www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Hands-on

PATC2014

Albert Segura, Alejandro Rico, Daniel Ruiz,
Filippo Mantovani, Oriol Vilarrubi
Barcelona Supercomputing Center

Outline

- Simple job submissions
  - Intel IMB benchmarks
- Tuning
  - Synthetic FP micro-benchmarks
- ARM + GPU
  - CUDA
- Porting (SW stack exploration)
  - High-Performance Linpack

Setup

- All exercises can be found in:

/PATC2014/exercises

- User account
- Good will and patience
  - Pedraforca cluster is an experimental cluster which never served 20+ users at once.

Outline

- Simple job submissions
  - Intel IMB benchmarks
- Tuning
  - Synthetic FP micro-benchmarks
- Porting (SW stack exploration)
  - High-Performance Linpack
- ARM + GPU
  - CUDA

Exercise 1 - Manage your job

- Access to Pedraforca using the information on the page that we have provided to you

- Copy everything from **/PATC2014/exercises/imb** to somewhere in your $HOME directory

- **Manage your job "myjob.job"**
  - Modify the content properly, submit it, check the queue, cancel it, check the output when it is COMPLETED and the properties of the job

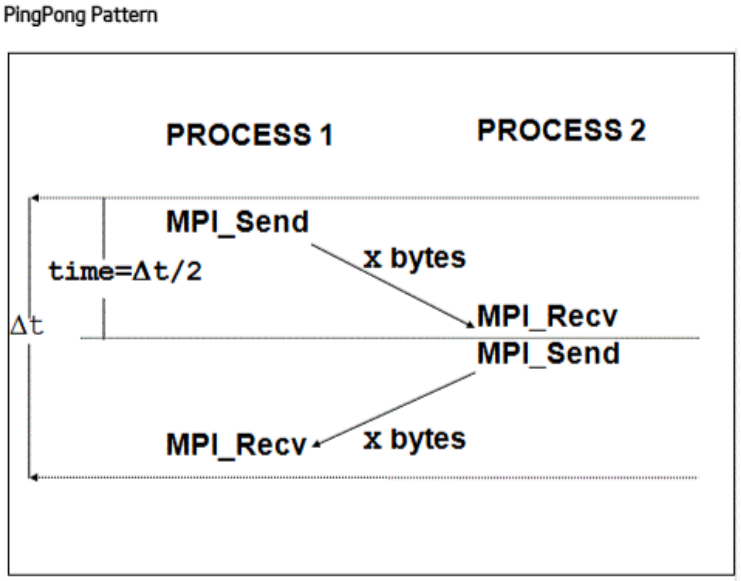
Exercise 1 - Manage your job

- Cheat sheet:
  - ntasks
  - cpus-per-task
  - ntasks-per-node
  - nodes
- Commands:
  - sbatch
  - squeue
  - scancel
  - sinfo
  - scontrol

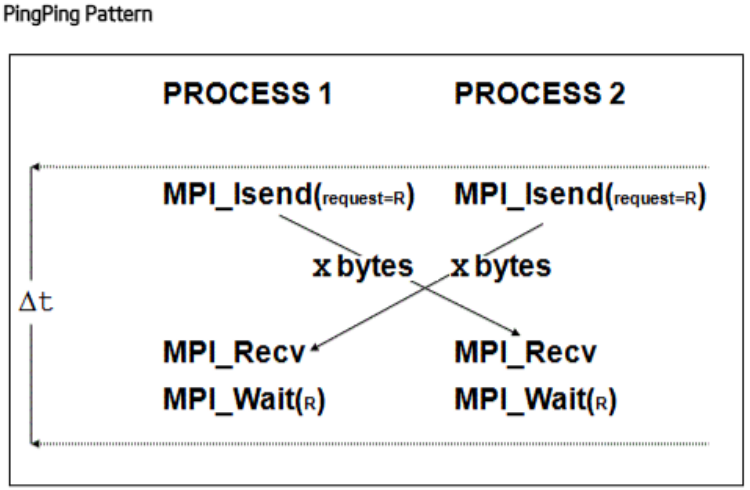
Intel MPI Benchmark

- It is a suite of benchmarks to assess performance of the cluster network, MPI libraries on communication
- Single/Parallel transfer and Collective types

