HPC-3

Распределенная память: MPI

перпендикулярный взгляд

И.Н. Коньшин[1,2,3]

[1]Институт вычислительной математики им. Г.И. Марчука, РАН
[2]Московский физико-технический институт
[3]Сеченовский университет

План

- отличие от случая общей памяти
- библиотека обменов MPI
- оценка параллельной эффективности алгоритма
- примеры программ
- результаты численных экспериментов

Классификация компьютеров

- ***Общая*** память (рабочие станции, персональные компьютеры, ноутбуки, телефоны, ...)
- ***Распределенная*** память (суперкомпьютеры, кластеры, ...)

- *'Разделяемая'* $\longleftrightarrow$ *'Разделенная'*

¡ Давайте перейдем к ***'Распределенной'*** *памяти !*

OpenMP vs. MPI: свойства

OpenMP

- Память общая, распараллеливать просто
- Макро, директивы препроцессора, иногда функции
- OpenMP — это работа по распараллеливанию арифметики
- Оценка //-ной эфф-ти — через закон Амдала

MPI

- Память распределенная (разделенная), распараллеливать сложно
- Только функции — никакого баловства, все серьезно!
- MPI — это работа при распараллеленных данных
- Оценка //-ной эфф-ти — закон нужно придумать самим...

OpenMP vs. MPI: модели работы

OpenMP

- В одной не очень большой комнате над общим делом слаженно работает бригада мастеров (*нити, потоки*)
- У них все данные рядом, на виду, под руками (*общая память*)
- Лишь иногда повздорят кому первому схватить кирпич (*конфликты памяти*)
- Да иногда кто-то окончит свою работу раньше других (*дисбаланс*)

MPI

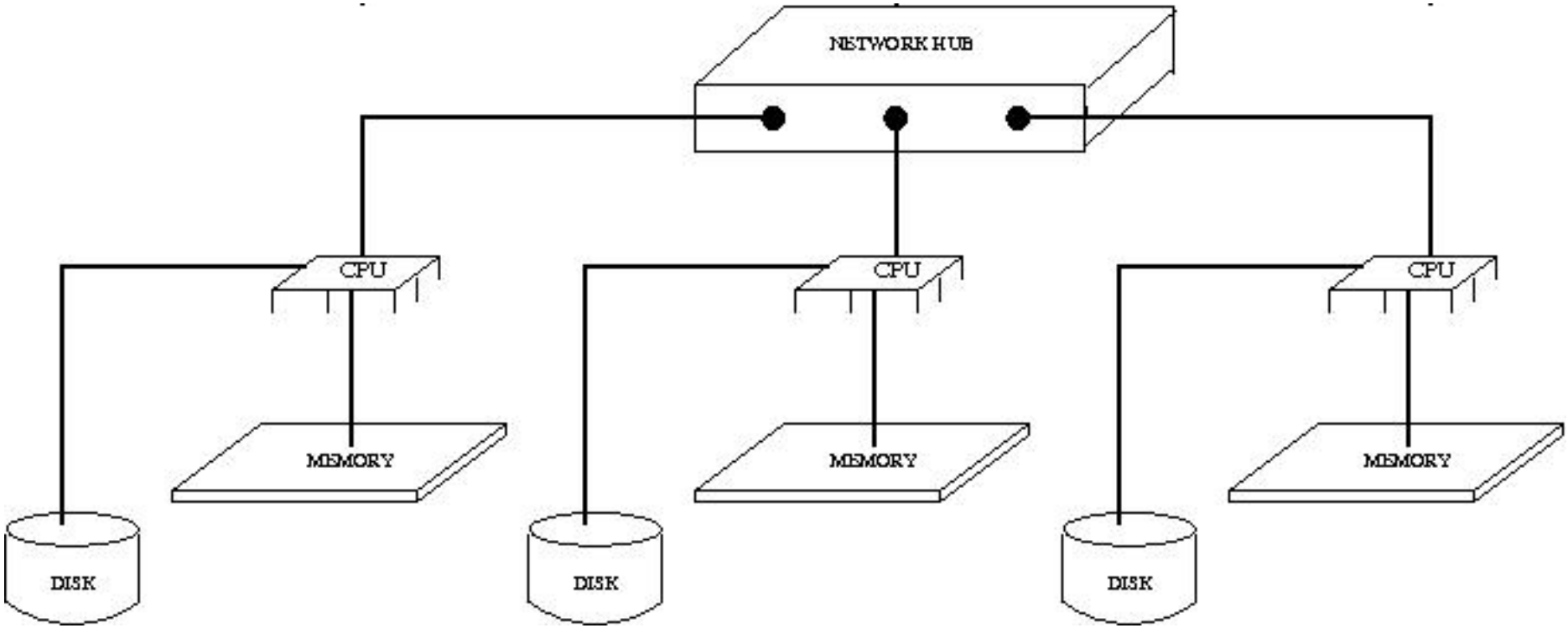
- В одной подводной лодке с маленькой командой: работник (*процесс MPI*) и радист (организует связь с другими *процессами-подлодками*)
- Свои данные под руками, но о данных соседей ничего не известно (знаем только *номера* подлодок)
- Сделать работу со своими данными можно достаточно успешно (*арифметика быстрая*)
- А вот запросить данные (азбукой Морзе) и получить их с соседних подлодок — это очень(!) долго... (*передача данных медленная*)

