (bytes): 2097152, Stride (bytes): 256, read+write: 1.83 ns
Size (bytes): 2097152, Stride (bytes): 512, read+write: 1.14 ns
Size (bytes): 4194304, Stride (bytes):   4, read+write: -2.56 ns
Size (bytes): 4194304, Stride (bytes):   8, read+write: -2.50 ns
Size (bytes): 4194304, Stride (bytes): 16, read+write: -2.03 ns
Size (bytes): 4194304, Stride (bytes): 32, read+write: -1.33 ns
Size (bytes): 4194304, Stride (bytes): 64, read+write: 1.17 ns
Size (bytes): 4194304, Stride (bytes): 128, read+write: 1.68 ns
Size (bytes): 4194304, Stride (bytes): 256, read+write: 1.89 ns
Size (bytes): 4194304, Stride (bytes): 512, read+write: 1.03 ns
Size (bytes): 8388608, Stride (bytes):   4, read+write: -2.53 ns
Size (bytes): 8388608, Stride (bytes):   8, read+write: -2.51 ns
Size (bytes): 8388608, Stride (bytes): 16, read+write: -2.11 ns
Size (bytes): 8388608, Stride (bytes): 32, read+write: -1.25 ns
Size (bytes): 8388608, Stride (bytes): 64, read+write: 1.19 ns
Size (bytes): 8388608, Stride (bytes): 128, read+write: 1.50 ns
Size (bytes): 8388608, Stride (bytes): 256, read+write: 2.09 ns
Size (bytes): 8388608, Stride (bytes): 512, read+write: 1.03 ns
Size (bytes): 16777216, Stride (bytes):   4, read+write: -2.57 ns
Size (bytes): 16777216, Stride (bytes):   8, read+write: -2.53 ns
Size (bytes): 16777216, Stride (bytes): 16, read+write: -2.13 ns
Size (bytes): 16777216, Stride (bytes): 32, read+write: -1.15 ns
Size (bytes): 16777216, Stride (bytes): 64, read+write: 1.19 ns
Size (bytes): 16777216, Stride (bytes): 128, read+write: 1.75 ns
Size (bytes): 16777216, Stride (bytes): 256, read+write: 2.09 ns
Size (bytes): 16777216, Stride (bytes): 512, read+write: 0.87 ns
Size (bytes): 33554432, Stride (bytes):   4, read+write: -2.34 ns
Size (bytes): 33554432, Stride (bytes):   8, read+write: -2.10 ns
Size (bytes): 33554432, Stride (bytes): 16, read+write: -1.23 ns
Size (bytes): 33554432, Stride (bytes): 32, read+write: 0.99 ns
Size (bytes): 33554432, Stride (bytes): 64, read+write: 5.46 ns
Size (bytes): 33554432, Stride (bytes): 128, read+write: 7.45 ns
Size (bytes): 33554432, Stride (bytes): 256, read+write: 7.35 ns
Size (bytes): 33554432, Stride (bytes): 512, read+write: 6.26 ns
Size (bytes): 67108864, Stride (bytes):   4, read+write: -2.36 ns
Size (bytes): 67108864, Stride (bytes):   8, read+write: -1.99 ns
Size (bytes): 67108864, Stride (bytes): 16, read+write: -0.79 ns
Size (bytes): 67108864, Stride (bytes): 32, read+write: 1.94 ns
Size (bytes): 67108864, Stride (bytes): 64, read+write: 7.45 ns
Size (bytes): 67108864, Stride (bytes): 128, read+write: 11.32 ns
Size (bytes): 67108864, Stride (bytes): 256, read+write: 14.01 ns
Size (bytes): 67108864, Stride (bytes): 512, read+write: 12.52 ns

Databar 3

Instructions for completing the databar exercise.

Make sure you have a linux terminal on your computer (bash, ubuntu, ...)
Open thinlinc, navigate to learn.inside.dtu.dk and download all the files for week 4.
'hello_mpi.c', 'communication.c', 'run.txt'
Place it in a folder you'll remember, and rename run.txt to run.sh
Open run.sh and insert your student number where it say so, don't make any other alterations.
Close thinlinc.

Need to access the LSF 10 Cluster, which could be done via thinlinc, but easier with a linux terminal.
Follow instructions here: http://www.cc.dtu.dk/?page_id=2501

Open the linux terminal, and type in:
ssh userid@login1.gbar.dtu.dk
Remember to change 'userid' to your student id,and then type in your password, if you screw up, hit ctrl+c to abort.

Type in 'linuxsh' to access another node.
Use 'cd' and 'ls -l' to navigate to the correct folder.

Type in:
'module load mpi'
'mpicc -o hello_mpi hello_mpi.c'
'bsub < run.sh'

You've now compiled it using MPI, and submitted it to the queue, to have a look at the queue, type in
'bjobs'
After a minute or two expect that the job will disappear, so you have a limited amount of time to test it.

Part 2 (different program, same process)

Create a copy of run.sh
Change the name to something different, 'run1.sh' will suffice.
Alter the content so that both output and input are different.
We want the input to be the compiled file of 'communication.c'.

