Hackaton Saturday May 20 2017

The Dutch C++ Group

@TheDutchCppGrp

Welcome

Sponsor

optiver

Welcome

Summary

Knights Landing (KNL) is the first self-boot Intel Xeon Phi processor

Many improvements for performance and programmability

Significant leap in scalar and vector performance

Significant increase in memory bandwidth and capacity

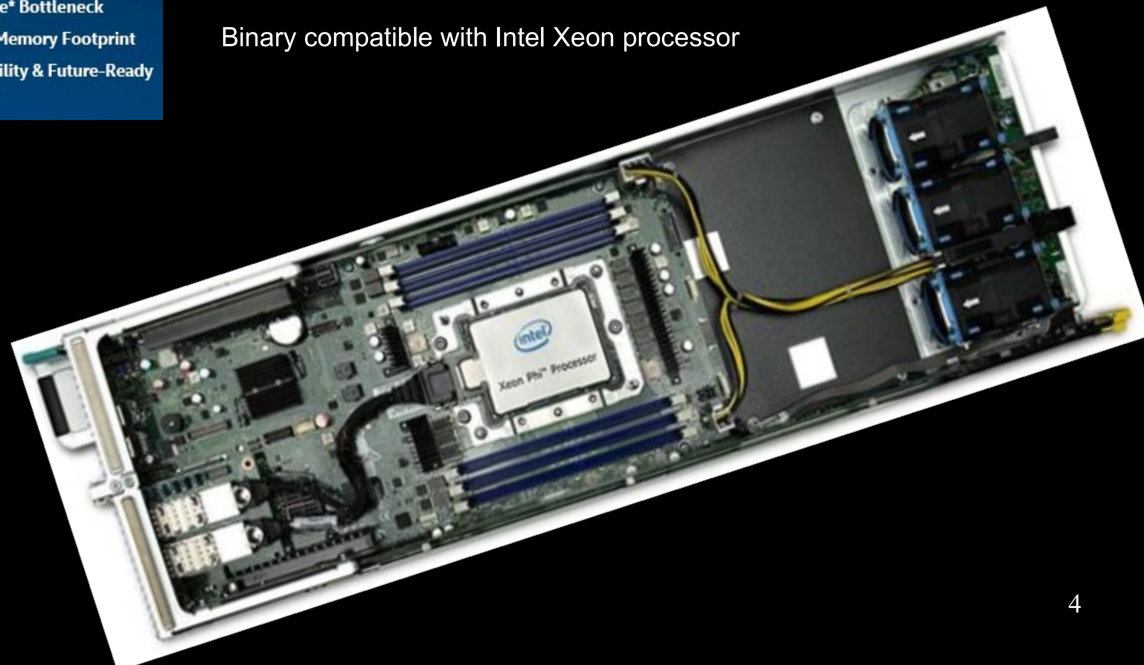
Binary compatible with Intel Xeon processor

Common programming models between Intel Xeon processor and Intel Xeon Phi processor

KNL offers immense amount of parallelism (both data and thread)

Future trend is further increase in parallelism for both Intel Xeon processor and Intel Xeon Phi processor

Developers need to prepare software to extract full benefits from this trend



Sponsor

Bright Computing

@TheDutchCppGrp

#XEONPHI

and/or

#DUTCHCPP

Day Program

09:00 Plenary start

09:10 Introduction Xeon Phi

10:15 Coffee break

10:45 Teams start coding

12:30 Lunch

Day Program

13:15 Teams continue coding

15:00 Coffee break

15:30 Teams continue coding

17:30 Diner

18:30 Teams continue coding

20:00 Team presentations

20:30 Wrap up

Xeon Phi

The Intel® Xeon Phi™ processor is a bootable host processor that delivers massive parallelism and vectorization to support the most demanding high-performance computing applications. The integrated and power-efficient architecture delivers significantly more compute per unit of energy consumed versus comparable platforms to give you an improved total cost of ownership.1 The integration of memory and fabric topples the memory wall and reduces cost to help you solve your biggest challenges faster.

Xeon Phi

The Intel® Xeon Phi™ processor is a bootable host processor that delivers massive parallelism and vectorization to support the most demanding high-performance computing applications. The integrated and power-efficient architecture delivers significantly more compute per unit of energy consumed versus comparable platforms to give you an improved total cost of ownership. The integration of memory and fabric topples the memory wall and reduces cost to help you solve your biggest challenges faster.

