大型矩阵乘法的 FOX 算法 MPI 流水并行
--并行应用实例

工学院　洪瑶　1801111621

目录

一、应用问题描述和应用价值分析

1.1 问题背景

稠密线性代数运算对模式识别和生物信息学等领域有着许多实际应用都十分重要，而矩阵的乘法就是十分重要的一项运算，矩阵乘法在数值预报中就必须要用到，因此研究在大型数值预报时就要用到大型矩阵乘法。

我们知道大型矩阵乘法的计算量十分庞大，如果没有任何优化，两个 N*N 的矩阵相乘的标量运算复杂性为 $O(N^3)$。20 世纪 60 年代末期，strassen 提出了一个新的快速算法，其标量运算只需要 $O(N^{\log_2 7})$。但是当矩阵规模十分大时，对计算器而言仍然是个十分巨大的挑战，因此考虑采用分布式并行计算是十分必要的。

1.2 数学描述

设有 M*K 个数据 a_{ij} 的二阶矩阵 ***A***，有 K*N 个数据的 b_{ij} 二阶矩阵 ***B***，计算矩阵 ***C:***

$$\boldsymbol{C=AB}$$

其中 c_{ij} 计算原则为:

$$c_{ij}=\sum_{k=0}^{K-1}a_{ik}b_{kj};\quad (i=0,N-1),\quad (j=0,M-1)$$

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

$\times$

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

$=$

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

二、应用问题的计算特征分析、影响数据处理效率的关键因素

2.1 计算特征分析

该问题最大的特征就是每个元素 C_{ij} 都需要 A 矩阵的第 i 行和 B 矩阵第 j 列的数据，因此若考虑并行计算数据划分是最大的困难，尤其对于大型稠密矩阵而言，全部数据存储在每个进程中将导致储存空间不足的问题，并起不到并行的目的。

2.2 影响数据处理效率的关键因素

本问题的数据依赖性特别高，因此在进行计算和规约时，如果不行合理的算法，将导致 R-R 冲突和 R-W 冲突大大降低计算效率，因此必须设计合理的流水并行算法，减少数据的 R-R 冲突。

三、任务分解方案、数据访问冲突分析

3.1 任务分解方案

利用高等代数的分块矩阵的方法，将 A、B、C 矩阵分别进行分块划分，划分方案如下（这里尤其要注意分块子矩阵的下标界限）：

$$A=\begin{pmatrix}\begin{pmatrix}a_{11} & a_{12} & \cdot & a_{1,\frac{K}{q}}\\ a_{21} & a_{22} & \cdot & \cdot\\ \cdot & \cdot & \cdot & \cdot\\ \cdot & \cdot & \cdot & \cdot\\ a_{\frac{M}{q},1} & \cdot & \cdot & a_{\frac{M}{q},\frac{K}{q}}\end{pmatrix}_{A_{11}} & & A_{12} & \cdot & A_{1q}\\ & & & & & \\ & A_{21} & & \cdot & \cdot & \cdot\\ & \cdot & & \cdot & \cdot & \cdot\\ & A_{q1} & & \cdot & \cdot & A_{qq}\end{pmatrix}_{MK}$$

$$B=\begin{pmatrix}\begin{pmatrix}b_{11} & b_{12} & \cdot & b_{1,\frac{N}{q}}\\ b_{21} & b_{22} & \cdot & \cdot\\ \cdot & \cdot & \cdot & \cdot\\ \cdot & \cdot & \cdot & \cdot\\ b_{\frac{K}{q},1} & \cdot & \cdot & b_{\frac{K}{q},\frac{N}{q}}\end{pmatrix}_{B_{11}} & & B_{12} & \cdot & B_{1q}\\ & & & & & \\ & B_{21} & & \cdot & \cdot & \cdot\\ & \cdot & & \cdot & \cdot & \cdot\\ & B_{q1} & & \cdot & \cdot & B_{qq}\end{pmatrix}_{KN}$$

$$C = \begin{pmatrix} \begin{pmatrix} c_{11} & c_{12} & \cdot & c_{1,\frac{N}{q}} \\ c_{21} & c_{22} & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ c_{\frac{M}{q},1} & \cdot & \cdot & c_{\frac{M}{q},\frac{N}{q}} \end{pmatrix}_{C_{11}} & & C_{12} & \cdot & C_{1q} \\ & C_{21} & & \cdot & \cdot & \cdot \\ & \cdot & & \cdot & \cdot & \cdot \\ & C_{q1} & & \cdot & \cdot & C_{qq} \end{pmatrix}_{MN}$$

将矩阵进行划分以后，可以按照高等代数里的知识：

$$\begin{aligned} C_{ij} &= \sum_{k=0}^{q-1} A_{ik} B_{kj} \\ &= A_{i0} B_{0j} + A_{i1} B_{1j} + \ldots + A_{ii-1} B_{i-1j} \\ &+ A_{ii} B_{ij} \\ &+ A_{i,i+1} B_{i+1j} + \ldots + A_{i,q-1} B_{q-1,j} \end{aligned}$$

注意，此时的 C_{ij} 和 c_{ij} 的区别，$A_{ij}/B_{ij}/C_{ij}$ 分别 A/B/C 的子矩阵。

到这里，可以根据 C 矩阵的划分原则，将其进行并行化。

3.2 数据访问冲突分析

如下图所示，在进行第一次的计算时，如果按行优先的计算原则，那么 A00 必然会 R-R 冲突，而 B 矩阵的读取不会发生冲突，因此规避 A 子矩阵的冲突时必然要做的事情，于是可以考虑用行通信子将 A_{00} 广播给每一行的 C_{0j} 即可。

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

×

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

=

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

四、目标并行计算机体系结构概述

4.1 硬件资源

【未名一号】

节点配置

节点类别	型号	主要规格	节点数
CPU 节点	NeXtScale nx360 M5	2*Intel Xeon E5-2697A V4，128G	140
		2*Intel Xeon E5-2697A V4，256G	51
		2*Intel Xeon E5-2697A V4，512G	5
GPU 节点	X3650 M5	2*Intel Xeon E5-2643 V4，256G，2* NVIDIA Tesla P100	10
KNL 节点	KNL	Intel Xeon Phi7250，192G	8
四路胖节点	X3850 X6	4 * Intel Xeno E7-8867V4，512G	1
八路胖节点	X3950 X6	8 * Intel Xeno E7-8867V4，4096G	2
合计		每个节点均配备 240GB 到 480GB SSD 硬盘，双口千兆，Intel OPA Single-port PCIe 3.0 x16 HFA	227
登录节点	X3650 M5	2*Intel Xeon E5-2690A V4，128G	2
管理节点	X3650 M5	2*Intel Xeon E5-2690A V4 ， 256G	4
存储节点	X3650 M5	2*Intel Xeon E5-2690A V4 ， 256G	4

节点性能

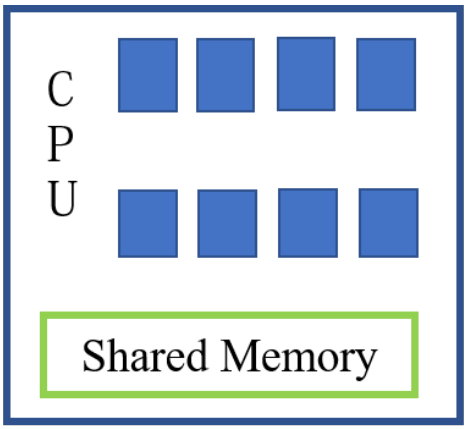
节点类别	单核主频	单节点核心数	单周期指令执行数	节点数	理论峰值/GFlops
CPU 节点	2.6GHz	32	16	196	260915.2
GPU/CPU	3.4GHz	12	16	10	6528
GPU/GPU	1328MHz	1792	NA	10	94000
KNL 节点	1.4GHz	68	32	8	24371.2
四路胖节点	2.4GHz	72	16	1	2764.8
八路胖节点	2.4GHz	144	16	2	11059.2
登录/管理节点	2.6GHz	28	16	6	6988.8
存储节点	2.6GHz	28	16	4	4659.2
合计				237	411286.4
计算节点：227 个，管理节点：10 个、CPU 核心：7032 个、KNL 核心：544 个、GPU 卡：20 块					

存储配置

型号	主要规格	套数
联想 GSS24	2*X3650M5 4*58*6TB NL SAS， 1392TB 4*2*200GB SSD， 1600GB	2
存储容量合计：2784.8 TB，聚合读带宽： 58GB/s，聚合写带宽：38GB/s		