Традиционный взгляд на MPI:

справочники, примеры применения

А.С.Антонов, Параллельное программирование с использованием технологии MPI, МГУ, Москва, 2004
http://dodo.inm.ras.ru/konshin/HPC/bib/MPI-Antonov.pdf

Распределенная память



MPI — распределенная память

MPI (Message Passing Interface) для C, C++, Fortran

MPI состоит из набора:

- библиотечных функций
- и переменных окружения

1994: MPI 1.0
2015: MPI 3.1

MPI: концепция

- Коммуникаторы:
  - MPI_COMM_WORLD
- Обмены точка-точка:
  - MPI_Send
  - MPI_Recv
  - MPI_Isend, MPI_Irecv
- Коллективные обмены:
  - MPI_Bcast
  - MPI_Gather
  - MPI_Reduce
  - MPI_Allreduce
  - MPI_Barrier
- Специальные функции:
  - MPI_Init
  - MPI_Finalize
  - MPI_Wtime

MPI: самые-самые базовые функции

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

int MPI_Comm_size(
  MPI_Comm comm,
    // group id
  int* size
    // (OUT) group size
);

int MPI_Comm_rank(
  MPI_Comm comm,
    // group id
  int* rank
    // (OUT) number of the calling process in the group
);

double MPI_Wtime( void );

int MPI_Finalize( void );
```

Прием/передача сообщений с блокировкой

MPI_Send

```
int MPI_Send(
  void* buf,
    // the start address of the send message buffer
  int count,
    // the number of transmitted elements
  MPI_Datatype datatype,
    // type of transmitted elements
  int dest,
    // destination process number
  int msgtag,
    // message id
  MPI_Comm comm
    // group id
);
```

MPI_Recv

```
int MPI_Recv(
  void* buf,
    // (OUT) the start address of the receive message buffer
  int count,
    // the maximal number of received elements
  MPI_Datatype datatype,
    // type of transmitted elements
  int source,
    // sender process number
  int msgtag,
    // message id
  MPI_Comm comm,
    // group id
  MPI_Status *status
    // (OUT) received message parameters
);
```

Прием/передача сообщений без блокировки

MPI_Isend

```
int MPI_Isend(
  void* buf,
    // the start address of the send message buffer
  int count,
    // the number of transmitted elements
  MPI_Datatype datatype,
    // type of transmitted elements
  int dest,
    // destination process number
  int msgtag,
    // message id
  MPI_Comm comm,
    // group id
  MPI_Request *request
    // (OUT) asynchronous transfer identifier
);
```

MPI_Irecv

