SCALABLE COMPUTING LABORATORY

VANDERBILT UNIVERSITY

SC3260 / SC5260

Introduction to HPC

Lecture by: Ana Gainaru

High Performance Computing

Tool used by computational scientists and engineers to tackle problems that require more computing resources or time than they can obtain on the personal computers available to them.

Forms of parallelism in HPC

- A commodity cluster computer is composed of multiple nodes, connected via a network
  - Each node is composed of a system board, 1 or more chips, DRAM, coprocessors/accelerators, etc.
- Each chip contains multiple cores
  - Each core has its own L1 cache, but may share higher level caches with other cores
- Depending on its type, a core can
  - Execute multiple instructions simultaneously (instruction level parallelism)
  - Execute the same instruction on multiple pieces of data simultaneously (SIMD)

New technologies every year to increase the level of parallelism or the speed of data movement

VANDERBILT UNIVERSITY

Summit supercomputer

Located at Oak Ridge National Laboratory
Currently the fastest supercomputer in the world

4,608 nodes: 2 IBM POWER9 CPUs and 6 Nvidia Volta V100 GPUs per node

- Each node contains two 22-core CPUs and 6 80-core coprocessors. Total of 2,414,592 cores
- Each node contains 608 GB of coherent memory which is addressable by all CPUs and GPUs
  - 800 GB of non-volatile RAM that can be used as a burst buffer or as extended memory

VANDERBILT UNIVERSITY

What is a High-Performance Computer?

Hardware

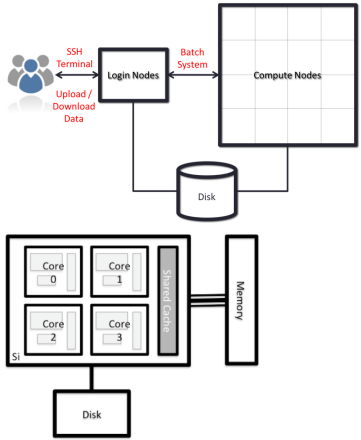
- Hierarchy of multiple processing units and memory modules connected by fast interconnect networks with access to multi layer large storage systems.
- Accessible through specialized interface nodes.

Application

- Applications for HPC must be split up into many smaller "programs" called processes/threads, corresponding to each CPU/core.
- Cores must be able communicate with each other efficiently.

Software

- Scheduler, Distributed Operating System, Filesystem

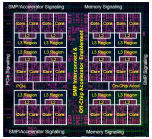


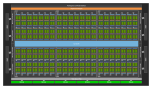
Source: https://epcced.github.io/hpc-intro/010-hpc-concepts/

VANDERBILT UNIVERSITY

General purpose CPUs and GPUs

IBM POWER9 CPUs and Nvidia Volta V100 GPUs

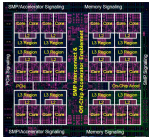




VANDERBILT UNIVERSITY

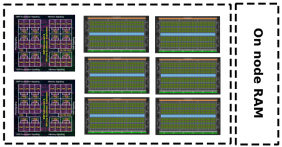
General purpose CPUs and GPUs

IBM POWER9 CPUs and Nvidia Volta V100 GPUs



Nodes (shared memory)

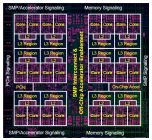
2 IBM POWER9 CPUs and 6 Nvidia Volta V100 GPUs

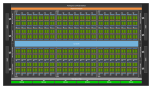


VANDERBILT UNIVERSITY

General purpose CPUs and GPUs

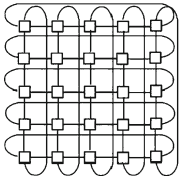
IBM POWER9 CPUs and Nvidia Volta V100 GPUs





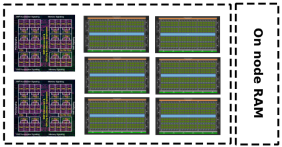
Connect nodes with a network

Torus topology



Nodes (shared memory)

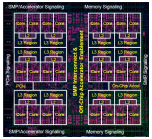
2 IBM POWER9 CPUs and 6 Nvidia Volta V100 GPUs



VANDERBILT UNIVERSITY

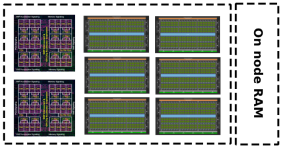
General purpose CPUs and GPUs

IBM POWER9 CPUs and Nvidia Volta V100 GPUs



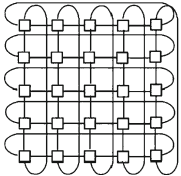
Nodes (shared memory)