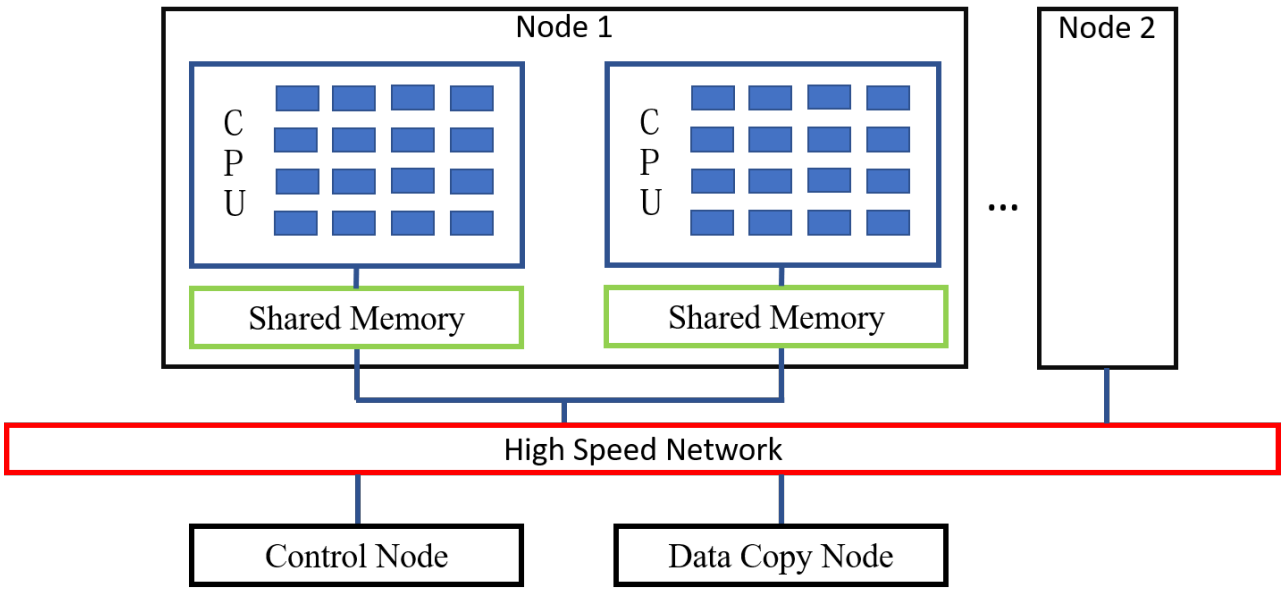
4. 2 计算机结构

4.2.1 单计算机 NUMA 结构

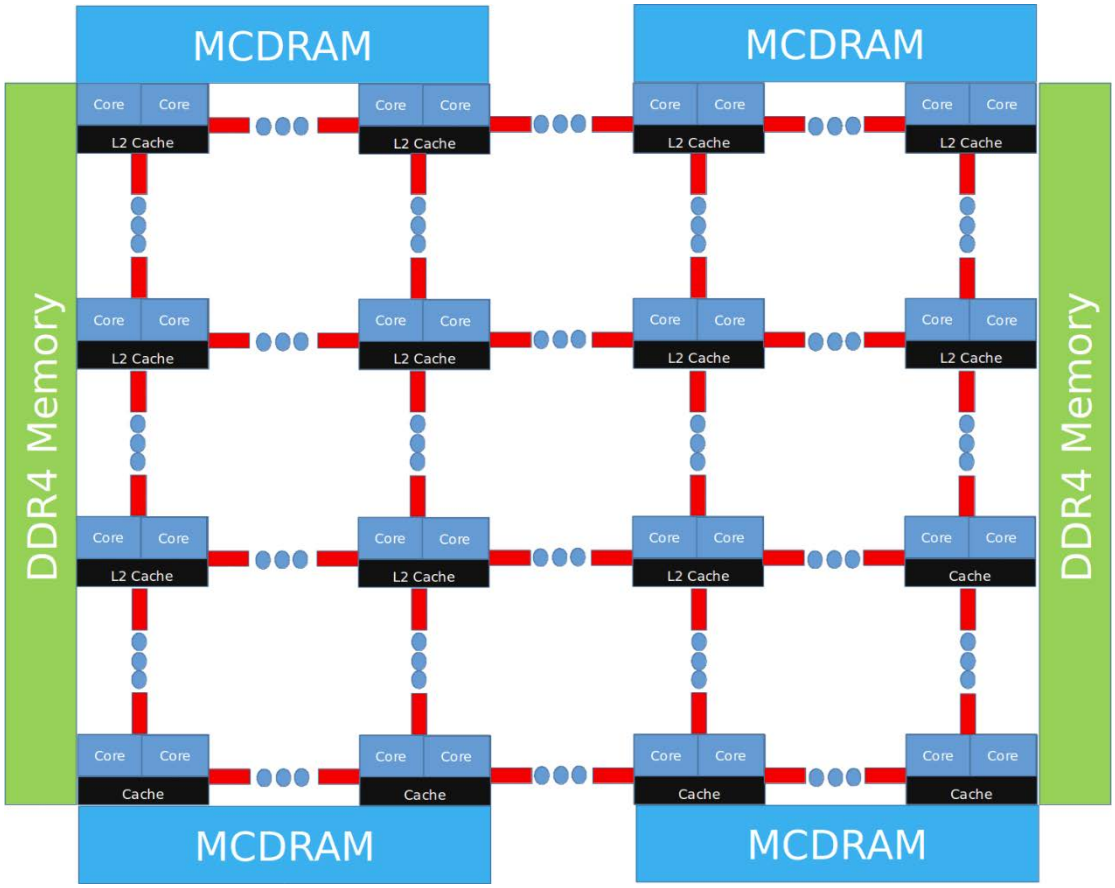


Intel® Core™ i7-6700K Processor
(8M Cache, up to 4.20 GHz)

4.2.2 未名一号集群 cluster 结构：



4.2.2 骑士降临第二代英特尔 XEON PHI 产品结构：



物理配置图：


Xeon Phi™ Processor
Xeon Phi™ Processor
with integrated
Intel® Omni-Path Fabric

五、计算划分方案、负载均衡分析、同步开销分析

5.1 计算划分方案

进程的资源分配及算法概述：

(1)矩阵 A 是 M×K 矩阵，矩阵 B 是 K×N 矩阵。

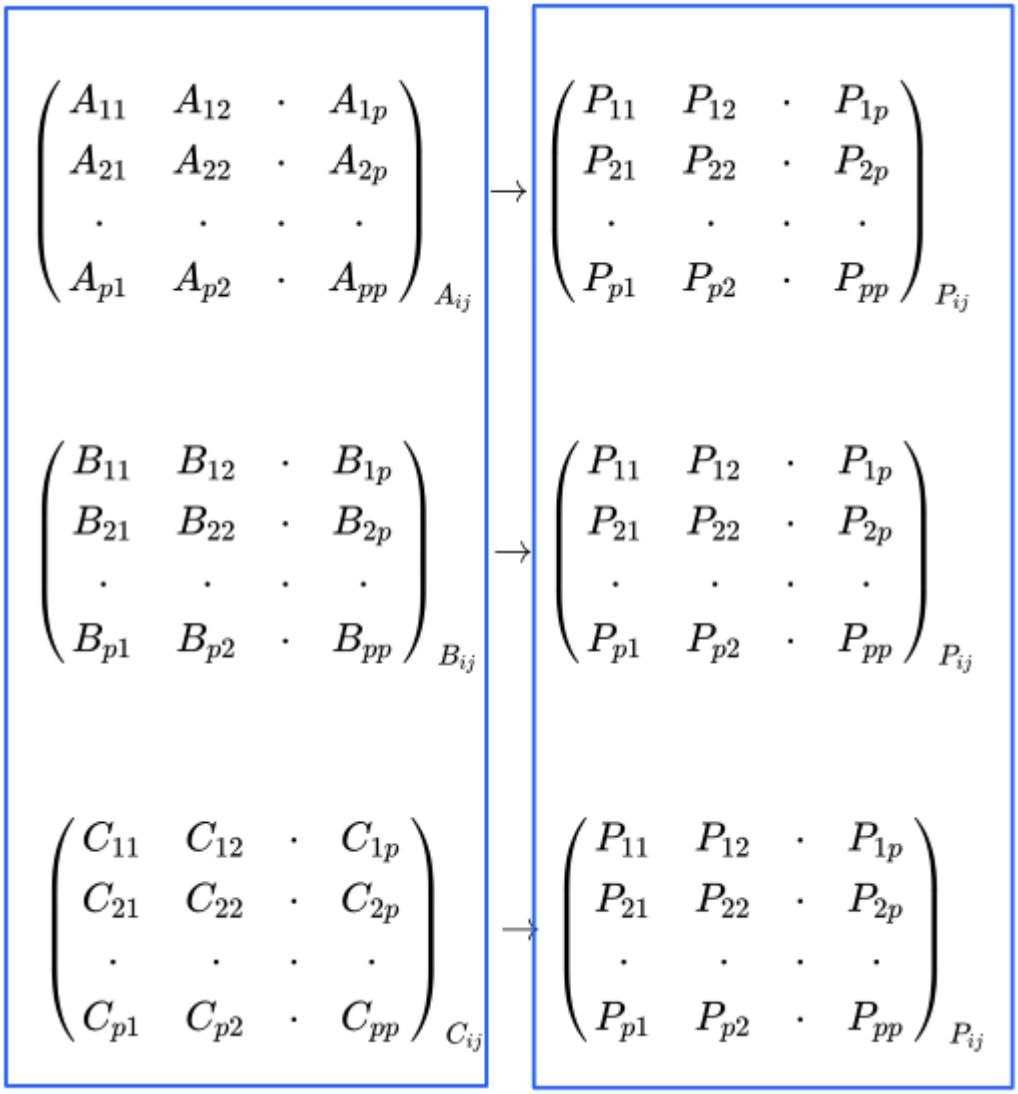
(2)并行计算矩阵 $\boldsymbol{C}=\boldsymbol{AB}$。

(3)设 p=num(处理器)是处理器的数目，q=$\sqrt{p}$是整数，使得它除以 M 和 N。

(4)建立具有网格 Pi j 的笛卡尔拓扑进程，i=0~q-1，j=0~q-1。

(5)定义 $\widehat{M}=\frac{M}{q},\ \widehat{K}=\frac{K}{q},\ \widehat{N}=\frac{N}{q}$。

