Large scale kronecker product on supercomputers

Claude Tadonki

Large scale kronecker product on supercomputers*

Claude Tadonki
MINES ParisTech - CRI (Centre de Recherche en Informatique) - Mathematiques et Systèmes
35, rue Saint-Honoré
77305, Fontainebleau-Cedex

Laboratoire de l'Accélérateur Linéaire/IN2P3/CNRS
University of Orsay, Faculty of Sciences, Bat. 200
91898 Orsay Cedex (France)
claude.tadonki@u-psud.fr

Abstract

The Kronecker product*, also called* tensor product*, is a fundamental matrix algebra operation, which is widely used as a natural formalism to express a convolution of many interactions or representations. Given a set of matrices, we need to multiply their Kronecker product by a vector. This operation is a critical kernel for iterative algorithms, thus needs to be computed efficiently. In a previous work, we have proposed a cost optimal parallel algorithm for the problem, both in terms of floating point computation time and interprocessor communication steps. However, the lower bound of data transfers can only be achieved if we really consider (local) logarithmic broadcasts. In practice, we consider a communication loop instead. Thus, it becomes important to care about the real cost of each broadcast. As this local broadcast is performed simultaneously by each processor, the situation is getting worse on a large number of processors (supercomputers). We address the problem in this paper in two points. In one hand, we propose a way to build a virtual topology which has the lowest gap to the theoretical lower bound. In the other hand, we consider a hybrid implementation, which has the advantage of reducing the number of communicating nodes. We illustrate our work with some benchmarks on a large SMP 8-Core supercomputer.*

1 Introduction

The *Kronecker product* is a basic matrix algebra operation, which is mainly used for multidimensional modeling in number of specialized fields[5]: *Stochastic Automata Networks* (SAN) [2, 3, 4], *Fast Fourier Transform* (FFT), *Fast Poisson Solver* (FPS) [9, 10], *Quantum Computation* (QC) [6] and *Lattice Quantum Chromodynamics* [8]. Considering a principal matrix expressed as a Kronecker product of several matrices, iterative schemes require to repeatedly multiply such a matrix by a vector. Formally, we are given N square matrices $A^{(i)}$ of sizes n_i, $i = 1, \cdots, N$, and a vector x of length $L = n_1 n_2 \cdots n_N$, and we need to compute y (of length L) given by

$$y = x(\bigotimes_{i=1}^{N} A^{(i)}). \tag{1}$$

It is well-known that we should not compute the matrix explicitly before performing the multiplication, as this would require a huge memory to store that matrix and will yield redundant computations. A cost optimal algorithm for this computation proceeds in a recursive way, consuming one matrix $A^{(i)}$ after another [7]. Consequently, traditional parallel routines for matrix-vector product cannot be considered. When starting with the recursive algorithm as a basis, any parallel scheme will involve a set of data communication at the end of each iteration. The cost of this communication is the main challenge for this problem, especially with a large number of processors, because there is a significant interleave between the (probably virtual) communication links. Moreover, in order to reduce the cache misses due to an increasing stride from one iteration to the next one, array reshuffling is sometimes considered, and this complicates the communication topology.

In [7], we have proposed an efficient parallel algorithm which achieves the multiplication without explicit shuffling and requires a minimal number of communication steps. However, the real cost of each communication step depends on the virtual topology and the way the transfers are really

*This work was partly supported by the PetaQCD (ANR) project.

performed. This problem was left open in the paper because of the modest size of the parallel computers considered (up to 256 processors). In this paper, we provide an algorithm to construct an efficient topology, in addition to a hybrid implementation using OpenMP[11] on the computing multicore nodes. With this contribution, we keep the global efficiency of the original algorithm on a larger number of processors as illustrated by some benchmark results. The rest of the paper is organized as follows. Section 2 gives an overview of the original algorithm. This is followed in section 3 by a discussion on its complexity and the position of the problem. We describe our heuristic to find an efficient topology in section 4. We discuss the hybrid implementation and section 5. In section 6, we display and comment our benchmark results. Section 7 concludes the paper.