2 IBM POWER9 CPUs and 6 Nvidia Volta V100 GPUs

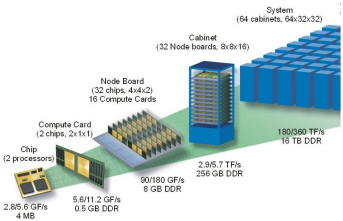


Connect nodes with a network

Torus topology

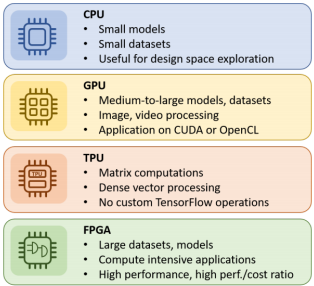


Organize nodes on levels



VANDERBILT UNIVERSITY

Quick survey of the different processing units



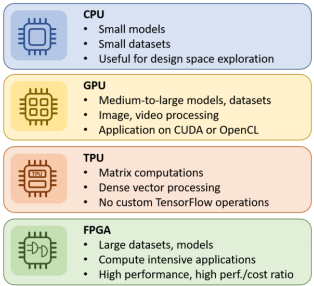
Source: CPU, GPU, FPGA or TPU: Which one to choose for my Machine Learning training? inaccel.com

- **CPU** Base model. Multiple and large caches. Can execute many functions (sophisticated control unit). Easy to program.

VANDERBILT UNIVERSITY

Processing units

Quick survey of the different processing units



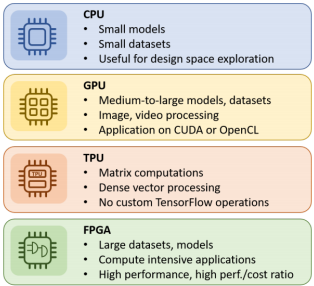
Source: CPU, GPU, FPGA or TPU: Which one to choose for my Machine Learning training? inaccel.com

- **CPU** Base model. Multiple and large caches. Can execute many functions (sophisticated control unit). Easy to program.
- **GPU, SIMD, Vector processors** Many parallel processing units. Specialized architecture for simple and iterative operations. Ideal for image/video processing applications, but offers flexibility for other needs.

VANDERBILT UNIVERSITY

Processing units

Quick survey of the different processing units



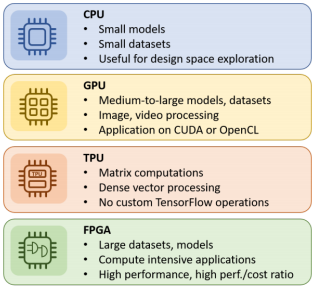
Source: CPU, GPU, FPGA or TPU: Which one to choose for my Machine Learning training? inaccel.com

- **CPU** Base model. Multiple and large caches. Can execute many functions (sophisticated control unit). Easy to program.
- **GPU, SIMD, Vector processors** Many parallel processing units. Specialized architecture for simple and iterative operations. Ideal for image/video processing applications, but offers flexibility for other needs.
- **TPU, VPU, NPU, QPU** For new type of application patterns (machine learning, vision, quantum). Optimized for a specific function (e.g. calculations of tensors for neural networks)

VANDERBILT UNIVERSITY

Processing units

Quick survey of the different processing units



Source: CPU, GPU, FPGA or TPU: Which one to choose for my Machine Learning training? inaccel.com

- **CPU** Base model. Multiple and large caches. Can execute many functions (sophisticated control unit). Easy to program.
- **GPU, SIMD, Vector processors** Many parallel processing units. Specialized architecture for simple and iterative operations. Ideal for image/video processing applications, but offers flexibility for other needs.
- **TPU, VPU, NPU, QPU** For new type of application patterns (machine learning, vision, quantum). Optimized for a specific function (e.g. calculations of tensors for neural networks)
- **FPGA** Configurable chip: avoids building a new chip for each need. The software chooses the connections to optimize the application.

