Implementation and Performance Analysis of LU Decomposition and Sieve of Eratosthenes

Relatório

Mestrado Integrado em Engenharia e Computação

Computação Paralela

Autor:
Diogo Serra Duque - up201406274@fe.up.pt

4 de Maio de 2018

Introdução

For the CPAR course, the class was proposed a study regarding the implementation and evaluation of sequential and parallel (distributed and shared) versions of two algorithms: LU Decomposition and Sieve of Eratosthenes.

All performance markers were taken in a Ubuntu laptop with an Intel Core i5-6200U (2 cores, 4 threads, with 2.30GHz base frequency but capable of reaching 2.80GHz using Turbo Boost) and a 3 MB SmartCache. Compilation was always made using the -O3 flag.

Problem Description

LU Decomposition

Given a matrix A of size *n* x *n*, we want to obtain a lower triangular matrix L of size *n* x *n* and an upper triangular matrix U of size *n* x *n* such that:

A = L·U

Sieve of Eratosthenes

Given any number *n*, we want to find every prime number belonging to the interval [2,n], { n ∈ ℕ | n > 2 }. A number is considered prime if it's only divisible by 1 and itself.

Algorithms Explanation

LU Decomposition (Doolittle Algorithm)

The Doolittle algorithm is used for the factorization of matrices into a lower triangular matrix and an upper triangular matrix. This algorithm iterates *k* from 0 to *n*, solving the *k*th row of U and *k*th column of *L*, thus solving the 2 matrices at the same time, from top-left to right-bottom. The core of the algorithm is shown below for better understanding:

```
for (int k = 0; k < n; k++)
{
    for (int m = k; m < n; m++)  //kth row of u
    {
        u[k][m] = a[k][m];
        for (int j = 0; j < k; j++)
        {
            u[k][m] -= l[k][j] * u[j][m];
        }
    }
    for (int i = k; i < n; i++) //kth column of l
    {
        l[i][k] = a[i][k];
        for (int j = 0; j < k; j++)
        {
            l[i][k] -= l[i][j] * u[j][k];
        }
        l[i][k] /= u[k][k];
    }
}
```

Fig. 1 - LU Decomposition (Doolittle) Algorithm

Sieve of Eratosthenes

The Sieve of Eratosthenes is used to find prime numbers up to a maximum number *n*. This algorithm creates a list of all the numbers up to *n*, that will be iterated repeatedly to find primes. In each iteration, the algorithm tries to find the first “unmarked” number on the list, which is then assumed as a prime. Then all its multiples are “marked” so that we know they cannot be primes. After this, the algorithm restarts until it finds no more unmarked numbers. In the implementation shown right below, the “marking” is made by setting the number to 0:

Fig. 2 - Sieve of Eratosthenes Algorithm

Note: the algorithm starts on number 3 because, during the initialization part, all even numbers (except for 2) were set to 0. This pre-calculation allows for steps of 2 in every *for* cycle, thus greatly reducing the number of iterations for each.

Algorithms Implementations

LU Decomposition (Doolittle) - Sequential

This implementation is the same as presented in the above Fig. 1.

LU Decomposition (Doolittle) - Shared Memory

This implementation is much like the sequential one, except for the parallelization in some loops. Against the “common sense”, the parallelization was not made on the intermediate loops to take advantage of less thread creation/destruction, but instead was made in the most inner loops and thus taking advantage of the possibility of reduction. Both methods were attempted, but the last one proved far better as we can see in Fig. 3.

Fig. 3 - Time performance of shared memory on LU Decomposition

LU Decomposition (Doolittle) - Distributed Memory

This implementation takes advantage of the fact that each cell's calculation depends only on cells within the same column/row from either triangular matrices, and assigns the cell's calculations to threads taking this into account. A cell's calculation belong to a thread according to the following rule: for the upper matrix, on a given *k* iteration, all cells where (*t - cellColumnIndex) mod threadsTotalNumber* belong to thread *t*, where *t* is the rank of the thread; for the lower matrix, on a given *k* iteration, all cells where (*t - cellRowIndex) mod threadsTotalNumber* belong to thread *t*, where *t* is the rank of the thread. At the end of the caculation of an upper matrix row, we broadcast the column that just got completed in that iteration to all other threads, as it will be needed for the next calculations. At the end of the caculation of an lower matrix column, we broadcast the row that just got completed in that iteration to all other threads. This ensures that every calculation will have the relevant information available.