- PingPong



- PingPing



Exercise 2 - Single Transfer benchmarks

- Run PingPong and PingPing within a node
  - Use --ntasks, --cpus-per-task and --ntasks-per-node properly.
- Run PingPong and PingPing between 2 cpus belonging to different nodes
- Each time run both at once and also try to use dependencies
  - Check --hold option

Exercise 2 - Single Transfer benchmarks

- Cheat sheet dependencies:
  - after:job_id[:job_id...]
    - This job can begin execution after the specified jobs have begun execution.
  - afterany:job_id[:job_id...]
    - This job can begin execution after the specified jobs have terminated.
  - afterok:job_id[:job_id...]
    - This job can begin execution after the specified jobs have successfully executed.
  - afternotok:job_id[:job_id...]
    - This job can begin execution after the specified jobs have terminated in some failed state.

Exercise 3 - Parallel Transfer benchmark

- Run Parallel Transfer benchmark Sendrecv:
  - Use different combinations of --ntasks, --cpus-per-task and --ntasks-per-node. Check the output.

Exercise 4. Collective benchmark

- Run Collective benchmarks Allgather and Alltoall:
  - Use different combinations of --ntasks, --cpus-per-task and --ntasks-per-node. Check the output.

Outline

- Simple job submissions
  - Intel IMB benchmarks
- Tuning
  - Synthetic FP micro-benchmarks
- ARM + GPU
- Porting (SW stack exploration)
  - High-Performance LINPACK

Synthetic FP-µbenchmarks

- To test the FP performance of Cortex-A9 CPU in Tegra3
- Developed to find out the peak performance
  - Tegra3 operates at max 1.3 GHz
- We will use it to test the importance of gcc architectural flags, and some optimizations

/PATC2014/exercises/ubenchs

FP addition - reduction

```
/**fpadd.c**/

double *A;
double accum;
… … …
gettimeofday(&start, 0);

for (j=0; j<t; j++) {
    acum  = 0;

    for (i=n; i!=0; i--) {
        acum  += A[i];
    }
}

gettimeofday(&end, 0);
```

- Sums all elements of an array
  - Double-precision FP
  - Repeats for a given number of times
  - VADD Expected peak performance?

FP multiply-add

```
/**fpma.c**/

double *A,*B;
double accum;
… … …
gettimeofday(&start, 0);

for (j=0; j<t; j++) {
    acum  = 0;

    for (i=0; i<n; i++) {
        acum  += A[i] * B[i];
    }
}

gettimeofday(&end, 0);
```

- Vector dot product
  - Double-precision FP
  - Repeats for a given number of times
- Same performance as before?
  - Floating multiply and add (VMLA) has the same operation throughput as the VADD

Exercise 1: Benchmark execution

- Execute synthetic benchmarks

```
make
sbatch job.slurm
```

- Observe the reported MFLOPS

- If you think it's not enough performance, what can we do to improve it?

Exercise 2: Compiler flags

Forgot about GCC flags?

- `-mcpu=cortex-a9 -mtune=cortex-a9`
  - Specifies the target CPU
    - gcc chooses the correct instructions to emit
    - Activates CPU-specific optimizations

- `-mfpu=vfpv3-d16`
  - Specifies floating point hardware that is available in the CPU

- `-funroll-loops`
  - Compiler directed loop-unrolling

Exercise 3: loop unrolling

```
/**fpadd_optimized.c**/

double *A;
double accum, ...;

gettimeofday(&start, 0);
for (j=0; j<t; j++) {
        acum  = 0; acum2 = 0;
        acum3 = 0; acum4=0;
        acum5 =0; acum6=0;
        acum7 =0;

        for (i=n; i!=0; i--) {
            acum  += A[i];
            acum2 += A[i];
            acum3 += A[i];
            acum4 += A[i];
            acum5 += A[i];
            acum6 += A[i];
            acum7 += A[i];
        }
}
gettimeofday(&end, 0);
```

- Sums all elements of an array
  - Double-precision FP
  - Repeats for a given number of times
- 90-100% peak performance

Outline

- Simple job submissions
  - Intel IMB benchmarks
- Tuning
  - Synthetic FP micro-benchmarks
- ARM + GPU
- Porting (SW stack exploration)
  - High-Performance LINPACK