VANDERBILT UNIVERSITY

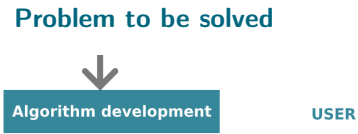
Applications development

Problem to be solved

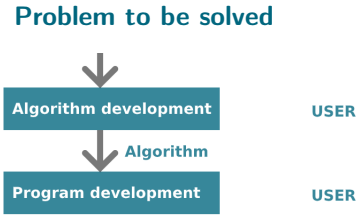
Results

VANDERBILT UNIVERSITY

Applications development



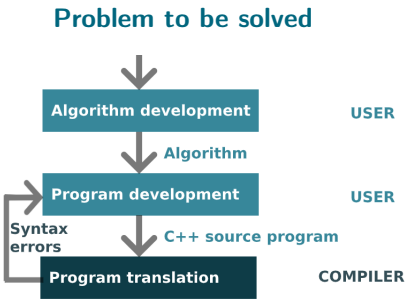
Applications development



Results

VANDERBILT UNIVERSITY

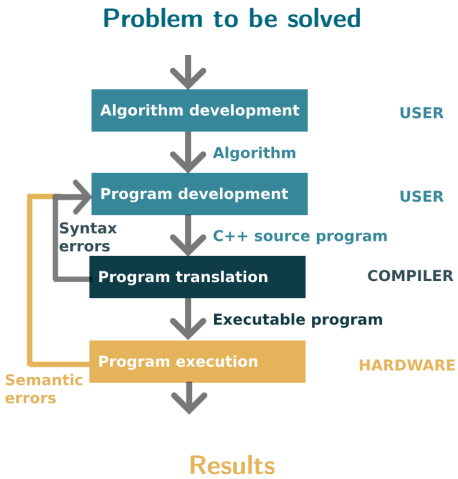
Applications development



Results

VANDERBILT UNIVERSITY

Applications development



VANDERBILT UNIVERSITY

Applications


Planetary Movments


Climate Change


Plate Tectonics


Weather


Galaxy Formation

Rush Hour Traffic

High Performance Computing has created two new scientific paradigms

Big data

- Dig through large amounts of data (sensors, transaction records, genome and protein databanks)

Scientific computing

- Simulate physical phenomena when real experiments are very costly

- Motivated the development of major computational infrastructures

VANDERBILT UNIVERSITY

Traditional Applications


Planetary Movments


Climate Change


Plate Tectonics


Weather


Galaxy Formation

Rush Hour Traffic

Historically, parallel computing has been considered to be "the high end of computing", and has been ised to model difficult problems in many ares of science and engineering:

- Atmosphere, Earth, Environment
- Phisics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
- Bioscience, Biotechnology, Genetics
- Chemestry, Molecular Sciences
- Geology, Seismology
- Mechanical Engineering - from phrosthetics to spacecraft
- Electrical Engineering, Circuit Design, Microelectronics
- Mathematics
- Defense, Weapons

VANDERBILT UNIVERSITY

New Generation Applications


Planetary Movments


Climate Change


Plate Tectonics


Weather

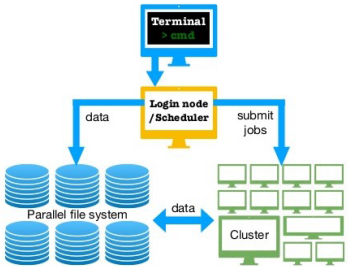

Galaxy Formation

Rush Hour Traffic

Today, commercial applications provide an equal driving force in the development of faster computers. These applications require the processing of large amounts of data in sophisticated ways.

- "Big data", databases, data mining
- Artificial Intelligence (AI)
- Web searches engines, web based business services
- Medical imaging and diagnosis
- Pharmaceutical design
- Finantial and economic modeling
- Management of national and multi-national corporations
- Advanced graphics and virtual reality
- Oil exploration
- Multi-media technologies

VANDERBILT UNIVERSITY



Scheduler

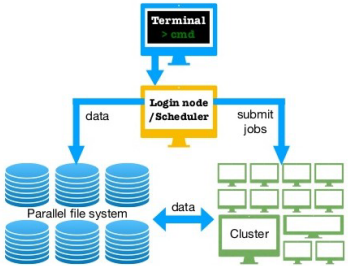
- Receives job submissions from users
- Finds available nodes in the cluster
  - That fit the requirements of the job
- Send for execution the job on the selected nodes
- Monitors the progress of the jobs

Uses different policies to decide what job needs to be scheduled first

- User fairness
- System utilization
- To minimize data movement
- To minimize communication cost

VANDERBILT UNIVERSITY

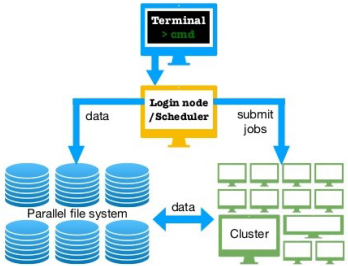
Middleware



Distributed operating system

- Same as for normal computers, just larger scale
  - Based on Linux
- Responsible with memory management
- Responsible with the allocation, management and disposition of node resources
- Manages processes and communication
- Manages input/output

VANDERBILT UNIVERSITY



Filesystem

Examples: Lustre or GPFS

- Responsible with file management
  - Creating / removing files from disks
  - Keeping track of their metadata (for efficient access)
  - Fault tolerance
  - Data movement
- **They need to deal with huge sizes** Tens of thousands of client nodes, tens of petabytes (PB) of storage on hundreds of servers, and more than a terabyte per second (TB/s) of aggregate I/O throughput

VANDERBILT UNIVERSITY

Problem: Increase all elements of an array by 1

At the end write the results in a file

```
for(i = 0; i < N; i++)
  a[i] = a[i] + 1;
```

Assume the size of the array equals the number of nodes of our cluster

Problem: Increase all elements of an array by 1

At the end write the results in a file

```
for(i = 0; i < N; i++)
  a[i] = a[i] + 1;
```

Assume the size of the array equals the number of nodes of our cluster

- Break the array into chunks of size 1
- Processor P is responsible for computing a[P] = a[P] + 1

Problem: Increase all elements of an array by 1

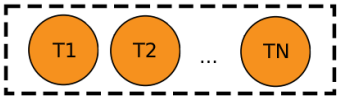
At the end write the results in a file

```
for(i = 0; i < N; i++)
  a[i] = a[i] + 1;
```