Fig. 4 - Time performance of distributed memory on LU Decomposition

Sieve of Eratosthenes - Sequential

This implementation is the same as presented in the above Fig. 2.

Sieve of Eratosthenes - Shared Memory

This implementation's logic is very similar to the sequential one, except for the parallelization of every outer loop. The first loop (responsible for putting all numbers up to *n* in the array) is parallelized and is also statically scheduled with a big chunk size, so there are not so much DCM, as each thread will do a lot of work on a specific memory zone of the array before “jumping” to another chunk. The second loop (main loop, where we mark non-primes) is dynamically scheduled for smaller chunk sizes, as the workload is different from where each thread is and it's important to prioritize the unmarking of lower numbers first. The third loop (where we pass all unmarked numbers to a new array) follows the same principles as the

first one. This last loop could have a critical zone when adding numbers to the new array, but it was not worth it as we can see in Fig. 4.

Fig. 5 - Time performance of shared memory on Sieve of Eratosthenes

Sieve of Eratosthenes - Distributed Memory

This implementation follows the iteration order of the larger loop (same as the sequential) but attempts to distribute the workload of each loop between threads. This means that each thread is responsible for a certain range of numbers, and in each iteration of the main loop it will mark every multiple of *k* (the prime in analysis) belonging to this range. This reduces the amount of comunication needed along the algorithm, only needing a single *broadcast* of the next *k* at the end of the each loop. When the main loop finishes, we only need to send the unmarked numbers (the ones who are not multiples of lower numbers, therefore being primes) to the root thread using a *reduction* (to know the total number of primes) and a *gather* (to really gather all the primes in one thread).

Fig. 6 - Time performance of distributed memory on Sieve of Eratosthenes

Results Analysis

LU Decomposition

From Fig. 7, it's easy to see that the better-scaling implementation is the one using shared memory. A likely cause for the discrepancy between both parallel versions (shared vs distributed) is due to fact that the shared one does not make so much “memory jumps” as the distributed, instead analyzing more contiguous memory. The “not so good” use of memory brings the distributed memory implementation closer to the sequential version that to the shared one. Another reason is that the shared approach takes advantage of the reduction possibility, making the loops more efficient that using a critical section.

Regarding the dilemma about the loops where the parallelization was made, even though the difference between both approaches aren't that high, it's still visible and should be taken into account. In this case, the drawback of using smaller loops and needing to create/destroy threads more often is greatly rewarded by the reduction, reducing the running time of the algorithm.

Fig. 7 - Overall time performance on LU Decomposition

Sieve of Eratosthenes

Even though the sequential algorithm is already rather fast and optimized, the parallel versions show that this performance can be further enhanced. As we can see on Fig. 8, we can obtain about 25% reduction on the 600000000 iteration, predicting higher percentages with maximum limits. Both parallel implementations take advantage of calculation by chunks, thus highly reducing DCM and making sure the algorithm is as fast as possible.

Regarding the dilemma about using a critical section it the shared version, the reason for the unexpected slowdown while using it is due to the fact that we are forcing so much communication in such a small loop that it gets slower, instead of faster.

Fig. 8 - Overall time performance on Sieve of Eratosthenes

Conclusions

To conclude, parallelization has proved itself as a great way of optimizing a program, although somewhat fickle and highly dependant on the expertise of the programmer. There are a lot of pitfalls and variables that can impact the performance, such as the Cache Misses and locks when waiting for other threads.

Even though both algorithms obtained rather good results from the shared memory approach, this is not an indicator that it's a better approach for both of the problems. Instead, it shows that distributed memory management, although more efficient as does not need to always be “in sync”, can become rather hard because it is up to the programmer to make sure the synchronization happens at the right time and with minimal impact on performance.

In this particular case, we can confirm that the parallelization really optimized the traditional and sequential approach, bringing exponentially better times in most cases.