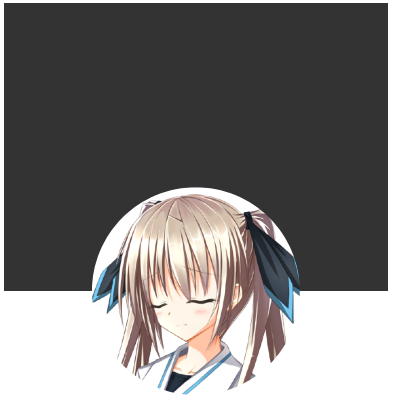


Rainmaker's Notebook
『求雨巫师的神奇之处在于他总是躲着不见你，却总说刚下完的雨是拜他所赐。』——《天真的人类学家》

Home

Archives

About

SiteXC

COSMA 算法分析

最近由于业务需要，我对 COSMA (Communication-Optimal S-partitioned Matrix-multiplication Algorithm) 做了一番比较深入的研究。本文简要概括了并行矩阵乘法的发展历史和思路，并分析了 COSMA 算法的思路和实现。

1. 矩阵乘法的理解

本文讨论一般稠密矩阵乘法（general matrix-matrix multiplication, GEMM）：

$$C = A \times B,\ A \in \mathbb{C}^{m \times k},\ B \in \mathbb{C}^{k \times n}, C \in \mathbb{C}^{m \times n}.$$

本文只考虑 $O(mnk)$ 复杂度的算法，不考虑 Strassen 一类的所谓的快速算法。

一个矩阵乘法的计算有两种理解方式：(1) mn 个独立的 k 维向量内积，(2) k 个 rank-1 update 累加。经典的并行 2D 算法 Cannon [1], PUMMA [2], 以及 SUMMA [3], 都可以视作并行计算 (1) 中的各个向量内积：这些 2D 算法都是对 C 做 2D 划分，然后每个进程计算其中一个 C 分块的结果。我们将这些 2D 划分称作『m 维度和 n 维度的并行』。同时，SUMMA 也可以视作并行计算 (2) 中的每一个 rank-1 update （分块版的 SUMMA 则是对每个 rank-b update 进行并行）。

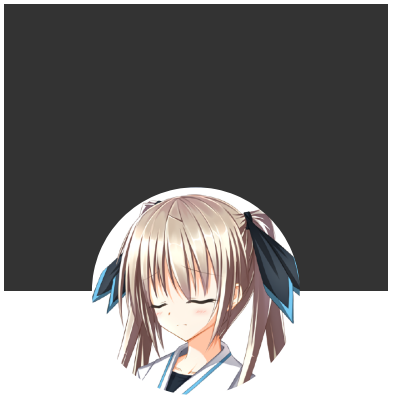
2. 3D 矩阵乘法：是什么，为什么

一个很自然的问题是，如果我们同时在 m, n, k 三个维度进行并行会怎么样？

当然可以。但是为什么要这么做？三个维度同时并行看起来复杂了不少，起码不是那么容易在草稿纸上画出简明易懂的示意图了。

对矩阵乘法进行三个维度的同时并行至少有三个理由：

1. 增加并行度。假设 m, n 很小，k 很大，那么只对 m, n 并行可能无法提供足够的并行度。比如说，在线性方程组迭代求解器里，有时候需要做 CholeskyQR，需要计算 $A^T \times A$，其中



Rainmaker's Notebook

『求雨巫师的神奇之处在于他总是躲着不见你，却总说刚下完的雨是拜他所赐。』——《天真的人类学家》

$m = n < 10$, 而 k 非常大。在具有几千个甚至上万个线程的 GPU 上面，只对 m, n 并行显然无法提供足够的并行度。

2. 增加矩阵乘法计算效率。矩阵乘法的计算访存比是 $2mnk/(mk + kn + mn)$。如果只对 m, n 并行，在进程数非常多的时候，计算访存比就会向 1 趋近，使得计算效率大幅下降。
3. 减少通信量（数据访问量）。这一点在 shared memory 并行中不太明显，但是在 distributed memory 并行中极为重要，因为跨进程通信的代价太大，往往是性能瓶颈所在。如何理解“减少通信量”呢？举个例子。我们有 16 个进程，使用 4×4 的网格。一开始，A, B 在 16 个进程上各自只有一个拷贝。而 2D 算法的计算本质使得在计算结束以后，每一个进程都拿到了 1/4 个 A 和 B 的数据，也就是至少有 3 份 A 和 B 的数据在进程之间相互交换了。但是采用 3D 算法以后，不需要有那么多份 A 和 B 的数据在进程之间进行交换，而代价是需要额外交换一些 C 的数据。对于这一点，[4] [5] [6] [7] 有详细的讨论，此略。