Assume the size of the array equals the number of nodes of our cluster

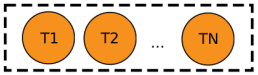
- Break the array into chunks of size 1
- Processor P is responsible for computing a[P] = a[P] + 1

- Divide the program into N tasks, $T_k = \{a[k] = a[k] + 1\}$
  - At the end all task communicate their values
  - Task T_{N+1} writes to the disk
- User writes the parallel program
- Compiles and gets one executable back with instructions



VANDERBILT UNIVERSITY

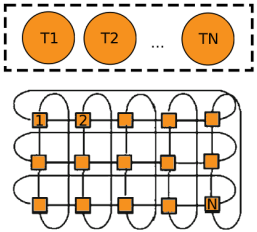
Application life cycle



1. Submit the executable to the **scheduler**
   - Run T = all{T_k with k=1,N} followed by T_{N+1}
   - Request N nodes
   - Request nodes with enough memory for one float
   - One extra nodes with enough memory for the entire array

VANDERBILT UNIVERSITY

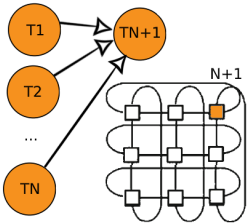
Application life cycle



1. Submit the executable to the **scheduler**
   - Run T = all{T_k with k=1,N} followed by T_{N+1}
   - Request N nodes
   - Request nodes with enough memory for one float
   - One extra nodes with enough memory for the entire array
2. The **operating system** creates the processes and the address space
   - Each tasks starts running on a separate node
   - T_k for k=1,N execute instruction ADD

VANDERBILT UNIVERSITY

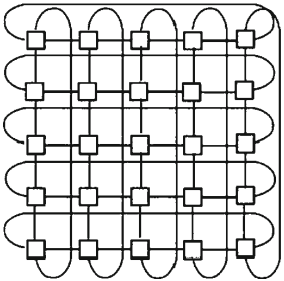
Application life cycle



1. Submit the executable to the **scheduler**
   - Run T = all{T_k with k=1,N} followed by T_{N+1}
   - Request N nodes
   - Request nodes with enough memory for one float
   - One extra nodes with enough memory for the entire array
2. The **operating system** creates the processes and the address space
   - Each tasks starts running on a separate node
   - T_k for k=1,N execute instruction ADD
   - The OS manages communication between the tasks
   - Once all tasks end, T_{N+1} starts
   - The **filesystem** manages the I/O to the file on storage

VANDERBILT UNIVERSITY

Application life cycle

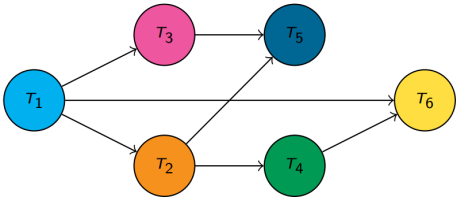


1. Submit the executable to the **scheduler**
   - Run T = all{T_k with k=1,N} followed by T_{N+1}
   - Request N nodes
   - Request nodes with enough memory for one float
   - One extra nodes with enough memory for the entire array
2. The **operating system** creates the processes and the address space
   - Each tasks starts running on a separate node
   - T_k for k=1,N execute instruction ADD
   - The OS manages communication between the tasks
   - Once all tasks end, T_{N+1} starts
   - The **filesystem** manages the I/O to the file on storage
3. The **scheduler** returns the result back to user

VANDERBILT UNIVERSITY

More complex tasks

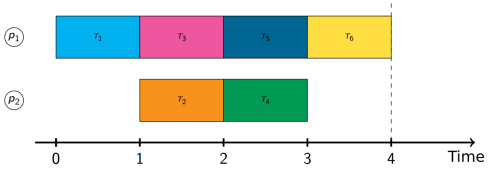
- Assume we have p processors (or nodes or cores, or more general, processing units)
- We can represent an application by a graph G = (V, E) :
  - Nodes represent the tasks that need to be executed
  - Edges are dependencies between tasks



VANDERBILT UNIVERSITY

More complex tasks

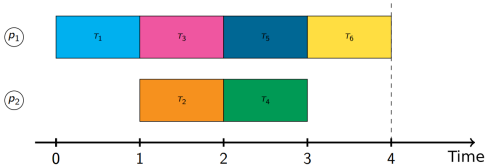
- Scheduling for 2 processors

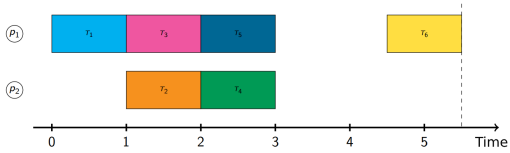


VANDERBILT UNIVERSITY

More complex tasks

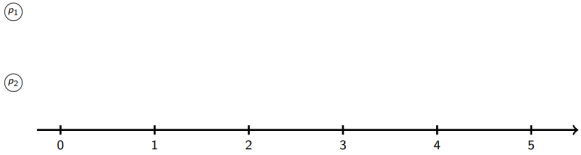
- Scheduling for 2 processors

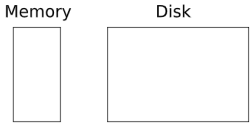




- Theoretical algorithms for mapping tasks to processors need to consider hazards

