Parallel Computer Architectures

10th Summer School in Statistics for Astronomers

Pierre-Yves Taunay

Research Computing and Cyberinfrastructure
224A Computer Building
The Pennsylvania State University
University Park
py.taunay@psu.edu

June 2014

PENNSTATE

Introduction

Objectives

1. Understand necessity for "going parallel"
2. Get familiar with HPC terms and architectures
3. Discover some strategies to leverage HPC resources
4. Learn about general problematics in HPC

PENNSTATE

Some acronyms and definitions

- **FLOPs** Floating Point Operation per Second
- **MPI** Message Passing Interface
- **HPC** High Performance Computing
- **GPU** Graphics Processing Unit
- **CPU** Central Processing Unit. Made of multiple **cores**.
- **Core** Smallest compute unit on a CPU **cluster**
- **Cluster** Ensemble of compute nodes, connected with high-perf. interconnect.
- **Compute node** Server/Machine that executes code. Contains 1 or more CPU.
- **Accelerator** "Add-on" card used to improve ("accelerate") execution time of code.

PENNSTATE

Motivation for HPC

- Why do we build big and expensive supercomputers ?
  - ↪ (At constant run time) Solve **larger** problems
  - ↪ (At constant problem size) Solve problems **faster**
- Need to increase the number of FLOPs, memory bandwidth
- A single core is not enough anymore !

PENNSTATE

Motivation for HPC

Examples

- 2012–2014 – PIConGPU:
  - ↪ 18,000 GPUs (!), 7.1 PFlops at peak
  - ↪ 1 s of runtime at peak = 34 days on a single core

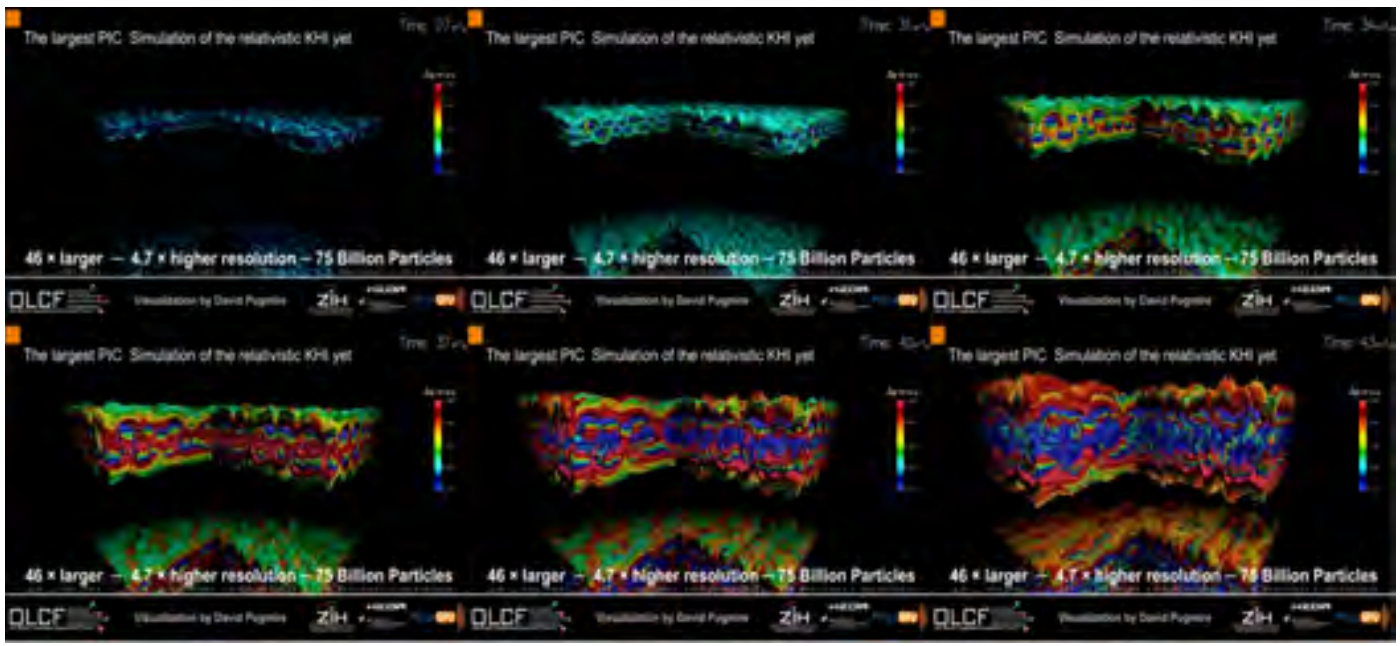


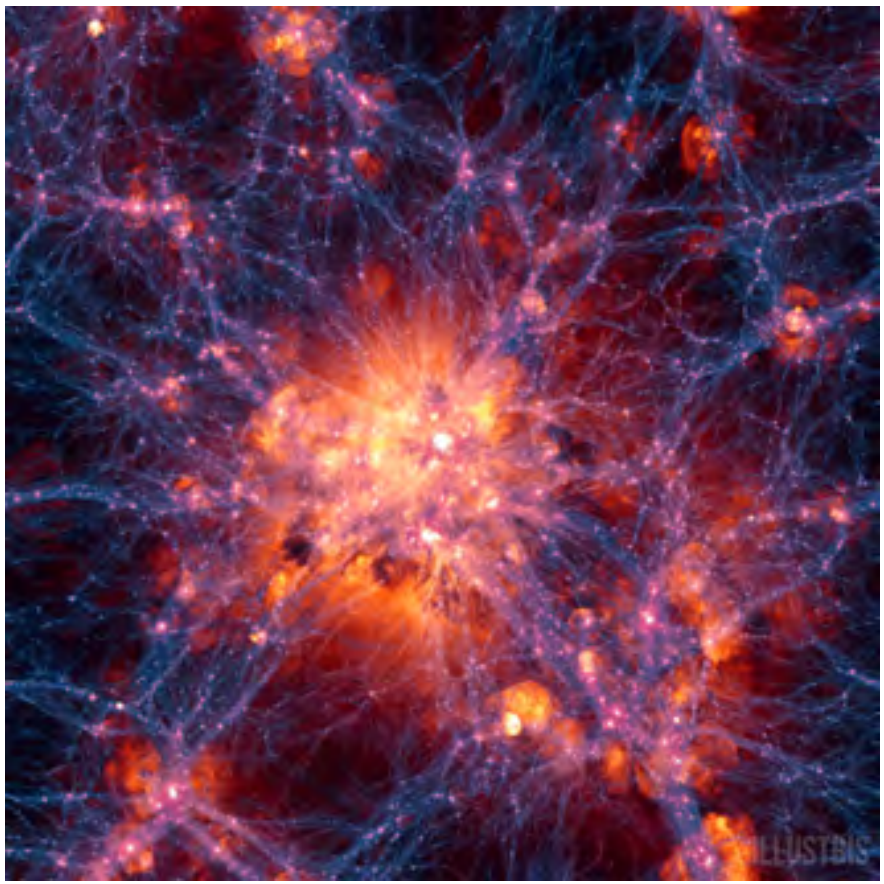
Figure: Kelvin–Helmholtz instability.