3. 先行者：3D 与 2.5D 算法

最早使用 3D 进程网格进行并行的是经典的 3D 算法 [8]，此后2.5D 算法 [9] 稍微放松了 3D 算法的限制，并且和 3D 算法的形式比较类似。因此，一般将 3D 和 2.5D 算法放到一起讨论。

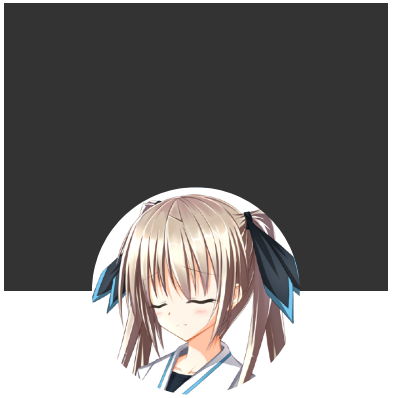
对 m, n, k 三个维度的同时并行意味着需要一个三维的进程网格。下记进程网格大小为 $p_m \times p_n \times p_k$，表示在 m, n, k 方向上有多少个进程，$P = p_m \times p_n \times p_k$ 为总进程数。对于 3D 算法，$p_m = p_n = p_k = P^{1/3}$. 对于 2.5D 算法，$p_m = p_n = \sqrt{P/c}$, $p_k = c$, 此外 p_m 必须是 c 的倍数。 那么这么一些 3D 进程网格具体在做什么呢?

我们先看 3D 算法。一个最简单的例子是假设 A, B, C 都被划分成了 2×2 个子矩阵。那么我们有：

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}.$$

假设原来有 4 个进程，使用 2D 算法，则每个进程独立计算一个 C 的分块。现在我们把进程数增加到 8 个，对应 $2 \times 2 \times 2$ 的进程网格。这意味着每个 C 的分块现在需要被 $p_k = 2$ 个进程计算。比如说，$C_{01} = A_{00}B_{01} + A_{01}B_{11}$, 则 $A_{00}B_{01}$ 由进程 $(0, 0, 1)$ 计算，$A_{01}B_{11}$ 由进程 $(0, 1, 1)$ 计算。对于更大的进程网格，计算是类似的。

解决了『每个进程要算些什么』以后，剩下的问题就是『数据从哪里来』，因为进程之间的数据交换需要显式声明。由上面的描述，我们可以观察到一个简单的规律：进程 P_{ijk} 计算 $C_{ijk} = A_{ij} \times B_{jk}$. 也就是说 P_{ijk} 需要 A_{ij}, B_{jk}, 并且最后结果要累加到 C_{ik} 上。由此我们可以得出一个简单的数据初始分布



Rainmaker's Notebook

『求雨巫师的神奇之处在于他总是躲着不见你，却总说刚下完的雨是拜他所赐。』——《天真的人类学家》

Home

Archives

About

SiteXC

和通信方式：P_{ij0} 在一开始的时候持有 A_{ij} 并将其广播到 $P_{ij:}$，P_{0jk} 在一开始的时候持有 B_{jk} 并将其广播到 $P_{:jk}$，P_{ijk} 将其 C_{ijk} reduce-sum 到 P_{i0k} 得到 C_{ik}.