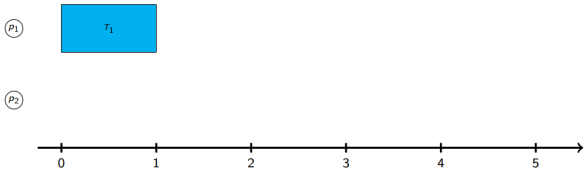
More complex tasks

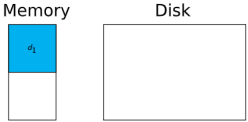




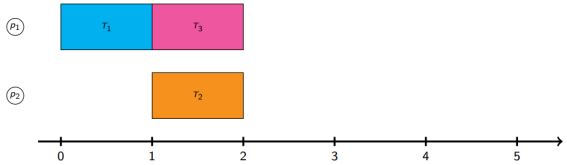
VANDERBILT UNIVERSITY

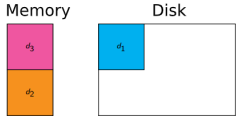
More complex tasks



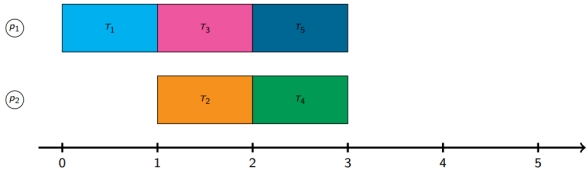


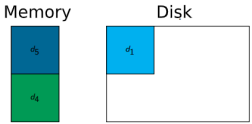
More complex tasks



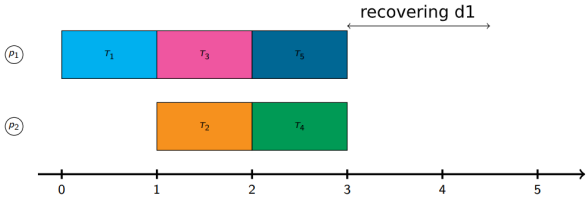


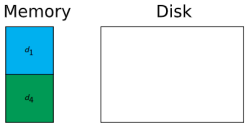
More complex tasks





More complex tasks





VANDERBILT UNIVERSITY

Evolution of computing systems

Three types of computing systems

- Grid, Internet, Volunteer computing
- Cloud, Fog, Edge
- Cluster, HPC

Grid computing

- Basically, distributed computing resources used together in a coordinated way
- Autonomous, distributed among multiple users
- Often heterogeneous and geographically distributed

Grid 5000

Grid computing system in France shared among 8 datacenters

- 8 sites, 31 machines, 828 nodes, 12328 cores
- 10Gbps dedicted network between sites
- 550 users

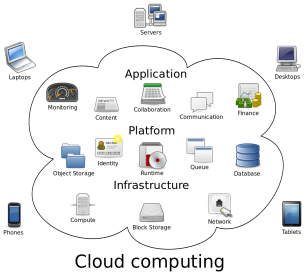
`https://www.grid5000.fr/`

VANDERBILT UNIVERSITY

Cloud computing

Cloud Originally by Amazon to use their unused machines (EC2)

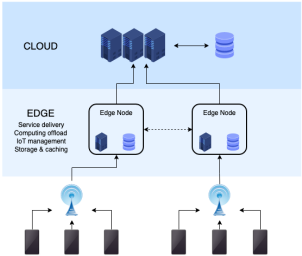
- On-demand access to a shared pool of dynamically configured computing resources and higher-level services
- Computing resources are located over multiple servers in clusters but managed centrally
- "Pay only for what you need" paradigm



Source: Wikipedia

Edge / Fog .. then their unused IOTs.

- Bringing processing closer to the creation of data
- Fog at local area network (LAN) level
- Edge at endpoints



VANDERBILT UNIVERSITY

High performance computing

- Large centralized machines
  - Used for large scientific applications (https://en.wikipedia.org/wiki/Grand_Challenges)
  - Capable of handling a lot of computation, a lot of data
- HPC system evolve together with large monolithic applications
  - Focus on performance
  - Developed by the community for years, tuned to scale and run on large-scale infrastructures

VANDERBILT UNIVERSITY

High performance computing

NUMA

Hardware threading

VANDERBILT UNIVERSITY

High performance computing

NUMA
Hardware threading

Accelerators
Memory hierarchy
High-bandwidth memory
Burst buffers, Vectorization

VANDERBILT UNIVERSITY

High performance computing

- NUMA
- Hardware threading
- Accelerators
- Memory hierarchy
- High-bandwidth memory
- Burst buffers, Vectorization
- NVLink, NVMe, NVSwitch, GPUDirect
- Unified Virtual Memory

VANDERBILT UNIVERSITY

High performance computing

NUMA

Hardware threading

Accelerators

Memory hierarchy

High-bandwidth memory

Burst buffers, Vectorization

NVLink, NVMe, NVSwitch, GPUDirect

Unified Virtual Memory

Chiplet architectures

Configurable Spatial Accelerator architecture

Computing on switch

VANDERBILT UNIVERSITY

Why parallel?

