矩阵计算并行算法

主要内容

- 并行算法基础知识
- 矩阵向量乘积的并行算法
- 矩阵矩阵乘积的并行算法

并行算法基础知识

一些基本概念

- 加速比 $$S_p(q) = \frac{T_s}{T_p(q)}$$

其中 T_s 串行程序运行时间，$T_p(q)$ 为 q 个进程的运行时间

- 并行效率 $$E_p(q) = \frac{S_p(q)}{q}$$

程序性能优化

串行程序性能优化 —— 并行程序性能优化的基础

- 调用高性能库。如：BLAS、LAPACK、FFTW
- 选择编译器优化选项：-O2、-O3
- 合理定义数组维数
- 注意嵌套循环次数：数据访问的空间局部性和时间局部性
- 循环展开 例：ex4performance.c
- 数据分块

程序性能优化

并行程序性能优化

- 设计好的并行算法和通信模式
- 减少通信次数、提高通信粒度
- 多进程通信时尽量使用高效率的聚合通信算法
- 负载平衡
- 减少进程的空闲时间
- 通信与计算的重叠
- 通过引入重复计算来减少通信

矩阵并行算法

一些记号和假定

- 假设有 p 个处理器，每个处理器上运行一个进程
- P_j 表示第 j 个处理器，P_{myid} 表示当前的处理器
- `send`$(x; j)$ 表示在 P_{myid} 中把数据块 x 发送给 P_j 进程
- `recv`$(x; j)$ 表示从 P_j 进程接收数据块 x
- i mod p 表示 i 对 p 取模运算

程序设计与机器实现是密不可分的，计算结果的好坏与编程技术有很大的关系，尤其是在并行计算机环境下，开发高质量的程序对发挥计算机的性能起着至关重要的作用

主要内容

- 并行算法基础知识
- 矩阵向量乘积的并行算法
- 矩阵矩阵乘积的并行算法

矩阵向量乘积

$$y = Ax \qquad A \in \mathrm{R}^{m \times n},\ x \in \mathrm{R}^{n}$$

串行算法

- 实现方法一：i-j 循环

```
for i=1 to m
  y(i)=0.0
  for j=1 to n
    y(i)=y(i)+A(i,j)*x(j)
  end for
end for
```

矩阵向量乘积

串行算法

- 实现方法二：j-i 循环

```
y=0 % 先赋初值
for j=1 to n
   for i=1 to m
      y(i)=y(i)+A(i,j)*x(j)
   end for
end for
```

例：mat_vec.c

矩阵向量乘积

并行算法一

- 矩阵的划分方法：按行划分和按列划分
- 按行划分并行算法

将矩阵 A 按行划分成如下的行块子矩阵

$$A = [A_0^T, A_1^T, \ldots, A_{p-1}^T]^T$$

则

$$\begin{aligned} Ax &= [A_0^T, A_1^T, \ldots, A_{p-1}^T]^T x \\ &= [(A_0x)^T, (A_1x)^T, \ldots, (A_{p-1}x)^T] \end{aligned}$$

将 A_i 存放在结点 P_i 中，每个结点计算 A_ix，最后调用 MPI_GATHER 或 MPI_GATHERV 即可

矩阵向量乘积

并行算法二

按列划分并行算法

将矩阵 A 按列划分，并对 x 也做相应的划分

$$A = [A_0, A_1, \ldots, A_{p-1}],\ x = (x_0^T, x_1^T, \ldots, x_{p-1}^T)^T$$

其中 x_i 的长度与 A_i 的列数相同，则有

$$Ax = \sum_{i=0}^{p-1} A_i x_i^T$$

将 A_i 和 x_i 存放在结点 P_i 中，每个结点计算 $A_i x_i^T$，最后调用 MPI_REDUCE 或 MPI_ALLREDUCE 即可

矩阵向量乘积示例

- 例：按行划分，用 p 个进程并行计算矩阵向量乘积，其中

$$A = (a_{ij}) \in \mathrm{R}^{h \times n}, \quad x = [1, 1, \ldots, 1]^T \in \mathrm{R}^{h},$$

$$a_{ij} = \frac{1}{i + j - 1}$$

示例程序：mpi_matvec.c
（n=1024, 1024*32）

主要内容

- 并行算法基础知识
- 矩阵向量乘积的并行算法
- 矩阵矩阵乘积的并行算法

矩阵矩阵乘积

$$C = A * B \qquad A \in \mathrm{R}^{m \times l},\ B \in \mathrm{R}^{l \times n}$$