PENNSTATE

Motivation for HPC

Examples

- 2014 – Illustris project: cosmological simulation
  - ↪ 8,192 cores, 19M CPU-hours; 2,000 yrs on a single core



PENNSTATE

Figure: Dark matter density overlaid with gas velocity field.

HPC system overview

- **Cluster** Ensemble of compute nodes, connected with high-perf. interconnect.

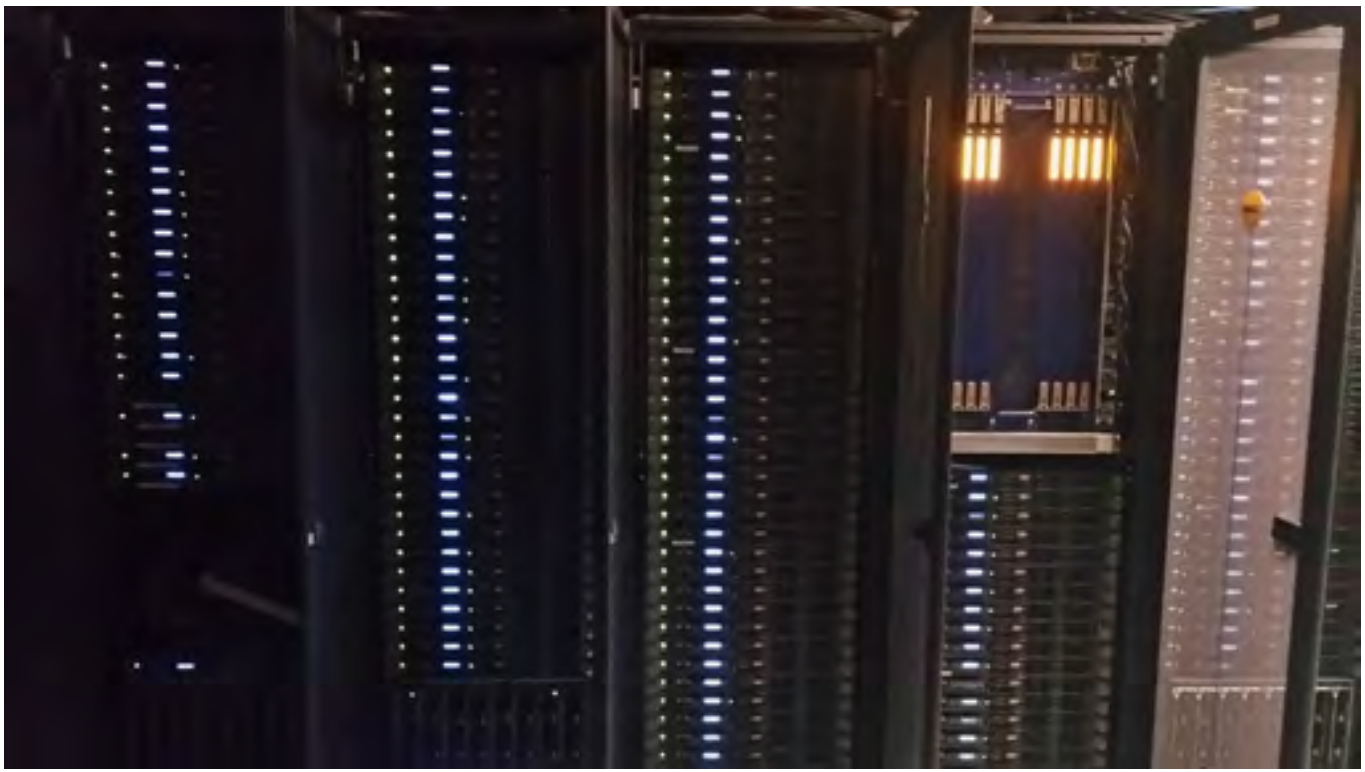


Figure: Cluster example

HPC system overview

- **Compute node** Server/Machine that executes code. Contains 1 or more CPU.