Submit the job to the queue once more
bsub < run1.sh

==========

Databar 4

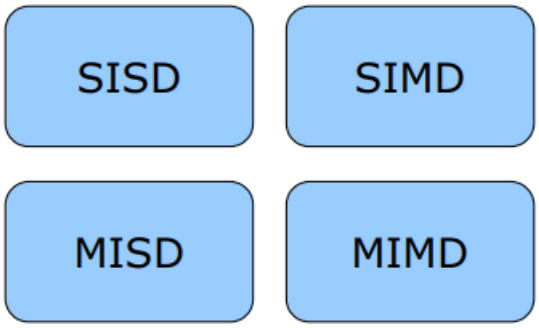
Part 8
Question: Why does half the processes send before receiving and vice versa?
Answer: Because it's a blocking operation, so it waits to ensure that the content is sent, and it waits to ensure that contents are received.

Question: What would happen if all send before receivin?

Need to know aspects

Flynn's taxonomy.

Flynn's taxonomy is a classification of computer architectures



Single Instruction stream, Single Data stream
Single Instruction stream, Multiple Data stream
Multiple Instruction stream, Single Data stream
Multiple Instruction stream, Multiple Data stream

Examples

SISD (LC-3)
SIMD (Parallel computing such as multithreading) (Most GPU use a SIMD model)
MISD (Multiple computers having to agree with the same data, example space shuttle flights)
MIMD (Distributed systems (i.e. networking))

SISD

The original Von Neumann Architecture that does not employ any kind of parallelism. The sequential processor takes data from a single address in memory and performs a single instruction on the data. All single processor systems are SISD.

Common usage
- Older Computers
- Microcontrollers

Advantages
- Low power requirements as only a single core
- Simpler architecture than others therefore cheaper and easier to manufacture

SIMD

A single instruction is executed on multiple different data streams. These instructions can be performed sequentially, taking advantage of pipelining, or in parallel using multiple processors. Modern GPUs, containing Vector processors and array processors, are commonly SIMD systems.

Common usage
- Graphics Processing Units

Advantages
- Very efficient where you need to perform instruction on large amounts of data

MISD

In this architecture multiple instructions are performed on a single data stream. An uncommon type commonly used for fault tolerance. Different systems perform operations on the data and all the results must agree. Used on flight control systems where fault detection is critical.

Common Usage
- Not used commercially.
- Some specific use systems (space flight control)

Advantages
- Excellent for situation where fault tolerance is critical

M I M D

Multiple autonomous processors perform operations on difference pieces of data, either independently or as part of shared memory space.

Common Usage

- Most modern desktop / laptop / mobile processors are MIMD processors.

Advantages

- Great for situations where you need to perform a variety of processor and data intensive tasks (such as video editing, game rendering)

Foster's methodology

A well-known design methodology that is used to architect and implement distributed-memory systems (DMS) is the Foster's Methodology. This methodology has as foundations four steps – partitioning, communication, agglomeration and mapping

tl;dr version (Foster's methodology)

1. Partition the problem solution into tasks.
2. Identify the communication channels between the tasks.
3. Aggregate the tasks into composite tasks. (combine stuff)
4. Map the composite tasks to cores.

Peterson's algorithm

Is a concurrent programming algorithm for mutual exclusion that allows two or more processes to share a
single-use resource without conflict, using only shared memory for communication.

Computing paradigmns

- In concurrent computing, a program is one in which multiple tasks can be in progress at any instant [4].
- In parallel computing, a program is one in which multiple tasks cooperate closely
  to solve a problem.
- In distributed computing, a program may need to cooperate with other programs
  to solve a problem.

Distributed Computing : ...
Parallel Computing : ...

Cluster Computing : ...
Grid Computing : ...
Utility Computing : ...
Edge Computing : ...
Fog Computing : ...
Cloud Computing : ...