Xeon Phi

The Intel® Xeon Phi™ processor is a bootable host processor that delivers massive parallelism and vectorization to support the most demanding high-performance computing applications. The integrated and power-efficient architecture delivers significantly more compute per unit of energy consumed versus comparable platforms to give you an improved total cost of ownership. The integration of memory and fabric topples the memory wall and reduces cost to help you solve your biggest challenges faster.

Flynn's taxonomy

SISD - Single instruction stream single data stream.

SIMD - Single instruction stream, multiple data streams.

MISD - Multiple instruction streams, single data stream.

MIMD - Multiple instruction streams, multiple data streams.

Xeon Phi

The Intel® Xeon Phi™ processor is a bootable host processor that delivers massive parallelism and vectorization to support the most demanding high-performance computing applications. The integrated and power-efficient architecture delivers significantly more compute per unit of energy consumed versus comparable platforms to give you an improved total cost of ownership. The integration of memory and fabric topples the memory wall and reduces cost to help you solve your biggest challenges faster.

Vectorization example

This code example shows a simple loop that can be auto vectorized by the compiler. Intel provides a document* about auto vectorization with more examples.

```
for (j = 0;j < 256; j++) {
 a[i] += b[i];
}
```

*Red text are hyperlinks

Cray XC with Xeon Phi

Cray XC with Xeon Phi

Xeon Phi Supercomputers

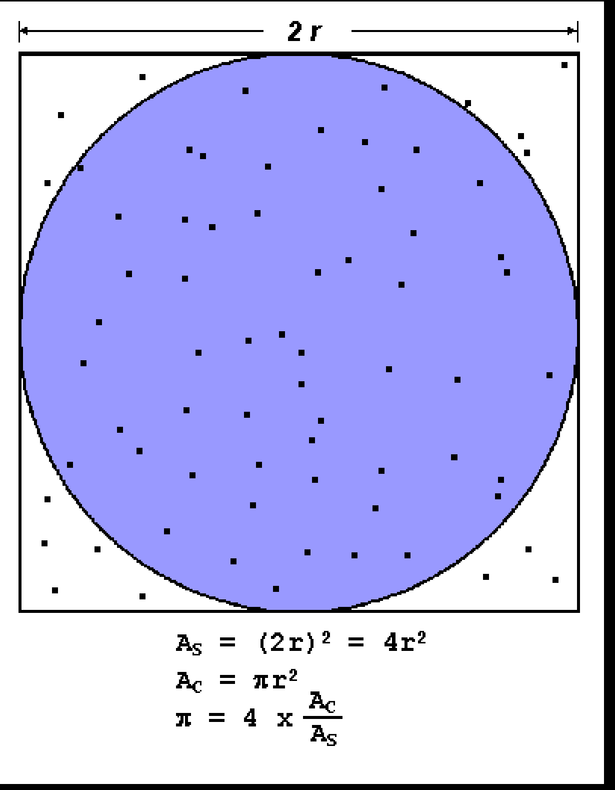
#	Name	Technology	Cores
1	National Supercomputing Center in Wuxi China	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway	10,649,600
2	National Super Computer Center in Guangzhou China	Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P	3,120,000
3	DOE/SC/Oak Ridge National Laboratory United States	Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x	560,640
4	DOE/NNSA/LLNL United States	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom	1,572,864
5	Joint Center for Advanced High Performance Computing Japan	Oakforest-PACS - PRIMERGY CX1640 M1, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path	556,104

Xeon Phi Applications

- Earth sciences and weather
- Energy
- Financial services
- Life sciences
- Manufacturing
- Material sciences
- Physics
- Visualization

Xeon Phi Programming

Inscribe a circle in a square. Randomly generate points in the square. Determine the number of points in the square that are also in the circle. Let r be the number of points in the circle divided by the number of points in the square. PI ~ 4r. Note that the more points generated, the better the approximation