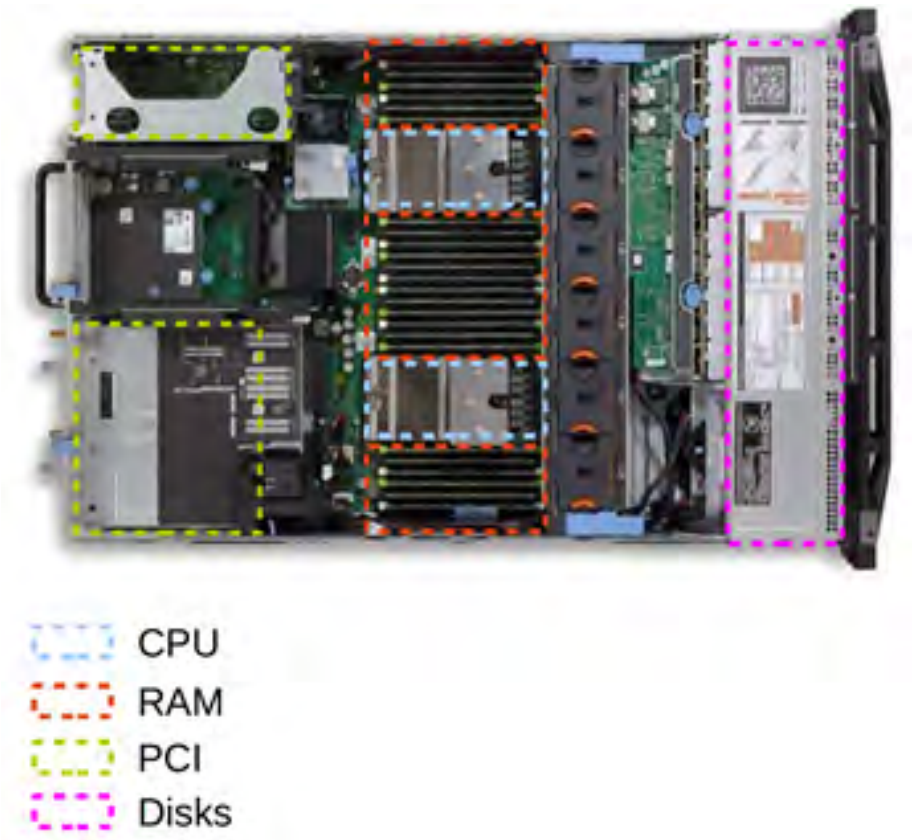


Figure: Dell R720

HPC system overview

- **Core** Smallest compute unit on a CPU cluster

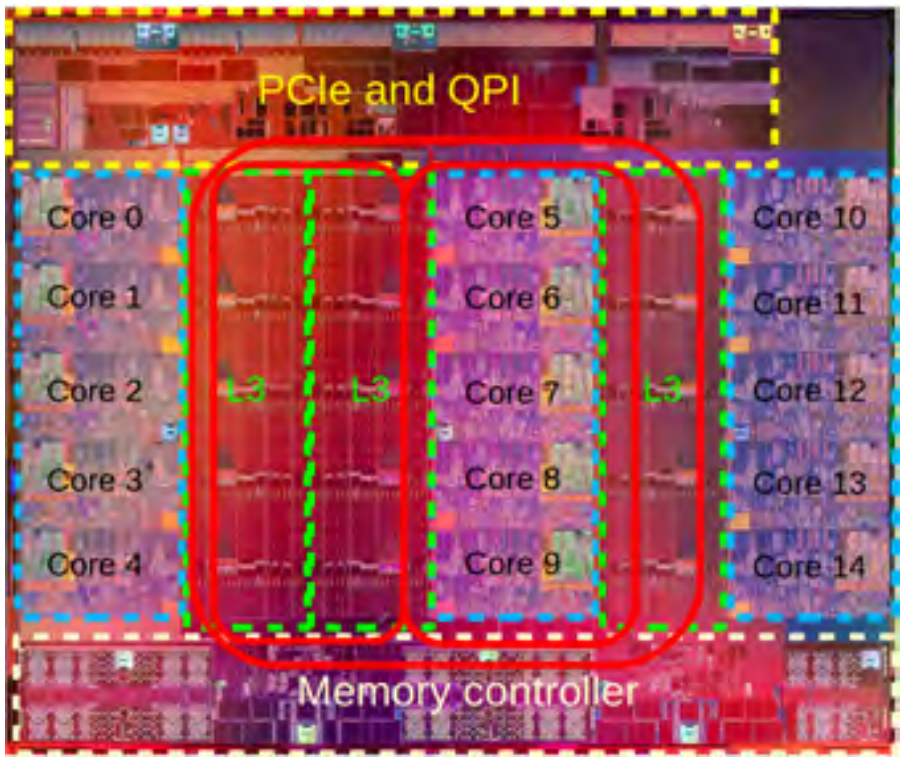
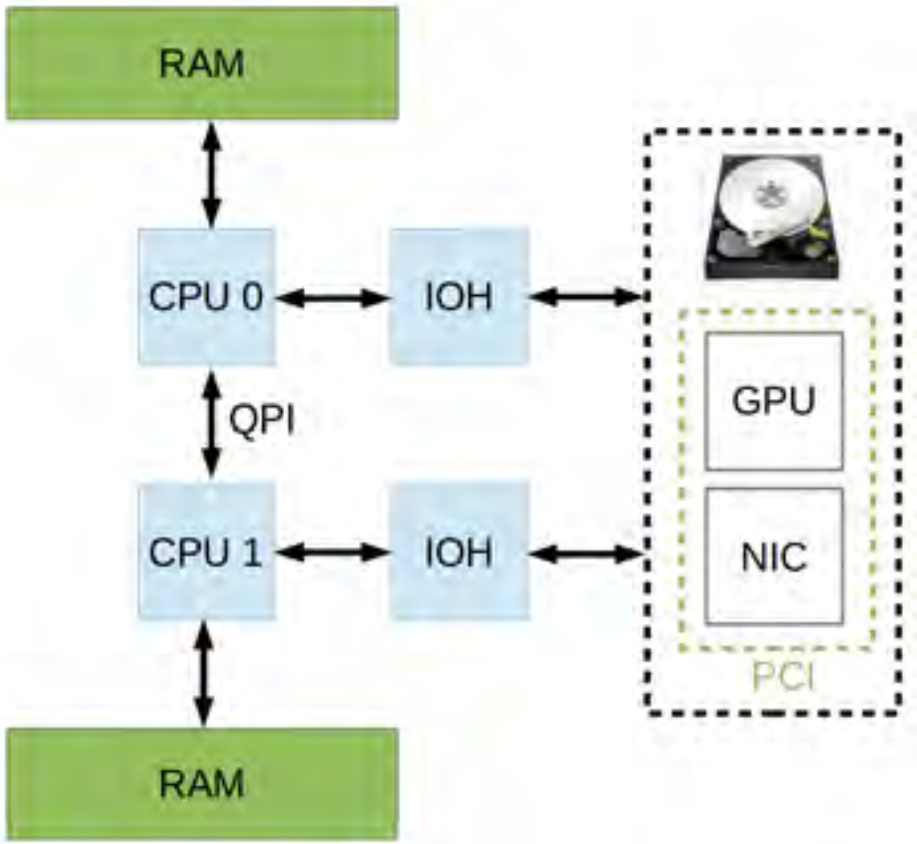


Figure: Ivybridge EX die

PENNSTATE

HPC system overview

Compute node



- **NIC** Network Interface Card
- **QPI** Quick Path Interconnect
- **IOH** Input/Output Hub

PENNSTATE

HPC system overview

Some numbers

Components	Theoretical speed
FDR Infiniband 4x	56 Gb/s
PCIe	32 GB/s
DRAM	10–200* GB/s
HDD	100–500 MB/s
QPI	40 GB/s

*low performance can be due to bad cache utilization

PENNSTATE

Some theory

Speed up and scaling – 1/2

- **Speed-up** and **scaling** are of utmost concern in HPC

Speed up

Factor of decrease in total execution time compared to a single core program.
$SU = t_{\mathrm{Nproc}}/t_{\mathrm{1proc}}$

Scaling

How well programs behave when you throw more resources at it.
Strong scaling: Measure the effect of a change in the number of processors while the total workload of a program stays constant. We have linear scaling if the speed-up is equal to the total number of processors.
$S_s = t_{\mathrm{1proc}}/(N \cdot t_{\mathrm{Nproc}}) \cdot 100$
Weak scaling: Measure the effect of a change in the number of processors while the workload per processor stays constant. We have linear scaling if the total run time stays constant.
$S_w = t_{\mathrm{1proc}}/(\cdot t_{\mathrm{Nproc}}) \cdot 100$

