Основы программирования вычислительных систем с распределенной памятью

(Занятие 3)

Игорь Николаевич Коньшин

МГУ - 03.07.2012

Распределенная память

- Общая память → распределенная память
- Процессоры имеют свою локальную память
- Доступ к «чужой» памяти через обмены
- MPI – библиотека для обменов

- Большинство компьютеров из TOP-500
- Распределенная память имеется и на GPU

Парадигма

- Доступ к нужным данным есть не всегда
- Нужны обмены
- Обмены достаточно медленные

- *Для общей памяти параллельность «алгоритмическая», а для распределенной – параллельность «по данным»*

// методы линейной алгебры

- Число, вектор, матрица
- СТАТИЧЕСКОЕ распараллеливание
- Распараллеливание циклов
- Крупно-блочный параллелизм

Оценка ускорения

p - количество используемых процессоров

T(p) - время решения задачи на p процессорах

S - ускорение, S=T(1)/T(p)

E - эффективность, E=S/p

La - общее количество арифметических операций алгоритма

Lc - общая длина всех обменов данными для алгоритма

ta - среднее время выполнения одной арифметической операции

tc - среднее время выполнения обмена одним числом

Ta = La ta - время затраченное на арифметику (вычисления)

Tc = Lc tc - время затраченное на коммуникации (обмены)

tau = tc / ta - общая характеристика параллельного компьютера

L = La / Lc - общая характеристика параллельности алгоритма

$$S = S(p) = T(1) / T(p) = Ta / (Ta/p + Tc) = p\,Ta / (Ta + p\,Tc) = p / (1 + p\,Tc/Ta) =$$
$$= p / (1 + p\,(Lc\,tc) / (La\,ta)) = p / (1 + p\,tau / L)$$

Идеальное распараллеливание

(S=p, E=1, linear speedup)

- (a) Сложение двух векторов:

  Zi = Xi + Yi, i=1,...,n.

- (b) Нормализация вектора:

  Xi = a * Xi, i=1,...,n, a = 1 / || X ||

- (c) Операция AXPY (BLAS1):

  Zi = a * Xi + Yi, i=1,...,n.

- *Вычисления независимы, поэтому ресурс параллелизма (максимальное возможное количество процессоров, которые можно задействовать) равен n.*

Скалярное произведение

c = Sum Xi, i=1,...,n.

- La=n/p, Lc~p-1, L=La/Lc~n/((p-1)*p),

  S(L,p,tau) = p/(1+tau/L) = p/(1+(p-1)*p*tau/n).
- Если n=1000000 и tau=10, то

  S(p=1)=1; S(p=100)~90; S(p=1000)~100.

- MPI_AllReduce(...)
- *Суммирование сдваиванием*

MVM: плотная матрица

$$Yi = Aij * Xj,\ \ i,j = 1,...,n$$

- (a) строчная схема хранения:

```
[=]    [== == ==]    [=]
---    ----------    ---
[=] =  [== == ==] *  [=]
---    ----------    ---
[=]    [== == ==]    [=]
```

- Матрица хранится по строкам, вектор точно также (и перед началом операции и после!).
- Дано, n - размерность матрицы, p - количество процессоров (П).
- Перед началом вычислений нужно собрать весь вектор на каждом П.

  La=n*n/p, Lc=(n/p)*(p-1), L=La/Lc=n/(p-1),

  S(L,p,tau)=p/(1+tau/L)=p/(1+(p-1)tau/n)=S(n,p,tau).
- Если n=1000 и tau=10, то S(p=1)=1; S(p=10)~9; S(p=100)~50.

MVM: плотная матрица

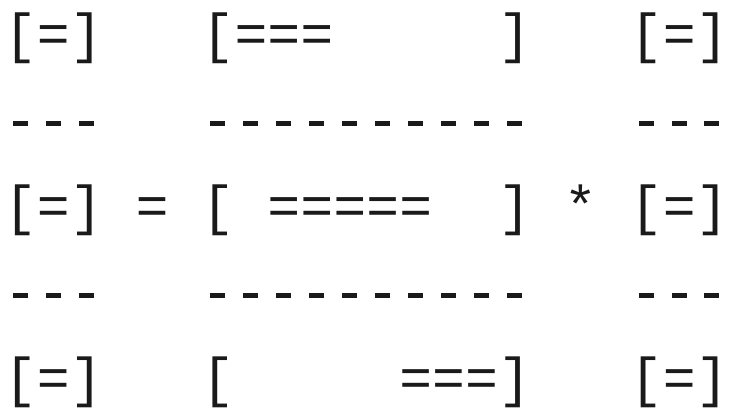
$$Yi = Aij^T * Xj,\ \ i,j = 1,...,n$$

- (b) столбцовая схема хранения матрицы:

```
[=]   [:: :: ::]   [=]
---   [:: :: ::]   ---
[=] = [:: :: ::] * [=]
---   [:: :: ::]   ---
[=]   [:: :: ::]   [=]
```

- Матрица хранится по столбцам, а вектор, естественно, по строкам.
- Для начала вычислений ничего передавать не нужно.
- Каждых процессор вычисляет частичную сумму и передает ее соответствующему П.
- Потом правда каждый П должен будет собрать частичные суммы и получить свою часть результата.
- Расчет ускорения точно такой же.

