

Методы и средства параллельного программирования

3 курс кафедры СКИ
сентябрь – декабрь 2016 г.

Лектор доцент Н.Н.Попова

Лекция 9. Часть 2.
14 ноября 2016 г.

Тема

- Параллельные алгоритмы матричного умножения.

Матричное умножение

Рассматриваем задачу: параллельная реализация $C = AxV$

Существует множество вариантов решения этой задачи на многопроцессорных системах.

Алгоритм решения существенным образом зависит от того, производится или нет распределение матриц по процессорам, и какая топология процессоров при этом используется.

Распределение матриц по процессам

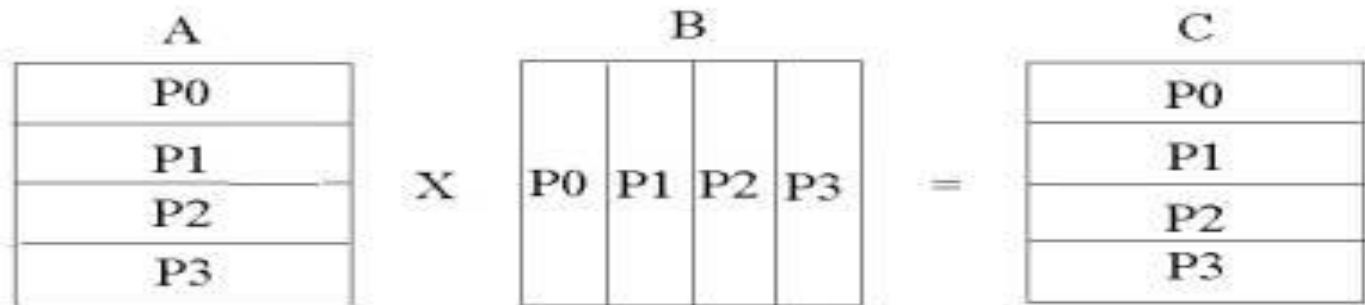
Каждая из трех матриц (A, B и C) может быть распределена одним из способов:

- копии матриц находятся в каждом процессе;
- распределена по столбцам на одномерную сетку;
- распределена по строкам на одномерную сетку;
- распределена на двумерную или трехмерную процессную сетку.

Могут использоваться и различные комбинации. Все зависит от решаемой задачи.

Распределение матриц по процессам

- Возможное распределение матриц:
 - матрица A и C – 1D распределение (ленточное, по строкам)
 - Матрица B – 1D (ленточное, по столбцам)



Блочные алгоритмы матричного умножения

- Алгоритм Фокса
- Алгоритм Кеннона
- Алгоритм SUMMA

Блочный алгоритм

Делим матрицы на подматрицы.

Пусть $p = s^2$ и матрицы разделены на s^2 подматриц.

Каждый блок содержит $n/s \times n/s$ элементов.

Обозначим $A_{p,q}$ - блок матрицы A

for (p = 0; p < s; p++)

for (q = 0; q < s; q++) {

C_{p,q} = 0;

/ обнуление блоков */*

for (r = 0; r < m; r++)

*/*блочное умножение &*/*

C_{p,q} = C_{p,q} + A_{p,r} * B_{r,q}; */*и сложение блоков*/*

}

Строка

C_{p,q} = C_{p,q} + A_{p,r} * B_{r,q};

означает умножение блоков $A_{p,r}$ и $B_{r,q}$, используя матричное умножение

Идея алгоритма Кеннона

- Алгоритм Кеннона определяет порядок суммирования членов во внутреннем цикле:

$$C(i,j) = \sum_{k=0}^{s-1} A(i, (i + j + k) \bmod s) * B((i + j + k) \bmod s, j)$$

таким образом, чтобы в каждом процессе на каждом шаге алгоритма находился один из блоков матриц A и B. Предусматривается первоначальное распределение блоков матриц таким образом, чтобы минимизировать обмены блоками в процессе выполнения алгоритма.