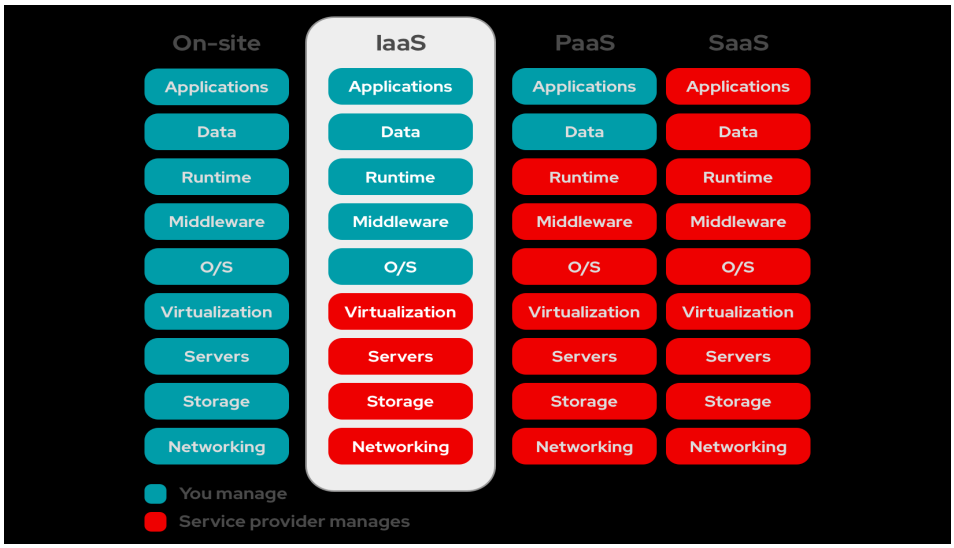
Distributed computing:	Distributed computing is defined as a type of computing where multiple computer systems work on a single problem. Here all the computer systems are linked together and the problem is divided into sub-problems where each part is solved by different computer systems. The goal of distributed computing is to increase the performance and efficiency of the system and ensure fault tolerance.
Parallel computing:	Parallel computing is defined as a type of computing where multiple computer systems are used simultaneously. Here a problem is broken into sub-problems and then further broken down into instructions. These instructions from each sub-problem are executed concurrently on different processors.
Cluster computing:	A cluster is a group of independent computers that work together to perform the tasks given. Cluster computing is defined as a type of computing that consists of two or more independent computers, referred to as nodes, that work together to execute tasks as a single machine. The goal of cluster computing is to increase the performance, scalability and simplicity of the system.
Grid computing:	Grid computing is defined as a type of computing where it is constitutes a network of computers that work together to perform tasks that may be difficult for a single machine to handle. All the computers on that network work under the same umbrella and are termed as a virtual super computer. The tasks they work on is of either high computing power and consist of large data sets. All communication between the computer systems in grid computing is done on the “data grid”.
Utility computing:	Utility computing is defined as the type of computing where the service provider provides the needed resources and services to the customer and charges them depending on the usage of these resources as per requirement and demand, but not of a fixed rate.

	Utility computing involves the renting of resources such as hardware, software, etc. depending on the demand and the requirement. The goal of utility computing is to increase the usage of resources and be more cost-efficient.
Edge computing:	Edge computing is defined as the type of computing that is focused on decreasing the long distance communication between the client and the server. This is done by running fewer processes in the cloud and moving these processes onto a user's computer, IoT device or edge device/server. The goal of edge computing is to bring computation to the network's edge which in turn builds less gap and results in better and closer interaction.
Fog computing:	Fog computing is defined as the type of computing that acts a computational structure between the cloud and the data producing devices. It is also called as "fogging". This structure enables users to allocate resources, data, applications in locations at a closer range within each other. The goal of fog computing is to improve the overall network efficiency and performance.
Cloud computing:	Cloud is defined as the usage of someone else's server to host, process or store data. Cloud computing is defined as the type of computing where it is the delivery of on-demand computing services over the internet on a pay-as-you-go basis. It is widely distributed, network-based and used for storage. Three type of cloud are public, private, hybrid and community and some cloud providers are Google cloud, AWS, Microsoft Azure and IBM cloud.

Source: https://www.geeksforgeeks.org/different-computing-paradigms/

There are 4 types of cloud services:

IaaS - Infrastructure as a service
SaaS - Software as a service
PaaS - Platform as a service
Serverless



Inter-dependency

Abbreviations & terminology

CPU == Core == Processor

Socket == chip

GPU

ILP - Instruction Level Parallelism
Instruction-level parallelism (ILP) is a measure of how many of the instructions in a computer program can be executed simultaneously.
ILP must not be confused with concurrency: (source: https://en.wikipedia.org/wiki/Instruction-level_parallelism)

Steady-state

Cache line == cache block

Data dependency

Node
A computing element, which we will generally refer to as a node, can be either a hardware device or a software process.

Open group, any node is allowed to join the distributed system, effectively meaning that it can send messages to any other node in the system.

Closed group
In contrast, with a closed group, only the members of that group can

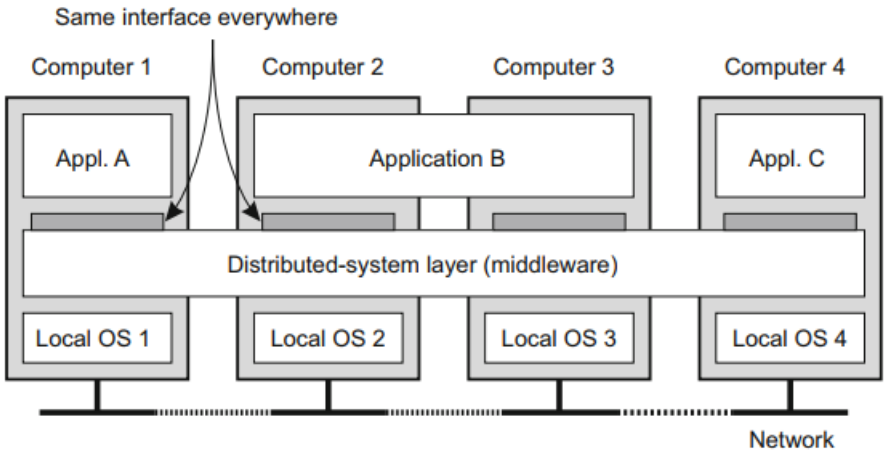
communicate with
each other and a separate mechanism is needed to let a node join or leave the group.