PENNSTATE

Some theory

Speed up and scaling – 2/2

Case study: PIConGPU

PENNSTATE

Some theory

Speed up and scaling – 2/2

- Speed up for 16–18,432 GPUs: 794 (ideal: 1,152)

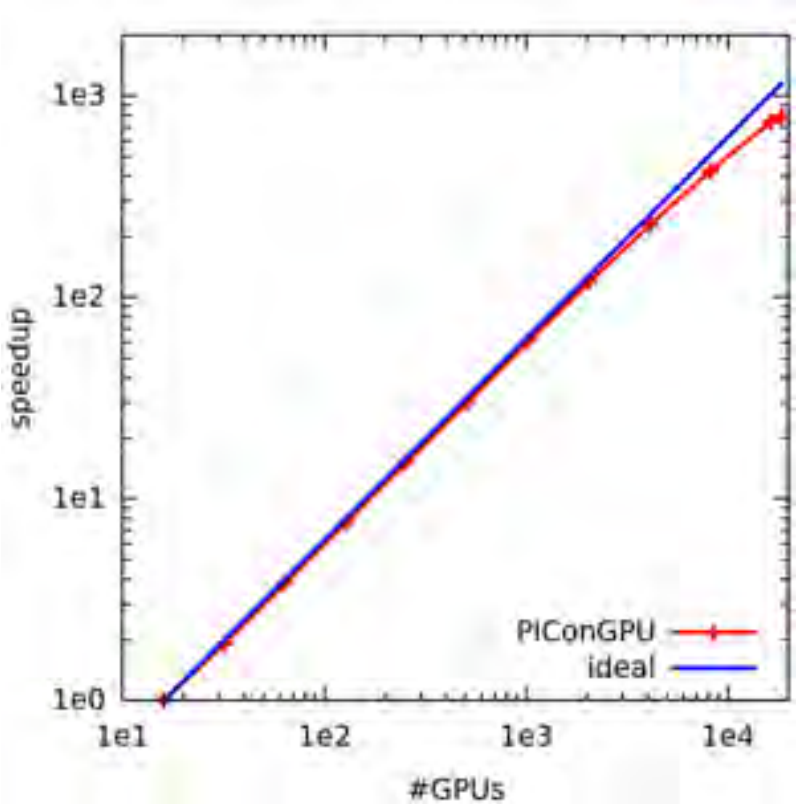


Figure: Speed up vs. number of GPUs

PENNSTATE

Some theory

Speed up and scaling – 2/2

- Strong scaling for 16–18,432 GPUs

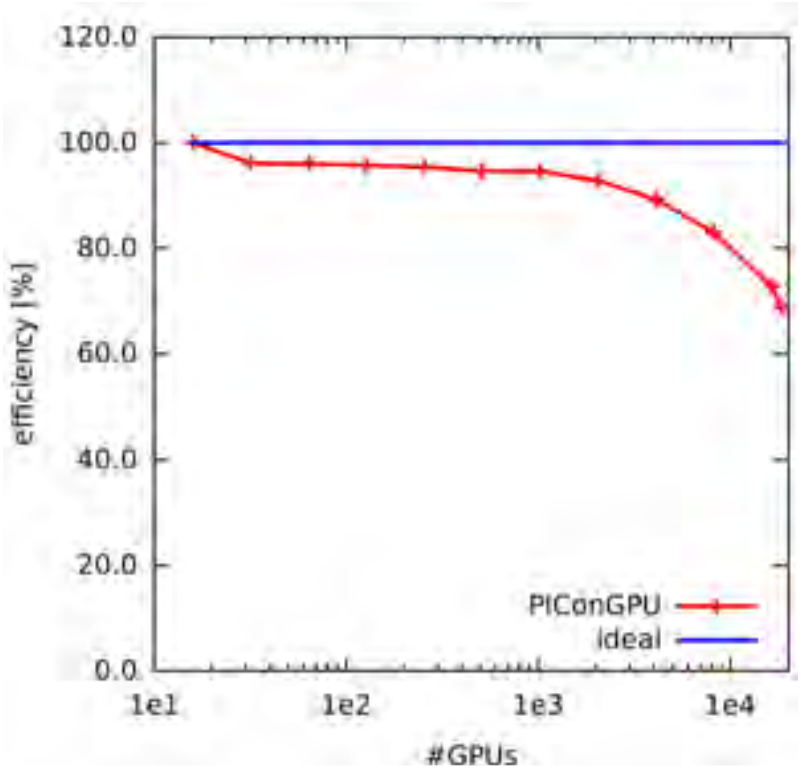


Figure: Strong scaling vs. number of GPUs

PENNSTATE

Some theory

Speed up and scaling – 2/2

- Weak scaling for 1–18,432 GPUs

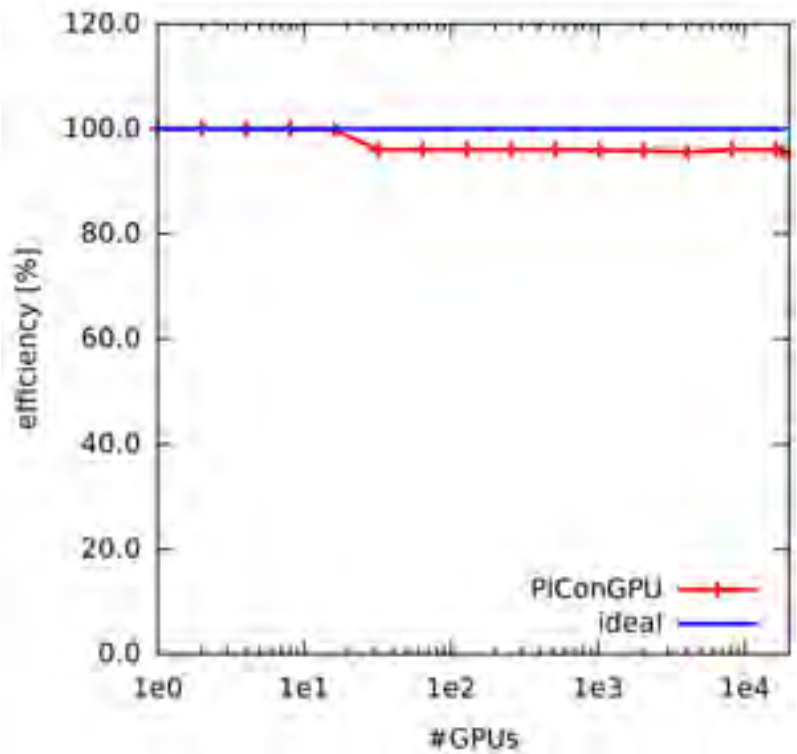


Figure: Weak scaling vs. number of GPUs

PENNSTATE

Some theory

Amdahl's law

Amdahl's law

Theoretical speed-up S can be estimated with

$$S = \frac{1}{(1-P) + P/N}, \tag{1}$$

where P is the percentage of code that can be parallelized, and N the total number of processor cores.