(6)在 p 进程上按块分配 A 和 B，使得 A_{ij} 是 M×K 块，B_{ij} 是 K×N 块，存储在进程 P_{ij} 上。



Maping

上图为分块矩阵数据最初存储在进程中匹配情况，之后的数据交换仅仅为数据块的交换，对数据进行传输时，以打包的方式进行传输。

(7) 使得 Aij 是 M×K 块，Bij 是 K×N 块，存储//在进程 Pij 上,进行一次 FOX kernel 的向量乘法计算，既在 P_{ij} 计算 $A_{ik}B_{kj}$，其中 k 代表第 k 个超级计算步，每个计算步最后都得进行一次规约：

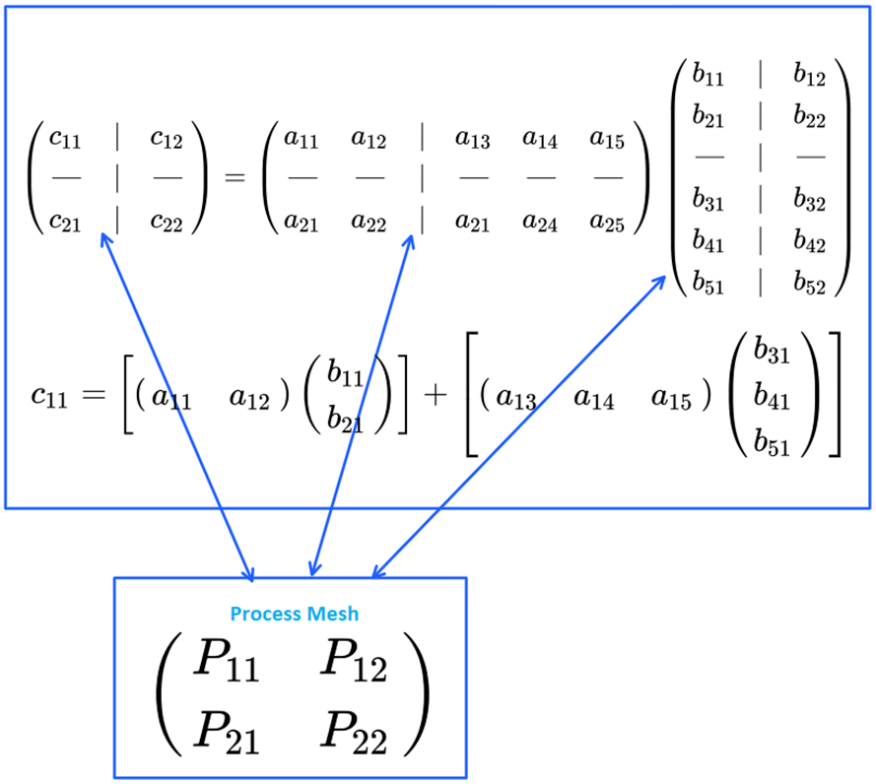
Super Step	Mathematical Operation
0	$\boldsymbol{C_{ij} = A_{ii}B_{ij}}$
1	$\boldsymbol{C_{ij} = C_{ij} + A_{ii+1}B_{i+1j}}$
2	$\boldsymbol{C_{ij} = C_{ij} + A_{ii+2}B_{i+2j}}$
…	**…**
q-2-i	$\boldsymbol{C_{ij} = C_{ij} + A_{iq-2}B_{q-2j}}$
q-1-i	$\boldsymbol{C_{ij} = C_{ij} + A_{iq-1}B_{q-1j}}$
…	$\boldsymbol{C_{ij} = C_{ij} + A_{i1}B_{1j}}$
…	$\boldsymbol{C_{ij} = C_{ij} + A_{i2}B_{2j}}$
…	**…**
q-1	$\boldsymbol{C_{ij} = C_{ij} + A_{ii-1}B_{\mathrm{i-1}j}}$

5.2 计算负载均衡性分析

由于我们把行列都设为能被进程数的数量的开根号（既 q）整除，于是在计算步的层面上，本问题不存在负载不均衡，也就意味着每一次同步，所有处理器上都进行了一次 FOX kernel 运算并进行了规约。

但是从每个超级步之中的不同进程计算量来看，k 的值没有规定能被 q 整除，因此其将导致负载的不均衡，如下图所示，当一个最简单的负载不均衡存在时，不同处理器的计算量不一样，将导致同步开

销的增加。



5.3 同步开销分析

同步开销体现在一个超级步上的时间不均衡性上面，这部分不均衡分为两个部分：

（1）从 5.2 中知道，由于负载不均衡，负载最重的处理器显然所需要的时间更长，同步的等待时间开销一部分来自这里。

（2）另一部分同步开销来自数据的通信导致，由于**拥有者原则**的存在，一部分进程不需要进行数据通信，直接进行计算，而另一部分进程则需要先通信数据再计算，这部分时间不均衡也导致同步开销增加。

六、实现设计

超级步中实现如下具体步骤：

Stage	Algorithm Operation
0	1. 进程 P_{ij} 拥有 A_{ij},和B_{ij} ， 但是仍需要 A_{ii} (对每个 i) 2. 进程 P_{ii} $boardcast$ A_{ii} 到进程网格的第 i 行的所有进程中 *3.* 进程 P_{ij}计算 $C_{ij} = A_{ii}B_{ij}$
1	*1.* P_{ij} 拥有 A_{ij} 和 B_{ij} ， 但是需要 A_{ii+1} 和 B_{i+1j} 1.1 将B_{ij}中的数据统一上翻一行 (第 0行 转到第$q-1$行) 1.2 P_{ii+1} broadcast A_{ii+1} 到第 i 行的所有进程 2. 进程 P_{ij} 计算 $C_{ij} = C_{ij} + A_{ii+1}B_{i+1j}$
2	*1.* P_{ij} 拥有 A_{ij} 和 B_{ij} ， 但是需要 A_{ii+2} and B_{i+2j} 1.1 将B_{ij}中的数据统一上翻一行 (第 **0**行 转到第$\boldsymbol{q}-\mathbf{1}$行) 1.2 P_{ii+2} broadcast A_{ii+2}到第 i 行的所有进程 2. 进程 P_{ij} 计算 $C_{ij} = C_{ij} + A_{ii+2}B_{i+2j}$
…	…
q-2-i	$C_{ij} = C_{ij} + A_{iq-2}B_{q-2j}$
q-1-i	$C_{ij} = C_{ij} + A_{iq-1}B_{q-1j}$
…	$C_{ij} = C_{ij} + A_{i1}B_{1j}$
…	$C_{ij} = C_{ij} + A_{i2}B_{2j}$
…	…
q-1	$C_{ij} = C_{ij} + A_{ii-1}B_{\mathrm{i}-1j}$

以处理器数量为 16 为例，可视化上述步骤如下：

在通过传播之后（相当于点乘了！）

七、实验评估与实验结果分析

7.1 计算量与通信量分析

Stage	计算量和通信量分析
0	$\boldsymbol{Broadcast}$操作的通信量: $$(q-1)\times q\times\frac{M}{q}\times\frac{K}{q}$$ 每个进程的计算量: $$\frac{M}{q}\times\frac{K}{q}\times\frac{N}{q}$$ 全部计算量: $$\left(\frac{M}{q}\times\frac{K}{q}\times\frac{N}{q}\right)\times q\times q$$
1	S$\boldsymbol{hift}$操作的通信量: $$(q\times q)\times(\frac{K}{q}\times\frac{N}{q})$$ $\boldsymbol{Broadcast}$操作的通信量: $$[(q-1)\times q]\times(\frac{M}{q}\times\frac{K}{q})$$ 全部通信量: $$(q\times q)\times\left(\frac{K}{q}\times\frac{N}{q}\right)+[(q-1)\times q]\times(\frac{M}{q}\times\frac{K}{q})$$ 每个进程的计算量: $$\frac{M}{q}\times\frac{K}{q}\times\frac{N}{q}$$ 全部计算量: $$\left(\frac{M}{q}\times\frac{K}{q}\times\frac{N}{q}\right)\times q\times q$$
2	S$\boldsymbol{hift}$操作的通信量: $$(q\times q)\times(\frac{K}{q}\times\frac{N}{q})$$

	$\boldsymbol{Broadcast}$操作的通信量: $$[(\boldsymbol{q}-\mathbf{1})\times\ \boldsymbol{q}]\times(\frac{\boldsymbol{M}}{\boldsymbol{q}}\times\frac{\boldsymbol{K}}{\boldsymbol{q}})$$ 全部通信量: $$[(\boldsymbol{q}-\mathbf{1})\times\ \boldsymbol{q}]\times(\frac{\boldsymbol{M}}{\boldsymbol{q}}\times\frac{\boldsymbol{K}}{\boldsymbol{q}})$$ 每个进程的计算量: $$\frac{\boldsymbol{M}}{\boldsymbol{q}}\times\frac{\boldsymbol{K}}{\boldsymbol{q}}\times\frac{\boldsymbol{N}}{\boldsymbol{q}}$$ 全部计算量: $$\left(\frac{\boldsymbol{M}}{\boldsymbol{q}}\times\frac{\boldsymbol{K}}{\boldsymbol{q}}\times\frac{\boldsymbol{N}}{\boldsymbol{q}}\right)\times\ \boldsymbol{q}\ \times\ \boldsymbol{q}$$
…	
q-2-i	
q-1-i	
…	
…	
…	
q-1	

总通信量:

$$\begin{aligned} Comm &= \left[(q-1)q\times\frac{M\times K}{q^2}\times q\right] \\ &+\left[(q\times q)\times\frac{K\times N}{q^2}\times(q-1)\right] \\ &=(M\times K+K\times N)\times(q-1) \end{aligned}$$

总计算量:

$$\begin{aligned} Comput &= \left[\left(\frac{M}{q}\times\frac{K}{q}\times\frac{N}{q}\right)\times(q\times q)\right]\times q \\ &=M\times K\times N \end{aligned}$$

