High Performance Computing
Synopsis of Technical and Programming Concepts

Claude TADONKI
MINES ParisTech – PSL Research University
Paris - France
INTEL BROADWELL

- 22x2 = 44 cores
- 2.2 Ghz/core
- 3.6 GHz Boost
- Hyperthreading
- 256-bit vectors
- 256 Gb RAM
- 76.8 Gb/s
- 500 Gb disk
- 1.54 Tflops SP
- 0.78 Tflops DP

Tflops is 1000 000 000 000 (1 billion) floating point operations per seconds

**Hardware**

- **CPU Name:** Intel Xeon E5-2699 v4
- **CPU Characteristics:** Intel Turbo Boost Technology up to 3.60 GHz
- **CPU MHz:** 2200
- **FPU:** Integrated
- **CPU(s) enabled:** 44 cores, 2 chips, 22 cores/chip, 2 threads/core
- **CPU(s) orderable:** 1,2 chip
- **Primary Cache:** 32 KB I + 32 KB D on chip per core
- **Secondary Cache:** 256 KB I+D on chip per core
- **L3 Cache:** 55 MB I+D on chip per chip
- **Other Cache:** None
- **Memory:** 256 GB (16 x 16 GB 2Rx4 PC4-2400T)
- **Disk Subsystem:** 1 x SATA, 500 GB, 7200 RPM
- **Other Hardware:** None
NVIDIA DGX-1 — NVIDIA supercomputing solution
- 8 Tesla P100 GPUs (Pascal GPU based)
- Dual Intel Xeon processors (host)
- 170 Tflops FP16 peak perf
- 7 Tb of SSD Storage
- Aggregate bandwidth 768 Gb/s
- Perf throughput 250 x86 servers
- Pascal GPU: 3584 CUDA Cores; 1480 MHz; 16 GB RAM at 720 Gb/s
- We should understand that GPU is specialized for specific tasks where it is likely to show up noticeable performances

**NVIDIA DGX-1 Delivers 75X Faster Training**

- Relative Performance (Based on Time to Train)
- NVIDIA DGX-1 CPU: 150 Hours (2.25 Days) → 2 Hours

**NVIDIA DGX-1 Delivers 56X More Performance**

- Performance in teraFLOPS
- NVIDIA DGX-1 CPU: 3 TFLops → 170 TFLops

High Performance Computing
Claude TADONKI – UFRJ – RJ – April 27, 2016
TIANHE-2 (MILKYWAY-2)

- In China
- Intel Xeon E5
- 260,000 nodes
- 3 million cores
- 54 PFlops peak
- 33 PFlops (61%)
Performances Evolution

- We are moving toward ExaFlops \((E = \text{Exa} = 10^{18})\)
# TOP500

## Top 5 sites - Top500 - Nov 2015

<table>
<thead>
<tr>
<th>RANK</th>
<th>SITE</th>
<th>SYSTEM</th>
<th>CORES</th>
<th>RMAX (TFLOP/S)</th>
<th>RPEAK (TFLOP/S)</th>
<th>POWER (KW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>National Super Computer Center in Guangzhou China</td>
<td>Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT</td>
<td>3,120,000</td>
<td>33,862.7</td>
<td>54,902.4</td>
<td>17,808</td>
</tr>
<tr>
<td>2</td>
<td>DOE/SC/Oak Ridge National Laboratory United States</td>
<td>Titan - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.</td>
<td>560,640</td>
<td>17,590.0</td>
<td>27,112.5</td>
<td>8,209</td>
</tr>
<tr>
<td>3</td>
<td>DOE/NNSA/LLNL United States</td>
<td>Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM</td>
<td>1,572,864</td>
<td>17,173.2</td>
<td>20,132.7</td>
<td>7,890</td>
</tr>
<tr>
<td>4</td>
<td>RIKEN Advanced Institute for Computational Science (AICS) Japan</td>
<td>K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu</td>
<td>705,024</td>
<td>10,510.0</td>
<td>11,280.4</td>
<td>12,660</td>
</tr>
<tr>
<td>5</td>
<td>DOE/SC/Argonne National Laboratory United States</td>
<td>Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM</td>
<td>786,432</td>
<td>8,586.6</td>
<td>10,066.3</td>
<td>3,945</td>
</tr>
</tbody>
</table>