Overlay network
A distributed system is often organized as an overlay network.
An overlay network is a telecommunications network that is built on top of another network and is supported by its infrastructure. An overlay network decouples network services from the underlying infrastructure by encapsulating one packet inside of another packet.

- Structured overlay
In this case, each node has a well-defined set of neighbors with whom it can communicate. For example, the nodes are organized in a tree or logical ring.

- Unstructured overlay
In these overlays, each node has a number of references to
randomly selected other nodes.

Middleware
Distributed systems are often organized to have a separate layer of software that is
logically placed on top of the respective operating systems of the computers that are
part of the system. This organization is shown in Fig. 1, leading to what is known as
middleware.



RPC - Remote Procedure Call
Common communication service.
An RPC service allows an application to invoke a function that is implemented and executed on a remote computer as if it was locally available.

Transparancy for distributed systems (Maarten & Taanembaum page 974)

Table 1 Different forms of transparency in a distributed system (see ISO [31])

Transparency	Description
Access	Hide differences in data representation and how an object is accessed
Location	Hide where an object is located
Relocation	Hide that an object may be moved to another location while in use
Migration	Hide that an object may move to another location
Replication	Hide that an object is replicated
Concurrency	Hide that an object may be shared by several independent users
Failure	Hide the failure and recovery of an object

WSC's
- Warehouse Scale Computers
Computing platforms that power cloud computing and all the great web services we use every day.

IaaS
- Infrastructur as a Service

DAS
- Directly Attached Storage
When drives are connected directly to compute servers (processors)

NAS
- Network Attached Storage

QoS
- Quality of Service

IOPS
- Input/Output Operations Per Second
IOPS (Input/Output Operations Per Second, pronounced i-ops) is a common performance measurement used to benchmark computer storage devices like hard disk drives (HDD), solid state drives (SSD), and storage area networks (SAN)

TCO
- Total Cost of Ownership

NIC
- Network Interface Controller
is a computer hardware component that connects a computer to a computer network

UPS
- Uninterruptable Power Supplies

SLO
- Service Level Object

DNNs
- Deep Neural Networks

N N
- Neural Networks

Computing paradigmns

Cloud computing

Edge computing

Formula

Foster's methodology

Reference to 'Ian Foster' who wrote a book called:
'Designing and Building Parallel Programs"
He provides an outline of steps, which is sometimes called
"**Foster's methodology**"

1. **Partitioning**. Divide the computation to be performed and the data operated on by the computation into small tasks. The focus here should be on identifying tasks that can be executed in parallel.
2. **Communication**. Determine what communication needs to be carried out among the tasks identified in the previous step.
3. **Agglomeration** or aggregation. Combine tasks and communications identified in the first step into larger tasks. For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.
4. **Mapping**. Assign the composite tasks identified in the previous step to processes/threads. This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work

tl;dr version
1. Partition the problem solution into tasks.
2. Identify the communication channels between the tasks.
3. Aggregate the tasks into composite tasks. (combine stuff)

4. Map the composite tasks to cores.

Time speedup

Speedup **S (capital S)**
Numbers of processes or threads **p**
Speedup: $S = \frac{T_{serial}}{T_{parallel}}$
Efficiency (E) = $\frac{\boldsymbol{S}}{\boldsymbol{p}}$
Problem size = **n**

So to calculate efficiency we say:

.....

= >

Example speedup

If program A use 4 time units, and program B use 6 time units, speedup become

$S = \frac{6}{4}$
S = 1½

p cores

$$E = \frac{S}{p} = \frac{\left(\frac{T_{serial}}{T_{parallel}}\right)}{p} = \frac{T_{serial}}{p \cdot T_{parallel}}$$

Amdahl's law

Amdahl's law

$$Speedup \leq \frac{1}{(1 - pctPar) + \frac{pctPar}{p}}$$

Where pctPar is the percentage that's parallel compared to serial.
p is the number of cores(threads)

MPI - Message Passing Interface

MPI stands for "Message Passing Interface"
To compile a code with MPI, you need to load an mpi module.

> module load mpi

After that, you can compile using the same format as gcc
Although instead of gcc, type 'mpicc'
Example:
> mpicc -o hello_world hello_world.c

MPI program rules

Need to include an 'mpi.h' header file.

- Must initialize at the start 'MPI_Init(&, &)' before any other mpi functions are called
and finish with MPI_Finalize();

In between MPI functions can be included, but not outside of those two.

Syntax
: int MPI Init(
int argc p / in/out /,
char argv p / in/out /);
: aka. MPI_Init(&argc,&argv)
If arguments aren't used, pass 'NULL' for both arguments.