7.2 实验结果分析

7.2.1 实验结果验证

通过打印矩阵，选择 2*2 个规模大小的进程进行代码验算，得到如下结果：

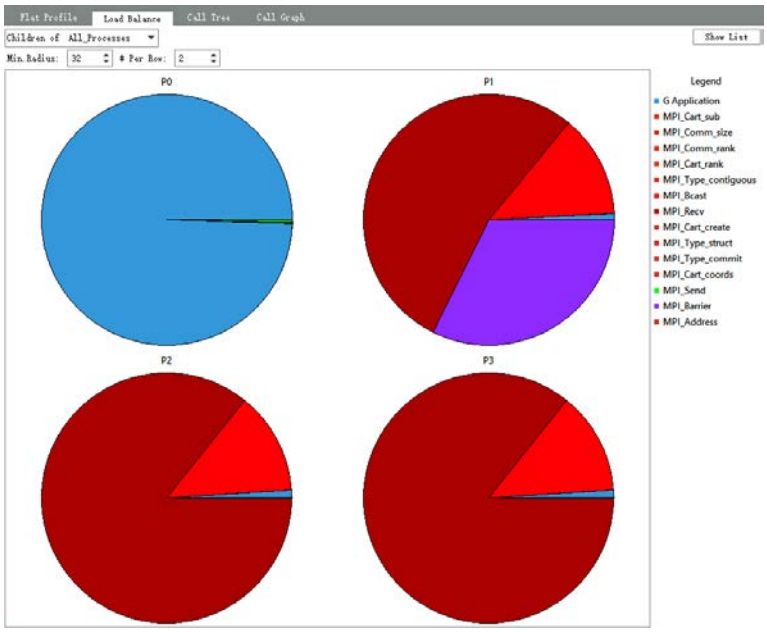
```
A =
 1.0   0.0   0.0   0.0
 0.0   1.0   0.0   0.0
 0.0   0.0   1.0   0.0
 0.0   0.0   0.0   1.0
B =
 1.0   2.0   5.0   6.0
 3.0   4.0   7.0   8.0
 9.0  10.0  13.0  14.0
11.0  12.0  15.0  16.0
Split of local matrix A
Process 0 > grid_row = 0, grid_col = 0
 1.0   0.0
 0.0   1.0
Process 1 > grid_row = 0, grid_col = 1
 0.0   0.0
 0.0   0.0
Process 2 > grid_row = 1, grid_col = 0
 0.0   0.0
 0.0   0.0
Process 3 > grid_row = 1, grid_col = 1
 1.0   0.0
 0.0   1.0
Split of local matrix B
Process 0 > grid_row = 0, grid_col = 0
 1.0   2.0
 3.0   4.0
Process 1 > grid_row = 0, grid_col = 1
 5.0   6.0
 7.0   8.0
Process 2 > grid_row = 1, grid_col = 0
 9.0  10.0
11.0  12.0
Process 3 > grid_row = 1, grid_col = 1
13.0  14.0
```

```
15.0 16.0
Split of local matrix C
Process 0 > grid_row = 0, grid_col = 0
 1.0   2.0
 3.0   4.0
Process 1 > grid_row = 0, grid_col = 1
 5.0   6.0
 7.0   8.0
Process 2 > grid_row = 1, grid_col = 0
 9.0 10.0
11.0 12.0
Process 3 > grid_row = 1, grid_col = 1
13.0 14.0
15.0 16.0
The product is
 1.0   2.0   5.0   6.0
 3.0   4.0   7.0   8.0
 9.0 10.0 13.0 14.0
11.0 12.0 15.0 16.0
```

可见结果正确，程序为正确。

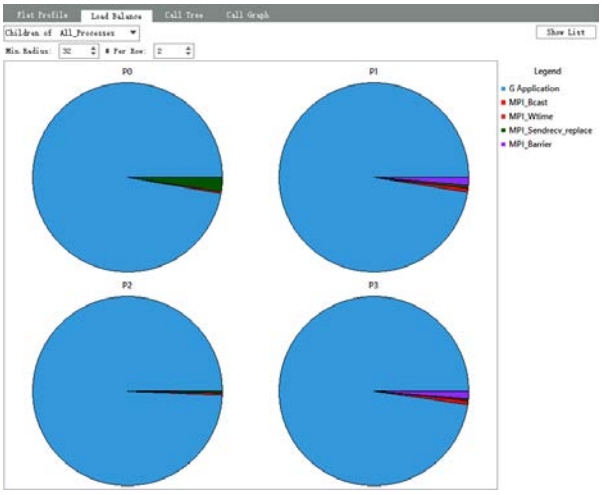
7.2.2 进程追踪分析

利用 intel trace analyzer 可以分析整个程序并行的时间线。

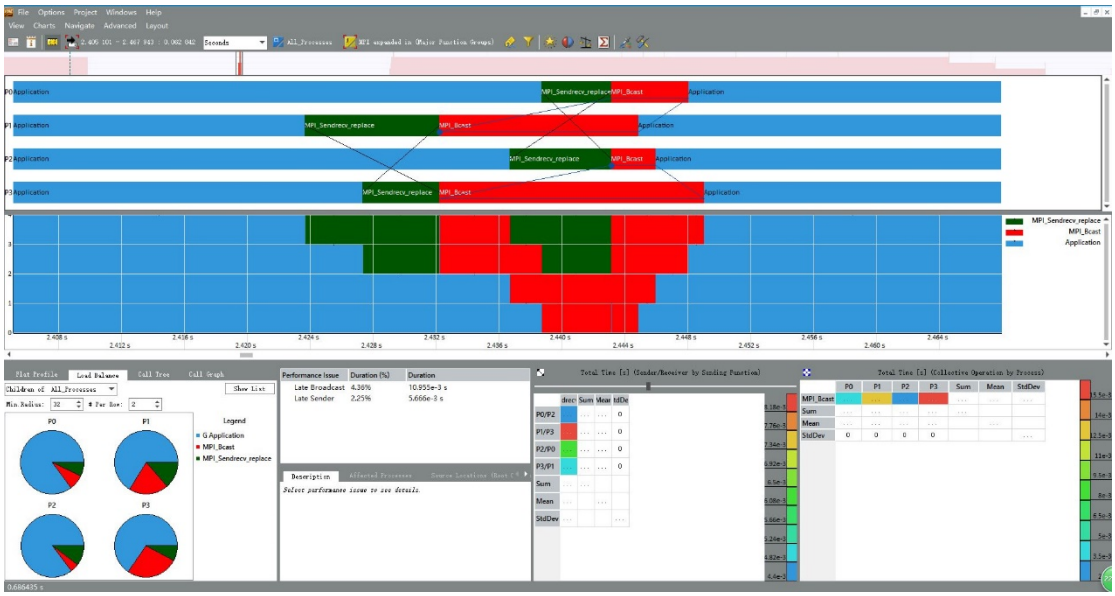


上图是整体程序的运算时间占比图，可以看到其实实际上程序矩阵的读入读出的时间还是占比很大的，这为我们以后的程序优化提供

了思路。



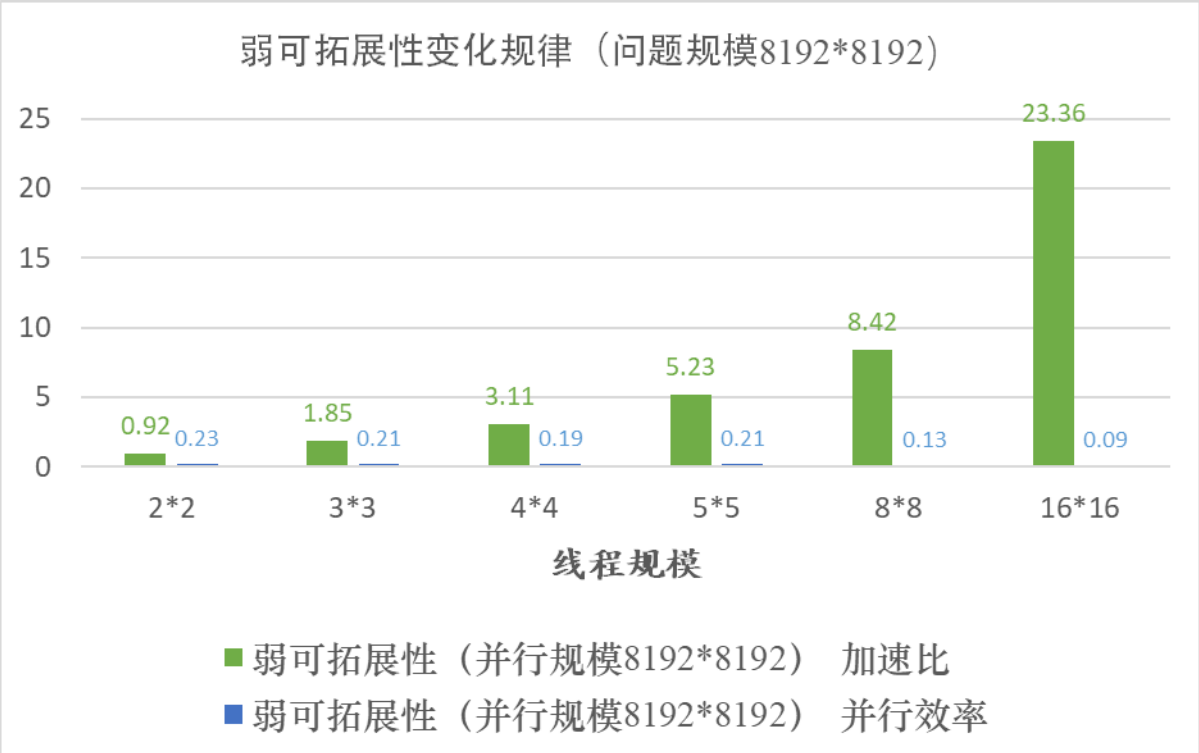
上图是在 fox kenel 函数中，计算时间与通信时间的占比（线程为 2*2，问题规模为 4096*4096），可以看见每个处理器的负载都比较均衡，这与我们之前分析的比较一致，因为我们所取的矩阵为方针。