2.5D 算法和 3D 算法类似，但是没有那么直观。算法过程稍微冗长一点，请自行参阅 [9] 中的 Algorithm 2. 与 3D 算法相比，2.5D 算法稍微灵活一点，允许用户选定 c 的值来控制内存使用量。在 2D 算法里，无论哪一步，每一个进程都只需要保存 $1/P$ 个 A, B, C 矩阵。但是在 3D 和 2.5D 算法里，需要保存的矩阵元素就更多了。因此 2.5D 算法引入的 c 参数可以调节内存用量，在 $c = 1$ 的时候回落到 Cannon 算法。这个设计可以说是很巧妙的。

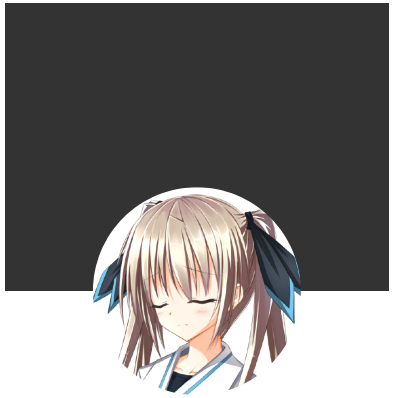
3D 和 2.5D 算法看起来已经解决了对矩阵乘法三维并行的问题了。但仔细观察一下，我们会发现它们有一个硬伤：3D 进程网格的要求太死板了。2.5D 算法说是说不要求 P 是个立方数，但是 p_m 必须是 c 的倍数，实际上要求 $P = u^2c^3, u = p_m/c$. 此外，如果 $m = n \ll k$ 且 m 非常小，3D 和 2.5D 算法依然可能无法提供足够的并行度。

4. CARMA：划时代的理想主义者

作为数值线性代数的祖师爷，James Demmel 自然不会止步于 2.5D 算法。2013 年，Communication-Avoding Recursive Matrix-multiplication Algorithm (CARMA) [10] 横空出世。CARMA 以一个简单优雅的方式处理了矩阵乘法的并行，其核心思想可以用一句话概括：将当前问题的最大尺寸进行对半划分，如果还有额外可用的内存则将进程对半划分每一半解决一个子问题，否则所有进程一起依次解决这两个子问题 (Algorithm 1 in [10])。CARMA 的划时代之处在于它将 1D, 2D, 3D 算法统一化了。用多少维度的并行完全取决于问题，再也不需要用户手动指定某些参数，或者有什么进程网格的限制。如此简单优雅的方式还使得 CARMA 的渐进通信复杂度达到了并行矩阵乘法的通信复杂度理论下界，也就是直接摸到了天花板。下面我们就 CARMA 算法中遇到的三种情况展开介绍其具体行为。

假设当前要处理的矩阵乘法问题 m 最大，那么我们切分 m 并得到两个子问题 $[C_{00}; C_{10}] = [A_{00}; A_{10}] \times B$. 如果我们有足够的可用内存，则将进程分成两组 PG_0, PG_1，前者算 $C_{00} = A_{00} \times B$, 后者算 $C_{10} = A_{10} \times B$. 原本所有进程持有一份 B 的拷贝，现在 PG_0, PG_1 各自需要持有一份 B 的拷贝。如果我们没有足够的可用内存，则所有进程一起先算 $C_{00} = A_{00} \times B$, 然后一起算 $C_{10} = A_{10} \times B$.

假设当前要处理的矩阵乘法问题 n 最大，那么我们切分 n 并得到两个子问题 $[C_{00}, C_{01}] = A \times [B_{00}, B_{01}]$. 如果我们有足够的可用内存，则将进程分成两组 PG_0, PG_1，前者算 $C_{00} = A \times B_{00}$, 后者算 $C_{01} = A \times B_{01}$. 原本所有进程持有一份 A 的拷贝，现在 PG_0, PG_1 各自



Rainmaker's Notebook
『求雨巫师的神奇之处在于他总是躲着不见你，却总说刚下完的雨是拜他所赐。』——《天真的人类学家》

Home

Archives

About

SiteXC

需要持有一份 A 的拷贝。如果我们没有足够的可用内存，则所有进程一起先算 $C_{00} = A \times B_{00}$，然后一起算 $C_{01} = A \times B_{01}$.