Xeon Phi Programming

```
#include <iostream>

unsigned long long int circleCount { 0 };
unsigned long long int numberOfPoints { 3000000000 };

int main() {
    std::cout << "Start calculating pi." << std::endl;
    srand(std::time(NULL));
    auto startTime = std::chrono::steady_clock::now();
    for (int i = 0; i < numberOfPoints; ++i) {
        auto r = (double)rand() / RAND_MAX;
        auto x = (2.0 * r) - 1;
        r = (double)rand() / RAND_MAX;
        auto y = (2.0 * r) - 1;
        if ( ((x * x) + ( y * y)) <= 1.0) {
            circleCount++;
        }
    }
    auto endTime = std::chrono::steady_clock::now();
    auto diff = std::chrono::duration_cast<std::chrono::seconds>(endTime-
startTime).count();
    std::cout.precision(20);
    std::cout << "Pi " << std::fixed << (4.0 * (double)circleCount/
(double)numberOfPoints)
              << " took " <<  diff << " seconds." << std::endl;
    return 0;
}
```

Xeon Phi Programming

Makefile

```
# Use Intel CPP compiler
CC = icpc

CXXFLAGS = -std=c++11

CXXFLAGS += -I"/usr/local/include/"

SimplePiCalculation: main.o
$(CC) $(CXXFLAGS) -o SimplePiCalculation main.o

main.o: main.cpp
$(CC) $(CXXFLAGS) -c main.cpp

clean:
rm -rf *.o SimplePiCalculation
```

Optimizations

```
CXXFLAGS += -I"/usr/local/include/" -O2 # or -O3
```

Xeon Phi Programming

```
#include <atomic>
#include <iostream>
#include <random>
#include <chrono>
#include "tbb/tbb.h"
#include "tbb/parallel_reduce.h"
#include "tbb/blocked_range.h"

size_t circleCount { 0 };
size_t numberOfPoints { 300000000ull }; // ull is unsigned long long

// For a visualisation of this estimation at work, see:
// https://academo.org/demos/estimating-pi-monte-carlo/
bool calculatePi(double random_number, double random_number2) {
    auto x = (2.0 * random_number) - 1;
    auto y = (2.0 * random_number2) - 1;
    return ( ((x * x) + ( y * y)) <= 1.0);
}
```

```
int main() {
    std::cout << "Start calculating pi." << std::endl;
    auto startTime = std::chrono::steady_clock::now();
    circleCount = tbb::parallel_reduce(
            // reduce on a range of 0.. numberOfPoints with stepsize 1
            tbb::blocked_range<size_t>(0ull, numberOfPoints, 1ull),
            // initial value is zero
            0ull,
            // lambda that does calculation for part of the range
            [](const tbb::blocked_range<size_t>& r, size_t value) -> size_t {
                // rand() uses hidden state and is not thread safe
            // we need to have a random number generator per thread
                std::uniform_real_distribution<double> unif(0.0, 1.0);
                std::default_random_engine re;

                for (size_t i=r.begin(); i<r.end(); i++) {
                    double random_number = unif(re);
                    double random_number2 = unif(re);
                    // do the estimation test and increment local per thread counter
                    if (calculatePi(random_number, random_number2))
                        value++;
                }
                return value;
            },
            // accumulate all the local (per thread) counters with std::plus to get
         // the absolute total
            std::plus<size_t>()
    );
    auto endTime = std::chrono::steady_clock::now();
    auto diff = std::chrono::duration_cast<std::chrono::seconds>(endTime-startTime).count();
    std::cout.precision(20);
    std::cout << "Pi " << std::fixed << (4.0 * (double)circleCount/(double)numberOfPoints)
              << " took " <<  diff << " seconds." << std::endl;
    return 0;
}
```

Xeon Phi Programming

Makefile

```
# Use Intel CPP compiler
CC = icpc

CXXFLAGS = -std=c++11

CXXFLAGS += -O2 -ltbb -I"/usr/local/include/"

LDFLAGS = -L/opt/intel/lib -L/opt/intel/lib/intel64 -L/opt/intel/tbb/lib
-L/opt/intel/mkl/lib

SimplePiCalculationParallel: main.o
$(CC) $(CXXFLAGS) -o SimplePiCalculationParallel main.o $(LDFLAGS)

main.o: main.cpp
$(CC) $(CXXFLAGS) -c main.cpp
clean:
rm -rf *.o SimplePiCalculationParallel
```

Xeon Phi Programming

- Intel parallel studio
- Make
- Provided example makefile*
- Example BS_Hackaton.cpp (Black-Scholes formula)*

*https://github.com/TheDutchCppGrp/Hackaton-20-May-2017

Challenges - Starters

The example contains a basic C++ program to calculate a theoretical "european" option price based on the Black-Scholes formula with a set of input parameters.