- **Remark**: Amdahl's law assumes that the problem size is fixed – the total workload of a program does not change with the number of processors (**strong scaling**).

PENNSTATE

Some theory

Amdahl's law

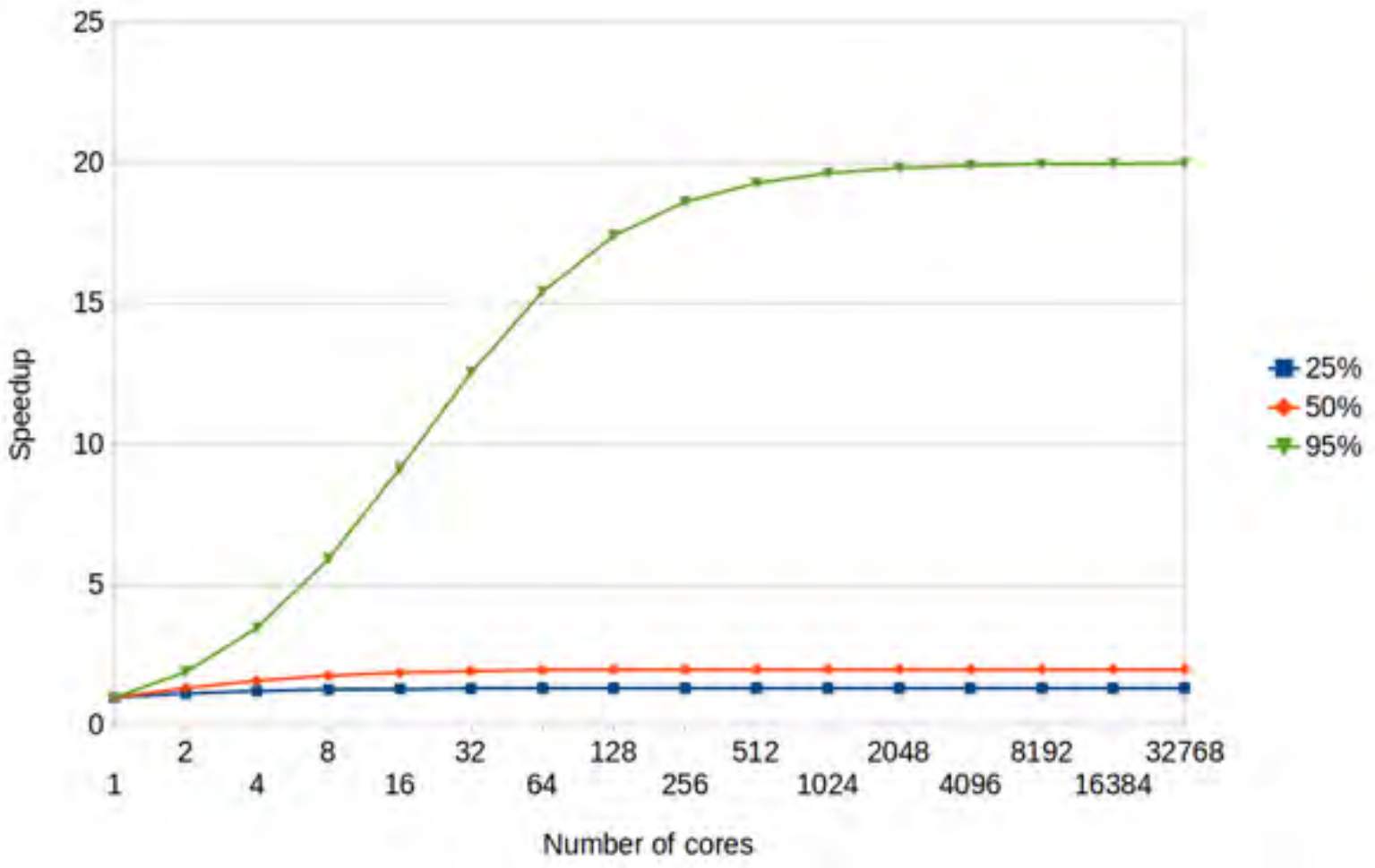


Figure: Amdahl's law

Some theory

Gustafson's law

Gustafson's law

Theoretical speed-up S can be estimated with

$$S = N - (1 - P) \cdot (N - 1), \tag{2}$$

where P is the percentage of code that can be parallelized, and N the total number of processor cores.

Some theory

Example

- Assume that 98% of the code can be parallelized. You run code on a CPU with 12 cores.
  - ↪ Amdahl: $S = 9.84$
  - ↪ Gustafson: $S = 11.8$

PENNSTATE

Summary

We now know about

- Motivation for HPC
- HPC systems
- A bit of theory about scaling and speed-up

PENNSTATE

What is HPC ?

- Using one or more forms of parallelism to improve the performance and scaling of your code
  - ↪ Vector architecture
  - ↪ Shared memory parallelism
  - ↪ Distributed memory parallelism
  - ↪ (Not today) Accelerators e.g., Graphics Processing Units

PENNSTATE

Vectorization

Definition

Vectorization

Parallelization of a program which performs scalar operations to a program that performs vector operations. A vector operations allow for operating on multiple pairs of scalar at once. When a processor perform the same instruction on multiple scalars at once, we talk about "Single Instruction Multiple Data" (SIMD).