There will be an amount of different processes.
Each process is specified by a 'rank' let's call it 'p' (non-negative integers 0,1,2,..., p-1)
And the max amount of processes.
In order to pass this information to the processes, we use 2 functions
MPI_Comm_size(.., ..) ;
MPI_Comm_rank(..,..) ;

See example

Code syntax

```
int MPI_Init(
      int*     argc_p  /* in/out */,
      char***  argv_p  /* in/out */);

int MPI_Finalize(void);

int MPI_Comm_size(
      MPI_Comm  comm         /* in  */,
      int*      comm_sz_p    /* out */);

int MPI_Comm_rank(
      MPI_Comm  comm         /* in  */,
      int*      my_rank_p    /* out */);

int MPI_Send(
      void*         msg_buf_p     /* in */,
      int           msg_size      /* in */,
      MPI_Datatype  msg_type      /* in */,
      int           dest          /* in */,
      int           tag           /* in */,
      MPI_Comm      communicator  /* in */);
```

//This is a blocking type operation, that sends data, first 3 arguments are about the data being sent, remaining 2 arguments are about the receiver
Take care not to cause a deadlock, since there must be 1 (and only 1) matching receive somewhere.

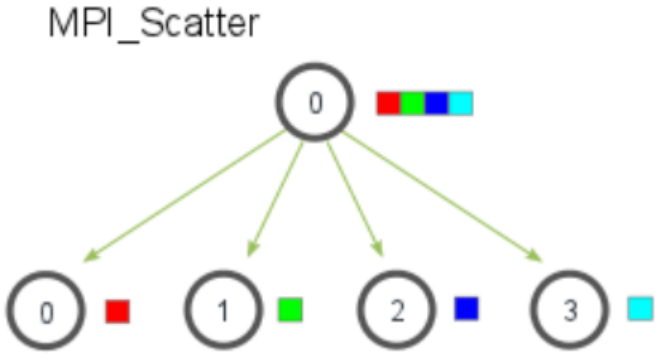
```
int MPI_Recv(
      void*         msg_buf_p     /* out */,
      int           buf_size      /* in  */,
      MPI_Datatype  buf_type      /* in  */,
      int           source        /* in  */,
      int           tag           /* in  */,
      MPI_Comm      communicator  /* in  */,
      MPI_Status*   status_p      /* out */);

int MPI_Get_count(
        MPI_Status*  status_p  /* in  */,
        MPI_Datatype type      /* in  */,
        int*         count_p   /* out */);
```

// Not particularly relevant, it can be used to determine the data within a MPI_Status structure, after having used the relevant MPI call using the status argument.

```
int MPI_Scatter(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      int            src_proc    /* in  */,
      MPI_Comm       comm        /* in  */);
```

//MPI_Scatter sends chunks of an array to different processes.



send_count is often equal to the number of elements in the array divided by the number of processes

```
int MPI_Gather(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      int            dest_proc   /* in  */,
      MPI_Comm       comm        /* in  */);
```

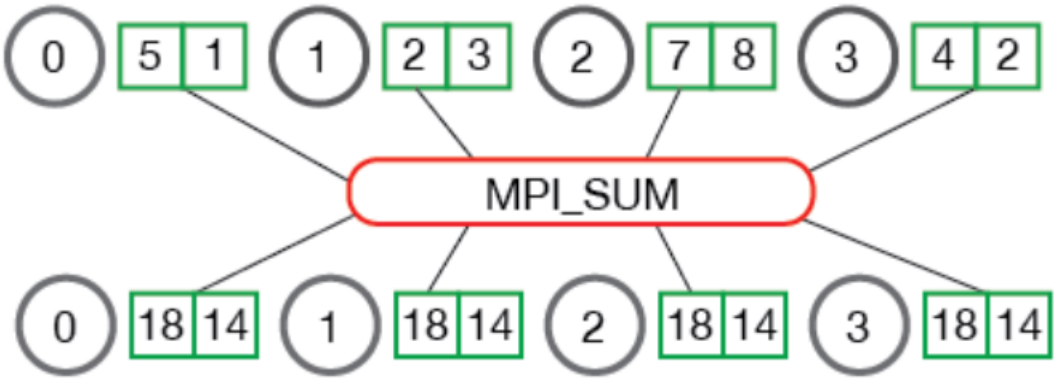
```
int MPI_Reduce(
      void*          input_data_p   /* in  */,
      void*          output_data_p  /* out */,
      int            count          /* in  */,
      MPI_Datatype   datatype       /* in  */,
      MPI_Op         operator       /* in  */,
      int            dest_process   /* in  */,
      MPI_Comm       comm           /* in  */);
```

//Can be used to gather information from multiple processes (Max, min, sum, ...)