- It is not always obvious that a parallel algorithm has benefits, unless we want to do things ...
  - faster: doing the same amount of work in less time
  - bigger: doing more work in the same amount of time

- Both of these reasons can be argued to produce *better results*, which is the only meaningful outcome of program parallelization

Bigger/Faster example

Let's take a weather prediction simulation

- Suppose the atmosphere of the earth is divided into 5x10^8 cubes, each 1x1x1 mile and stacked 10 miles high
- It takes 200 floating point operations per cube to complete one time step
- 10^4 time steps are needed for a 7 day forecast
- Then 10^{15} floating point operations must be performed
- **This takes 10^6 seconds (= 10 days) on a 1 GFLOP machine**

Bigger/Faster example

Let's take a weather prediction simulation

- Suppose the atmosphere of the earth is divided into 5x10^8 cubes, each 1x1x1 mile and stacked 10 miles high
- It takes 200 floating point operations per cube to complete one time step
- 10^4 time steps are needed for a 7 day forecast
- Then 10^{15} floating point operations must be performed
- **This takes 10^6 seconds (= 10 days) on a 1 GFLOP machine**

On a perfect machine with all data in cache

VANDERBILT UNIVERSITY

Floating Points Ops per Second

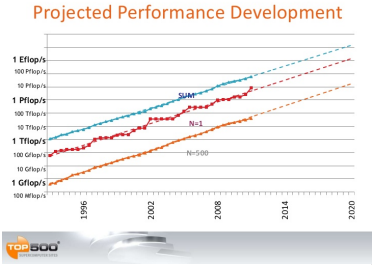
FLOPS = sockets * (cores per socket) * (number of clock cycles per second) * (number of floating point operations per cycle)

Let's take Intel Core i7-970

- 6 cores
- running at 3.2 GHz

FLOPS = 1 (socket) * 6 (cores) * 3.2 * 10^9 (cycles per second) * 8 (single-precision FLOPs per second)
= 153.6 * 10^9 single-precision FLOPs per second or
Performance (double-precision) of 76.8 GFLOP/s

What's your laptop's top performance?

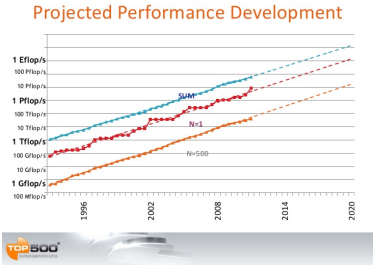


Source: Top500 SC11 BOF Slides

VANDERBILT UNIVERSITY

Bi-annual list of the most powerful (FLOPS) supercomputers in the world

- Uses the Linpack benchmark to compute the FLOPS
  - Measure how fast a computer solves a dense n by n system of linear equations Ax = b
  - Not a lot of communication or memory accesses



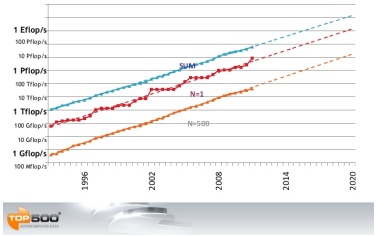
Source: Top500 SC11 BOF Slides

VANDERBILT UNIVERSITY

Bi-annual list of the most powerful (FLOPS) supercomputers in the world

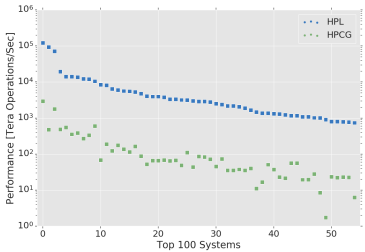
- Uses the Linpack benchmark to compute the FLOPS
  - Measure how fast a computer solves a dense n by n system of linear equations Ax = b
  - Not a lot of communication or memory accesses

Projected Performance Development



Source: Top500 SC11 BOF Slides

HPCG is a sparse solver benchmark that models the data access patterns of real-world applications



Summit has 196 Peta FLOPs. What is its theoretical performance?

VANDERBILT UNIVERSITY

Objective function

Connecting **applications** and **machines** while trying to optimize an **objective function**

What is the objective function?