		Data	
		Single	Multiple
Instructions	Single	Single core, serial $a = b + c$	Single core, vectorized $\vec{a} = \vec{b} + \vec{c}$
	Multiple	Uncommon	Distributed systems $\vec{a} = \vec{b} + \vec{c}$ $x = y * z$

Table: Flynn's taxonomy

Example

Likelihood estimation – 1/2

- Multivariate normal PDF for $\mathbf{X} \in \mathbb{R}^N$

$$\mathrm{F}\left(\mathbf{X}; \boldsymbol{\mu}, \Sigma\right) = \frac{1}{\sqrt{(2\pi)^N \left|\Sigma\right|}} \cdot \mathrm{e}^{-\frac{1}{2}(\mathbf{X}-\boldsymbol{\mu})^{\mathrm{T}} \Sigma^{-1} (\mathbf{X}-\boldsymbol{\mu})}$$

- Likelihood for M observations:

$$\mathrm{L} = \prod_{i=1}^{M} \mathrm{F}\left(\mathbf{X}_i; \boldsymbol{\mu}, \Sigma\right)$$

PENNSTATE

Example

Likelihood estimation – 2/2

- Log–likelihood for M observations:

$$\ln \mathrm{L} = \sum_{i=1}^{M} \ln \mathrm{F}\left(\mathbf{X}_i; \boldsymbol{\mu}, \Sigma\right)$$

$$\ln \mathrm{L} = -1/2 \cdot \left[MN \ln 2\pi + M \ln |\Sigma| + \sum_{i=1}^{M} \left(\mathbf{X_i} - \boldsymbol{\mu}\right)^{\mathrm{T}} \Sigma^{-1} \left(\mathbf{X_i} - \boldsymbol{\mu}\right) \right]$$

- Calculate for various M; data is provided from multivariate normal PDF with random cov. and expected val.
- Size of input data:
  - ↪ Random variable observations: $\mathbf{X} = [NOBS \times SIZE]$
  - ↪ Expected val.: $\boldsymbol{\mu} = [1 \times SIZE]$
  - ↪ Covar.: $\Sigma^{-1} = [SIZE \times SIZE]$

Example I

Python – 1/2

```
1  def log_likelihood_naive(obs,nobs,isig,mu,det_sig):
2    sum = 0
3    # Sum all members
4    for i in range(0,nobs):
5      LV = obs[i]-mu;
6      RV = np.transpose(LV);
7      sum += np.dot(np.dot(LV,isig),RV)
8
9    # Size of random variable
10   N = mu.shape[0]
11
12   # Add the rest to sum
13   sum += nobs*N*math.log(2*np.pi) + nobs*math.log(abs(
         det_sig))
14   sum *= -0.5
```

Example II

Python – 1/2

```
1  def log_likelihood_vec(obs,nobs,isig,mu,det_sig):
2
3    # Size of random variable
4    N = mu.shape[0]
5
6    # Left hand side: (X-mu)T*SIG
7    LV = obs[:]-mu
8    tmp = np.dot(LV,isig)
9
10   # Right hand side: (X-mu)
11   RV = np.reshape(LV,[nobs*N,1])
12
13   # Reshape left hand side
14   LV = np.reshape(tmp,[1,nobs*N])
15
16   # Perform single sum through dot product
17   sum = np.dot(LV,RV)
18
19   # Rest is the same
```

PENNSTATE

Example

Python – 2/2

M	Non-vectorized	Vectorized	Speed–up
10^1	9.63×10^{-5}	3.22×10^{-5}	2.95
10^2	7.87×10^{-4}	8.22×10^{-5}	9.57
10^4	7.61×10^{-2}	5.43×10^{-3}	14.0
10^6	7.56	3.23×10^{-1}	23.4

Table: Timings of two Python implementation of likelihood estimation

PENNSTATE

Example I

R – 1/2

```r
log_likelihood_naive <- function(obs, nobs, isig, mu, det_sig)
{
  sum = 0
  for (i in seq(1,nobs))
  {
    LV <- obs[i,] - mu
    RV <- t(LV)
    sum <- sum + LV %*% isig %*% RV
  }

  # Size of random variable
  N <- dim(mu)[2]
  sum <- sum + nobs*N*log(2*pi) + nobs*log(abs(det_sig))
  sum <- sum*(-0.5)
  return(sum)
}
```

Example II

R – 1/2

```r
log_likelihood_vec <- function(obs, nobs, isig, mu, det_sig)
{
  sum = 0

  # Size of random variable
  N <- dim(mu)[2]

  # Create a temporary container for subtraction X-mu
  tmp <- matrix(1, nrow=nobs, ncol=1)
  tmp <- tmp %*% mu
  LV <- X-tmp

  # First part of the product (X-mu)T*SIG
  tmp <- LV %*% isig

  # RHS
  RV <- matrix(LV, nrow=1, byrow=TRUE)
  RV <- t(RV)

  # Make that an array
```

PENNSTATE

Example III

R – 1/2

```
21    LV <- matrix(tmp, nrow=1, byrow=TRUE)
22
23    # Single sum through dot product
24    sum <- LV %*% RV
25
26    sum <- sum + nobs*N*log(2*pi) + nobs*log(abs(det_sig))
27    sum <- sum*(-0.5)
28
29    return(sum)
30  }
```

PENNSTATE

Example

R – 2/2

M	Non-vectorized	Vectorized	Speed–up
10^1	1.31×10^{-3}	1.10×10^{-3}	1.19
10^2	3.50×10^{-3}	1.17×10^{-3}	3.00
10^4	2.38×10^{-1}	1.52×10^{-2}	16.0
10^6	23.9×10^{1}	1.63	14.7

Table: Timings of two R implementation of likelihood estimation

PENNSTATE

Example I

IDL – 1/2

```
1  FUNCTION log_likelihood_naive, OBS, NOBS, ISIG, MU, DET_SIG
2    SUM = 0
3    FOR i = 0,NOBS-1 DO BEGIN
4      LV = OBS[*,i]-MU
5      RV = TRANSPOSE(LV)
6      SUM = SUM + LV ## ISIG ## RV
7    ENDFOR
8
9    ; Size of a random variable
10   N = SIZE(MU,/DIMENSIONS)
11
12   SUM = SUM + NOBS*N*ALOG(2*!PI) + NOBS*ALOG(ABS(DET_SIG))
13   SUM = -0.5*SUM
14
15   RETURN, SUM
16 END
```

PENNSTATE

Example II

IDL – 1/2

```
1  FUNCTION log_likelihood_vec, OBS, NOBS, ISIG, MU, DET_SIG
2    SUM = 0
3
4    ; Size of a random variable
5    N = SIZE(MU,/DIMENSIONS)
6
7    ; Create a tempprary container for subtraction of X-mu
8    TMP = MAKE_ARRAY(1,NOBS,/DOUBLE, VALUE=1)
9    TMP = TMP ## MU
10   LV = OBS-TMP
11
12   ; First part of the product
13   TMP = LV ## ISIG
14
15   ; Right hand side
16   RV = REFORM(LV,NOBS*N,1)
17   RV = TRANSPOSE(RV)
18
19   ; Reform TMP into LV as a row-array
20   LV = REFORM(TMP,NOBS*N,1)
```

Example III

IDL – 1/2

```
21
22    ; Single sum through dot product
23    SUM = LV ## RV
24
25    SUM = SUM + NOBS*N*ALOG(2*!PI) + NOBS*ALOG(ABS(DET_SIG))
26    SUM = -0.5*SUM
27
28    RETURN, SUM
29 END
```