//if Count > 1, then it can be used on arrays rather than scalars (???)
//Beware of aliasing (using same output for multiple processes)
//Similar to MPI_Gather, MPI_Reduce takes an array of input elements on each process and //returns an array of output elements to the root process. The output elements contain the reduced //result.

```
int MPI_Allreduce(
          void*         input_data_p    /* in  */,
          void*         output_data_p   /* out */,
          int           count           /* in  */,
          MPI_Datatype  datatype        /* in  */,
          MPI_Op        operator        /* in  */,
          MPI_Comm      comm            /* in  */);
```

MPI_Allreduce



https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

```
int MPI_Bcast(
          void*         data_p          /* in/out */,
          int           count           /* in     */,
          MPI_Datatype  datatype        /* in     */,
          int           source_proc     /* in     */,
          MPI_Comm      comm            /* in     */);
```

//Broadcast

MPI predefined types

Table 3.1 Some Predefined MPI Datatypes

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Constant:
MPI_STATUS_IGNORE

MPI_ANY_SOURCE
- Can be used by MPI_Recv to receive in the order that they finish

MPI_COMM_WORLD
- seems to be a global communicator (verification required)

MPI_Status
- Is a struct datatype, alternate option is to use "MPI_STATUS_IGNORE"
Otherwise, initialize with "MPI_Status status"
Struct contains 3 types of data.
- MPI_SOURCE
- MPI_TAG
- MPI_ERROR.

Table 3.2 Predefined Reduction Operators in MPI

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

Example

```
/****************
 * hello_mpi.c
 * 02346, F20, DTU
 * For instructions, see databar exercise 3.
*/
#include "mpi.h"
#include <stdio.h>
#define MASTER 0

int main(int argc, char *argv[]) {
 int  numtasks, rank, len, rc;
 char hostname[MPI_MAX_PROCESSOR_NAME];

  rc = MPI_Init(&argc,&argv);
  if (rc != MPI_SUCCESS) {
    printf ("Error starting MPI program. Terminating.\n");
    MPI_Abort(MPI_COMM_WORLD, rc);
   }

  MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Get_processor_name(hostname, &len);
```

```
  printf ("Number of tasks= %d My rank= %d Running on %s\n", numtasks,rank,
hostname);

  if (rank == MASTER) {
    printf ("The master is running on %s\n", hostname);
   }

  MPI_Finalize();
}
```

Example (start and end of array that's been scattered)

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int MAXSIZE = 100000;

int main(int argc, char **argv) {
int comm_sz, comm_rank;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
MPI_Comm_rank(MPI_COMM_WORLD, &comm_rank);

int   A[3];
int   B[MAXSIZE];
//int   D[MAXSIZE];

Broadcast A
Broadcast B

int start, end, avg;
if(comm_rank == 0) start = 1;
//Is the array evenly distributed among the cores?
if(MAXSIZE%size == 0 ){ //Yes it is
    avg = MAXSIZE/comm_sz;
    start = comm_rank * avg;
    end = (comm_rank+1) * avg;

} else { //No it isn't.
    avg = MAXSIZE/(comm_sz)+1;
    start = comm_rank * avg;
    if (comm_rank != comm_sz - 1){
        end = (comm_rank+1) * avg;
    } else {
        end = MAXSIZE;
    }
```

```
}

int E[stride];

if(rank != size -1)
for (int i = 0; i< avg ; i++){
    E[i] = 0;
    for (int j=0; j < 3; j++)
        E[i] += A[j] * B[start + i + j];
}

if(rank == size -1)
for (int i=1; i < end-2; i++){
    E[i] = 0;
    for (int j=0; j < 3; j++)
        E[i] += A[j] * B[start + i + j];
}


allgather E --> D
MPI_Finalize();

}
```

OpenMP - Open Multi-platform

Source:
- Book, pancheco
- https://gcc.gnu.org/onlinedocs/libgomp/index.html#SEC_Contents

Compile :

```
$ gcc -g -Wall -fopenmp -o name filename.c
$ export OMP_NUM_THREADS=2
$ ./omp_hello
```

Scope
-wtf is this
default scope of a variable can change with other directives, OpenMP provides clauses to modify the default scope

WARNING, THIS IS POSSIBLE, BUT NOT RECOMMENDED UNLESS YOU KNOW WHAT YOU'RE DOING. (see Pragmas paragraph, - schedule)

```
$ export OMP_schedule="static,1"
```

WARNING THIS IS POSSIBLE, BUT DO NOT DO THIS UNLESS YOU KNOW WHAT YOU'RE DOING!

$ export TEST_VAR="hello"

Variables declared within parallel blocks are private
Variables declared prior to parallel blocks can be defined as private using a private() command, in the pragma.
#pragma omp parallel private(VARIABLENAME)

Syntax

#include <omp.h>

Focus on these two functions to get the ID/rank and max threads respectively.

```
int omp_get_thread_num(void);
int omp_get_num_threads(void);
```

Returns the number of threads in the current team.

```
int omp_get_num_threads(void);
```

Return the maximum number of threads used for the current parallel region that does not use the clause *num_threads.*

```
int omp_get_max_threads(void);
```

Specifies the number of threads used by default in subsequent parallel sections, if those do not specify a num_threads clause. The argument of omp_set_num_threads shall be a positive integer.

Specifies the number of threads used by default in subsequent parallel sections, if those do not specify a *num_threads* clause. The argument of *omp_set_num_threads* shall be a positive integer.