上图为矩阵进行行通信子和列通信子进行通信时各线程之间通信的情况图，绿色为 MPI_sendreiv_replace 进行 B 矩阵行交换的通信时间，红色为 A 矩阵进行通信时 MPI_Bcast 所占用的时间，可惜看到通信情况良好，为阻塞式通信，没有发生越级。

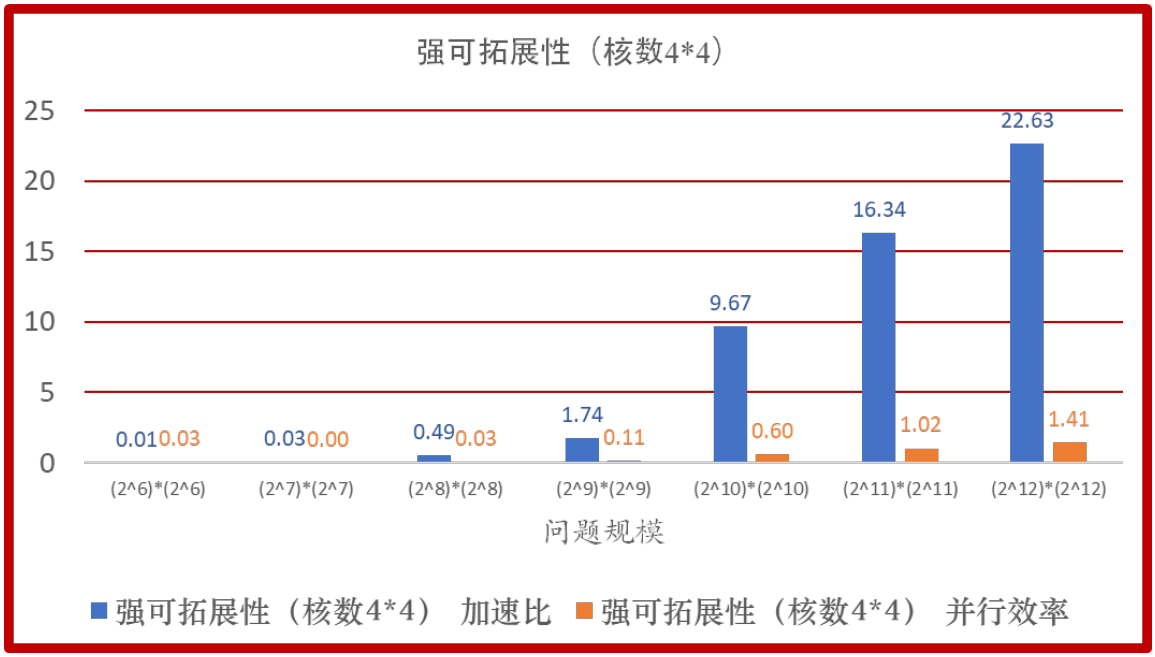
7.2.3 未名一号 Xeon E5 cluster

弱可拓展性分析（矩阵规模 8192*8192）



分析：从图中可以看出在线程规模较小时，加速比较小，甚至小于 1，这是由于线程规模较小时通信耗时太长，同步开销太大，导致加速比小于 1.但是随着线程规模的增加，不同进程之间的计算量明显减小，因此加速比越来愈大，但是我们可以看到，并行效率是越来越低的，这是因为线程数量越多，通信量越多，单个进程的计算率降低导致的。

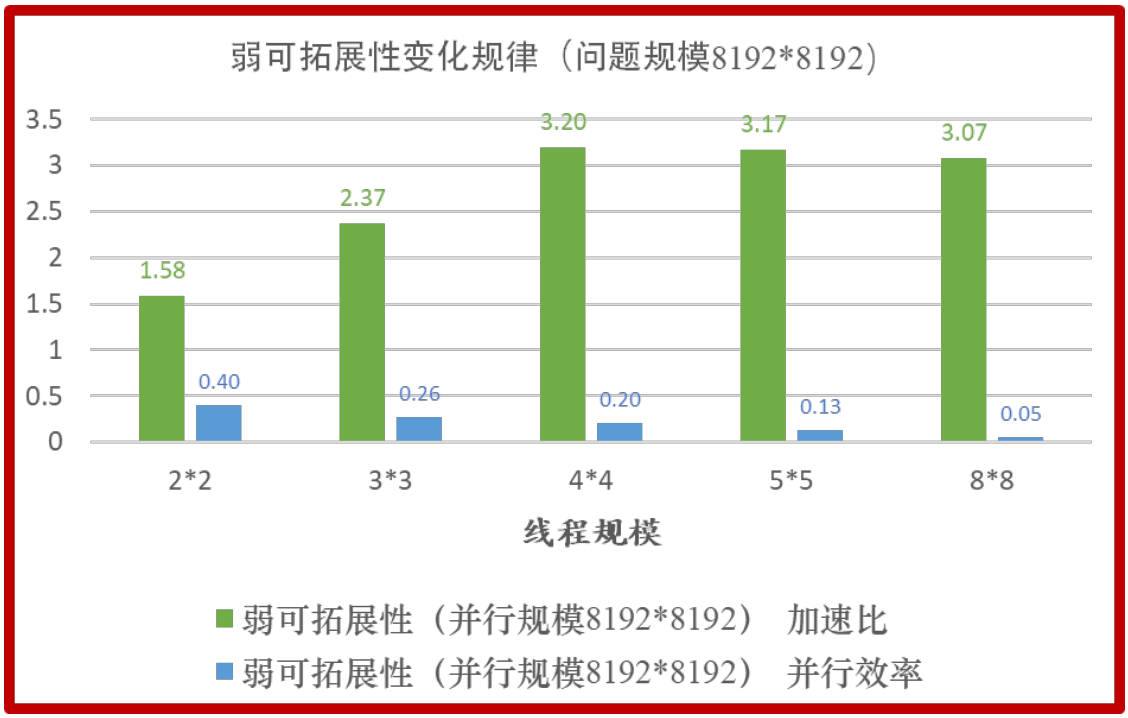
强可拓展性分析（矩阵规模 8192*8192）



分析：从图中可以看出并行加速比在问题规模较小时极小，接近于 0，但随着问题规模的增大，加速比开始逐渐增大，这是因为在数据规模较小时，通信占比较多。而之后出现的并行效率大于 1 是因为采用了 OPENMP 进行进程中的线程并行，加速了其计算。

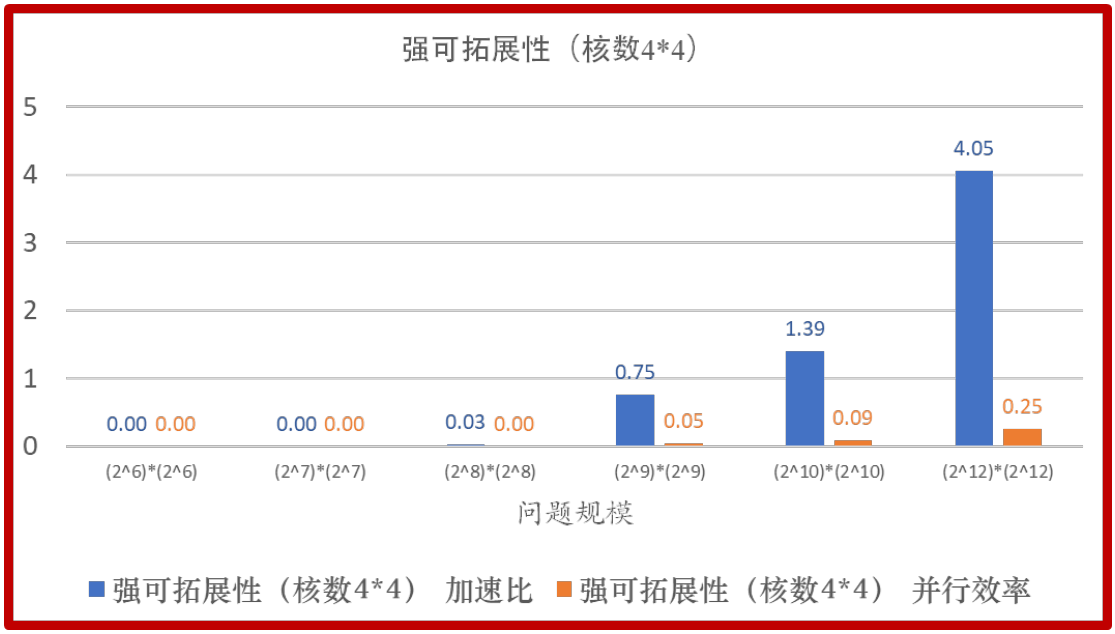
7.2.4 未名一号 **Xeon PHI**

弱可拓展性分析（矩阵规模 8192*8192）



分析：从图中可以看出在线程规模的改变对加速比的影响并不是很明显，但是并行效率随着线程规模的增家，并行效率减小，由于至强 phi 的单核计算效率并不高，导致其计算的加速比不会稳固的增长。

