Faculdade de Engenharia da Universidade do Porto

Parallel and Distributed Computing
Project 1

Computação Paralela e Distribuída– 23/24– *Licenciatura em Engenharia Informática e Computação*
Turma 01 – Equipa 16

Students:
Florentia Diamantopoulou up202311252@up.pt
Maria Mitkou up202311354@up.pt

Problem Description

In this problem we studied the effect on the processor performance of the memory hierarchy when accessing large amounts of data for the first part and for the second we evaluated the performance of a multi-core implementation. For the analysis we used multiplication of two matrices which is a common operation in many engineering applications.

Algorithm Explanation

In this project, we implemented three different multiplication algorithms and measured their time and cache … in order to observe the performance of a single core. The difference between the three algorithms is how they multiply the elements of the matrices. We implemented them in C++ and Java. In more detail the three algorithms are:

1. Matrix Multiplication

This algorithm was already given in a C++ language and we translated it to Java. It is the simple Matrix Multiplication which multiplies one line of the first matrix by each column of the second matrix. In general, if the length of the matrix is N, the total time complexity would be O(N^3).

Algorithm 1: The Naive Matrix Multiplication Algorithm

```
Data: S[A][B], P[G][H]
Result: Q[][]
if B == G then
    for m = 0; m < A; m++ do
        for r = 0; r < H; r++ do
            Q[m][r] = 0;
            for k = 0; k < G; k++ do
                Q[m][r] += S[m][k] * P[k][r];
            end
        end
    end
end
```

2. Line Matrix Multiplication

In this algorithm we tried to optimize the first one by multiplying an element from the first matrix by the correspondent line of the second matrix. The complexity of the algorithm is O(N^3). For the second part …

3. Block Matrix Multiplication

In this algorithm we try to optimize the multiplication even more. It divides the matrices in blocks and uses the same sequence of computation as in the second algorithm. The advantage of this approach is that the small blocks can be moved into the fast local memory and their elements can then be repeatedly used.This technique is commonly used in high-performance computing to optimize matrix operations, especially for large matrices. Though the complexity of the algorithm is O(N^3) same as the others.

Blocked

```
for (i = 0; i < n; i=i+bs)
for (j = 0; j < n; j=j+bs)
{
 for (ii = i; ii < i+bs && ii < n; ii++)
 for (jj = j; jj < j+bs && jj < n; jj++)
  c[ii*p+jj] = 0;
   for (k = 0; k < n ; k=k+bs)
      for (ii = i; ii < i+bs && ii < n; ii++)
      for (jj = j; jj < j+bs && jj < n; jj++)
      for (kk = k; kk < k+bs && kk < n; kk++)
       c[ii*p+jj] += a[ii*n+kk] * b[kk*p+jj];
}
```

Performance Metrics

In order to register the performance metrics we used a Performance API also known as PAPI for the C++ version of the algorithms. PAPI provides a standardized interface for collecting performance counter data from diverse hardware and software components, including CPUs. It is a widely-used library for collecting hardware performance metrics, offering portability and ease of us. And for the Java version we could only calculate the first metric To evaluate and compare the performance of the different algorithm versions, we selected the following performance metrics:

1. **Execution Time:** the amount of time taken by each algorithm to complete its execution

2. **Memory Usage:** the amount of memory required by each algorithm during execution.

3. **Cache Efficiency:** measures how effectively the algorithms utilize the cache memory hierarchy. We used this only for the part 2.