```
int MPI_Irecv(
  void* buf,
    // (OUT) the start address of the receive message buffer
  int count,
    // the maximal number of received elements
  MPI_Datatype datatype,
    // type of transmitted elements
  int source,
    // sender process number
  int msgtag,
    // message id
  MPI_Comm comm,
    // group id
  MPI_Request *request
    // (OUT) asynchronous transfer identifier
);
```

MPI_Probe, MPI_Get_count

```
int MPI_Probe(
  int source,
    // sender process number or MPI_ANY_SOURCE
  int msgtag,
    // expected message identifier or MPI_ANY_TAG
  MPI_Comm comm,
    // group id
  MPI_Status *status
    // (OUT) parameters of the detected message
);

int MPI_Get_count(
  MPI_Status *status,
    // parameters of the received message
  MPI_Datatype datatype,
    // element type of the received message
  int *count
    // (OUT) number of message elements
);
```

MPI_Wait, MPI_Waitall

```
int MPI_Wait(
  MPI_Request *request,
    // identifier for asynchronous send or receive
  MPI_Status *status
    // (OUT) message parameters
);

int MPI_Waitall(
  int count,
    // number of identifiers
  MPI_Request requests[],
    // array of asynchronous send or receive identifiers
  MPI_Status statuses[]
    // (OUT) message parameters
);
```

...и более продвинутые функции для приема/передачи без блокировки

MPI_Wait_any, MPI_Wait_some

```
int MPI_Waitany(
  int count,
    // number of identifiers
  MPI_Request requests[],
    // array of asynchronous receive or transmit identifiers
  int *index,
    // number of the completed exchange operation
  MPI_Status *status
    // message parameters
);

int MPI_Waitsome(
  int incount,
    // number of identifiers
  MPI_Request requests[],
    // array of asynchronous receive or transmit identifiers
  int *outcount,
    // number of identifiers of completed exchange
        operations
  int indexes[],
    // array of numbers of completed exchange operations
  MPI_Status statuses[]
    // parameters of completed messages
);
```

MPI_Test

```
int MPI_Test( MPI_Request *request, int *flag, MPI_Status *
status );
```

- `request` — идентификатор асинхронного приема или передачи
- (OUT) `flag` — признак завершенности операции обмена
- (OUT) `status` — параметры сообщения

Проверка завершенности асинхронных процедур `MPI_Isend` или `MPI_Irecv`, ассоциированных с идентификатором `request`.
В параметре `flag` возвращается значение `1`, если соответствующая операция завершена, и значение `0` в противном случае.
Если завершена процедура приема, то атрибуты и длину полученного сообщения можно определить обычным образом с помощью параметра `status`.

MPI_Testall

```
int MPI_Testall( int count, MPI_Request requests[], int *
flag, MPI_Status statuses[] );
```

- `count` — число идентификаторов
- `requests` — массив идентификаторов асинхронного приема или передачи
- (OUT) `flag` — признак завершенности операций обмена
- (OUT) `statuses` — параметры сообщений

В параметре `flag` возвращает значение `1`, если все операции, ассоциированные с указанными идентификаторами, завершены (с указанием параметров сообщений в массиве `statuses`).
В противном случае возвращается `0`, а элементы массива `statuses` неопределены.

MPI_Testany

```
int MPI_Testany( int count, MPI_Request requests[], int *
index, int *flag, MPI_Status *status );
```

- `count` — число идентификаторов
- `requests` — массив идентификаторов асинхронного приема или передачи
- (OUT) `index` — номер завершенной операции обмена
- (OUT) `flag` — признак завершенности операции обмена
- (OUT) `status` — параметры сообщения

Если к моменту вызова функции хотя бы одна из операций обмена завершилась, то в параметре `flag` возвращается значение `1`, `index` содержит номер соответствующего элемента в массиве `requests`, а `status` — параметры сообщения.

MPI_Testsome