强可拓展性分析（矩阵规模 8192*8192）



分析：从图中可以看出并行加速比在问题规模较小时极小，接近于 0，但随着问题规模的增大，加速比开始逐渐增大，这是因为在数据规模较小时，通信占比较多。而之后出现的并行效率大于 1 是因为采用了 OPENMP 进行进程中的线程并行，加速了其计算。

附录程序：

```
/* fox_floats_timer_caching_omp.c -- uses Fox's
algorithm to multiply two square matrices
 *
 * Input:
 *     n: global order of matrices
 *     A,B: the factor matrices
 * Output:
 *     C: the product matrix
 *
 * Notes:
 *     1.  Assumes the number of processes is a perfect
square
 *     2.  The array member of the matrices is
statically allocated
 *
 * See Chap 7, pp. 113 & ff and pp. 125 & ff in PPMPI
 */

/* Compiler command:
* mpiicc -O3 -qopenmp -xMIC-AVX512 -qopt-report-
phase=vec -qopt-report=3 fox_floats_timer_caching_omp.c
```

```
-o fox_floats_timer_caching_omp
*
* Run command:
* mpirun -n -4 ./fox_floats_timer_caching_omp
*/


/* Head files */
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <mpi.h>
#include <omp.h>

// define float precision, 4 byte single-precision
float or 8 byte double-precision float
#define FLOAT double
#define FLOAT_MPI MPI_DOUBLE


// Define threads speed-up affnity in the computing
// 8 for Intel Core i7-6700K
```

```
// 134 for Intel Xeon Phi 7250 (68-1)*2 Hardware
Threads
#define NUM_THREADS 2
// Define threads affinity "scatter" or "compact"
#define AFFINITY "KMP_AFFINITY = compact"


/* Type define structure of process grid */
typedef struct {
    int        p;              /* Total number of
processes    */
    MPI_Comm   comm;           /* Communicator for entire
grid */
    MPI_Comm   row_comm;       /* Communicator for my row
*/
    MPI_Comm   col_comm;       /* Communicator for my col
*/
    int        q;              /* Order of grid
*/
    int        my_row;         /* My row number
*/
    int        my_col;         /* My column number
```

```
*/
    int        my_rank;        /* My rank in the grid
comm     */
} GRID_INFO_T;


/* Type define structure of local matrix */
#define MAX 65536  // Maximum number of elements in the
array that store the local matrix (2^16)
typedef struct {
    int     n_bar;
    #define Order(A) ((A)->n_bar)
// defination with parameters
    FLOAT  entries[MAX];
    #define Entry(A,i,j) (*(((A)->entries) +
((A)->n_bar)*(i) + (j)))    // defination with
parameters, Array dereference
} LOCAL_MATRIX_T;


/* Function Declarations */
LOCAL_MATRIX_T*  Local_matrix_allocate(int n_bar);
void             Free_local_matrix(LOCAL_MATRIX_T**
```

```
local_A);

void                Read_matrix_A(char* prompt,
LOCAL_MATRIX_T* local_A,
                        GRID_INFO_T* grid, int n);
// for continuous memory access, local A(i,k)*B(k,j) =
A(i,k)*B^{T}(j,k)
void                Read_matrix_B(char* prompt,
LOCAL_MATRIX_T* local_B,
                        GRID_INFO_T* grid, int n);
// for continuous memory access, local A(i,k)*B(k,j) =
A(i,k)*B^{T}(j,k)
void                Print_matrix(char* title,
LOCAL_MATRIX_T* local_A,
                        GRID_INFO_T* grid, int n);
void                Print_matrix_B(char* title,
LOCAL_MATRIX_T* local_B,    // speical print function
for local matrix B^{T}(j,k)
                        GRID_INFO_T* grid, int n);

void                Set_to_zero(LOCAL_MATRIX_T* local_A);
void                Local_matrix_multiply(LOCAL_MATRIX_T*
```

```
local_A,
                        LOCAL_MATRIX_T* local_B,
LOCAL_MATRIX_T* local_C);

void                Build_matrix_type(LOCAL_MATRIX_T*
local_A);
MPI_Datatype        local_matrix_mpi_t;

LOCAL_MATRIX_T*  temp_mat;        // global
LOCAL_MATRIX_T* type pointer

void                Print_local_matrices(char* title,
LOCAL_MATRIX_T* local_A,
                        GRID_INFO_T* grid);
void                Print_local_matrices_B(char* title,
LOCAL_MATRIX_T* local_B, // speical print function for
local matrix B^{T}(j,k)
                        GRID_INFO_T* grid);

/*********************************************************
***/
main(int argc, char* argv[]) {
```

```
    int             p;
    int             my_rank;
    GRID_INFO_T     grid;
    LOCAL_MATRIX_T* local_A;
    LOCAL_MATRIX_T* local_B;
    LOCAL_MATRIX_T* local_C;
    int             n;
    int             n_bar;
    double          timer_start;
    double          timer_end;

    void Setup_grid(GRID_INFO_T*  grid);
    void Fox(int n, GRID_INFO_T* grid, LOCAL_MATRIX_T*
local_A,
             LOCAL_MATRIX_T* local_B, LOCAL_MATRIX_T*
local_C);

    // SPMD Mode start from here (Processess fork from
here)
    MPI_Init(&argc, &argv);
// MPI initializing
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

```
// Get my process id in the MPI communicator

    // Initial OpenMP Environment
    omp_set_num_threads(NUM_THREADS);
    kmp_set_defaults(AFFINITY);

    Setup_grid(&grid);
// Set up Processess grid
    if (my_rank == 0) {
        printf("What's the order of the matrices?\n");
        scanf("%d", &n);
// Overall Matrix's Order
    }

    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
// MPI broadcast the overall matrix's order
    n_bar = n/grid.q;
// \bar n is the local matrix's order

    local_A = Local_matrix_allocate(n_bar);
// Allocate local matrix A
    Order(local_A) = n_bar;
```

```
// Local matrix A's order
    Read_matrix_A("Enter A", local_A, &grid, n);
// Read local matrices A from process 0 by using stdin,
and send them to each process (Procedure)
    Print_matrix("We read A =", local_A, &grid, n);
// Print local matrices A from process 0 by using
stdout, and send them to each process (Procedure)

    local_B = Local_matrix_allocate(n_bar);
// Allocate local matrix
    Order(local_B) = n_bar;
// Local matrix B's order
    Read_matrix_B("Enter B", local_B, &grid, n);
// Read local matrix B as it's local transpose from
process 0 by using stdin, and send them to each process
(Procedure)
    Print_matrix_B("We read B =", local_B, &grid, n);
// Print local matrix B as it's local transpose from
process 0 by using stdout, and send them to each
process (Procedure)

    Build_matrix_type(local_A);
```

```
// Buid local_A's MPI matrix data type
    temp_mat = Local_matrix_allocate(n_bar);
// Allocate temporary matrix of order n $\time$ n

    local_C = Local_matrix_allocate(n_bar);
// Allocate matrix local_C
    Order(local_C) = n_bar;
// Set matrix local_C's order

    MPI_Barrier(MPI_COMM_WORLD);
// Set the MPI process barrier
    timer_start = MPI_Wtime();
// Get the MPI wall time
    Fox(n, &grid, local_A, local_B, local_C);
// FOX parallel matrix multiplication Algorithm
implement function
    timer_end = MPI_Wtime();
// Get the MPI wall time
    MPI_Barrier(MPI_COMM_WORLD);
// Set the MPI process barrier

    Print_matrix("The product is", local_C, &grid, n);
```

```
// Print matrix local_C (parallel matrix multiplication
result)

    Print_local_matrices("Split of local matrix A",
                         local_A, &grid);
// Print matrix A split in processess
    Print_local_matrices_B("Split of local matrix B",
                         local_B, &grid);
// Print matrix B split in processess, special for row-
major storage
    Print_local_matrices("Split of local matrix C",
                         local_C, &grid);
// Print matrix C split in processess

    Free_local_matrix(&local_A);
// Free local matrix local_A
    Free_local_matrix(&local_B);
// Free local matrix local_B
    Free_local_matrix(&local_C);
// Free local matrix local_C

    if(my_rank == 0)
```

```
        printf("Parallel Fox Matrix Multiplication
Elapsed time:\n %30.20E seconds\n", timer_end-
timer_start);

    MPI_Finalize();
// MPI finalize, processes join and resource recycle

}  /* main */


/*********************************************************
***/
void Setup_grid(
         GRID_INFO_T*  grid  /* out */) {
    int old_rank;
    int dimensions[2];
    int wrap_around[2];
    int coordinates[2];
    int free_coords[2];

    /* Set up Global Grid Information */
    MPI_Comm_size(MPI_COMM_WORLD, &(grid->p));
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &old_rank);

    /* We assume p is a perfect square */     // but
what if it's not a perfect square
    grid->q = (int) sqrt((double) grid->p);
    dimensions[0] = dimensions[1] = grid->q;

    /* We want a circular shift in second dimension. */
    /* Don't care about first                          */
    wrap_around[0] = wrap_around[1] = 1;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions,
        wrap_around, 1, &(grid->comm));
    MPI_Comm_rank(grid->comm, &(grid->my_rank));
    MPI_Cart_coords(grid->comm, grid->my_rank, 2,
        coordinates);
    grid->my_row = coordinates[0];
    grid->my_col = coordinates[1];

    /* Set up row communicators */
    free_coords[0] = 0;
    free_coords[1] = 1;
    MPI_Cart_sub(grid->comm, free_coords,
```

```
        &(grid->row_comm));

    /* Set up column communicators */
    free_coords[0] = 1;
    free_coords[1] = 0;
    MPI_Cart_sub(grid->comm, free_coords,
        &(grid->col_comm));
} /* Setup_grid */