- 串行算法一：i-j-k 循环

```
for i=1 to m
  for j=1 to l
    C(i,j)=0
    for k=1 to n
      C(i,j)=C(i,j)+A(i,k)*B(k,j)
    end for
  end for
end for
```

矩阵矩阵乘积

$$C = A * B \qquad A \in \mathrm{R}^{m \times l},\ B \in \mathrm{R}^{l \times n}$$

- 串行算法二：j-k-i 循环

```
C=0
for j=1 to l
  for k=1 to n
    for i=1 to m
      C(i,j)=C(i,j)+A(i,k)*B(k,j)
    end for
  end for
end for
```

并行矩阵乘积

$$C = A * B \qquad A \in \mathrm{R}^{m \times l},\ B \in \mathrm{R}^{l \times n}$$

- 假定：

  m, l, n 均能能 p 整除，其中 p 为进程个数

- 基于 A、B 的不同划分，矩阵乘积的并行算法可分为
  - 行列划分
  - 行行划分
  - 列列划分
  - 列行划分

行列划分

- **行列划分：*A* 按行划分、 *B* 按列划分**

$$AB = \begin{bmatrix} A_0 \\ A_1 \\ \vdots \\ A_{p-1} \end{bmatrix} \begin{bmatrix} B_0 & B_1 & \cdots & B_{p-1} \end{bmatrix} = \begin{bmatrix} A_0B_0 & A_0B_1 & \cdots & A_0B_{p-1} \\ A_1B_0 & A_1B_1 & \cdots & A_1B_{p-1} \\ \vdots & & \ddots & A_0B_{p-1} \\ A_{p-1}B_0 & A_{p-1}B_1 & \cdots & A_{p-1}B_{p-1} \end{bmatrix}$$

- 令 $C=(C_{ij})$，其中 $C_{ij}=A_iB_j$
- 将 A_i, B_j 和 C_{ij} ($j=0, 1, ..., p-1$) 存放在第 i 个处理器中
  (这样的存储方式使得数据在处理器中不重复)
- P_i 负责计算 C_{ij} ($j=0, 1, ..., p-1$)
- 由于使用 p 个处理器，每次每个处理器只计算一个 C_{ij}，
  故计算出整个 C 需要 p 次完成
- C_{ij} 的计算是按对角线进行的

行列划分

- 并行算法一：行列划分

```
for i=0 to p-1
  j=(i+myid) mod p
  Cj=A*B
  src = (myid+1) mod p
  dest = (myid-1+p) mod p
  if (i!=p-1)
    send(B,dest)
    recv(B,src)
  end if
end for
```

- 本算法中，$C_j = C_{myid,j}, A = A_{myid}, B$ 在处理器中每次循环向前移动一个处理器，即每次交换一个子矩阵数据块，共交换 $p-1$ 次

行列划分程序示例

- 例：按行列划分并行计算矩阵乘积，其中

$$A=(a_{ij})\in \mathrm{R}^{n\times n},\quad a_{ij}=\frac{1}{i+j-1}$$

$$B=(b_{ij})\in \mathrm{R}^{n\times n},\quad b_{ij}=i+j-1$$

示例程序：`mpi_matmul01.c`

行行划分

■ 行行划分：A 按行划分、 B 按行划分

将矩阵 A, B 分别划分成如下的行块子矩阵

$$A = [A_0^T, A_1^T, \ldots, A_{p-1}^T]^T, \quad B = [B_0^T, B_1^T, \ldots, B_{p-1}^T].$$