MVM: ленточная матрица

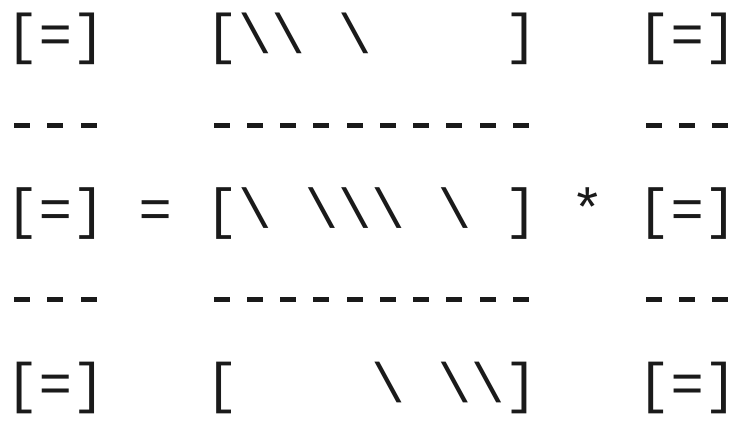


- Пусть r - полуширина ленты.
- Перед началом вычислений нужно от двух соседних П принять r компонент вектора.
- La=(2r+1)*n/p, Lc=2r, L=La/Lc=(2r+1)*n/(p*2r)~n/p,

  S(L,p,tau)=p/(1+tau/L)=p/(1+p*tau/n).
- (!Note! Формула для S почти в точности та же (и почти не зависит от r).

  Уменьшилось количество вычислений, но уменьшилось и количество обменов.)
- Если n=1000000 и tau=10, то S(p=10)~9.999; S(p=1000)~990.

MVM: диагонально-разреженная



- Пусть d - количество диагоналей, а r - как и прежде, полуширина ленты.
- Перед началом вычислений нужно от двух соседних П принять r компонент вектора.
- La=d*n/p, Lc=2r, L=La/Lc=dn/(2rp),

  S(L,p,tau)=p/(1+tau/L)=p/(1+2r*p*tau/(dn)).
- Если n=1000000, r=1000, d=5, tau=10, то S(p=10)~9.6; S(p=100)~70.

Пример MPI программы

```
#include "mpi.h"
#include "stdio.h"
int main(int argc, char** argv)
{
   int np, myid, namelen;
   char proc_name[MPI_MAX_PROCESSOR_NAME];
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &np);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
   MPI_Get_processor_name(proc_name, &namelen);
   printf("I am %d of %d with name: %s\n",
                  myid, np, proc_name);
   MPI_Finalize();
}
```

MPI: основные функции

```
int MPI_Init( int* argc, char*** argv )

int MPI_Finalize( void )

int MPI_Comm_size(
  MPI_Comm comm, //- идентификатор группы
  int* size //- (OUT) размер группы
)

int MPI_Comm_rank(
  MPI_Comm comm, //- идентификатор группы
  int* rank //- (OUT) номер вызывающего процесса в группе comm
)
```

MPI: замер времени

```
double MPI_Wtime( void )


int MPI_Barrier(
  MPI_Comm comm //- идентификатор группы
)


double t;
MPI_Barrier(MPI_COMM_WORLD);
t = MPI_Wtime();
:::::::::
:::::::::
MPI_Barrier(MPI_COMM_WORLD);
t = MPI_Wtime() - t;
```

MPI_Send

```
int MPI_Send(
  void* buf, //- адрес начала буфера посылки сообщения
  int count, //- число передаваемых элементов в сообщении
  MPI_Datatype datatype, //- тип передаваемых элементов
  int dest, //- номер процесса-получателя
  int msgtag, //- идентификатор сообщения
  MPI_Comm comm //- идентификатор группы
)
```

MPI_Recv

```
int MPI_Recv(
  void* buf, //- (OUT) адрес начала буфера приема сообщения
  int count, //- максимальное число элементов в принимаемом сообщении
  MPI_Datatype datatype, //- тип элементов принимаемого сообщения
  int source, //- номер процесса-отправителя
  int msgtag, //- идентификатор принимаемого сообщения
  MPI_Comm comm, //- идентификатор группы
  MPI_Status *status //- (OUT) параметры принятого сообщения
)
```

MPI_Allreduce

```
int MPI_Allreduce(
  void *sbuf, //- адрес начала буфера для аргументов
  void *rbuf, //- (OUT) адрес начала буфера для результата
  int count, //- число аргументов у каждого процесса
  MPI_Datatype datatype, //- тип аргументов
  MPI_Op op, //- идентификатор глобальной операции
  MPI_Comm comm //- идентификатор группы
)

//- op = MPI_MAX,  MPI_MIN,  MPI_SUM,  MPI_PROD
```

Список практических заданий (1)

Параметры // компьютера (nodes x ppn):

- максимальное число nodes
- максимальное число ppn
- максимальная память на 1 node
- максимальная память на 1 ppn

Обмены:

- внутри узла (ppn>1)
- между узлами (ppn=1)
- синхронные обмены "пинг-понг"
- синхронные обмены "чет./нечет."
- асинхронные обмены "по кольцу"
- асинхронные обмены "все всем"

Список практических заданий (2)

Замер времени:

- время одного вызова MPI_Wtime()
- время одного вызова MPI_Barrier()
- время ta одной арифметической операции: y=a*x+y
- время tc передачи одного числа
- коэффициент // эф-ти компьютера: tau=tc/ta
- измерение латентности t0 передачи данных: Tc=t0+tc*Lc

Зависимость скорости обменов от:

- длины сообщения
- количества процессоров
- числа ppn
- типа передачи (синхр./асинхр.)