假设当前要处理的矩阵乘法问题 k 最大，那么我们切分 k 并得到两个子问题 $C = [A_{00}, A_{01}] \times [B_{00}; B_{10}]$. 如果我们有足够的可用内存，则将进程分成两组 PG_0, PG_1，前者算 $C^{(0)} = A_{00} \times B_{00}$，后者算 $C^{(1)} = A_{01} \times B_{10}$，然后两个进程组累加得到 $C = C^{(0)} + C^{(1)}$。原本所有进程持有一份 A 和 B 的拷贝，现在 PG_0 需要持有 A_{00}, B_{00}，PG_1 需要持有 A_{01}, B_{10}。如果我们没有足够的可用内存，则所有进程一起先算 $C^{(0)} = A_{00} \times B_{00}$，然后一起算 $C^{(1)} = A_{01} \times B_{10}$.

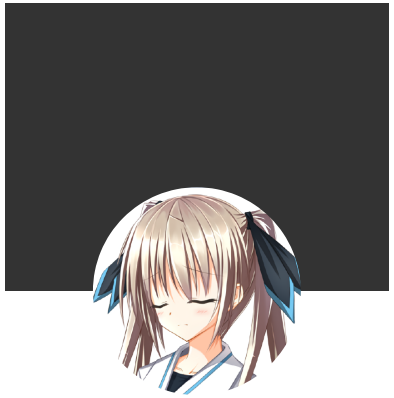
讲完了丰满的理想，我们来看看现实的骨感。对于 CARMA，第一个很容易问出来的问题是『如果在某一步 P 不是 2 的倍数怎么办？』这个问题有不同的解决方法，但是实现起来都会出现各种棘手的情况。第二个问题也很蛋疼：『既然每一步划分子问题都对矩阵有不同的存储要求，那么这些矩阵一开始要怎么分布？』这两个问题搅在一起，就更蛋疼了。我们引用论文中的几句话结束本节：

> The MPI version of CARMA only communicates one of the three matrices at each BFS step. It requires that each of the two halves of the other two matrices already resides entirely on the corresponding half of the processors... The recursive data layout that the distributed version of CARMA uses is different from any existing linear algebra library; hence CARMA cannot be directly incorporated into, for example, ScaLAPACK.
>
> In fact, even if a new library is designed for CARMA, there is a complication.

5. COSMA：一个名称，各自表述

时间来到了 2019 年。在 SC19，Torsten Hoefler 教授带领他的团队大杀特杀，不仅砍下了十分之一的发表论文（8篇），还拿下了 GB 奖和 best student paper, 后者正是 COSMA [11]。COSMA 也是一个将 1D, 2D, 3D 算法统一化了的算法，实测性能比 CARMA 还要高一截，并且作者称 COSMA 在绝大多数情况下可以『贴着』并行矩阵乘法的通信复杂度理论下界，而不是差一个常倍数（『摸着』下界）。下面我们一起来看看 COSMA 是如何做到如此牛逼的。

根据论文，COSMA 里面每个进程计算一个 $a \times a \times b$ 的子问题，也就是计算一个 $a \times b$ 的 A 子矩阵乘以一个 $b \times a$ 的 B 子矩阵。通过求解一个最优化问题，找到给定可用内存大小 S 限制下的（近）最优 a, b 解，得到一个（近）最优的串行计算策略。随后，根据 red-blue pebbling game 理论，对于计算依赖有向无环图 （computational directed acyclic graph, CDAG) 进行划分，得到通信代价最小的



Rainmaker's Notebook
『求雨巫师的神奇之处在于他总是躲着不见你，却总说刚下完的雨是拜他所赐。』——《天真的人类学家》

Home

Archives

About

SiteXC

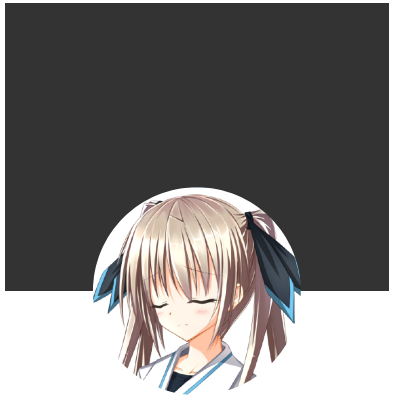
并行计算策略。这个并行计算策略的通信代价可以达到或者非常接近理论最小通信代价。完整的算法参见下图。

Algorithm 1 COSMA

Input: matrices $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$,
number of processors: p, memory size: S, computation-I/O tradeoff ratio ρ
Output: matrix $C = AB \in \mathbb{R}^{m \times n}$