Алгоритм Кеннона: распределение блоков матриц

\sqrt{P}

A0,0	A0,1	A0,2	A0,3
A1,0	A1,1	A1,2	A1,3
A2,0	A2,1	A2,2	A2,3
A3,0	A3,1	A3,2	A3,3

B0,0	B0,1	B0,2	B0,3
B1,0	B1,1	B1,2	B1,3
B2,0	B2,1	B2,2	B2,3
B3,0	B3,1	B3,2	B3,3

A0,0	A0,1	A0,2	A0,3
B0,0	B1,1	B2,2	B3,3
A1,1	A1,2	A1,3	A1,0
B1,0	B2,1	B3,2	B0,3
A2,2	A2,3	A2,0	A2,1
B2,0	B3,1	B0,2	B1,3
A3,3	A3,0	A3,1	A3,2
B3,0	B0,1	B1,2	B2,3

Первоначальное
перераспределение

Алгоритм Кеннона: обмены блоками матриц

A0,1	A0,2	A0,3	A0,0
B1,0	B2,1	B3,2	B0,3
A1,2	A1,3	A1,0	A1,1
B2,0	B3,1	B0,2	B1,3
A2,3	A2,0	A2,1	A2,2
B3,0	B0,1	B1,2	B2,3
A3,0	A3,1	A3,2	A3,3
B0,0	B1,1	B2,2	B3,3

Первый
сдвиг

A0,2	A0,3	A0,0	A0,1
B2,0	B3,1	B0,2	B1,3
A1,3	A1,0	A1,1	A1,2
B3,0	B0,1	B1,2	B2,3
A2,0	A2,1	A2,2	A2,3
B0,0	B1,1	B2,2	B3,3
A3,1	A3,2	A3,3	A3,0
B1,0	B2,1	B3,2	B0,3

Второй сдвиг

A0,3	A0,0	A0,1	A0,2
B3,0	B0,1	B1,2	B2,3
A1,0	A1,1	A1,2	A1,3
B0,0	B1,1	B2,2	B3,3
A2,1	A2,2	A2,3	A2,0
B1,0	B2,1	B3,2	B0,3
A3,2	A3,3	A3,0	A3,1
B2,0	B3,1	B0,2	B1,3

Третий
сдвиг

Схема алгоритма Кеннона

```
for all (i=0 to s-1)      // начальное распределение блоков матрицы A
  Циклический сдвиг влево строки i матрицы A на j
  так, чтобы на место A(i,j) была записана подматрица A(i,(i+j) mod s)
end
for  for all (i=0 to s-1) // начальное распределение блоков матрицы B
  Циклический сдвиг вверх столбца j матрицы B на j
  так, чтобы на место B(i,j) была записана подматрица B((i+j) mod s,j)
  end
for  for k=0 to s-1
  for all (i=0 to s-1, j=0 to s-1)
    C(i,j) = C(i,j) + A(i,j)*B(i,j)
```

Схема алгоритма Кеннона

Циклический сдвиг влево каждой строки матрицы A на 1 так, чтобы на место $A(i,j)$ была записана подматрица $A(i, (j+1) \bmod s)$

Циклический сдвиг вверх каждого столбца матрицы B на 1 так, чтобы на место $B(i,j)$ была записана подматрица $B((i+1) \bmod s, j)$

end for

end for

Алгоритм Кеннона: основной цикл

```
dims[0] = dims[1] = sqrt(P);  
periods[0] = periods[1] = 1;
```

```
MPI_Cart_Create(comm,2,dims,periods,1,&comm_2d);  
MPI_Comm_rank(comm_2d, &my2drank);  
MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);
```

```
MPI_Cart_shift(comm_2d, 0, -1, &rightrank, &leftrank);  
MPI_Cart_shift(comm_2d, 1, -1, &downrank, &uprank);
```

```
nlocal = n/dims[0];
```

