Performance evaluation of a cluster tile (a single node)

Alberto José Proença & João Garcia Barbosa

---

1. Context

Consider the SeARCH cluster, with several heterogeneous computing nodes, each node containing several independent number-crunching elements (aka processing-elements or computing cores), all accessing a single-address-space shared memory.

This training exercise for the larger UCE project will focus on performance evaluation issues related to a single node.

At least 8 generations of Intel Xeon chips are populating the current 77 nodes in this cluster; each node with at least 2 CPU sockets and 1GB RAM per core. This training exercise will only consider the following nodes: (i) with 2x quad-core Xeon-Nehalem devices (**i5520**) and (ii) with 2x dodeca-core Opteron (**AMD 6174**).

Three generations of accelerating FP devices (gpGPU) are also in the cluster. However, CUDA environment only supports two and this training exercise will only consider the newer Fermi devices, which are in nodes with dodeca-core Opteron (**Tesla C2050**)

Several metrics to present the results are available (e.g., CPI, CPE, MFLOP), but only the last one will be used to present and discuss the results. Students should comment this.

The problem to be used to evaluate the individual performance of the nodes is the matrix multiplication function known as xGEMM, within a range of pre-defined sizes. This training exercise further focus this operation by limiting it to only one GEMM function (DP operations): `C=AxB`, where `A`, `B` and `C` are squared matrices.

Several algorithms can be explored to improve performance, but this is outside the scope of topics covered during the 1st semester of this UCE. So, this training exercise supplies a less naive multi-threaded program (code listing below; file `ompMM_main.c` available through CPD/PI website), and each student will develop one variation of this algorithm that explores the block multiplication approach.

2. Action

To limit the extremely wide range of available options to select the experimental settings to evaluate the performance of such multiplication function in a cluster node, the overall training exercise will be divided into 3 scenarios with some constraints:

- target: to produce an individual report (max. 15-page) to deliver on Jan-24, 10h00, and present a 30 minute oral communication on Jan-25;
- aims: to measure GEMM performance on 3 scenarios, and to analyse, specify and fully characterize the scenarios with a critical evaluation of the measured results;
- constraints:
    - for each scenario, consider only 3 multi-threaded program codes, related to, (i) the number of available EM64T cores in the node, (ii) half that value, and (iii) double that value;
    - the scalar coefficients in the GEMM operation are 1.0;
    - in the 3 scenarios, the 2nd must use the BLAS library (either for the EM64T or for the GPU), the 3rd must use the GPU (without any math library).

3. Individual Tasks

a) For each hardware platform evaluate the **Peak Theoretical Performance** in MFLOPS

b) Options for **Scenario A** (using both i5520 and AMD 6174, no BLAS, no Fermi)

   i. Compiler: `gcc`
   Vector Extensions: `SSE3`
   Core allocation: all cores, no distinction, # threads related to # cores

   ii. Compiler: `gcc`
   Vector Extensions: `SSE3`
   Core allocation: explore core affinity for 4 & 8 threads

   iii. Compiler: `icc`
   Vector Extensions: `SSE3`
   Core allocation: all cores, no distinction, # threads related to # cores

   iv. Compiler: `icc`
   Vector Extensions: `SSE4.1`
   Core allocation: explore core affinity for 8 & 16 threads

c) Options for **Scenario B** (with the library BLAS)

   i. CPU: i5520
   Compiler/switches: `gcc` or `icc`
   Library: BLAS at Intel MKL

   ii. CPU: i5520
   Compiler/switches: `gcc` or `icc`
   Library: BLAS in GoToBLAS

   iii. CPU: AMD 6174 + Fermi
   Compiler/switches: `gcc` or `icc`
   Library: BLAS GoToBLAS or IMKL + CUBLAS

d) Options for **Scenario C** (with Fermi, no libraries)

   i. CPU/GPU: multiplication only in Fermi
   Compiler: `gcc` or `icc`
   Techniques: explore loop unroll

   ii. CPU/GPU: multiplication in both CPU & Fermi
   Compiler: `gcc` or `icc`
   Techniques: explore loop unroll

e) **Graphics** for all scenarios:

   i. `x` axis: # rows at each square matrix;
   `y` axis: performance in MFLOPS/GFLOPS

   ii. Contents: at least 6 functions, relating # threads (# cores), w/-w/o BLAS, w/-w/o GPU

f) Task for **extra points**: assess and eventually install and use **PerfExpert**, a new performance bottleneck diagnosis tool for HPC application writers, which automatically evaluates the core, chip, and node-level performance.

g) Specific task options to be allocated to each student: by email.

Code listing of `ompMM_main.c` :

```
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define N 2048

void initMatrix(float* A) {

#pragma omp parallel for
        for (int ii=0; ii<N; ii++)
           for (int jj=0; jj<N; jj++)
               A[ii*N+jj] = (float)rand()/(float)RAND_MAX;

}

void clearMatrix(float* A) {

#pragma omp parallel for
        for (int ii=0; ii<N; ii++)
           for (int jj=0; jj<N; jj++)
               A[ii*N+jj] = 0.0;

}

void printMatrix(float* A) {
        for (int ii=0; ii<N; ii++)
           for (int jj=0; jj<N; jj++)
               printf("%f\t",A[ii*N+jj]);
}

void thrMM(float* A, float* B, float* C) {

        // Note: OpenMP allows
        //    more robust workload distribution among threads
        //    assign threads to specific cores (affinity)

        #pragma omp parallel for
        for (int ii=0; ii<N; ii++) {
           for (int kk=0; kk<N; kk++) {
               float r =  A[ii*N+kk];
               for (int jj=0; jj<N; jj++) {
                      C[ii*N+jj] += r * B[kk*N+ii];
               }
           }
        }
}

int main (int argc, const char * argv[]) {

        float *A = (float*)malloc(sizeof(float)*N*N);
        float *B = (float*)malloc(sizeof(float)*N*N);
        float *C = (float*)malloc(sizeof(float)*N*N);

        // The #threads should be dynamically set and commented
        omp_set_num_threads(2);

        initMatrix(A);
        initMatrix(B);
        initMatrix(C);

        //Be carefull with time measuring (avg?????)
        double start = omp_get_wtime();
        for (int i=0; i<5; i++) {
           thrMM(A, B, C);
        }
        double end = omp_get_wtime();

        //Use a more adequate metric to measure performance
        printf("Avg. time: %f\n",(end-start)/5);
}
```