```
int MPI_Testsome( int incount, MPI_Request requests[], int *
outcount, int indexes[], MPI_Status statuses[] );
```

- `incount` — число идентификаторов
- `requests` — массив идентификаторов асинхронного приема или передачи
- (OUT) `outcount` — число идентификаторов завершившихся операций обмена
- (OUT) `indexes` — массив номеров завершившихся операции обмена
- (OUT) `statuses` — параметры завершившихся операций

Данная функция работает так же, как и `MPI_Waitsome`, за исключением того, что возврат происходит немедленно.
Если ни одна из указанных операций не завершилась, то значение `outcount` будет равно нулю.

MPI_Iprobe

```
int MPI_Iprobe( int source, int msgtag, MPI_Comm comm, int *
flag, MPI_Status *status );
```

- `source` — номер процесса-отправителя или `MPI_ANY_SOURCE`
- `msgtag` — идентификатор ожидаемого сообщения или `MPI_ANY_TAG`
- `comm` — идентификатор группы
- (OUT) `flag` — признак завершенности операции обмена
- (OUT) `status` — параметры обнаруженного сообщения

Получение информации о поступлении и структуре ожидаемого сообщения без блокировки.
В параметре `flag` возвращает значение `1`, если сообщение с подходящими атрибутами уже может быть принято (в этом случае ее действие полностью аналогично `MPI_Probe`), и значение `0`, если сообщения с указанными атрибутами еще нет.

Объединение запросов на взаимодействие: *Зачем?*

Несколько приемов/посылок могут быть совмещены в одном вызове MPI чтобы снизить накладные расходы при передаче информации между процессом и сетевым контроллером

MPI_Send_init

```
int MPI_Send_init( void *buf, int count, MPI_Datatype
datatype, int dest, int msgtag, MPI_Comm comm, MPI_Request *
request );
```

- `buf` — адрес начала буфера посылки сообщения
- `count` — число передаваемых элементов в сообщении
- `datatype` — тип передаваемых элементов
- `dest` — номер процесса-получателя
- `msgtag` — идентификатор сообщения
- `comm` — идентификатор группы
- (OUT) `request` — идентификатор асинхронной передачи

Формирование запроса на выполнение пересылки данных.
Все параметры точно такие же, как и у функции `MPI_Isend`, однако в отличие от нее пересылка не начинается до вызова функции `MPI_Startall`.

MPI_Recv_init

```
int MPI_Recv_init( void *buf, int count, MPI_Datatype
datatype, int source, int msgtag, MPI_Comm comm, MPI_Request
 *request );
```

- (OUT) `buf` — адрес начала буфера приема сообщения
- `count` — число принимаемых элементов в сообщении
- `datatype` — тип принимаемых элементов
- `source` — номер процесса-отправителя
- `msgtag` — идентификатор сообщения
- `comm` — идентификатор группы
- (OUT) `request` — идентификатор асинхронного приема

Формирование запроса на выполнение приема данных.
Все параметры точно такие же, как и у функции `MPI_Irecv`, однако в отличие от нее реальный прием не начинается до вызова функции `MPI_Startall`.

MPI_Startall

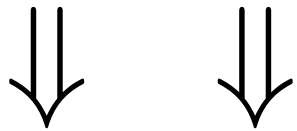
```
MPI_Startall( int count, MPI_Request *requests );
```

- `count` — число запросов на взаимодействие
- (OUT) `requests` — массив идентификаторов приема/передачи

Запуск всех отложенных взаимодействий, ассоциированных вызовами функций `MPI_Send_init` и `MPI_Recv_init` с элементами массива запросов `requests`.
Все взаимодействия запускаются в режиме без блокировки, а их завершение можно определить обычным образом с помощью функций `MPI_Wait` и `MPI_Test`.

Будет ли эффект от объединения запросов?