Example

IDL – 2/2

M	Non-vectorized	Vectorized	Speed–up
10^1	2.78×10^{-5}	2.01×10^{-5}	1.38
10^2	2.45×10^{-4}	1.22×10^{-4}	2.01
10^4	2.39×10^{-2}	5.90×10^{-3}	4.03
10^6	2.38	4.59×10^{-1}	5.19

Table: Timings of two IDL implementation of likelihood estimation

PENNSTATE

Example I

Julia – 1/2

```
function log_likelihood_naive(obs::Array{Float64,2},nobs::
    Int64,isig::Array{Float64,2},mu::Array{Float64,1},
    det_sig::Float64)
  @assert(size(obs,1)==length(mu)==size(isig,1)==size(isig
      ,2))
  N = size(obs,1)
  RV = Array(Float64,1)

  tot = 0.0
  for i=1:nobs
    RV = obs[:,i] - mu
    tot += RV'*isig*RV
  end

  tot += nobs*N*log(2*pi) + nobs*log(abs(det_sig))
  tot *= (-0.5)

  return tot
end
```

Example II

Julia – 1/2

```
1  function log_likelihood_vec(obs::Array{Float64,2},nobs::
       Int64,isig::Array{Float64,2},mu::Array{Float64,1},
       det_sig::Float64)
2    @assert(size(obs,1)==length(mu)==size(isig,1)==size(isig
         ,2))
3    tot = 0.0
4    N = size(mu,1)
5    RV = Array{Float64,2}
6
7    # X-mu
8    RV = obs[:,1:nobs].-mu
9
10   # X-mu * isig
11   tmp = isig*RV
12
13   # Reshape to single array
14   # Some transpose to do to flatten the array
15   RV = reshape(RV,nobs*N,1)
16   tmp = reshape(tmp,nobs*N,1)
17
```

Example III

Julia – 1/2

```
18    tot = (transpose(RV)*tmp)[1,1]
19
20    tot += nobs*N*log(2*pi) + nobs*log(abs(det_sig))
21    tot *= (-0.5)
22
23    return tot
24 end
```

PENNSTATE

Example

Julia – 2/2

M	Naive	Vectorized	Speed–up
10^1	3.29×10^{-5}	1.07×10^{-5}	3.07
10^2	2.17×10^{-4}	4.71×10^{-5}	4.61
10^4	2.36×10^{-2}	8.01×10^{-3}	2.95
10^6	2.39	6.49×10^{-1}	3.68

Table: Timings of two Julia implementation of likelihood estimation

Compiled code vectorization

- Compiled code can benefit from vectorization
- Some Intel processors support vectorization through specific sets of instructions (SSE, AVX, etc.)
- Enable it at compile time:
  - ↪ `-xHost` enables best support for vectorization for a particular machine
  - ↪ or use specific set of instructions: `-xavx` enables AVX. See manual for more.

PENNSTATE

Compiled code vectorization

Example – 1/2

```
1  #define N 1000000
2
3  int main() {
4    ...
5    double tic = omp_get_wtime();
6
7    for(int i = 0;i<N;i++)
8      R[i] = V1[i] + V2[i]*V3[i];
9
10   double toc = omp_get_wtime();
11
12   toc -= tic;
13
14   printf("Total time: %f\n", toc);
15   ...
16   return 0;
17 }
```

PENNSTATE

Compiled code vectorization

Example – 2/2

- -O1 -xHost: 0.00431141
- -O1: 0.0056924
- Intructions executed at core level
  - ↪ Vectorized: `vmulsd` (r13,rbp,8), xmm0, xmm1
  - ↪ Not vectorized: `mulsd` (r13,rbp,8), xmm0, xmm1

PENNSTATE

Summary

- SIMD vectorization happens at the processor core level
- Can control vectorization with some interpreted languages
- Compiled languages: compile time to enable specific instructions
- Next parallel level: **multi–core**, **shared memory**

PENNSTATE

Multi-core architectures

Why multi-core ?

- Increase performance on a single core
  - ↪ Increase the clock–speed
  - ↪ Cram more transistors
- BUT power dissipation problems
- Processors from '00 to '14 have similar clock-rates
  - ↪ More recent ones have more cores

Xeon processor	Year released	Number of cores	Clock freq. (GHz)
Nehalem	2010	8	2.26
Westmere	2011	10	2.4
Sandy Bridge	2012	8	3.1
Ivy Bridge	2013	15	2.8

Table: Clock speed and number of cores for recent CPUs

PENNSTATE

Multi–Core architecture

Ivybridge EX

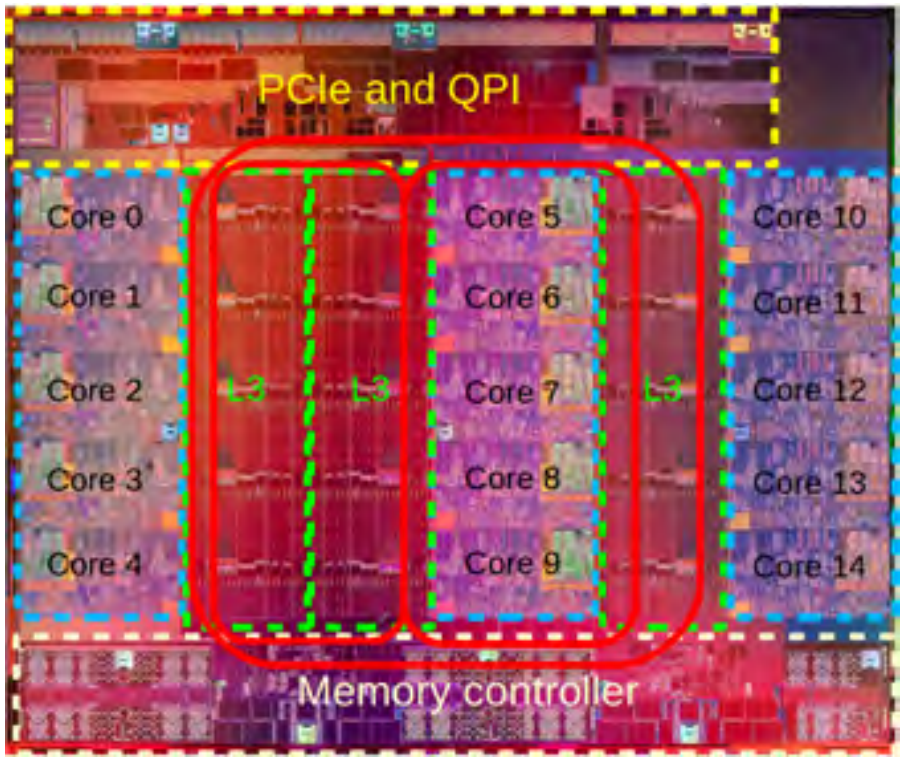


Figure: Ivybridge EX die

- Each core has a unique cache (L1, L2)
- L3 cache is shared by all cores

Cache interlude – 1/3

Cache

Smaller, faster memory that contains copy of data from DRAM, and/or instructions

- What happens when CPU core tries to access memory ?
  - ↪ Cache hit: data read from the cache. Fast.
  - ↪ Cache miss: copy data from DRAM, then data is read. Slow.
- Large cache size ↔ slow cache. Hence hierarchy (L1, L2, ...)