Adjust this program to calculate theoretical prices for a large set of strikes, several expiries and several underlyings in parallel.

Usually one calculates option theoreticals for the whole set of strikes available for a particular underlying and expiry (e.g. 10, 11, 12, 13, 14, 15, 18, 20, etc). This is where the vectorization comes in handy to do that in fewer CPU instructions for several strikes at once. Parallelization comes in handy when calculating theoretical prices for many underlings at the same time, e.g. DAX, AEX, BMW, etc.

Challenges - Experienced

Use the Black-Scholes formula (see the example C++ program for the formula) to calculate the “implied” volatility in every available strike and expiry. First you need to optimize the BS formula and secondly you use the optimized version to revert the formula and, based on the current option market prices, calculate the implied volatility.

Challenges – Your own

?

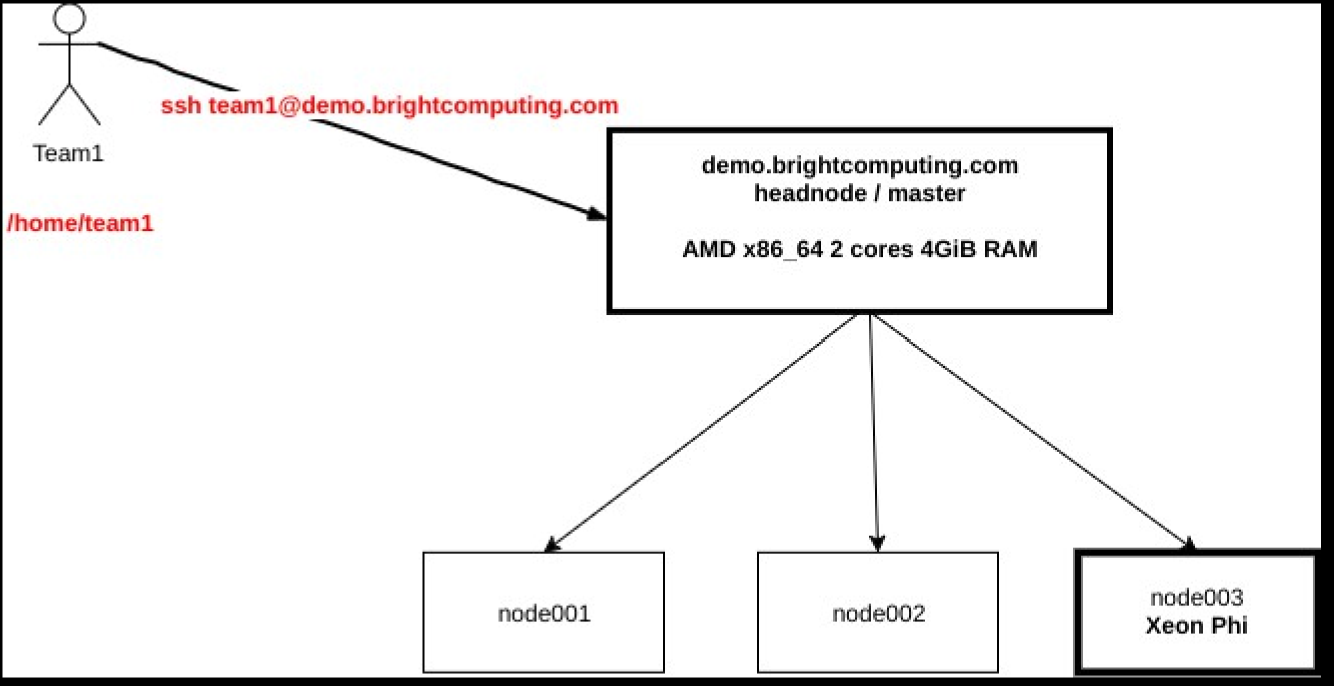
Challenges – Resources

https://software.intel.com/en-us/tools-by-segment/technical-enterprise

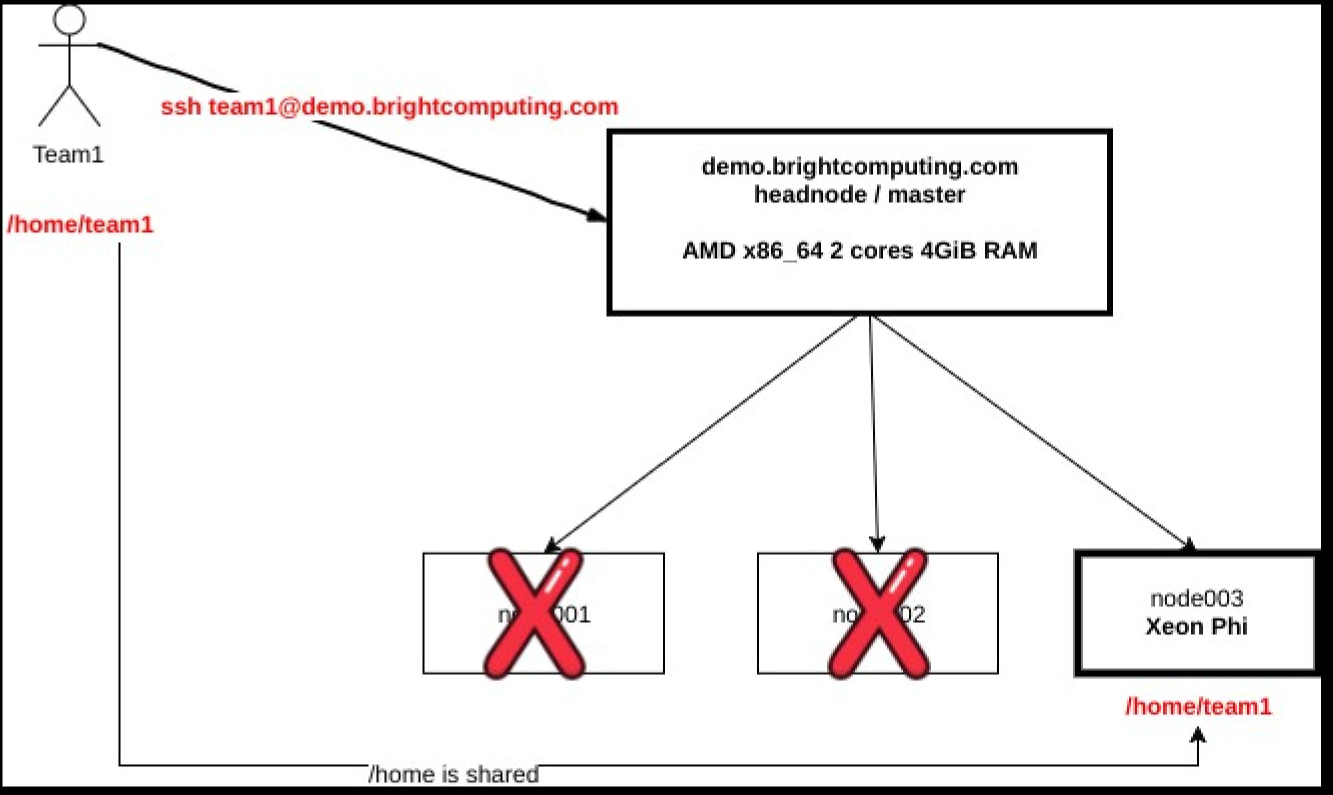
https://software.intel.com/en-us/xeon-phi/mic/programming