---

**High Performance Computing**  
Claude TADONKI – UFRJ – RJ – April 27, 2016
Peak Performance Evaluation

<table>
<thead>
<tr>
<th>RANK</th>
<th>SITE</th>
<th>SYSTEM</th>
<th>CORES</th>
<th>RMAX (TFLOP/S)</th>
<th>RPEAK (TFLOP/S)</th>
<th>POWER (KW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>National Super Computer Center in Guangzhou China</td>
<td>Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT</td>
<td>3,120,000</td>
<td>33,862.7</td>
<td>54,902.4</td>
<td>17,808</td>
</tr>
</tbody>
</table>

Getting Tianhe-2 RPEAK:

- CPU-core frequency: 2.2 Ghz = 2.2 GFlops
- Considering the vector capability (256-bit wide - 4 DP): $4 \times 2.2 = 8.8$ GFlops
- Given the CPU can do ADD and MUL in one cycle (FMA): $2 \times 8.8 = 17.6$ GFlops
- Finally the total number of cpu-cores: $3,120,000 \times 17.6$ Ghz = 54.912 PFlops

Clearly, we should exploit all levels of parallelism, if we need to harvest an acceptable fraction of the peak performance.
Peak vs Sustained

Not counted in peak performance:

- Memory accesses
- Interprocessor communications

We got here a sustained performance per core of 500 Mflops over 9 GFlops

How to Program a Supercomputer

- **Message passing** between nodes (MPI, …) [1]
- **Shared memory** between cores (Pthreads, OpenMP, …) [2]
- **Vector computing** inside a core (SSE, AVX, …) [3]
Message Passing Programming

- This is the typical way to execute across several independent compute nodes.
- The whole program is decomposed at runtime into several processes.
- Processes exchange data among themselves using message passing routines.
- The standard programming model is **SPMD** (Single Program Multiple Data).

```c
#include <stdio.h>  /* printf and BUFSIZE defined there */
#include <stdlib.h> /* exit defined there */
#include <mpi.h>   /* all MPI-2 functions defined there */

int main(argc, argv)
int argc;
char *argv[];
{
    int rank, size, length;
    char name[BUFSIZE];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(name, &length);

    printf("%s: hello world from process %d of %d\n", name, rank, size);
    MPI_Finalize();
    exit(0);
}
```
Message Passing Programming

- MPI code is compiled with `mpicc -o myprogram myprogram.c`
- Our MPI program is launched with the command `mpirun myprogram -np 8`
- The value passed through “-np” is the number of processes
- The number of processes can be higher or lower than the number of processors.
- The scalability of your MPI code will mainly depend on data exchanges overhead

```
$ mpiexec -n 8 hellow2
bc08: hello world from process 0 of 8
bc21: hello world from process 2 of 8
bc09: hello world from process 1 of 8
bc33: hello world from process 3 of 8
bc34: hello world from process 5 of 8
bc03: hello world from process 4 of 8
bc35: hello world from process 6 of 8
bc32: hello world from process 7 of 8
```

- Every MPI command starts with the prefix “MPI_”
- There are several implementations and versions of MPI, but portability is preserved
Message Passing Programming

- MPI commands can be roughly grouped into three categories
  - Environment Management Routines
  - Communication Routines (point-to-point – collective - synchronization)
  - Group Communicator Management Routines

- From here you just need to delve into MPI documentation for details & specific needs
- The global performance of your program will depend on both the parallel algorithm behind and the quality of the corresponding parallel program. **2 skills involved!!!**
Although we can still use the message passing approach for multicore machines, it is important to know that there is a specific paradigm for this context.
Multithreaded Programming

HISTORICAL CONTEXT AND TREND

We observe a stagnation of the processor frequency (tends to decreases)

We need to keep following the trend of Moore’s Law (transistors count)

In order to scale up with processor speed, we need more cores per chip

The number of cores per chip is increasing, but with complex memory system

Historically:
Boost single-stream performance via more complex chips.

Now:
Deliver more cores per chip (+ GPU, NIC, SoC).

The free lunch is over for today’s sequential apps and many concurrent apps. We need killer apps with lots of latent parallelism.
The cores always share the main memory and there are different cache levels.

Cache memories are distributed among the cores depending on the packaging.

A given core might be able to get data from non-local unshared caches.

Cache coherency is guarantee by the hardware and associated protocols.
Multithreaded Programming

Packaging and NUMA Consideration

Shared Memory Architecture – NUMA

Yinan Li et al.