```
MPI_Isend(..);   ...; MPI_Isend(..);
MPI_Irecv(..);   ...; MPI_Irecv(..);
...
MPI_Wait(..);
```



```
MPI_Send_init(..);   ...; MPI_Send_init(..);
MPI_Recv_init(..);   ...; MPI_Recv_init(..);
MPI_Startall(..);
...
MPI_Wait(..);
```

Ответ полностью зависит от реализации MPI.
Бывает “да”, а бывает и “нет”.

Совмещенные прием/передача сообщений

$$\begin{array}{r} \text{Send} \\ \text{Receive} \end{array} \quad \Longleftrightarrow \quad \begin{array}{l} \text{Receive} \\ \text{Send} \end{array}$$

MPI_Sendrecv

```
int MPI_Sendrecv( void *sbuf, int scount, MPI_Datatype stype
, int dest, int stag, void *rbuf, int rcount, MPI_Datatype
rtype, int source, MPI_Datatype rtag, MPI_Comm comm,
MPI_Status *status );
```

- `sbuf` — адрес начала буфера посылки сообщения
- `scount` — число передаваемых элементов в сообщении
- `stype` — тип передаваемых элементов
- `dest` — номер процесса-получателя
- `stag` — идентификатор посылаемого сообщения
- (OUT) `rbuf` — адрес начала буфера приема сообщения
- `rcount` — число принимаемых элементов сообщения
- `rtype` — тип принимаемых элементов
- `source` — номер процесса-отправителя
- `rtag` — идентификатор принимаемого сообщения
- `comm` — идентификатор группы
- (OUT) `status` — параметры принятого сообщения

Коллективные взаимодействия процессов

В операциях коллективного взаимодействия процессов участвуют ***все*** процессы коммуникатора.
Соответствующая процедура должна быть вызвана ***каждым*** процессом, быть может, со своим набором параметров.
Возврат из процедуры коллективного взаимодействия может произойти в тот момент, когда участие процесса в данной операции уже закончено.
Как и для ***блокирующих*** процедур, возврат означает то, что разрешен свободный доступ к буферу приема или посылки, но не означает ни того, что операция завершена другими процессами, ни даже того, что она ими начата (если это возможно по смыслу операции).

MPI_Bcast

```
int MPI_Bcast( void *buf, int count, MPI_Datatype datatype,
int source, MPI_Comm comm );
```

- (OUT) `buf` — адрес начала буфера посылки/приема сообщения
- `count` — число передаваемых элементов в сообщении
- `datatype` — тип передаваемых элементов
- `source` — номер рассылающего процесса
- `comm` — идентификатор группы

Широковещательная рассылка сообщения от процесса `source` всем процессам, включая рассылающий процесс.

При возврате из процедуры содержимое буфера `buf` процесса `source` будет скопировано в локальный буфер процесса.
Значения параметров `count`, `datatype` и `source` должны быть одинаковыми у всех процессов.

MPI_Gather

```
int MPI_Gather( void *sbuf, int scount, MPI_Datatype stype,
void *rbuf, int rcount, MPI_Datatype rtype, int dest,
MPI_Comm comm );
```

- `sbuf` — адрес начала буфера посылки
- `scount` — число элементов в посылаемом сообщении
- `stype` — тип элементов отсылаемого сообщения
- (OUT) `rbuf` — адрес начала буфера сборки данных
- `rcount` — число элементов в принимаемом сообщении
- `rtype` — тип элементов принимаемого сообщения
- `dest` — номер процесса, на котором происходит сборка данных
- `comm` — идентификатор группы
- (OUT) `ierror` — код ошибки

Сборка данных со всех процессов в буфере `rbuf` процесса `dest`. Каждый процесс, включая `dest`, посылает содержимое своего буфера `sbuf` процессу `dest`.

Собирающий процесс сохраняет данные в буфере `rbuf`, располагая их в порядке возрастания номеров процессов.

MPI_Allreduce