https://software.intel.com/en-us/xeon-phi/mic/training

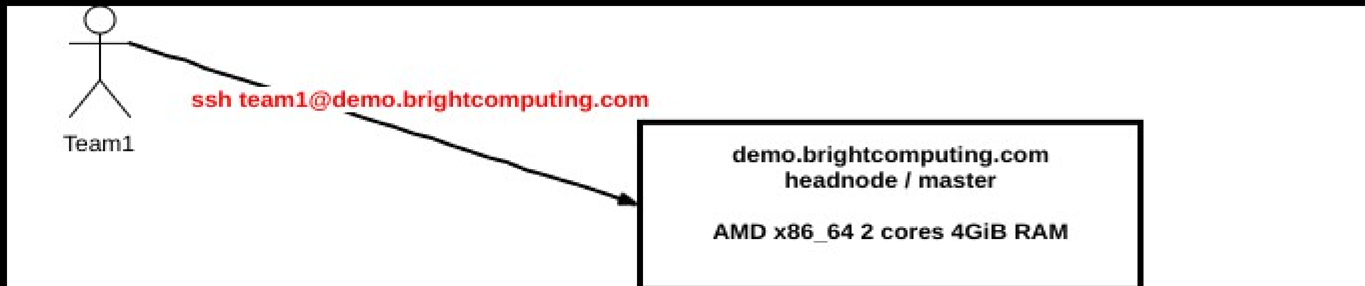
Server with the Xeon Phi



Server with the Xeon Phi



Environment files




```
[team1@demo ~]$ icpc
-bash: icpc: command not found
[team1@demo ~]$ module avail intel

------------------------------------ /cm/shared/modulefiles ------------------
intel/compiler/32/2017/17.0.1          intel/mpi/32/2017/1.132
intel/compiler/64/2017/17.0.1          intel/mpi/64/2017/1.132
intel/daal/32/2017/1.132               intel/mpi/mic/2017/1.132
intel/daal/64/2017/1.132               intel/studio  <<< ignore this one
intel/ipp/32/2017/1.132                intel/tbb/32/2017/1.132
intel/ipp/64/2017/1.132                intel/tbb/64/2017/1.132
intel/mkl/32/2017/1.132                intel/tbb/mic/2017/1.132
intel/mkl/64/2017/1.132                intel-tbb-oss/ia32/2017_20160722oss
intel/mkl/mic/2017/1.132               intel-tbb-oss/intel64/2017_20160722oss
[team1@demo ~]$ module load intel/compiler/64/2017/17.0.1
[team1@demo ~]$ icpc
icpc: command line error: no files specified; for help type "icpc -help"
[team1@demo ~]$
```

Environment files

```
[team1@demo parallel-version]$ make
icpc -std=c++11  -O2 -ltbb -I"/usr/local/include/" -c main.cpp
main.cpp(5): catastrophic error: cannot open source file "tbb/tbb.h"
  #include "tbb/tbb.h"
                      ^

compilation aborted for main.cpp (code 4)
make: *** [main.o] Error 4
[team1@demo parallel-version]$ module load intel/tbb/64/2017/1.132
[team1@demo parallel-version]$ make
icpc -std=c++11  -O2 -ltbb -I"/usr/local/include/" -c main.cpp
icpc -std=c++11  -O2 -ltbb -I"/usr/local/include/" -o SimplePiCalculationParallel main.o -L/opt/int
el/lib -L/opt/intel/lib/intel64 -L/opt/intel/tbb/lib -L/opt/intel/mkl/lib
[team1@demo parallel-version]$
```

Running your program..