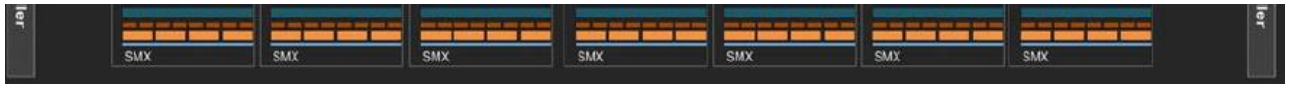
ARM+CUDA

- Each Pedraforca node includes a NVIDIA K20 GPU
  - Kepler architecture
  - Peak SP perf: **3520** GFLOPS, DP perf: **1170** GFLOPS

- Let's see how to use it and how it performs

- Copy files from

/PATC2014/exercises/arm+cuda/



CUDA Matrix multiplication

- Matrix multiplication is a GPU-friendly operation
- Four versions:
  - **`matmul-blocked.c`**: matmul using blocking on CPU
  - **`matmul-blas.c`**: matmul using BLAS on CPU
  - **`matmul-cuda.cu`**: matmul using tiling and shared memory on GPU
  - **`matmul-cublas.cu`**: matmul using CUBLAS on GPU

- Input matrices:
  - **`inputMatrix0.txt`**: 32 x 32
  - **`inputMatrix1.txt`**: 128 x 128
  - **`inputMatrix2.txt`**: 512 x 512
  - **`inputMatrix3.txt`**: 2048 x 2048
  - **`inputMatrix4.txt`**: 8192 x 8192
  - **`inputMatrix5.txt`**: A: 10000 x 6000, B: 6000 x 8000

Exercise 1: Compile CPU and GPU codes

- Floating point precision compilation option:
  -DDOUBLE_PRECISION=X
  - X=1 for DP, X=0 for SP

- Compiler: **gcc** for CPU, **nvcc** for GPU
- Tip: append suffix .s or .d for different FP precision binaries

- matmul-blocked.c: No specific flags
- matmul-blas.c:
  -I/usr/local/atlas/include -L/usr/local/atlas/lib -lsatlas
- matmul-cuda.cu: No specific flags
- matmul-cublas.cu:
  -lcublas

Exercise 2: Run and compare CPU and GPU codes

- Run different binaries with different input matrices
- It shows the achieved performance in GFLOPS
- For GPU shows computation only and computation+transfer performance

- On CPU use up to 2048x2048 matrices

- How different is performance for different FP precisions?

- When does it pay off to offload the computation to the GPU?
- Is the answer different for SP and DP FP?

Outline

- Simple job submissions
  - Intel IMB benchmarks
- Tuning
  - Synthetic FP micro-benchmarks
- ARM + GPU
- Porting (SW stack exploration)
  - High-Performance LINPACK

High Performance LINPACK

- Official Top500 list benchmark
  - Developed by Jack Dongarra et al. in University of Tennessy
  - Rank HPC Machines by the rate of solving the dense systems of linear equations in double precision arithmetic
  - Lets see how good is Pedraforca

/PATC2014/exercises/hpl/

Exercise 1: Port HPL to Pedraforca

- Untar hpl-2.1.tar.gz and check README.txt
- Follow instructions and compile it

- ATLAS as BLAS backend: /usr/local/atlas
- MPICH2 as MPI backedn: /usr/lib/mpich2

Exercise 2: Single node performance

- Execute HPL with the following parameters

- Set the block size Nb=160
- Set the problem size as N=X*Nb so that it fits in ~1.5 GB of memory, where $X \in \boldsymbol{N}$
  - **Limit to 1.5GB in order to have decent runtime**
- Set the process grid map to P=2 Q=2
- Execute on **one** node with **4** processes per node
- Keep the output files for the later comparison

Exercise 3: Effects of problem partitioning

- Execute HPL with the following parameters:
  - **N** same as in previous step
  - **Nb**=1600
  - **P, Q** same as in previous step
  - Execute on **one** node with **4** processes per node
  - Keep the output files for the comparison

- What can you observe compared to the previous case with N=160 ?

Exercise 4: Shared memory vs Message Passing partitioning

- Execute LINPACK in three different configurations:
  - **N** same as in previous step
  - **Nb = 160**
  - **P,Q** same as in previous step, (2,2)
  - Modify job script in order to provide following process mappings:
    - 4 processes, 4 processes per node
    - 4 processes, 2 processes per node
    - 4 processes, 1 process per node
  - Keep the output files for comparison

- What can you observe between three runs? Which case provides best performance? How you explain this?