这时，需要将 A 的子块 A_i 进一步按列分块，且与 B 的行分块相对应：

$$A_i = [A_{i0}, A_{i1}, \ldots, A_{i,p-1}], \quad i = 0, 1, \ldots, p-1.$$

设 C_i 为与 A_i 相应的 C 的第 i 个行分块，则有

$$C_i = A_i * B = \sum_{j=0}^{p-1} A_{ij} B_j.$$

将 A_i, B_i 存放在第 i 个处理器 P_i 中，处理器 P_i 负责计算 C_i。整个计算过程也需要 p 次来完成。

行行划分

算法 2 行行划分矩阵乘积并行算法

$$
\begin{aligned}
&for\ (i = 0\ to\ p-1)\ do\\
&\quad j = (i + myid) \bmod p\\
&\quad C = C + A_j * B\\
&\quad source = (myid + 1) \bmod p\\
&\quad dest = (myid - 1) \bmod p\\
&\quad if\ (i \neq p-1),\ send(B, dest),\ recv(B, source)\\
&end\{for\}
\end{aligned}
$$

- 在本算法中，$C = C_{myid}$，$A_j = A_{myid,j}$，B 在处理器中每次循环向前移动一个处理器。本算法中的数据交换量和计算量均与算法 1 相同，不同的只是计算 C 的方式。

列列划分

- **列列划分：A 按列划分、 B 按列划分**

将矩阵 A, B 分别划分成如下的列块子矩阵

$$A = [A_0, A_1, \ldots, A_{p-1}], \quad B = [B_0, B_1, \ldots, B_{p-1}].$$

这时，需要将 B 的子块 B_j 进一步按行分块，且满足与 A 的列分块相对应：

$$B_j = [B_{0j}^T, B_{1j}^T, \ldots, A_{p-1,j}^T]^T, \quad j = 0, 1, \ldots, p-1.$$

此时，设 C 的划分与 B 的划分相对应，即按列分块。同样将 A_j, B_j 存放在处理器 P_j 中，C_j 也存放处理器 P_j 中。有

$$C_j = A * B_j = \sum_{i=0}^{p-1} A_i B_{ij}.$$

注意此时计算过程中需交换的数据矩阵是 A，而不是 B。

列列划分

算法 3 列列划分矩阵乘积并行算法

$$
\begin{aligned}
&for\ (i = 0\ to\ p-1)\ do\\
&\quad k = (i + myid) \bmod p\\
&\quad C = C + A * B_k\\
&\quad source = (myid + 1) \bmod p\\
&\quad dest = (myid - 1) \bmod p\\
&\quad if\ (i \neq p-1),\ send(A, dest),\ recv(A, source)\\
&end\{for\}
\end{aligned}
$$

- 在本算法中，$C = C_{myid}$，$B_k = B_{k,myid}$，A 在处理器中每次循环向前移动一个处理器。本算法的计算量与算法 1 和算法 2 的是相同的，当 $m \neq n$ 时，通信量略有不同，所以在具体应用中可以按通信量大小选择算法就能得到好的并行效率。

列行划分

- **列行划分：A 按列划分、 B 按行划分**

将矩阵 A, B 分别划分成如下的行块子矩阵和列块子矩阵

$$A = [A_0, A_1, \ldots, A_{p-1}], \quad B = [B_0^T, B_1^T, \ldots, B_{p-1}^T]^T,$$