Алгоритм Кеннона: основной цикл

```
/* Initial Matrix Alignment */
```

```
MPI_Cart_shift(comm_2d, 0, -mycoords[0], &shiftsource,  
               &shiftdest);
```

```
MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,  
                     shiftdest, 1, shiftsource, 1, comm_2d, &status);
```

```
MPI_Cart_shift(comm_2d, 1, -mycoords[1], &shiftsource,  
               &shiftdest);
```

```
MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,  
                     shiftdest, 1, shiftsource, 1, comm_2d, &status);
```

Алгоритм Кеннона: основной цикл

```
/* Main Computation Loop */
for(i=0; i<dims[0]; i++){
    MatrixMultiply(nlocal,a,b,c); /* c=c+a*b*/

    /* Shift matrix a left by one */
    MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,
        leftrank, 1, rightrank, 1, comm_2d, &status);

    /* Shift matrix b up by one */
    MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,
        uprank, 1, downrank, 1, comm_2d, &status);
}
```

Алгоритм Кеннона: основной цикл

```
/* Restore original distribution of a and b */  
MPI_Cart_shift(comm_2d, 0, +mycoords[0], &shiftsource,  
               &shiftdest);  
MPI_Sendrecv_replace(a, nlocal*nlocal, MPI_DOUBLE,  
                     shiftdest, 1, shiftsource, 1, comm_2d, &status);  
  
MPI_Cart_shift(comm_2d, 1, +mycoords[1], &shiftsource,  
               &shiftdest);  
MPI_Sendrecv_replace(b, nlocal*nlocal, MPI_DOUBLE,  
                     shiftdest, 1, shiftsource, 1, comm_2d, &status);
```

■ Алгоритм Фокса

Алгоритм Фокса

$$C_{ij} = A_{i0}B_{0j} + A_{i1}B_{1j} + A_{i2}B_{2j} \dots A_{i,n-1}B_{n-1,j}$$

- Шаг 0


- Процесс(i, j) : $C_{ij} = A_{ii} \times B_{ij}$

- Шаг 1

- Процесс(i, j) : $C_{ij} = C_{ij} + A_{i,i+1} \times B_{i+1,j}$

- Шаг k

- Процесс(i, j) : $C_{ij} = C_{ij} + A_{i,i+k} \times B_{i+k,j}$



Алгоритм Фокса

	i	ii
Stage 0	$a_{00} \rightarrow$	$c_{00} += a_{00}b_{00}$
	$\leftarrow a_{11}$	$c_{01} += a_{00}b_{01}$
	$\leftarrow a_{22}$	$c_{02} += a_{00}b_{02}$
Stage 1	$\leftarrow a_{01}$	$c_{10} += a_{11}b_{10}$
	$\leftarrow a_{12}$	$c_{11} += a_{11}b_{11}$
	$a_{20} \rightarrow$	$c_{12} += a_{11}b_{12}$
Stage 2	$\leftarrow a_{02}$	$c_{20} += a_{22}b_{20}$
	$a_{10} \rightarrow$	$c_{21} += a_{22}b_{21}$
	$\leftarrow a_{21}$	$c_{22} += a_{22}b_{22}$

Алгоритм Фокса

Шаг 1. Широковещательная рассылка диагонального элемента каждой строки матрицы A по всем процессорам своей строки.

Каждый процессор (i, j) выполняет

$$C(i, j) = A(i, i) * B(i, j)$$

Столбец матрицы B циклически сдвигается вверх по своему столбцу, замещая элемент $B(i, j)$.

Шаг 2. Широковещательная рассылка элемента матрицы A , находящегося справа от диагонального, по всем процессорам своей строки.

Каждый процессор (i, j) выполняет

$$C(i, j) = C(i, j) + A(i+1, i) * B(i+1, j)$$

Столбец матрицы B циклически сдвигается вверх по своему столбцу

Шаг k . Широковещательная рассылка очередного $(i+k) \bmod s$ элемента строки матрицы A по всем процессорам своей строки.