Serious source of scalability issues

High Performance Computing
Claude TADONKI – UFRJ – RJ – April 27, 2016
Multithreaded Programming

 THREAD

- In a program, an independent section or a routine can be executed as a thread.
- A **multi-threaded** program is a program that contains several concurrent threads.
- A **thread** can be seen as a lightweight process (memory is shared among threads).
- A **thread** is a child of a (OS) process. Thus it uses the main resources of the process (shared between all running threads), while keeping its own
  - Stack pointer
  - Registers
  - Scheduling properties (policy, priority)
  - Set of pending and blocked signals
  - Thread specific data.
Multithreaded Programming

A threaded program is built from a classical program by embedding the execution of some of its subroutines within the framework of associated threads.

Typical scenario to design a threaded program implies:
- calls to a specialized library (thread implementation)
- programming directives for threads creation
- appropriate compiler directives

There are several (incompatible) implementations of threads depending on the target architecture (vendors) or the operating system. This impacts on programs portability.

Two standard implementations of threads are: **POSIX Threads** and **OpenMP**.
Multithreaded Programming

**OpenMP**

*Directives oriented compiler for multithreaded programming*

```plaintext
PROGRAM HELLO
!$OMP PARALLEL
PRINT *,"Hello World"
!$OMP END PARALLEL
STOP
END
```

```plaintext
#include <iostream>
#include "omp.h"
int main() {
    #pragma omp parallel
    {
        std::cout << "Hello World\n"
    }
    return 0;
}
```

**OMP COMPILER DIRECTIVES**

**Compiler Commands**

- **intel:** ifort -openmp -o hi.x hello.f
- **pgi:** pgfortran -mp -o hi.x hello.f
- **gnu:** gfortran -fopenmp -o hi.x hello.f

**Export**

OMP_NUM_THREADS=4

```
./hi.x
```

**OMP ENVIRONMENTAL VARIABLE**

**NOTE:** example hello.f90
Multithreaded Programming

Pthread

Pthread library contains hundred of routines that can be grouped into 4 categories:

- **Thread management**: Routines to create, terminate, and manage the threads.
- **Mutexes**: Routines for synchronization (through a "mutex" ≈ mutual exclusion).
- **Condition variables**: Routines for communications between threads that share a mutex.
- **Synchronization**: Routines for the management of read/write locks and barriers.

All identifiers of the Pthreads routines and data types are prefixed with « pthread_ »

Example: pthread_create, thread_join, pthread_t, …

For portability, the pthread.h header file should be included in each source file

The generic compile command is

« cc -lpthread » or « cc -pthread »,
cc = compiler
Multithreaded Programming

```c
#include <pthread.h>
#include <stdio.h>
define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread %ld\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)&t);
        if (rc){
            printf("ERROR: return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }

    /* Last thing that main() should do */
    pthread_exit(NULL);
}
```

In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
Hello World! It's me, thread #1!
Hello World! It's me, thread #2!
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
A SIMD machine simultaneously operates on tuples of atomic data (one instruction).

SIMD is opposed to SCALAR (the traditional mechanism).

SIMD is about exploiting parallelism in the data stream (DLP), while superscalar SISD is about exploiting parallelism in the instruction stream (ILP).

SIMD is usually referred as VECTOR COMPUTING, since its basic unit is the vector.

Vectors are represented in what is called packed data format stored into vector registers.

On a given machine, the length/number of the vector registers are fixed

SIMD can be implemented on using specific extensions MMX, SSE, AVX, …
Vector Programming

SIMD Implementation

SIMD: Continuous Evolution

<table>
<thead>
<tr>
<th>Year</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999</td>
<td>SSE</td>
</tr>
<tr>
<td>2000</td>
<td>SSE2</td>
</tr>
<tr>
<td>2004</td>
<td>SSE3</td>
</tr>
<tr>
<td>2006</td>
<td>SSSE3</td>
</tr>
<tr>
<td>2007</td>
<td>SSE4.1</td>
</tr>
<tr>
<td>2008</td>
<td>SSE4.2</td>
</tr>
<tr>
<td>2009</td>
<td>AES-NI</td>
</tr>
<tr>
<td>2010/11</td>
<td>AVX</td>
</tr>
</tbody>
</table>

- 70 instr
  - Single-Precision Vectors
  - Streaming operations

- 144 instr
  - Double-precision Vectors
  - 8/16/32
  - 64/128-bit vector integer

- 13 instr
  - Complex Data

- 47 instr
  - Video
  - Graphics building blocks
  - Advanced vector instr
  - String/XML processing
  - POP-Count CRC

- 8 instr
  - Encryption and Decryption
  - Key Generation

- 7 instr

- ~100 new instr.
  - ~300 legacy
  - SSE instr updated
  - 256-bit vector
  - 3 and 4-operand instructions

Then AVX2, MIC, ...