则

$$C = A * B = \sum_{i=0}^{p-1} A_i B_i.$$

此时 C 的计算是通过计算 p 个规模和 C 相同的矩阵之和得到的。从对问题的划分可以看出，并行算法的关键是计算矩阵的和，设计有效地计算矩阵和的算法，对发挥分布式并行系统的效率起着重要作用。假设结果矩阵 C 也是按列分块存放在处理器中的，记 $B_i = [B_{i0}, B_{i1}, \ldots, B_{i,p-1}]$，则

$$C_j = \sum_{i=0}^{p-1} A_i B_{ij}.$$

列行划分

算法 4 列行划分矩阵乘积并行算法

$C = A * B_{myid}$

$for\ (i = 1\ to\ p-1)\ do$

$j \equiv (i + myid) \bmod p,\ k \equiv (myid - i) \bmod p$

$T = A * B_j$

$send(T, j),\ recv(T, k)$

$C = C + T$

$end\{for\}$

- 如果采用按行分块方式计算 C，算法 4 也同样适合，且通信量不变，因此选择何种方式计算 C 可根据需要而定。

Cannon 算法

假设矩阵 A，B 和 C 都可以分成 $m \times m$ 块矩阵，即: $A = (A_{ij})_{m\times m}$，$B = (B_{ij})_{m\times m}$ 和 $C = (C_{ij})_{m\times m}$，其中 A_{ij}，B_{ij} 和 C_{ij} 是 $n \times n$ 矩阵，进一步假定有 $p = m \times m$ 个处理器。为了讨论 Cannon 算法，引入块置换矩阵 $Q = (Q_{ij})$ 使得

$$Q = \begin{pmatrix} \mathbf{0} & \boldsymbol{I} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{I} & \cdots & \mathbf{0} \\ \vdots & & & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \boldsymbol{I} \\ \boldsymbol{I} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \end{pmatrix}, \quad \text{即 } Q_{ij} = \begin{cases} \boldsymbol{I}_n, \ j \equiv (i+1) \mod m \\ \mathbf{0}_n, \ \text{other case} \end{cases}$$

则 QA 就是将 A 的所有行向上移动一个位置，即第 m 行移到第 $m-1$ 行，第 $m-1$ 行移到第 $m-2$ 行，以此类推，其中第 1 行移到第 m 行。而 AQ 则是将 A 的所有列向右移动一个位置。

Cannon 算法

- 定义块对角矩阵 $D_A^{(l)} = \mathrm{diag}(D_i^{(l)}) = \mathrm{diag}(A_{i,i+l \bmod m})$，容易证明 $A = \sum_{l=0}^{m-1} D_A^{(l)} Q^l$，于是

$$C = AB = \sum_{l=0}^{m-1} D_A^{(l)} Q^l B$$

$$= D_A^{(0)} B^{(0)} + D_A^{(1)} B^{(1)} + \cdots + D_A^{(m-1)} B^{(m-1)},$$

其中 $B^{(l)} = Q^l B = Q B^{(l-1)}$, $l = 0, 1, \ldots, m-1$。

- 利用这个递推关系式，并把处理器结点编号从一维映射到二维，即有 $P_{myid} = P_{myrow,mycol}$，数据 A_{ij}，B_{ij} 和 C_{ij} 存放在 P_{ij} 中，容易得到下面的在处理器 P_{myid} 结点上的算法。

