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 7 generations of Intel Xeon chips are populating the over 70 nodes in this cluster; each node with 2 Xeon sockets and 1GB RAM per core. This training exercise will only consider nodes with 2 Xeon generations, those with quad-core devices.

Two generations of accelerating FP devices (gpGPU) are also in the cluster. However, only the latest one is supported by the CUDA environment. This training exercise will only consider these latter devices, which are in nodes with quad-core Xeons.

Several metrics to present the results are available (e.g., CPI, CPE, MFLOP), but only 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 and to 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 to be used as **case A** (code listing below; file `ompMM_main.c` available through CPD/PI website), and each student will develop and present for discussion only one variation of this algorithm, **case B**, which will explore 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 executed in 2 separate stage/phases and some constraints are further imposed.

Phase 1

- **target**: to produce an 8-page report to deliver on Jan-11, 12h00;
- **aims**: to analyse, specify and fully characterize a set of 8 possible experimental scenarios to measure the node performance (paper work, no experimental results required).

Phase 2

- **target**: to produce a 15-page report to deliver on Feb-01, 12h00, and present a 30 minute oral communication on Feb-02;
- **aims**: to measure GEMM performance on 3 out of the previously reported 8 scenarios (these 3 will be selected by the instructors team), and to complement the previous work with a critical evaluation of the measured results.

Further 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 final 3 scenarios, one must use the BLAS library (either for the EM64T or for the GPU), another must use the GPU (without any math library).

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);

}
```