$a \leftarrow FindSeqSchedule(S, m, n, k, p)$ ▷ sequential I/O optimality (§ 5)
$b \leftarrow ParallelizeSched(a, m, n, k, p)$ ▷ parallel I/O optimality (§ 6)
$(\mathcal{G}, a_{opt}, b_{opt}) \leftarrow FitRanks(m, n, k, a, b, p, \delta)$
for all $p_i \in \{1 \dots p\}$ **do in parallel**
  $(A_l, B_l, C_l) \leftarrow GetDataDecomp(A, B, \mathcal{G}, p_i)$
  $s \leftarrow \left\lfloor \frac{S - a_{opt}^2}{2a_{opt}} \right\rfloor$ ▷ latency-minimizing size of a step (6.3)
  $t \leftarrow \left\lceil \frac{b_{opt}}{s} \right\rceil$ ▷ number of steps
  for $j \in \{1 \dots t\}$ **do**
    $(A_l, B_l) \leftarrow DistrData(A_l, B_l, \mathcal{G}, j, p_i)$
    $C_l \leftarrow Multiply(A_l, B_l, j)$ ▷ compute locally
  end for
  $C \leftarrow Reduce(C_l, \mathcal{G})$ ▷ reduce the partial results
end for

a, b 求解的优化问题如下：



Rainmaker's Notebook

『求雨巫师的神奇之处在于他总是躲着不见你，却总说刚下完的雨是拜他所赐。』——《天真的人类学家》

Home

Archives

About

SiteXC

The local domain $\mathcal{D}_j$ is a grid of size $[a \times a \times b]$, containing b outer products of vectors of length a. The optimization problem of finding $\mathcal{P}_{opt}$ using the computational intensity (Lemma 2) is formulated as follows:

$$\text{maximize } \rho = \frac{a^2 b}{ab + ab + a^2} \tag{3}$$

$$\text{subject to:}$$

$$a^2 \leq S \text{ (the I/O constraint)}$$

$$a^2 b = \frac{mnk}{p} \text{ (the load balance constraint)}$$

$$pS \geq mn + mk + nk \text{ (matrices must fit into memory)}$$

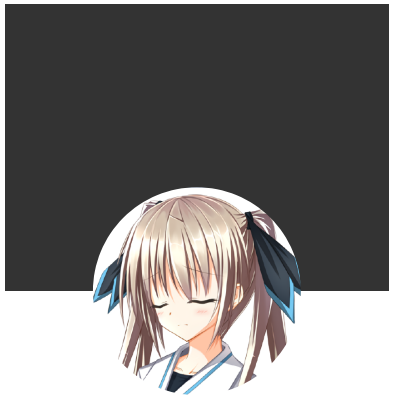
The I/O constraint $a^2 \leq S$ is binding (changes to equality) for $p \leq \frac{mnk}{S^{3/2}}$. Therefore, the solution to this problem is:

$$a = \min\left\{\sqrt{S}, \left(\frac{mnk}{p}\right)^{1/3}\right\},\ b = \max\left\{\frac{mnk}{pS}, \left(\frac{mnk}{p}\right)^{1/3}\right\} \tag{4}$$

The I/O complexity of this schedule is:

$$Q \geq \frac{a^2 b}{\rho} = \min\left\{\frac{2mnk}{p\sqrt{S}} + S, 3\left(\frac{mnk}{p}\right)^{\frac{2}{3}}\right\} \tag{5}$$

那么数据是如何排布，计算过程的通信是怎么样的呢？很不幸，论文里并没有讨论 `GetDataDecomp()` 和 `DistrData()` 这两个函数到底长什么样子。论文仅仅提到：



Rainmaker's Notebook

『求雨巫师的神奇之处在于他总是躲着不见你，却总说刚下完的雨是拜他所赐。』——《天真的人类学家》

7.2 Enhanced Communication Pattern

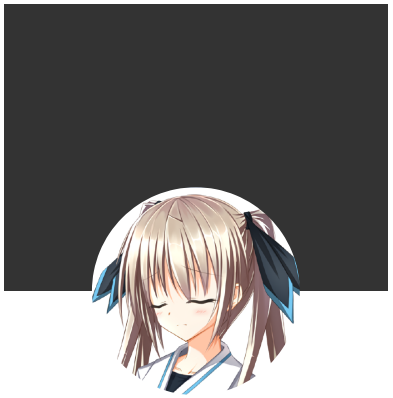
As shown in Algorithm 1, COSMA by default executes in $t = \frac{2ab}{S-a^2}$ rounds. In each round, each processor receives $s = ab/t = (S-a^2)/2$ elements of A and B. Thus, the input matrices are broadcast among the **i** and **j** dimensions of the processor grid. After the last round, the partial results of C are reduced among the **k** dimension. The communication pattern is therefore similar to ScaLAPACK or CTF.