算法 5 矩阵乘积 Cannon 并行算法

$C = 0$

$for\ (i = 0\ to\ m-1)\ do$ % 第 i 步计算的是第 $i+1$ 项

$k = (myrow + i) \bmod m$

$mp1 = (mycol + 1) \bmod m$

$mm1 = (mycol - 1) \bmod m$

$if\ (mycol = k)\ then$

% 将 $A_{myrow,k}$ 广播给 $P_{myrow,j},\ j = 0, \ldots, m-1, j \neq k$

$send(A, (myrow, mp1));\ copy(A, tmpA)$

$else$

$recv(tmpA, (myrow, mm1))$

$if\ (k \neq mp1),\ send(tmpA, (myrow, mp1))$

$end\{if\}$

$C = C + tmpA * B$

$mp1 = myrow + 1 \bmod m$

$mm1 = myrow - 1 \bmod m$

$if\ (i \neq m-1)\ then$ % 在同列中滚动 B

$send(B, (mm1, mycol));\ recv(B, (mp1, mycol))$

$end\{if\}$

$end\{for\}$

Cannon 算法示例

- 以 3×3 分块为例：9 个进程，进行三轮计算

- A、B 的起始存放位置：

A_{00}	A_{01}	A_{02}
A_{10}	A_{11}	A_{12}
A_{20}	A_{21}	A_{22}

B_{00}	B_{01}	B_{02}
B_{10}	B_{11}	B_{12}
B_{20}	B_{21}	B_{22}

- 第一轮：计算 $D_A^{(0)} B^{(0)}$

A_{00}	A_{00}	A_{00}
A_{11}	A_{11}	A_{11}
A_{22}	A_{22}	A_{22}

B_{00}	B_{01}	B_{02}
B_{10}	B_{11}	B_{12}
B_{20}	B_{21}	B_{22}

Cannon 算法示例

- 第二轮：计算 $D_A^{(1)} B^{(1)}$

A_{01}	A_{01}	A_{01}
A_{12}	A_{12}	A_{12}
A_{20}	A_{20}	A_{20}

B_{10}	B_{11}	B_{12}
B_{20}	B_{21}	B_{22}
B_{00}	B_{01}	B_{02}

- 第三轮：计算 $D_A^{(2)} B^{(2)}$

A_{02}	A_{02}	A_{02}
A_{10}	A_{10}	A_{10}
A_{21}	A_{21}	A_{21}

B_{20}	B_{21}	B_{22}
B_{00}	B_{01}	B_{02}
B_{10}	B_{11}	B_{12}

Cannon 算法

- 该算法具有很好的负载平衡，其特点是在同一行中广播 A，计算出 C 的部分值之后，在同列中滚动 B。
- 本算法也可以通过在水平方向（向左）滚动 A，在垂直方向（向上）滚动 B 来实现。
- 由于计算量对每个处理器来说是相同的，因此在选择算法时只需考虑通信量。对于方阵的乘积，所给出的五个计算矩阵乘积的并行算法中，当 $p \geq 4$ 时，Cannon 算法具有优越性。

Cannon 算法

本算法也可以通过在水平方向（向左）滚动 A，在垂直方向（向上）滚动 B 来实现。但此时 A, B 在处理器中存放位置如下：

A_{00}	A_{01}	A_{02}
A_{11}	A_{12}	A_{10}
A_{22}	A_{20}	A_{21}

B_{00}	B_{11}	B_{22}
B_{10}	B_{21}	B_{02}
B_{20}	B_{01}	B_{12}

即：A 的第 i 行向左移 i 格，而 B 的第 j 列向上移 j 格。在计算时，每计算完一轮后，沿水平方向向左滚动 A，并同时沿垂直方向向上滚动 B。

- 起始存放位置：

$$\begin{array}{ccc|ccc} A_{00} & A_{01} & A_{02} & B_{00} & B_{11} & B_{22} \\ A_{11} & A_{12} & A_{10} & B_{10} & B_{21} & B_{02} \\ A_{22} & A_{20} & A_{21} & B_{20} & B_{01} & B_{12} \end{array}$$

- 第一轮计算：

$$\begin{array}{ccc|ccc} A_{00} & A_{01} & A_{02} & B_{00} & B_{11} & B_{22} \\ A_{11} & A_{12} & A_{10} & B_{10} & B_{21} & B_{02} \\ A_{22} & A_{20} & A_{21} & B_{20} & B_{01} & B_{12} \end{array}$$

- 第二轮计算：

$$\begin{array}{ccc|ccc} A_{01} & A_{02} & A_{00} & B_{10} & B_{21} & B_{02} \\ A_{12} & A_{10} & A_{11} & B_{20} & B_{01} & B_{12} \\ A_{20} & A_{21} & A_{22} & B_{00} & B_{11} & B_{22} \end{array}$$

- 第三轮计算：

$$\begin{array}{ccc|ccc} A_{02} & A_{00} & A_{01} & B_{20} & B_{01} & B_{12} \\ A_{10} & A_{11} & A_{12} & B_{00} & B_{11} & B_{22} \\ A_{21} & A_{22} & A_{20} & B_{10} & B_{21} & B_{02} \end{array}$$