Objective function

From the system's perspective

Administrators want to keep the system utilized

- Utilization (max) : percentage of the CPU time that is spent computing
- Power consumption (min)
- User fairness : give space on the machine to all users

Objective function

From the system's perspective

Administrators want to keep the system utilized

- Utilization (max) : percentage of the CPU time that is spent computing
- Power consumption (min)
- User fairness : give space on the machine to all users

From the user's perspective

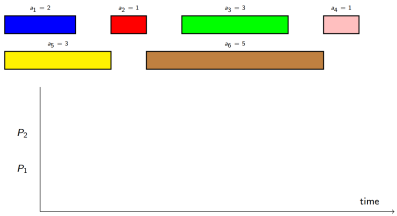
Users want their job to compute as fast as possible

- Makespan (min) : time to complete the job from start to end
- Response time (min) : time to complete the job from submission to end
- Stretch (min) : ration between the response time and the ideal execution time

Scheduling policies

The scheduler can be used to balance all the metrics

- User fairness
- System utilization
- Application response time

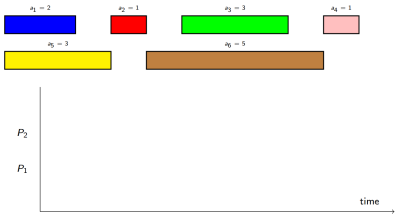


VANDERBILT UNIVERSITY

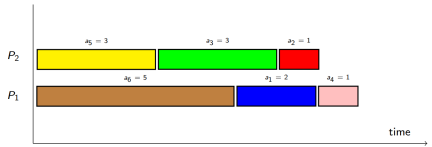
Scheduling policies

The scheduler can be used to balance all the metrics

- User fairness
- System utilization
- Application response time



Longest job first

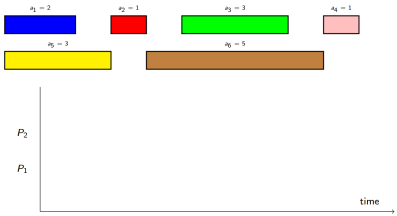


VANDERBILT UNIVERSITY

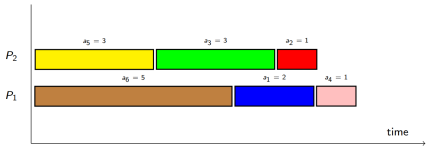
Scheduling policies

The scheduler can be used to balance all the metrics

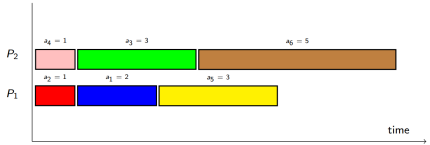
- User fairness
- System utilization
- Application response time



Longest job first



Shortest job first

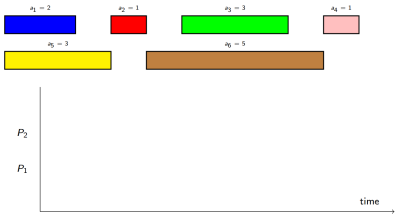


VANDERBILT UNIVERSITY

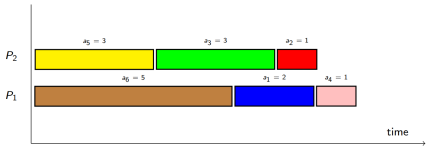
Scheduling policies

The scheduler can be used to balance all the metrics

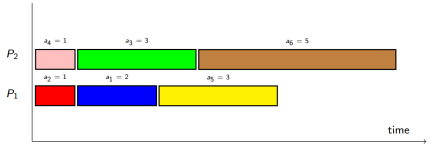
- User fairness
- System utilization
- Application response time



Longest job first



Shortest job first



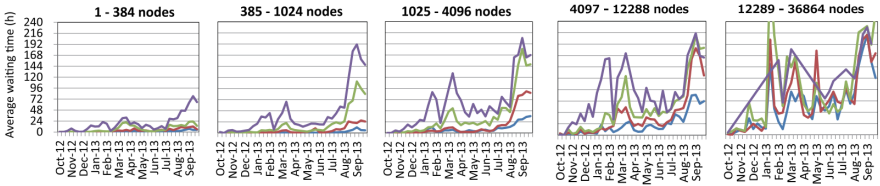
Priority based scheduling (with many optimizations)

VANDERBILT UNIVERSITY

What to remember

- Remember the hardware you will be running on
  - Type of processing units, memory layout, network
- Be aware that you are sharing resources.
  - Do not ask for 1k nodes until you know you need them
  - If you use less resources than you need, you might encounter penalties
  - If you use more resources than you need, your application will be killed

Example penalty Queue wait time is a function of the requested walltime, requested cores, platform occupancy, and platform policy.



Source: K. Yamamoto et al., "The K computer Operations: Experiences and Statistics", Elsevier Computer Science, Volume 29, 2014, Pages 576-585

VANDERBILT UNIVERSITY

When not parallel?

- Bad parallel programs can be worse than their sequential counterparts
  - **Slower**: because of communication overhead, resource contention
  - **Scalability**: some parallel algorithms are only faster when the problem size is very large

Understand the problem and use common sense!

- Focus on both designing good parallel algorithms and on non-parallel optimizations.