2 Original parallel algorithm

We restate our parallel algorithm in order to provide a self-contained material, the reader could refer to [7] for more details. From (1) and using the so-called *canonical factorization*, we obtain the recursive scheme defined by (2)

$$\begin{cases} V^{(N+1)} = x \\ V^{(s)} = V^{(s+1)}(I_{n_1 \cdots n_{s-1}} \otimes A^{(s)} \otimes I_{n_{s+1} \cdots n_N}) \end{cases} \quad (2)$$

which leads at the last step to $V^{(1)} = x \otimes_{i=1}^{N} A^{(i)}$. Our parallelization of the recursive computation expressed by equation (2) can be defined as follows. Given p processors (assuming that p divides $L = n_1 n_2 ... n_N$), we proceed as follows. We first compute a sequence of N integers p_i such that $p = p_1 p_2 ... p_N$ and p_i divides $n_i, i = 1, 2, ..., N$. Considering a multidimensional indexation, we say that each processor $(a_1, a_2, \cdots, a_N)$ computes the entries $(b_1, b_2, \cdots, b_N)$ of $V^{(s)}$ such that $b_i \bmod p_i = a_i$, $i = 1, 2, \cdots, N$. A complete description of the parallel algorithm is given by Alg. 1. Note that the *send* and *receive* occurrences can be combined into a single *sendreceive* call because of the symmetry of the topology.

3 Communication complexity

Our scheduling onto p processors is based on a decomposition $(p_1, p_2, \cdots, p_N)$ such that p_i divides n_i, and $p_1 p_2 \cdots p_N = p$. In theory, algorithm Alg. 1 performs $log(p)$ parallel communication steps when executed with p processors. Indeed, one local broadcast occurs at the end of each step s, thus we do $log(p_1)+log(p_2)+\cdots+log(p_N) = log(p_1 p_2 \cdots p_N) = log(p)$ parallel communication steps. This assumes that, at a given step i, we perform $log(p_i)$ parallel transfers (local broadcast to p_i processors by each processor). However, in practice, we issue $p_i - 1$ transfers (communication loop). Thus, the gap between $p_i - 1$ and $log(p_i)$ becomes important for larger p_i. Actually, each processor performs $p_1 + p_2 + \cdots + p_N$ transfers in total. On a larger cluster, there will be an additional overhead coming from the gap between the virtual topology and the physical topology. We first focus on how to find a decomposition which reduces the measure $p_1 + p_2 + \cdots + p_N$.

$\pi \leftarrow 1$; $r \leftarrow 1$; $\ell \leftarrow c_1 c_2 ... c_N = L/p$ /* $c_i = \frac{n_i}{d_i}$ */
$y \leftarrow x(Q_{1w_1}, Q_{2w_2}, ..., Q_{Nw_N})$
For $s \leftarrow N$ **downto** 1 **do**
 $\ell \leftarrow \ell / c[s]$
 $ws = [w \mathbf{div}(\pi)] \mathbf{mod}(d[s]) + 1$
 $e \leftarrow (ws - 1) \times c[s]$
 $v \leftarrow 0$
 $i \leftarrow 1$
 For $a \leftarrow 1$ **to** ℓ **do**
  For $j \leftarrow e + 1$ **to** $e + c[s]$ **do**
   For $b \leftarrow 1$ **to** r **do**
    For $t \leftarrow e + 1$ **to** $e + c[s]$ **do**
     $v[i] \leftarrow v[i] + A(s, t, j) y[I + (t - j)r]$
    end do
    $i \leftarrow i + 1$
   end do
  end do
 end do
 If $(ws = 1)$ **then** $H \leftarrow d$ **else** $H \leftarrow ws - 1$
 For $T = ws + 1$ **to** $ws + d[s] - 1$ **do**
  $G \leftarrow$ **mod**$(T - 1, d[s]) + 1$
  $idest \leftarrow w + (G - ws) \times \pi$
  $isender \leftarrow w + (H - ws) \times \pi$
  send(y, $idest$, ws)
  recv(u, $isender$, H)
  $e \leftarrow (H - 1) \times c[s]$
  $i \leftarrow 1$
  For $a \leftarrow 1$ **to** ℓ **do**
   For $j \leftarrow 1$ **to** $c[s]$ **do**
    For $b \leftarrow 1$ **to** r **do**
     For $t \leftarrow e + 1$ **to** $e + c[s]$ **do**
      $v[i] \leftarrow v[i] + A(s, t, j) u[I + (t - j)r]$
     end do
     $i \leftarrow i + 1$
    end do
   end do
  end do
  If $(H = 1)$ **then** $H \leftarrow d[s]$ **else** $H \leftarrow H - 1$
 end do
 $r \leftarrow r \times c[s]$
 $\pi \leftarrow \pi \times d[s]$
 If $(s > 1)$ **then** $y \leftarrow v$
end do
$z(Q_{1w_1}, Q_{2w_2}, ..., Q_{Nw_N}) \leftarrow v$

Alg. 1 : Implementation of the matrix-vector product.