Каждый процессор (i, j) выполняет:

$$C(i, j) = C(i, j) + A(i, (i+k) \bmod s) * B((i+k) \bmod s, j)$$

Столбец матрицы B циклически сдвигается вверх по своему столбцу, замещая собой текущий элемент $B(i, j)$

Производительность двухточечных операций MPI

- Простая модель для оценки времени выполнения двухточечных операций:

Время передачи = α + размер сообщения / β

- **Latency** – время запуска обмена = время пересылки нулевого сообщения, не зависит от размера сообщения
- **Bandwith** – пропускная способность, число байт в секунду
- **Цена обмена** = **latency** * **bandwith** – число байт, которые могли бы быть переданы за время запуска обмена.
- Если **размер сообщения** >> **цене обмена** – производительность канала обмена близка к пропускной способности канала
- Если **размер сообщения** == **цене обмена** – производительность равно половине пропускной способности

Latency & Bandwidth

- Модель:
 - в случае коротких сообщений во времени передачи доминирует latency
 - в случае длинных сообщений - bandwidth
- Critical message size = latency * bandwidth

Эффективность алгоритма Кеннона

```
forall i=0 to s-1      ... s = sqrt(p)
    циклический сдвиг строки i матрицы A на i ... t ≤ s*(α + β*n2/p)
forall i=0 to s-1
    циклический сдвиг столбца i матрицы B на i ... t ≤ s*(α + β*n2/p)
for k=0 to s-1
    forall i=0 to s-1 and j=0 to s-1
        C(i,j) = C(i,j) + A(i,j)*B(i,j) ... t = 2*(n/s)3 = 2*n3/p3/2
        left-circular-shift each row of A by 1 ... t = α + β*n2/p
        up-circular-shift each column of B by 1 ... t = α + β*n2/p
```

- ° Общее время **Total Time = 2*n³/p + 4* s*α + 4*β*n²/s**
- ° Эффективность **Parallel Efficiency = 2*n³ / (p * Total Time)**
= 1/(1 + a * 2*(s/n)³ + b * 2*(s/n))
- Стремится к 1 при n/s = n/sqrt(p) = sqrt(data per processor) растёт
- ° Лучше, чем 1D распределение, при котром **Efficiency = 1/(1 + O(p/n))**