PENNSTATE

Cache interlude – 2/3

- How does the data get in the caches ?
  - ↪ Compiler
  - ↪ Hardware
  - ↪ OS
- Programmer is still responsible for some data access patterns !

```
1  // Flattened matrix
2  SIZE_X=2048;
3  SIZE_Y=2048;
4
5  double * data = (double*) malloc(SIZE_X*SIZE_Y*sizeof(double));
6    for (int i=0; i<SIZE_X; i++)
7      for (int j=0; j<SIZE_Y; j++)
8        data[j+SIZE_Y*i] = 10.0 * 3.14;
9        //bad data access
10       //data[i+SIZE_Y*j] = 10.0 * 3.14;
```

- Good access: 0.576 s
- Bad access: 1.04 s

PENNSTATE

Cache interlude – 3/3

- Use valgrind and cachegrind to measure cache hit rate

```
valgrind --tool=cachegrind ./myprogram
```

```
[pzt5044@lionxg cache]$ valgrind --tool=cachegrind ./a.out
Bad access: 1.041008
==25486==
==25486== I   refs:      63,328,247
==25486== I1  misses:         1,059
==25486== LLi misses:         1,053
==25486== I1  miss rate:       0.00%
==25486== LLi miss rate:       0.00%
==25486==                     Read ops          Write ops
==25486== D   refs:      42,098,558  (33,668,615 rd   + 8,429,943 wr)
==25486== D1  misses:     4,200,008  (     4,632 rd   + 4,195,376 wr)
==25486== LLd misses:     4,198,736  (     3,444 rd   + 4,195,292 wr)
==25486== D1  miss rate:        9.9% (       0.0%     +       49.7%  )
==25486== LLd miss rate:        9.9% (       0.0%     +       49.7%  )
==25486==
==25486== LL refs:        4,201,067  (     5,691 rd   + 4,195,376 wr)
==25486== LL misses:      4,199,789  (     4,497 rd   + 4,195,292 wr)
==25486== LL miss rate:         3.9% (       0.0%     +       49.7%  )
```

Instructions

Data

Lowest level

PENNSTATE

Multiple cores; same machine

Back to multi-core. How to leverage such architecture ?

- Usually through **threads** e.g. OpenMP or POSIX threads
- One **process** spawns multiple threads
- Each thread can **share memory** with other threads
- Usually map 1 thread to 1 processor core
  - ↪ You can oversubscribe cores... but perf. hits

PENNSTATE

Multiple cores; same machine

OpenMP

- Compiler directives, and environment variables for creating multi-threaded applications
  - ↪ Statements interpreted
  - ↪ Represents functionality like fork and join
- Directives indicate parallel sections of code e.g.:

```
1 #pragma omp parallel for
2 for (int i = 0; i <N; i++)
3   A[i] = B[i] + C[i];
```

- Standard managed by review board + HW vendors
- Compile with -openmp with Intel Compilers
- -fopenmp for GCC

PENNSTATE

Summary

- Multiple core processors are ubiquitous in supercomputers
- Possible to have 2 or more cores in a processor cooperate through **threads**
- Thread libraries: OpenMP, POSIX, Boost
- Cache effects can be treacherous ! Use `callgrind`
- Next parallel level: **distributed computations**

PENNSTATE

Distributed architectures

Multiple compute nodes

Motivation

- Some problems can not fit on only one compute node
  - ↪ Multi-TB problems ?
- Or need to use many cores for speedup
  - ↪ Remember Amdahl and Gustafson
- Communication across nodes usually relies on a very fast interconnect
  - ↪ 10 GigE : 10 Gb/s
  - ↪ Infiniband : up to 56 Gb/s

PENNSTATE

Leveraging distributed resources

Communication

- Sockets over TCP/IP (yikes !)
- **MPI** Message Passing Interface
  - ↪ Widely used standard for scientific apps.

PENNSTATE

Leveraging distributed resources

Core to process mapping

- In this approach: 1 processor core = 1 process (usually)
  - ↪ You launch multiple processes on multiple cores, executing different / the same instructions
- Programmer is responsible for defining each process task

PENNSTATE

Shared memory vs. distributed memory

Shared memory	**Distributed memory**
Limited to 1 machine	No limit
1 process = multiple cores	1 process = 1 core
Multiple cores can access same memory	Memory local to processor core
Data exchange through memory bus	Data exchange through network

Summary

- Use distributed computing for large parallelizable problems
- MPI for communication
- Underlying network: Infiniband (fast !)
- 1 core = 1 process

PENNSTATE

General problematics in HPC

General problematics

- Resource contention
- I/O
- Scalability
- Software dev.

PENNSTATE

General problematics

Resource contention

- Thread 1 wants to write at location A. At the same time, Thread 2 tries to read from location A. Who wins ?
- Programmer responsible for avoiding these
- `valgrind` and `helgrind` can detect race conditions in compiled code

```
valgrind --tool=helgrind ./myprog
```

PENNSTATE

General problematics

I/O

- I/O kills !
- Why ?
  - ↪ Control returned to the kernel, application stalls: **voluntary context switch**
  - ↪ Disk might be used by other processes
  - ↪ Lower BW; high latency
  - ↪ Parallel FS: additional BW and latency

PENNSTATE

Software development – 1/2

- Step 1: Prototype
- Step 2: Debug
- Step 3: Measure perf.
- Step 4: Target routines for optimization
- Go back to step 1

Donald Knuth

Premature optimization is the root of all evil.

PENNSTATE

Software development – 2/2

Use version control !

- Git
- SVN
- CVS
- RCS
- ...

Conclusion

- Motivation for HPC
  - ↪ Solve bigger problems, faster
- HPC system
  - ↪ One cluster = multiple compute nodes
  - ↪ One compute node = 1 or more CPU + memory
  - ↪ One CPU = multiple cores
- Leveraging HPC resources
  - ↪ Core level: vectorization
  - ↪ CPU level: multi-core programming
  - ↪ Cluster level: distributed programming

PENNSTATE

Questions ?