Cannon's algorithm

In computer science, **Cannon's algorithm** is a distributed algorithm for matrix multiplication for two-dimensional meshes first described in 1969 by Lynn Elliot Cannon.[1][2]

It is especially suitable for computers laid out in an $N \times N$ mesh.[3] While Cannon's algorithm works well in homogeneous 2D grids, extending it to heterogeneous 2D grids has been shown to be difficult.[4]

The main advantage of the algorithm is that its storage requirements remain constant and are independent of the number of processors.[2]

The Scalable Universal Matrix Multiplication Algorithm (SUMMA)[5] is a more practical algorithm that requires less workspace and overcomes the need for a square 2D grid. It is used by the ScaLAPACK, PLAPACK, and Elemental (http://code.google.com/p/elemental/) libraries.

Contents

Algorithm overview

When multiplying two n×n matrices A and B, we need n×n processing nodes p arranged in a 2d grid. Initially $p_{i,j}$ is responsible for $a_{i,j}$ and $b_{i,j}$.

```
// PE(i , j)
k := (i + j) mod N;
a := a[i][k];
b := b[k][j];
c[i][j] := 0;
for(l := 0; l < N - 1; l++){
    c[i][j] := c[i][j] + a * b;
        concurrently{
            send a to PE(i, (j + N - 1) mod N);
            send b to PE((i + N - 1) mod N, j);
        } with {
            receive a' from PE(i, (j + 1) mod N);
            receive b' from PE((i + 1) mod N, j );
        }
    a := a';
    b := b';
}
```

We need to select k in every iteration for every Processor Element (PE) so that processors don't access the same data for computing $a_{ik} * b_{kj}$.

Therefore processors in the same row / column must begin summation with different indexes. If for example *PE(0,0)* calculates $a_{00} * b_{00}$ in the first step, *PE(0,1) chooses* $a_{01} * b_{11}$ first. The selection of *k := (i + j) mod n* for *PE(i,j)* satisfies this constraint for the first step.

In the first step we distribute the input matrices between the processors based on the previous rule.

In the next iterations we choose a new *k' := (k + 1) mod n* for every processor. This way every processor will continue accessing different values of the matrices. The needed data is then always at the neighbour processors. *A PE(i,j)* needs then the a from *PE(i,(j + 1) mod n)* and the b from *PE((i + 1) mod n,j)* for the next step. This means that a has to be passed cyclically to the left and also b cyclically upwards. The results of the multiplications are summed up as usual. After n steps, each processor has calculated all $a_{ik} * b_{kj}$ once and its sum is thus the searched c_{ij}.

After the initial distribution of each processor, only the data for the next step has to be stored. These are the intermediate result of the previous sum, a a_{ik} and a b_{kj}. This means that all three matrices only need to be stored in memory once evenly distributed across the processors.

Generalisation

In practise we have much fewer processors than the matrix elements. We can replace the matrix elements with submatrices, so that every processor processes more values. The scalar multiplication and addition become sequential matrix multiplication and addition. The width and height of the submatrices will be $N = n/\sqrt{p}$.

The runtime of the algorithm is $T(n,p) = T_{coll}(n/N,p) + N * T_{seq}(n/N) + 2(N-1)(T_{start} + T_{byte}(n/N)^2)$, where T_{coll} is the time of the initial distribution of the matrices in the first step, T_{seq} is the calculation of the intermediate results and T_{start} and T_{byte} stands for the time it takes to establish a connection and transmission of byte respectively.

A disadvantage of the algorithm is that there are many connection setups, with small message sizes. It would be better to be able to transmit more data in each message.

See also

- Systolic array

References

1. Lynn Elliot Cannon, *A cellular computer to implement the Kalman Filter Algorithm (http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=905686)*, Technical report, Ph.D. Thesis, Montana State University, 14 July 1969.
2. Gupta, H.; Sadayappan, P.: Communication Efficient Matrix-Multiplication on Hypercubes (http://dbpubs.stanford.edu:8090/pub/1994-25), dbpubs.stanford.edu
3. 4.2 Matrix Multiplication on a Distributed Memory Machine (http://www.phy.ornl.gov/csep/la/node6.html), www.phy.ornl.gov
4. Ph.D. Research (https://web.archive.org/web/20090517034825/http://graal.ens-lyon.fr/~jfpineau/research.html), graal.ens-lyon.fr. The thesis itself is not available from the archived link.
5. Robert A. van de Geijn and Jerrell Watts, SUMMA: scalable universal matrix multiplication algorithm (http://onlinelibrary.wiley.com/doi/10.1002/(SICI)1096-9128(199704)9:4%3C255::AID-CPE250%3E3.0.CO;2-2/abstract), Concurrency: Practice and Experience. Volume 9, Issue 4, pages 255–274, April 1997.

External links

- Lecture at Berkeley (http://www.cs.berkeley.edu/~demmel/cs267/lecture11/lecture11.html)
- mu.oz.au (http://www.cs.mu.oz.au/498/notes/node30.html)