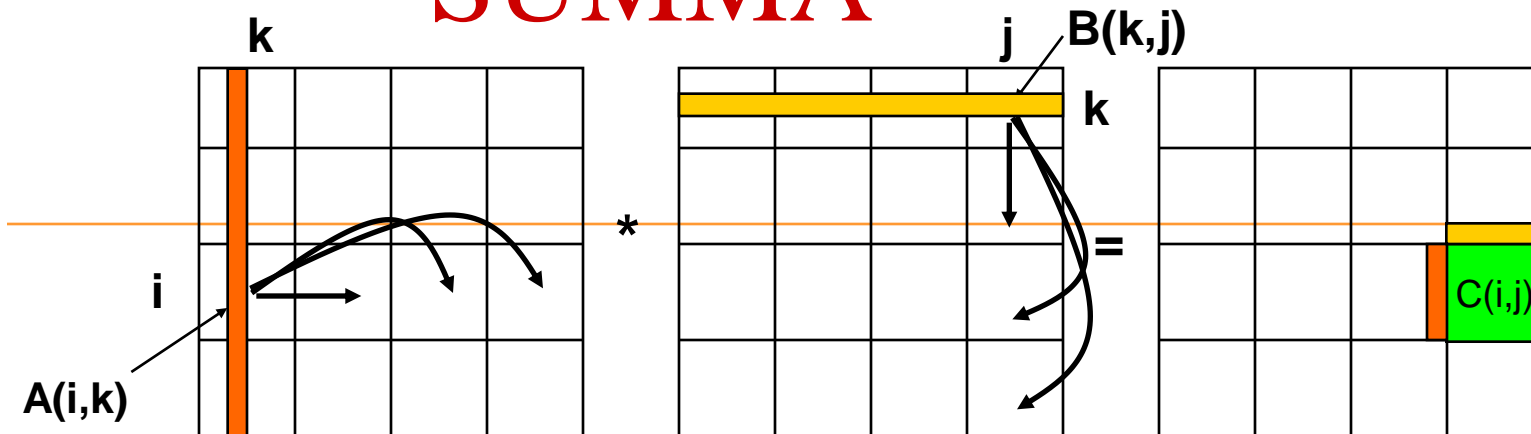
Недостатки алгоритмов Кеннона и Фокса

- Трудно обобщаются для случаев:
 - p не полный квадрат
 - A и B не квадратные
 - Размерности A , B не делятся нацело на $s=\sqrt{p}$
- Требуется дополнительная память для хранения копий блоков

Алгоритм SUMMA

- **SUMMA** = Scalable Universal Matrix Multiply *
 - Менее эффективный, чем алгоритм Кеннона, но проще и легче обобщается
 - на случай разных способов распределения данных
 - Пересылок в $\log p$ раз больше, чем в методе Кеннона
 - требует меньше дополнительной памяти, но в то же время, и больше пересылок
 - Используется на практике в PBLAS = Parallel BLAS
- * R. A. Van De Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. Concurrency: Pract. Ex. , 9(4):255–274, 1997

SUMMA



- Процессорная решетка не обязательно должна быть квадратной: $P = p_r * p_c$
- $b \ll N / \max(p_x, p_y)$
- k – блок с $b \geq 1$ строками или столбцами

$$C(i,j) = C(i,j) + \sum_k A(i,k) * B(k,j)$$

Эффективность SUMMA

- ° Для упрощения предположим, что $s = \sqrt{p}$

for $k=0$ to $n/b-1$

for all $i = 1$ to s ... $s = \sqrt{p}$

owner of $A(i,k)$ broadcasts it to whole processor row

... time = $\log s * (\alpha + \beta * b * n/s)$, используя дерево

for all $j = 1$ to s

owner of $B(k,j)$ broadcasts it to whole processor column

... time = $\log s * (\alpha + \beta * b * n/s)$, используя дерево

Receive $A(i,k)$ into A_{col}

Receive $B(k,j)$ into B_{row}

$C_{myproc} = C_{myproc} + A_{col} * B_{row}$

... time = $2 * (n/s)^2 * b$

- ° Общее время

$$\text{Total time} = 2 * n^3 / p + \alpha * \log p * n / b + \beta * \log p * n^2 / s$$

Эффективность SUMMA

- Total time = $2 \cdot n^3 / p + a \cdot \log p \cdot n / b + b \cdot \log p \cdot n^2 / s$
- Parallel Efficiency =
$$1 / (1 + a \cdot \log p \cdot p / (2 \cdot b \cdot n^2) + b \cdot \log p \cdot s / (2 \cdot n))$$
- \approx такое же слагаемое с b как и в Кенноне, за исключением множителя $\log p$
- (a) член может быть больше, в зависимости от b
Если $b=1$, получим $a \cdot \log p \cdot n$
С ростом b grows to n/s , term shrinks to
 $a \cdot \log p \cdot s$ (log p times Cannon)
- Дополнительная память - $2 \cdot b \cdot n / s$
- Можно изменять b , чтобы регулировать соотношение время выполнения – используемая память

2D параллельные алгоритмы матричного умножения

■ 2D

■ Cannon

- Эффективность = $1/(1 + O(\alpha * (\sqrt{p}/n)^3 + \beta * \sqrt{p}/n))$ – оптимальная
- Трудно обобщать на случай произвольного p , n , блочно-циклического распределения данных

■ SUMMA

- Эффективность = $1/(1 + O(\alpha * \log p * p / (b * n^2) + \beta * \log p * \sqrt{p}/n))$
- Легко обобщается
- b маленькое \Rightarrow меньше памяти, меньше эффективность
- b большое \Rightarrow больше памяти, выше эффективность
- Используется на практике (PBLAS)