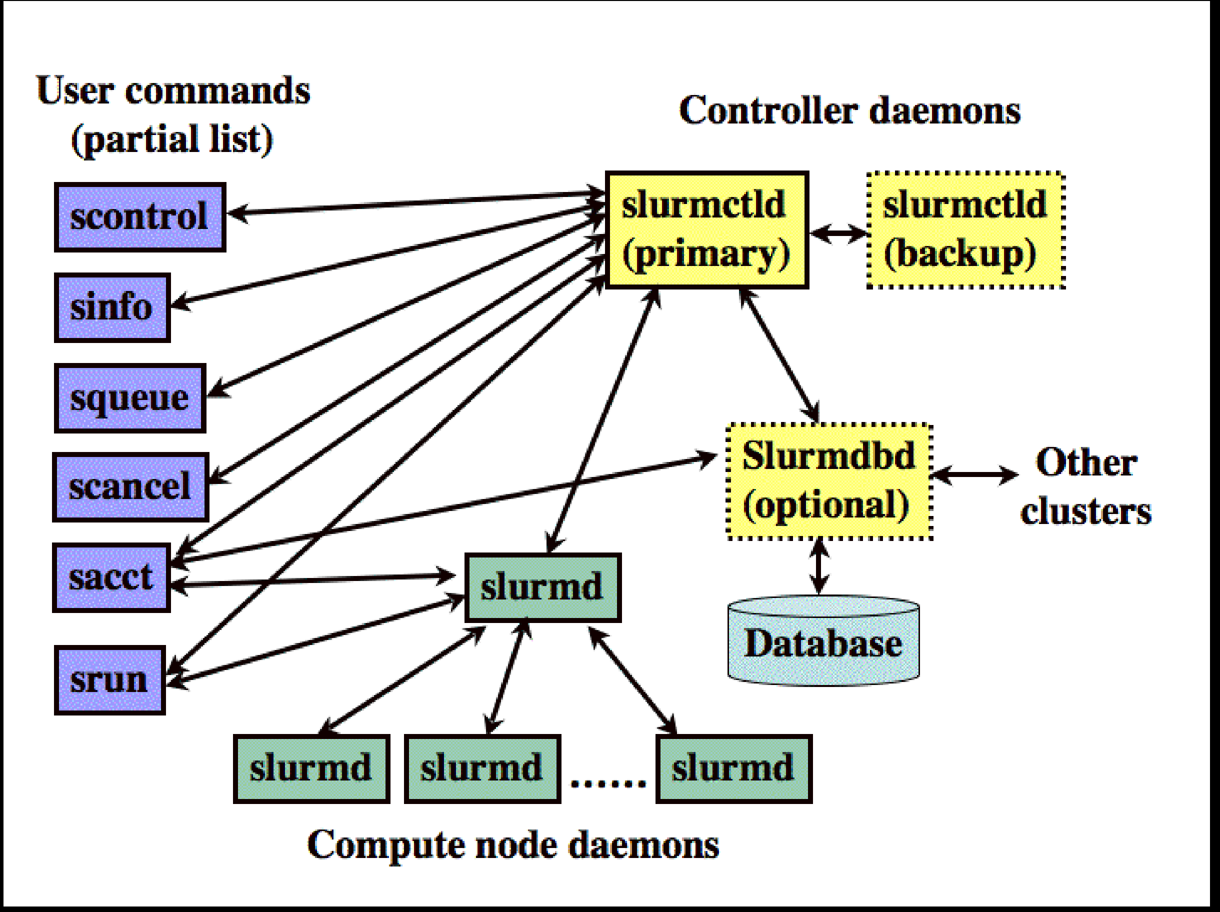
```
[team1@demo parallel-version]$ ./SimplePiCalculationParallel
Start calculating pi.
Pi 3.14113378666666687167 took 12 seconds.
[team1@demo parallel-version]$ srun -p xeonphi ./SimplePiCalculationParallel
Start calculating pi.
Pi 3.13402054666666662897 took 1 seconds.
[team1@demo parallel-version]$
```

srun is part of the Slurm Workload Manager..

-p xeonphi parameter specifies to use the queue “xeonphi”.

Which is a queue with only one node as a worker, node003.

More on slurm, see https://slurm.schedmd.com/quickstart.html



By example

```
[team1@demo parallel-version]$ srun hostname        # don't forget to specify the xeon phi queue :)
node001                                             # because Xeon Phi runs on node003 only
[team1@demo parallel-version]$ srun -p xeonphi hostname
node003
[team1@demo parallel-version]$ srun -p xeonphi nproc
272
[team1@demo parallel-version]$ srun -p xeonphi sleep 30 &
[1] 22789
[team1@demo parallel-version]$ srun -p xeonphi sleep 30 &
[2] 22809
[team1@demo parallel-version]$ srun: job 141 queued and waiting for resources
[team1@demo parallel-version]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
               141   xeonphi    sleep    team1 PD       0:00      1 (Resources)
               140   xeonphi    sleep    team1  R       0:04      1 node003
[team1@demo parallel-version]$
```

Chat

http://webchat.freenode.net/?channels=xeonphihackaton

- IRC
- Anonymous
- …

You can even join other channels like ##C++, ##C++-basic, #intel

Hackaton Saturday May 20 2017

The Dutch C++ Group

Sources

- Intel Corporation
- Wikipedia
- Inside HPC
- NVIDIA
- Top500.org
- Intel Xeon Phi Applications

Many thanks to

Optiver for hosting the Hackaton.

Bright Computing for providing a Xeon Phi.

Ray Burgemeestre for the installation and setup of the Xeon Phi.

Alexander Kristenko for creating the example Black Scholes program and makefile and ideas for the challenges.