我翻来覆去把论文看了好几次，都没有参透这背后的细节，可能是我的理论水平不够吧。好在 COSMA 的代码是开源的 [12]，我们可以直接看代码。COSMA 的代码是 C++ 写的，写得很整洁实用，可读性很好，没有用什么花里胡哨的 C++ 新语法。我们从 `miniapp/cosma_miniapp.cpp` 这个最简单的样例程序入手。

样例程序中，首先会调用 `Strategy` 类处理用户指定的并行策略。如果用户没有指定，那么 COSMA 就会计算最优的策略，这正是我们想知道的。此时，代码进入 `src/cosma/strategy.cpp:82` 的初始化函数，并且在 `src/cosma/strategy.cpp:328` 的 `square_strategy()` 函数计算并行策略。当我读完了这个函数以后，一种似曾相识的感觉浮上心头。首先，这个函数会通过 `src/cosma/math_utils.cpp` 中的 `balanced_divisors()` 穷举所有可能性计算出 `dm`，`dn`，`dk`，使得 $m/dm \approx n/dn \approx k/dk$. 随后， dm, dn, dk 被分解质因数，然后根据内存用量限制，决定每一步在 m, n, k 其中哪个方向进行并行划分或者串行划分。和 CARMA 相比，这个思路非常相似，只不过不再是机械地进行对半划分了。至于求解最优化问题，算 a, b 的最优值，以及 CDAG，到目前为止还没出现。

求得最优策略以后，样例程序就创建了三个分布式矩阵 A, B, C（这里我们先不管矩阵具体是怎么分布的），然后调用 `multiply()` 开始计算矩阵乘法。计算的实际入口在 `src/cosma/multiply.cpp:308`，这个 `multiply()` 会根据输入的策略调用 `parallel()` 或者 `sequantial()`。其中， `parallel()` 这个函数写了大量的详细注释。读完注释和代码，只能说这个函数在做的东西和 CARMA 不能说是一母同生，起码也是心有灵犀。下面摘录一点来自 `parallel()` 顶部的注释：

> If m split: Split matrix A and copy matrix B ...
>
> If n split: Split matrix B and copy matrix A ...
>
> If k split: Split both matrices A and B ...



Rainmaker's Notebook
『求雨巫师的神奇之处在于他总是躲着不见你，却总说刚下完的雨是拜他所赐。』——《天真的人类学家》

Home

Archives

About

SiteXC

作为对比，我们摘录一些 CARMA 论文里描述 CARMA 具体算法的 Algorithm 2 的句子：

> if m is the largest dimension then copy B to disjoint halves of the processors
>
> if n is the largest dimension then copy A to disjoint halves of the processors
>
> if k is the largest dimension then gather C from disjoint halves of the processors

只能说，COSMA 这个代码的实现是『以现代理念重新创作的官方精神续作』，解决了 CARMA 最大的痛点对半划分。在实际操作上，COSMA copy A/B 的时候用的是 allgather，算完 C 的部分结果以后用 reduce-scatter 得到最终的 C 结果并且均匀分布到各个进程上。当然，A 和 B 的初始分布依然是不那么直观，不过要算还是可以算清楚的。

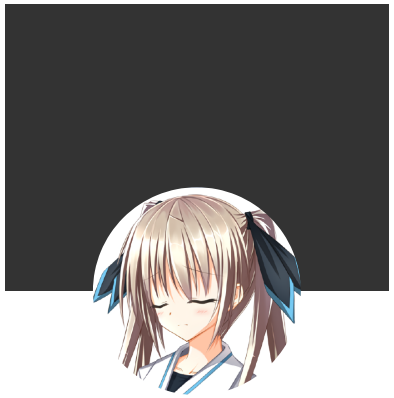
可怜的 red-blue pebbling game，仅仅在论文里面闪耀出场了。还是不得不佩服作者，竟然可以在理论和代码各走一条道的情况下，两边都达到了最优通信代价，真是条条大路通罗马啊。