```
void omp_set_num_threads(int num_threads);
```

Security practice

In case we don't know if OpenMP is included we can say:

```
#ifdef _OPENMP
#  include <omp.h>
#endif
```

Which can be expanded to:

```
#  ifdef _OPENMP
      int my_rank = omp_get_thread_num();
      int thread_count = omp_get_num_threads();
#  else
      int my_rank = 0;
      int thread_count = 1;
#  endif
```

This is to ensure that a generic 'int thread_count' is generated, and the program will work, even if it is just on a single core.

Included higher abstraction level mutex and semaphores to prevent race conditions.
This is done by a pragma directive called 'critical'

 # pragma omp critical

This creates a critical section, that only 1 thread can access at a time.

Pragmas (directives)

OpenMP pragmas always begin with
 # pragma omp

Syntax is then either with or without {}, but if used without, only the following line is considered parallel

Multiple lines can be created within the pragma, by including a '\' at the end of each line
*# pragma omp parallel *
num_threads(thread_count)

To create a parallel region can be created with
pragma omp parallel

It's possible to specify the number of threads using a 'parallel directive' called 'num_threads(int argument)', where 'arguments' is usually replaced with '**thread_count**'
 # pragma omp parallel num_threads(thread_count)

For sections that require preventions of race conditions, use 'critical' directive.
pragma omp critical

When acquiring a specific data from all threads to be combined into a single

function, we can use a pragma directive called 'reduction'

```
reduction(<operator>: <variable list>)
```

# pragma omp parallel num_threads(thread_count) reduction(+: global_result)
    global_result += Local_trap(double a, double b, int n);

Operator List

+, *, -, &, |, ^, &&, ||

Subtraction is be a bit problematic, therefore it's easier to store the combined value as a positive, then switch it to negative, to subtract from whatever it is we needed to.

For example, the serial code

```
result = 0;
for (i = 1; i <= 4; i++)
   result -= i;
```

stores the value -10 in result. If, however, we split the iterations among two threads, with thread 0 subtracting 1 and 2 and thread 1 subtracting 3 and 4, then thread
0 will compute -3 and thread 1 will compute -7 and, of course, -3 - (-7) = 4.

Loops: It is possible to parallelize for-loops but not while, or do-while loops.
It is done using the directive 'for', same rules apply when using for-loops, example:
Beware of exit (exit/break) clauses, as well as dependencies on previous loops (terminology 'loop-carried dependences')

```
for ( index = start  ;  index < end     ;  index++          )
                        index <= end        ++index
                        index >= end        index--
                        index > end         --index
                                            index += incr
                                            index -= incr
                                            index = index + incr
                                            index = incr + index
                                            index = index - incr
```

#pragma omp parallel for

Private or shared variables.
If you want a shared value, declare it outside of the parallel region
If you need a private value, declare it within the parallel region.
It is possible to use pragma directives to alter this, so that a shared value becomes private within the parallel region.

syntax is private(variable1, variable2, variable3, ...)
#pragma omp parallel private(derp)

Schedule clause - Something about breaking up the chunksize in different ways.
Syntax:

```
schedule(<type> [, <chunksize>])
```

Scheduling type list

The `type` can be any one of the following:

- `static`. The iterations can be assigned to the threads before the loop is executed.
- `dynamic` or `guided`. The iterations are assigned to the threads while the loop is executing, so after a thread completes its current set of iterations, it can request more from the run-time system.
- `auto`. The compiler and/or the run-time system determine the schedule.
- `runtime`. The schedule is determined at run-time.

The `chunksize` is a positive integer. In OpenMP parlance, a **chunk** of iterations is a block of iterations that would be executed consecutively in the serial loop. The number of iterations in the block is the `chunksize`. Only `static`, `dynamic`, and `guided` schedules can have a `chunksize`. This determines the details of the schedule, but its exact interpretation depends on the `type`.

Static
Word by word: "To get a cyclic schedule, we can add a schedule clause to the parallel for
directive:"