/*********************************************************
***/
void Fox(
        int               n         /* in  */,
        GRID_INFO_T*      grid      /* in  */,
        LOCAL_MATRIX_T*   local_A   /* in  */,
        LOCAL_MATRIX_T*   local_B   /* in  */,
        LOCAL_MATRIX_T*   local_C   /* out */) {

    LOCAL_MATRIX_T*  temp_A; /* Storage for the sub-
*/
                             /* matrix of A used during
```

```
*/
                                /* the current stage
*/
    int                 stage;
    int                 bcast_root;
    int                 n_bar;  /* n/sqrt(p)
*/
    int                 source;
    int                 dest;
    MPI_Status          status;

    n_bar = n/grid->q;
    Set_to_zero(local_C);

    /* Calculate addresses for row circular shift of B
*/
    source = (grid->my_row + 1) % grid->q;
    dest = (grid->my_row + grid->q - 1) % grid->q;

    /* Set aside storage for the broadcast block of A
*/
    temp_A = Local_matrix_allocate(n_bar);
```

```
    for (stage = 0; stage < grid->q; stage++) {
        bcast_root = (grid->my_row + stage) % grid->q;
        if (bcast_root == grid->my_col) {
            MPI_Bcast(local_A, 1, local_matrix_mpi_t,
                bcast_root, grid->row_comm);
            Local_matrix_multiply(local_A, local_B,
                local_C);
        } else {
            MPI_Bcast(temp_A, 1, local_matrix_mpi_t,
                bcast_root, grid->row_comm);
            Local_matrix_multiply(temp_A, local_B,
                local_C);
        }
        MPI_Sendrecv_replace(local_B, 1,
local_matrix_mpi_t,
            dest, 0, source, 0, grid->col_comm,
&status);
    } /* for */

} /* Fox */
```

```
/*********************************************************
***/
LOCAL_MATRIX_T* Local_matrix_allocate(int local_order)
{
    LOCAL_MATRIX_T* temp;

    temp = (LOCAL_MATRIX_T*)
malloc(sizeof(LOCAL_MATRIX_T));
    return temp;
}  /* Local_matrix_allocate */


/*********************************************************
***/
void Free_local_matrix(
        LOCAL_MATRIX_T** local_A_ptr  /* in/out */) {
    free(*local_A_ptr);
}  /* Free_local_matrix */


/*********************************************************
```

```
***/
/* Read and distribute matrix for matrix A:
 *     foreach global row of the matrix,
 *         foreach grid column
 *             read a block of n_bar floats on process
0
 *             and send them to the appropriate
process.
 */
void Read_matrix_A(
        char*           prompt   /* in  */,
        LOCAL_MATRIX_T* local_A  /* out */,
        GRID_INFO_T*    grid     /* in  */,
        int             n        /* in  */) {

    int         mat_row, mat_col;
    int         grid_row, grid_col;
    int         dest;
    int         coords[2];
    FLOAT*      temp;
    MPI_Status status;

```

```
    if (grid->my_rank == 0) {  // Process 0 read matrix
input from stdin and send them to other processess
        temp = (FLOAT*)
malloc(Order(local_A)*sizeof(FLOAT));
        printf("%s\n", prompt);
        fflush(stdout);
        for (mat_row = 0;  mat_row < n; mat_row++) {
            grid_row = mat_row/Order(local_A);
            coords[0] = grid_row;
            for (grid_col = 0; grid_col < grid->q;
grid_col++) {
                coords[1] = grid_col;
                MPI_Cart_rank(grid->comm, coords,
&dest);
                if (dest == 0) {
                    for (mat_col = 0; mat_col <
Order(local_A); mat_col++)
                        scanf("%lf",

(local_A->entries)+mat_row*Order(local_A)+mat_col);
                } else {
                    for(mat_col = 0; mat_col <
```

```
Order(local_A); mat_col++)
                        scanf("%lf", temp + mat_col);
                    MPI_Send(temp, Order(local_A),
FLOAT_MPI, dest, 0,
                        grid->comm);
                }
            }
        }
        free(temp);
    } else {  // Other processess receive matrix from
process 0
        for (mat_row = 0; mat_row < Order(local_A);
mat_row++)
            MPI_Recv(&Entry(local_A, mat_row, 0),
Order(local_A),
                FLOAT_MPI, 0, 0, grid->comm, &status);
    }

}  /* Read_matrix */


/*********************************************************
```

```
***/
/* Read and distribute matrix for local matrix B's
transpose:
 *     foreach global row of the matrix,
 *         foreach grid column
 *             read a block of n_bar floats on process
0
 *             and send them to the appropriate
process.
 */

void Read_matrix_B(
         char*            prompt   /* in  */,
         LOCAL_MATRIX_T*  local_B  /* out */,
         GRID_INFO_T*     grid     /* in  */,
         int              n        /* in  */) {

    int         mat_row, mat_col;
    int         grid_row, grid_col;
    int         dest;
    int         coords[2];
    FLOAT       *temp;
```

```
    MPI_Status status;

    if (grid->my_rank == 0) {  // Process 0 read matrix
input from stdin and send them to other processess
        temp = (FLOAT*)
malloc(Order(local_B)*sizeof(FLOAT));
        printf("%s\n", prompt);
        fflush(stdout);
        for (mat_row = 0;  mat_row < n; mat_row++) {
            grid_row = mat_row/Order(local_B);
            coords[0] = grid_row;
            for (grid_col = 0; grid_col < grid->q;
grid_col++) {
                coords[1] = grid_col;
                MPI_Cart_rank(grid->comm, coords,
&dest);
                if (dest == 0)
{                                                    //
process 0 (local)
                    for (mat_col = 0; mat_col <
Order(local_B); mat_col++)
                        scanf("%lf",
```

```
(local_B->entries)+mat_col*Order(local_B)+mat_row);
// switch rows and colums in local_B, for column major
storage
                        /* scanf("%lf",
(local_A->entries)+mat_row*Order(local_A)+mat_col); */
                } else {
                    for(mat_col = 0; mat_col <
Order(local_B); mat_col++)
                        scanf("%lf", temp + mat_col);
                    MPI_Send(temp, Order(local_B),
FLOAT_MPI, dest, 0,
                        grid->comm);
                }
            }
        }
        free(temp);
    } else {  // Other processess receive matrix from
process 0
        temp = (FLOAT*)
malloc(Order(local_B)*sizeof(FLOAT));                //
```

```
switch rows and colums in local_B, for column major
storage
        for (mat_col = 0; mat_col < Order(local_B);
mat_col++) {
            MPI_Recv(temp, Order(local_B),
                FLOAT_MPI, 0, 0, grid->comm, &status);
// switch rows and colums in local_B, for column major
storage
            for(mat_row = 0; mat_row < Order(local_B);
mat_row++)
                Entry(local_B, mat_row, mat_col) =
*(temp + mat_row);       // switch rows and colums in
local_B, for column major storage

            /* MPI_Recv(&Entry(local_A, mat_row, 0),
Order(local_A),
                FLOAT_MPI, 0, 0, grid->comm, &status);
*/
        }
        free(temp);
    }

```

```
}  /* Read_matrix_B */


/*********************************************************
***/
/* Recive and Print Matrix:
 *     foreach global row of the matrix,
 *         foreach grid column
 *             send n_bar floats to process 0 from each
other process
 *             receive a block of n_bar floats on
process 0 from other processes and print them
 */
void Print_matrix(
        char*           title    /* in  */,
        LOCAL_MATRIX_T*  local_A  /* out */,
        GRID_INFO_T*     grid     /* in  */,
        int             n        /* in  */) {
    int        mat_row, mat_col;
    int        grid_row, grid_col;
    int        source;
    int        coords[2];
```

```
    FLOAT*      temp;
    MPI_Status status;

    if (grid->my_rank == 0) {
        temp = (FLOAT*)
malloc(Order(local_A)*sizeof(FLOAT));
        printf("%s\n", title);
        for (mat_row = 0;  mat_row < n; mat_row++) {
            grid_row = mat_row/Order(local_A);
            coords[0] = grid_row;
            for (grid_col = 0; grid_col < grid->q;
grid_col++) {
                coords[1] = grid_col;
                MPI_Cart_rank(grid->comm, coords,
&source);
                if (source == 0) {
                    for(mat_col = 0; mat_col <
Order(local_A); mat_col++)
                        printf("%20.15E ",
Entry(local_A, mat_row, mat_col));
                } else {
                    MPI_Recv(temp, Order(local_A),
```

```
FLOAT_MPI, source, 0,
                        grid->comm, &status);
                    for(mat_col = 0; mat_col <
Order(local_A); mat_col++)
                        printf("%20.15E ",
temp[mat_col]);
                }
            }
            printf("\n");
        }
        free(temp);
    } else {
        for (mat_row = 0; mat_row < Order(local_A);
mat_row++)
            MPI_Send(&Entry(local_A, mat_row, 0),
Order(local_A),
                FLOAT_MPI, 0, 0, grid->comm);
    }

}  /* Print_matrix */
```

```
/*********************************************************
***/
/* Recive and Print Matrix for local matrix B's
transpose:
 *     foreach global row of the matrix,
 *         foreach grid column
 *             send n_bar floats to process 0 from each
other process
 *             receive a block of n_bar floats on
process 0 from other processes and print them
 */
void Print_matrix_B(
        char*            title    /* in  */,
        LOCAL_MATRIX_T*  local_B  /* out */,
        GRID_INFO_T*     grid     /* in  */,
        int              n        /* in  */) {
    int        mat_row, mat_col;
    int        grid_row, grid_col;
    int        source;
    int        coords[2];
    FLOAT*     temp;
    MPI_Status status;
```