Vector instructions can be used from their **native form** or through **intrinsics**

High Performance Computing
Claude TADONKI – UFRJ – RJ – April 27, 2016
Vector Programming

**MMX™**
- Vector size: 64bit
- Data types: 8, 16 and 32 bit integers
- VL: 2, 4, 8
- For sample on the left: $X_i$, $Y_i$ 16 bit integers

**Intel® SSE**
- Vector size: 128bit
- Data types:
  - 8, 16, 32, 64 bit integers
  - 32 and 64 bit floats
- VL: 2, 4, 8, 16
- Sample: $X_i$, $Y_i$ bit 32 int / float

**Intel® AVX**
- Vector size: 256bit
- Data types: 32 and 64 bit floats
- VL: 4, 8, 16
- Sample: $X_i$, $Y_i$ 32 bit int or float

**Intel® MIC**
- Vector size: 512bit
- Data types:
  - 32 and 64 bit integers
  - 32 and 64 bit floats
  - (some support for 16 bit floats)
- VL: 8, 16
- Sample: 32 bit float

**MMX = MultiMedia eXtension**
**SSE = Streaming SIMD Extension**
**AVX = Advanced Vector Extensions**
**MIC = Many Integrated Core**
Vector Programming

- **SSE = Streaming SIMD Extensions**

- SEE programming can be done either through *(inline) assembly* or from a high-level language (C and C++) using *intrinsics*.

- The `{x,e,p}`mmmintrin.h header file contains the declarations for the SSEx instructions intrinsics.
  - xmmmintrin.h -> SSE
  - emmintrin.h -> SSE2
  - pmmintrin.h -> SSE3

- SSE instruction sets can be enabled or disabled. If disabled, SSE instructions will not be possible. It is recommended to leave this BIOS feature enabled by default. In any case MMX (MultiMedia eXtensions) will still available.

- Compile your SSE code with "gcc -o vector vector.c -msse -msse2 -msse3"

- SSE intrinsics use types `__m128 (float)` , `__m128i (int, short, char)`, and `__m128d (double)`

- Variable of type `__m128, __m128i, and __m128d (exclusive use)` maps to the XMM[0-7] registers (128 bits), and automatically aligned on 16-byte boundaries.

- Vector registers are `xmm0, xmm1, ..., xmm7`. Initially, they could only be used for *single precision* computation. Since SSE2, they can be used for *any primitive data type*. 

---

High Performance Computing
Claude TADONKI – UFRJ – RJ – April 27, 2016
Vector Programming

SSE (Connecting vectors to scalar data)

- Vector variables can be connected to scalar variables (arrays) using one of the following ways

```c
float a[N] __attribute__((aligned(16)));
__m128 *ptr = (__m128*)a;

prt[i] or *(ptr+i) represents the vector
{a[4i], a[4i+1], a[4i+2], a[4i+3]}
```

```c
float a[N] __attribute__((aligned(16)));
__m128 mm_a;
mm_a = _mm_load_pd(&a[4i]); // here we explicitly load data into the vector

mm_a represents the vector
{a[4i], a[4i+1], a[4i+2], a[4i+3]}
```

- Using the above connections, we can now use SSE instruction to process our data. This can be done through
  - (inline) assembly
  - intrinsics (interface to keep using high-level instructions to perform vector operations)
Vector Programming

SSE (illustrations)

```c
void scalar_sqrt(float *a){
    int i;
    for(i = 0; i < N; i++)
        a[i] = sqrt(a[i]);
}
```

**Scalar version**

```c
void sse_sqrt(float *a){
    // We assume N % 4 == 0.
    int nb_iters = N / 4;
    __m128 *ptr = (__m128*)a;
    int i;
    for(i = 0; i < nb_iters; i++, ptr++, a += 4)
        _mm_store_ps(a, _mm_sqrt_ps(*ptr));
}
```

**Vector version (SSE)**

```
Tadonki@TADONKI-PC ~/vector
$ ./test
Running time of the scalar code: 0.286017
Running time of the SSE code: 0.031001
```

10 times faster !!!!!!!
Conclusion

HPC is making noticeable progresses, but we still need to skillfully use its elements and concepts in order to reach our performance expectations.

There is no free launch
Thanks for your attention