4 Heuristic for an efficient topology

We propose the algorithm Alg. 2 to find an efficient decomposition for a given number of processors p, which is a factor of $n_1 n_2 \cdots n_N$.

$d \leftarrow p$
{Starting decomposition }
For $i \leftarrow 1$ **to** N **do**
  $p_i \leftarrow$gcd(d, n_i)
  $d \leftarrow \frac{d}{p_i}$
enddo
{Recursive refinement }
For $i \leftarrow 1$ **to** N **do**
  For $j \leftarrow 1$ **to** N **do**
    $\alpha \leftarrow$gcd$(p_i, \frac{n_j}{p_j})$
    if $((\alpha > 1) \wedge (p_i > cp_j))$
      $p_i \leftarrow \frac{d_i}{\alpha}$
      $p_j \leftarrow \alpha d_j$
    endif
  enddo
enddo

Alg. 2 : Heuristic for an efficient decomposition

The principle of Alg. 2 is the following. We start with a gcd decomposition. Next, we refine it using the fact that if $p_i > \alpha p_j$, with α a non trivial factor of p_i, then $p_i/\alpha + \alpha p_j < p_i + p_j$. It is thus rewarding to replace p_i (resp. p_j) by p_i/α (resp. αp_j). Once this is done, it is clear that on a larger cluster (i.e. large value of p), all these simultaneous transfers will exacerbate the communication overhead and certainly slowdown the global performance. Fortunately, most modern supercomputers are built up with multicore nodes. Thus, a hybrid implementation, which combines the standard distributed memory implementation with a shared memory program (SMP) on the nodes, will overcome the problem by reducing the number of communicating nodes.

5 SMP implementation

We chose to use OpenMP to derive our shared memory code. Looking at Alg. 1, we decide to put the loop distribution pragma over the a loop. In order to do so, we first need to remove the $i \leftarrow i + 1$ incrementation and directly calculate the i index, which is given by $i \leftarrow c[s] \times r \times (a-1) + r \times (j-1) + b$. Now, the length ℓ where the loop blocking will occur varies with s ($\ell \leftarrow \ell / c[s]$). Thus, we need to keep it being a factor of the (fixed) number of threads. We achieve it by splitting the main loop into two parts, means isolating the case $s = N$ and then enclose the rest ($s = N-1, N-2, \cdots, 1$) into a parallel section. Moreover, since the number of nodes is now reduced to p/T (T is the number of OpenMP threads), we need to adapt our primarily decomposition such that ℓ remains a factor of T. The general way to do that is to split the loop over s at the right place (not only the extremal iteration), but this would be better implemented with Posix threads library, because we could dynamically manage the threads to handle desired loop partitioning (this is left for future work). We now show the impact of our strategy on benchmark results. Interested reader can download the source code at
http://www.omegacomputer.com/staff/tadonki/codes/kronecker.f

6 Experimental results

We consider a SMP 8-core cluster named JADE [12]. The whole cluster JADE is composed of 1536 compute nodes (i.e. 1536 x 8 = 12288 cores of Harpertown type processors) and 1344 compute nodes of nehalem type processor (1344 x 8 = 10 752 cores). The network fabric is an Infiniband (IB 4x DDR) double planes network for the first part of the machine (Harpertown), whereas 4 drivers InfiniBand 4X QDR provide 72 ports IB 4X QDR on output of each IRU of the second part of the machine (576 Go/s).

We choose $N = 6$ square matrices of orders 20, 36, 32, 18, 24, and 16, which means a principal matrix of order L = 159 252 480. We first show in table 1 the results of the pure MPI code. The decomposition obtained with our algorithm is marked with a star and is surrounded by two alternative decompositions (the one obtained by a basic gcd decomposition and the less distributed one) to illustrate the difference.

p	decomposition	time(s)
32	(4,1,8,1,1,1)	2.06 s
32	(2,2,2,2,2,1)*	1.62 s
32	(1,1,32,1,1,1)	4.14 s
180	(20,9,1,1,1,1)	0.75 s
180	(5,3,2,2,3,1)*	0.34 s
180	(10,6,1,1,3,1)	0.49 s
720	(20,36,1,1,1,1)	1.20 s
720	(10,3,2,2,3,2)*	0.23 s
720	(10,9,4,2,1,1)	0.35 s
2880	(20,36,4,1,1,1)	1.47 s
2880	(10,6,2,2,6,2)*	1.20 s
2880	(20,12,2,2,3,1)	1.32 s
4320	(20,36,2,3,1,1)	1.48 s
4320	(10,3,4,3,3,4)*	0.92 s
4320	(20,18,4,3,1,1)	1.34 s

Table 1. MPI implementation timings

From Table 1, we see that for a given number of processors, the partition obtained with our procedure can im-

prove the global performance by a factor from 2 to 5 (see $p = 720$). However, when the number or MPI processes increases, we see that we lose the scalability, because data communication severely dominates (the code is cost optimal for floating point operations). We now see how this is improved using a hybrid implementation. We reconsider the previous best decompositions as baseline and compare each of them with the corresponding hybrid configuration. For each number of cores in $\{4320, 2880, 720\}$, we consider a k-cores SMP clustering, $k \in \{1, 4, 8\}$.