```
    if (grid->my_rank == 0) {
        temp = (FLOAT*)
malloc(Order(local_B)*sizeof(FLOAT));
        printf("%s\n", title);
        for (mat_row = 0;  mat_row < n; mat_row++) {
            grid_row = mat_row/Order(local_B);
            coords[0] = grid_row;
            for (grid_col = 0; grid_col < grid->q;
grid_col++) {
                coords[1] = grid_col;
                MPI_Cart_rank(grid->comm, coords,
&source);
                if (source == 0) {
                    for(mat_col = 0; mat_col <
Order(local_B); mat_col++)
                        printf("%20.15E ",
Entry(local_B, mat_col, mat_row));      // switch rows
and colums in local_B, for column major storage
                        // printf("%20.15E ",
Entry(local_A, mat_row, mat_col));
                } else {
```

```
                    MPI_Recv(temp, Order(local_B),
FLOAT_MPI, source, 0,
                        grid->comm, &status);
                    for(mat_col = 0; mat_col <
Order(local_B); mat_col++)
                        printf("%20.15E ",
temp[mat_col]);
                }
            }
            printf("\n");
        }
        free(temp);
    } else {
        temp = (FLOAT*)
malloc(Order(local_B)*sizeof(FLOAT));
        for (mat_col = 0; mat_col < Order(local_B);
mat_col++) {
            for(mat_row = 0; mat_row < Order(local_B);
mat_row++)
                *(temp+mat_row) = Entry(local_B,
mat_row, mat_col);       // switch rows and colums in
local_B, for column major storage
```

```
            MPI_Send(temp, Order(local_B), FLOAT_MPI,
0, 0, grid->comm);
        }

        free(temp);
    }

} /* Print_matrix_B */


/*********************************************************
***/
/*
*  Set local matrix's element to zero
*/
void Set_to_zero(
        LOCAL_MATRIX_T*  local_A  /* out */) {

    int i, j;

    for (i = 0; i < Order(local_A); i++)
```

```
        for (j = 0; j < Order(local_A); j++)
            Entry(local_A,i,j) = 0.0E0;

}  /* Set_to_zero */


/*********************************************************
***/
void Build_matrix_type(
         LOCAL_MATRIX_T*  local_A  /* in */) {
    MPI_Datatype  temp_mpi_t;
    int           block_lengths[2];
    MPI_Aint      displacements[2];
    MPI_Datatype  typelist[2];
    MPI_Aint      start_address;
    MPI_Aint      address;

    MPI_Type_contiguous(Order(local_A)*Order(local_A),
        FLOAT_MPI, &temp_mpi_t);
// Creates a contiguous datatype
    /*
    Synopsis
```

```
            int MPI_Type_contiguous(int count,
                                    MPI_Datatype oldtype,
                                    MPI_Datatype *newtype)

    Input Parameters

    count
            replication count (nonnegative integer)
    oldtype
            old datatype (handle)
    */
    block_lengths[0] = block_lengths[1] = 1;

    typelist[0] = MPI_INT;
    typelist[1] = temp_mpi_t;

    MPI_Address(local_A, &start_address);
// Gets the address of a location in caller's memory
    MPI_Address(&(local_A->n_bar), &address);
    /*
    Synopsis
```

```
            int MPI_Address(const void *location,
MPI_Aint *address)

    Input Parameters

    location
            location in caller memory (choice)

    Output Parameters

            address
            address of location (address integer)
    */
    displacements[0] = address - start_address;

    MPI_Address(local_A->entries, &address);
    displacements[1] = address - start_address;

    MPI_Type_struct(2, block_lengths, displacements,
        typelist, &local_matrix_mpi_t);
// Creates a struct datatype
```

```
    /*
    Synopsis

    int MPI_Type_struct(int count,
                        const int *array_of_blocklengths,
                        const MPI_Aint
*array_of_displacements,
                        const MPI_Datatype
*array_of_types,
                        MPI_Datatype *newtype)

    Input Parameters

    count
        number of blocks (integer) -- also number of
entries in arrays array_of_types ,
array_of_displacements and array_of_blocklengths
    array_of_blocklengths
        number of elements in each block (array)
    array_of_displacements
        byte displacement of each block (array)
    array_of_types
```

```
        type of elements in each block (array of
handles to datatype objects)

    Output Parameters

    newtype
        new datatype (handle)
    */
    MPI_Type_commit(&local_matrix_mpi_t);
// Commits the datatype
    /*
    Synopsis

    int MPI_Type_commit(MPI_Datatype *datatype)

    Input Parameters

    datatype
        datatype (handle)
    */
}  /* Build_matrix_type */
```

```
/*****************************************************
***/
/* local matrix multiplication function
*  withing OpenMP Thread Acceleration
*/
void Local_matrix_multiply(
        LOCAL_MATRIX_T*  local_A  /* in  */,
        LOCAL_MATRIX_T*  local_B  /* in  */,
        LOCAL_MATRIX_T*  local_C  /* out */) {

    int i, j, k;

    // int my_rank;
    // MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
// Get my process id in the MPI communicator

    #pragma omp parallel for private(i, j, k)
shared(local_A, local_B, local_C)
num_threads(NUM_THREADS)       // Threads acceleration
upgrade, parallel task split
    for (i = 0; i < Order(local_A); i++) {
```

```
        // printf("Current in the Fox Kernel:\n my
process id is %d, my thread id
is %d\n",my_rank,omp_get_thread_num());
        for (j = 0; j < Order(local_A); j++)
            for (k = 0; k < Order(local_B); k++)
                Entry(local_C,i,j) = Entry(local_C,i,j)
// switch rows and colums in local_B, for column major
storage
                    +
Entry(local_A,i,k)*Entry(local_B,j,k);        //
continuous memory access, local matrix multiplication
A(i,k)*B^T(j,k)
                /* Entry(local_C,i,j) =
Entry(local_C,i,j)
                    +
Entry(local_A,i,k)*Entry(local_B,k,j);        // non-
continuous memory access, A(i,k)*B^T(j,k) is more
proper
                */
    }
}  /* Local_matrix_multiply */
```

```
/*********************************************************
***/
/* Recive and Print Local Matrix:
 *        Process 0 print local matrix local_A
 *        Other Processess send local matrix local_A to
process 0
 *        And process 0 receive local matrix local_A
from other processess
 */
void Print_local_matrices(
        char*           title    /* in */,
        LOCAL_MATRIX_T*  local_A  /* in */,
        GRID_INFO_T*     grid     /* in */) {

    int         coords[2];
    int         i, j;
    int         source;
    MPI_Status  status;

    // print by process No.0 in process mesh
    if (grid->my_rank == 0) {
```

```
        printf("%s\n", title);
        printf("Process %d > grid_row = %d, grid_col
= %d\n",
            grid->my_rank, grid->my_row, grid->my_col);
        for (i = 0; i < Order(local_A); i++) {
            for (j = 0; j < Order(local_A); j++)
                printf("%20.15E ", Entry(local_A,i,j));
            printf("\n");
        }
        for (source = 1; source < grid->p; source++) {
            MPI_Recv(temp_mat, 1, local_matrix_mpi_t,
source, 0,
                grid->comm, &status);
            MPI_Cart_coords(grid->comm, source, 2,
coords);
            printf("Process %d > grid_row = %d,
grid_col = %d\n",
                source, coords[0], coords[1]);
            for (i = 0; i < Order(temp_mat); i++) {
                for (j = 0; j < Order(temp_mat); j++)
                    printf("%20.15E ",
Entry(temp_mat,i,j));
```

```
                printf("\n");
            }
        }
        fflush(stdout);
    } else {
        MPI_Send(local_A, 1, local_matrix_mpi_t, 0, 0,
grid->comm);
    }

}  /* Print_local_matrices */



/*********************************************************
***/
/* Recive and Print Local Matrix for local matrix B's
transpose:
 *        Process 0 print local matrix local_A
 *        Other Processess send local matrix local_A to
process 0
 *        And process 0 receive local matrix local_A
from other processess
```

```
 */
void Print_local_matrices_B(
        char*             title    /* in */,
        LOCAL_MATRIX_T*   local_B  /* in */,
        GRID_INFO_T*      grid     /* in */) {

    int         coords[2];
    int         i, j;
    int         source;
    MPI_Status  status;

    // print by process No.0 in process mesh
    if (grid->my_rank == 0) {
        printf("%s\n", title);
        printf("Process %d > grid_row = %d, grid_col
= %d\n",
            grid->my_rank, grid->my_row, grid->my_col);
        for (i = 0; i < Order(local_B); i++) {
            for (j = 0; j < Order(local_B); j++)
                printf("%20.15E ", Entry(local_B,j,i));
// switch rows and colums in local_B, for column major
storage
```

```
            printf("\n");
        }
        for (source = 1; source < grid->p; source++) {
            MPI_Recv(temp_mat, 1, local_matrix_mpi_t,
source, 0,
                    grid->comm, &status);
            MPI_Cart_coords(grid->comm, source, 2,
coords);
            printf("Process %d > grid_row = %d,
grid_col = %d\n",
                    source, coords[0], coords[1]);
            for (i = 0; i < Order(temp_mat); i++) {
                for (j = 0; j < Order(temp_mat); j++)
                    printf("%20.15E ",
Entry(temp_mat,j,i));              // switch rows and
colums in local_B, for column major storage
                printf("\n");
            }
        }
        fflush(stdout);
    } else {
        MPI_Send(local_B, 1, local_matrix_mpi_t, 0, 0,
```

```
grid->comm);
    }

}  /* Print_local_matrices_B */
```