6. 番外：3D SUMMA

3D SUMMA [13] 本来不在本文的讨论范围内，但是后来经 STA 鞭策，觉得这个算法家族还是值得拎出来写一下。3D SUMMA 的论文是 2016 年出来的，比 COSMA 要早，然而可能由于题目不显眼，COSMA 并没有引用此文，也没有讨论 3D SUMMA.

2D SUMMA 的框架其实很容易将其变成 3D 算法：原来是切分 k-dim 以后若干个 low-rank update (LRU) 依次执行，现在只要把这些 LRU 并行执行就可以得到一个 3D SUMMA 了。不过作者的思路显然要更广一点：对原来的矩阵乘法先做一次一维切分，切成若干个子矩阵相乘，这是第一层并行；每个子矩阵相乘再调用 2D SUMMA，这是底下的两层并行。加起来，一共是三层并行，就是 3D 算法了。因此，3D SUMMA 有三个版本：stationary A/B/C 版，分别对应第一层在 n-dim, m-dim, k-dim 进行切分。不过在我看来，stationary A/B 本质上并不是 3D 算法，因为整体上仍然只对 m 和 n 两个维度进行了切分，没有对 k 维度进行切分。尽管如此，3D SUMMA 的优点还是很明显的：两层结构清晰易懂，可以复用 2D SUMMA 方便实现。

在我看来，3D SUMMA 的问题起码有两个。第一个是：三个版本哪个是最好的，取决于实际的问题尺寸和 3D 进程网格的大小。论文里并没有提到如何针对不同问题尺寸选择进程网格，仅仅讨论了两个正方形矩阵相乘时的通信复杂度和此时的最佳网格大小。也就是说，并没有一个 3D SUMMA 算法适合所有尺寸的问题。在这一点上，显然 COSMA 要做得更完善一点。还有一个问题在于，即使是算两个正方形矩阵相乘，使用最佳的 3D 进程网格，3D SUMMA 的 latency cost 会比 COSMA 大。原因在于 2D SUMMA 的 broadcast 是串行的，而每一次都是 $\log(p_m) + \log(p_n)$ 的 latency cost.



Rainmaker's Notebook

『求雨巫师的神奇之处在于他总是躲着不见你，却总说刚下完的雨是拜他所赐。』——《天真的人类学家》

Home

Archives

About

SiteXC

7. 番外的番外：世间安得双全法

那么问题来了：有没有一个算法，既有 3D SUMMA 框架的简洁性，又可以对所有问题尺寸自动选择合适的 3D 进程网格大小，还能有 COSMA 一样的最优通信代价？

世间安得双全法，不过允许一些瑕疵存在的话，满足上面三个要求的算法确实有了。这个问题留作习题，请读者诸君自己寻找答案吧！

致谢

笔者在此特别感谢 STA 的多次催更。没有他的 buff 加持，本文可能不会诞生。祝 STA 同志从此单抽出金，十连满命！

参考文献

1. A Cellular Computer to Implement the Kalman Filter Algorithm, link
2. PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers, link
3. SUMMA: Scalable Universal Matrix Multiplication Algorithm, link
4. I/O Complexity: The Red-Blue Pebble Game, link
5. Communication Lower Bounds for Distributed-Memory Matrix Multiplication, link
6. Minimizing Communication in Numerical Linear Algebra, link
7. Communication Lower Bounds and Optimal Algorithms for Numerical Linear Algebra, link
8. A three-dimensional approach to parallel matrix multiplication, link
9. Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms, link
10. Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication, link
11. Red-blue Pebbling Revisited: Near Optimal Parallel Matrix-Matrix Multiplication, link
12. GitHub: eth-cscs/COSMA, link
13. Parallel Matrix Multiplication: A Systematic Journey, link

发布于 2022-04-02

tags: { LinearAlgebra } { MPI } { GEMM }