Let's look at some examples

```
for (i = 0; i < N; i++)
    z[i] = x[i] + y[i];
```

Let's look at some examples

```
for (i = 0; i < N; i++)
    z[i] = x[i] + y[i];
```

- Easy to parallelize
- No communication between processes
- No memory access conflicts

Let's look at some examples

```
for (i = 0; i < N; i++)
   z[i] = x[i] + y[i];
```

```
int Collatz(int n)
{
  int step;
  for (step = 1; n != 1; step++)
  {
    if (n % 2 == 0)
          n = n / 2;
        else
          n = 3*n + 1;
  }
  return step;
}
```

- Easy to parallelize
- No communication between processes
- No memory access conflicts

Let's look at some examples

```
for (i = 0; i < N; i++)
   z[i] = x[i] + y[i];
```

- Easy to parallelize
- No communication between processes
- No memory access conflicts

```
int Collatz(int n)
{
  int step;
  for (step = 1; n != 1; step++)
  {
    if (n % 2 == 0)
        n = n / 2;
      else
        n = 3*n + 1;
  }
  return step;
}
```

Not all problems are amenable to parallelism

- Step i in Collatz requires step i-1 to finish before executing

Let's look at some examples

```
for (i = 0; i < N; i++)
   z[i] = x[i] + y[i];
```

- Easy to parallelize
- No communication between processes
- No memory access conflicts

```
int Collatz(int n)
{
  int step;
  for (step = 1; n != 1; step++)
  {
    if (n % 2 == 0)
        n = n / 2;
      else
        n = 3*n + 1;
  }
  return step;
}
```

Not all problems are amenable to parallelism

- Step i in Collatz requires step i-1 to finish before executing

```
int Fibonacci(int n)
{
  fib[0] = 0;
  fib[1] = 1;
  for (i = 2; i < n; i++)
    fib[i] = fib[i-1] + fib[i-2];
}
```

Let's look at some examples

```
for (i = 0; i < N; i++)
   z[i] = x[i] + y[i];
```

- Easy to parallelize
- No communication between processes
- No memory access conflicts

```
int Collatz(int n)
{
  int step;
  for (step = 1; n != 1; step++)
  {
    if (n % 2 == 0)
        n = n / 2;
      else
        n = 3*n + 1;
  }
  return step;
}
```

Not all problems are amenable to parallelism

- Step i in Collatz requires step i-1 to finish before executing

```
int Fibonacci(int n)
{
  fib[0] = 0;
  fib[1] = 1;
  for (i = 2; i < n; i++)
    fib[i] = fib[i-1] + fib[i-2];
}
```

- Not as easy to parallelize ...
- but there are two ways:
  one for the iterative and
  one for the recursive version

Speed-up

Definition: the speedup of an algorithm using P processors is defined as:

$S_P = \frac{t_s}{t_P}$, where

- t_s is the execution time of the best available sequential algorithm
- t_P is the execution time of the parallel algorithm

- The speedup is perfect or ideal if $S_P = P$
- The speedup is linear if $S_P \approx P$
- The speedup is superlinear when $S_P > P$

- Ideally, the parallel version runs P times faster (**linear speedup**)
  - Typically not the case, because of overhead in creating the processes
  - There is also overhead in communication, resource contention, and synchronization

Usually the speed-up is limited by the fraction of the computation that is sequential

Speed-up Limitations

Several factors can limit the speedup

- **Amdahl's Law** gives the maximum speedup one can achieve for a program (given its fraction of the computation that cannot be parallelized)
- **Data dependencies**
  - The Collatz iteration loop has a loop-carried dependence
- **Overhead of the parallel program**
  - Extra computations are performed in the parallel version
  - Synchronization, communication, ...

To increase parallelism

- Change the loops to remove dependences when possible
- Apply algorithmic changes by rewriting the algorithm
  - Be careful about changing the result of the output or approximating it

Programming for HPC systems takeaways

- Understand **the problem** to be solved
- Understand the **machine architecture constraints**
- **Redesign** the algorithm with parallel execution in mind
- **Partition** the data appropriately
- Think about the intrinsic sequential part of your problem
- **Debugging** is much more complicated
- **Performance analysis** is no longer optional

Future lectures

Running applications on HPC

- Intro to C
- Intro to Linux and the batch submission system
- Libs and tools to develop your applications
- Performance analysis tools (PAPI, Perf, Vampir, Scalasca, etc.)
- Bug detection (Valgrind, gdb, etc.)
- Parallel programming languages based on C (could be applied to Fortran and python)

Designing parallel algorithms

- Mapping on HPC hardware

Further reading

Ask us about good articles on any of the subjects

- *Contemporary High Performance Computing* by Jeffrey S. Vetter, chapter on the **Overview of HPC applications**
- *Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers* (2nd ed.) by B. Wilkinson and M. Allen, Prentice Hall, pages 3-12
- *Sourcebook of Parallel Programming* by J. Dongara, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White (eds), Morgan Kaufmann, chapters on **Speed-up and how to increase application parallelism**.