#MPI	decomposition	#threads	time	speedup
4320	(10,3,4,3,3,4)	1	0.92 s	1
1080	(5, 3, 2, 3, 3, 4)	4	0.16 s	5.75
540	(5, 3, 2, 3, 3, 2)	8	0.12 s	7.67
2880	(10,6,2,2,6,2)	1	1.20 s	1
360	(5, 3, 2, 3, 3, 4)	8	0.16 s	7.5
720	(10,3,2,3,3,2)	1	0.23 s	1
90	(5, 3, 2, 1, 3, 1)	4	0.45 s	0.51

Table 2. Hybrid (MPI+OpenMP) code timings

We can see from Table 2 that we are close to a linear (threads) speedup with 4320 and 2880 cores. This is due to the fact that the global computation time was really dominated by data communication and synchronization mechanism. For a smaller number of cores, we see that we start loosing the benefit of the SMP implementation. This is due the (predictable) cache misses penalty coming from the stride $(t-j)\times r$ in Alg. 1, which is increasingly bigger since r does. We could use larger number of cores, but we our experimental configuration sufficiently illustrative of what we need to show and how our solution contributes to the issues.

7 Conclusion

The problem of multiplying a vector by a Kronecker product of matrices is crucial in stochastic sciences and is a computationally challenging task for large instances. In order to avoid a memory bottleneck and redundant computation, a recursive scheme has been mathematically formulated, for which corresponding efficient implementations are expected. In one hand, the increasing loop stride needs to be handled carefully in order to reduce the impact of caches misses. This aspect really dominates and thus needs to be seriously taken into account in the performance analysis. In the other hand, the parallelization requires an important number of parallel transfers, which could become problematic on large clusters. This paper provides a contribution on both aspects, based on a cost optimal solution (floating point computation point of view) from the literature. Our solution is a combination of a heuristic procedure to build an efficient virtual topology and the use of hybrid programming paradigm. Our experimental results illustrate the improvement of our contribution, and evidence the need of a compromise on large clusters.

References

[1] M. Davio, *Kronecker Products and Shuffle Algebra*, IEEE Trans. Comput., Vol. C-30, No. 2, pp. 116-125, 1981.

[2] P. Fernandes, B. Plateau, and W. J. Stewart, *Efficient Descriptor-Vector Multiplications in Stochastic Automata Networks*, INRIA internal report No. 2935 , July 1996.

[3] Benoit, Anne and Fernandes, Paulo and Plateau, Brigitte and Stewart, William J., *On the benefits of using functional transitions and Kronecker algebra*, Performance Evaluation, Vol. 58(4), pp. 367-390, 2000.

[4] Benoit, Anne and Plateau, Brigitte and Stewart, William J., *Memory-efficient Kronecker algorithms with applications to the modeling of parallel systems*, Future Gener. Comput. Syst., Vol. 22(7), pp. 838-847, 2006.

[5] J. Granta, M. Conner, and R. Tolimieri, Recursive fast algorithms and the role of the tensor product, IEEE Transaction on Signal Processing, 40(12):2921-2930, December 1992.

[6] P. W. Shor, Quantum Computing, Proceeding of the ICM Conference, 1998.

[7] C. Tadonki and B. Philippe, Parallel Multiplication of a Vector by a Kronecker Product of Matrices, Journal of Parallel and Distributed Computing and Practices, Parallel Distributed Computing Practices PDCP, volume 3(3), 2000.

[8] C. Tadonki, G. Grosdidier, and O. Pene, *An efficient CELL library for lattice quantum chromodynamics*, ACM SIGARCH Computer Architecture News, Vol. 38(4), 2011.

[9] C.Tong and P. N. Swarztrauber, Ordered Fast Fourier Transforms on Massively Parallel Hypercube Multiprocessor, Journal of Parallel and Distributed Computing 12, 50-59, 1991.

[10] C. Van Loan, *Computational Framework for the Fast Fourier Transform*, SIAM, 1992.

[11] `http://openmp.org/`

[12] `http://www.cines.fr`