```
      sum = 0.0;
#     pragma omp parallel for num_threads(thread_count) \
         reduction(+:sum) schedule(static,1)
      for (i = 0; i <= n; i++)
         sum += f(i);
```

If we use the static-type, the chunksize iterations will be handled in a roundrobin fashion, example with 12 tasks.
schedule(static,1) --> becomes:
Thread 0: 0,3,6,9

Thread 1: 1,4,7,10

Thread 2: 2,5,8,11

schedule(static,2) --> becomes:
Thread 0: 0,1,6,7

Thread 1: 2,3,8,9

Thread 2: 4,5,10,11

and so on.
#pragma omp parallel for schedule(static,1)

There are some situations in which it's a good idea to explore some schedules before others:

- If each iteration of the loop requires roughly the same amount of computation, then it's likely that the default distribution will give the best performance.
- If the cost of the iterations decreases (or increases) linearly as the loop executes, then a `static` schedule with small chunksizes will probably give the best performance.
- If the cost of each iteration can't be determined in advance, then it may make sense to explore a variety of scheduling options. The `schedule(runtime)` clause can be used here, and the different options can be explored by running the program with different assignments to the environment variable `OMP_SCHEDULE`.

Section 5.7 in the book

Which schedule to use?

5.7.5 Which schedule?

If we have a `for` loop that we're able to parallelize, how do we decide which type of schedule we should use and what the `chunksize` should be? As you may have guessed, there *is* some overhead associated with the use of a `schedule` clause. Furthermore, the overhead is greater for `dynamic` schedules than `static` schedules, and the overhead associated with `guided` schedules is the greatest of the three. Thus, if we're getting satisfactory performance without a `schedule` clause, we should go no further. However, if we suspect that the performance of the default schedule can be substantially improved, we should probably experiment with some different schedules.

In the example at the beginning of this section, when we switched from the default schedule to `schedule(static,1)`, the speedup of the two-threaded execution of the program increased from 1.33 to 1.99. Since it's *extremely* unlikely that we'll get speedups that are significantly better than 1.99, we can just stop here, at least if we're only going to use two threads with 10,000 iterations. If we're going to be using varying numbers of threads and varying numbers of iterations, we need to do more experimentation, and it's entirely possible that we'll find that the optimal schedule depends on both the number of threads and the number of iterations.

It can also happen that we'll decide that the performance of the default schedule isn't very good, and we'll proceed to search through a large array of schedules and iteration counts only to conclude that our loop doesn't parallelize very well and *no* schedule is going to give us much improved performance. For an example of this, see Programming Assignment 5.4.

There are some situations in which it's a good idea to explore some schedules before others:

- If each iteration of the loop requires roughly the same amount of computation, then it's likely that the default distribution will give the best performance.
- If the cost of the iterations decreases (or increases) linearly as the loop executes, then a `static` schedule with small chunksizes will probably give the best performance.
- If the cost of each iteration can't be determined in advance, then it may make sense to explore a variety of scheduling options. The `schedule(runtime)` clause can be used here, and the different options can be explored by running the program with different assignments to the environment variable `OMP_SCHEDULE`.

Page 241 in the book.

#pragma omp parallel for schedule(static,1)

Barrier

Prevents threads from progressing untill all threads have reached the barrier.
#pragma omp barrier

Atomic
Similar to barrier, but it can only be used for a single line of code

Single
The omp single directive identifies a section of code that must be run by a single available thread.

Method 1, using arguments

Compile:
$ gcc -g -Wall -fopenmp -o omp hello_omp hello.c
$ ./omp_hello 4

This function is used when converting argc/argv to a long int

```
long strtol(
      const char*  number p     /* in  */,
      char**       end p        /* out */,
      int          base         /* in  */);
```

Example: int thread_count = strtol(argv[1], NULL, 10);

Then use:
#pragma omp parallel num_threads(int derp)
{
 /** Parallel code */
}

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void);  /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```

Method 2, using console

$ gcc -fopenmp -o name filename.c
$ export OMP_NUM_THREADS #
$ ./name

Trial exam

Found at eksamen.dtu.dk

code is: 00000

We can get partial points, but we cannot get half points

Question 1.1

Question 1.1
Explain in your own words the difference between message passing and shared memory programming models.
What are the advantages and disadvantages between the two?

General guideline

with 3 points, we're expected to state 3 things, but it's not necessarily limited to only 3 possible things to mention.

Examples:
shared vs distributed memory
Message passing have some advantages in terms of scalability
With distributed memory, it's possible to scale beyond a single computer.

Question 2

Assume you have a sequential program that runs in
forty eight (48) seconds. You profile the program and notice that forty (40) seconds of the program is spent on work that
can be done in parallel and that it can be made perfectly parallel.

Grading
Answer 1point
Method 2point
-1p for small errors
-2p for large obvious errors

Formula for speedup is

...

Otherwise use amdahl

Question 3

Pseudo code 2-3 points
Method 3-4 points

If we need to assume something, we should state it.

Read chapter 2 specifically page 66

Concepts:
- Partitioning
- Communication
- Aggregation
- Mapping

Partitioning
A <-- should be split in "bands"

Everyone needs a copy of B (broadcast)
We will split C same way as A
it's just that we won't need to initialize A

Communication
MPI scatter on A
MPIBcast on B (broadcast the entire array across different nodes)

Pseudo code

```
float A[100000][100000]; // <-- bands
float B[100000][100000]; // <-- broadcast
float C[100000][100000]; // <-- bands

Scatter A
Broadcast B

for (int i=0; i<100000; i++){
  for (int j=0; j<100000; j++){
    C[i][j] = 0.0;
    for (int k=0; k<100000; k++){
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}

Gather C
```

Note: https://www.rookiehpc.com/mpi/docs/
Figure: 3.11 Page 114

Pages 109-114 discuss details for how scatter/gather works in details

$n := 1000:$

$$\frac{8}{\left(\frac{40}{}\right)}$$

(10.3.1)