```
int MPI_Allreduce( void *sbuf, void *rbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm );
```

- `sbuf` — адрес начала буфера для аргументов
- (OUT) `rbuf` — адрес начала буфера для результата
- `count` — число аргументов у каждого процесса
- `datatype` — тип аргументов
- `op` — идентификатор глобальной операции
- `comm` — идентификатор группы

Выполнение `count` ***глобальных операций*** `op` с возвратом `count` результатов во всех процессах в буфере `rbuf`.

Значения параметров `count` и `datatype` у всех процессов должны быть одинаковыми.

Из соображений эффективности реализации предполагается, что операция `op` обладает свойствами ***ассоциативности*** и ***коммутативности***.

MPI_Reduce

```
int MPI_Reduce( void *sbuf, void *rbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm );
```

- `sbuf` — адрес начала буфера для аргументов
- (OUT) `rbuf` — адрес начала буфера для результата
- `count` — число аргументов у каждого процесса
- `datatype` — тип аргументов
- `op` — идентификатор глобальной операции
- `root` — процесс-получатель результата
- `comm` — идентификатор группы

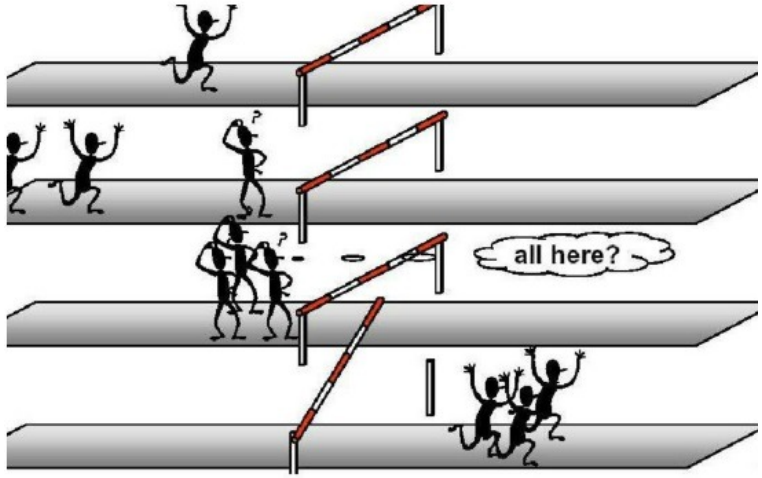
Функция аналогична `MPI_Allreduce`, только результат будет записан в буфер `rbuf` только у процесса `root`.

Синхронизация процессов

MPI_Barrier

```
int MPI_Barrier( MPI_Comm comm );
```

- `comm` — идентификатор группы



Блокирует работу процессов, вызвавших данную процедуру, до тех пор, пока все оставшиеся процессы группы `comm` также не выполнят эту процедуру.

¡¿ Зачем это может пригодиться ?!

Полностью опускаем работу с группами процессов...

- `MPI_Comm_split(..)` — образование новых коммуникаторов
- `MPI_Comm_free(..)` — освобождение ненужных коммуникаторов

MPI: предопределенные типы

`MPI_Status` — структура; атрибуты сообщений; содержит три обязательных поля:

- `MPI_Source` — номер процесса отправителя
- `MPI_Tag` — идентификатор сообщения
- `MPI_Error` — код ошибки

`MPI_Request` — системный тип; идентификатор операции посылки–приема сообщения

`MPI_Comm` — системный тип; идентификатор группы (коммуникатора)

- `MPI_COMM_WORLD`

MPI: некоторые константы

- `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD`
- `MPI_ANY_SOURCE`, `MPI_ANY_TAG`
- `MPI_SUCCESS`

MPI 1.0: базовые функции и замер времени

```c
#include <mpi.h>

int main (int argc, char **argv)
{
    int id, np;
    double t;

    MPI_Init (&argc, &argv);                /* starts MPI */
    MPI_Comm_rank (MPI_COMM_WORLD, &id); /* get current process id */
    MPI_Comm_size (MPI_COMM_WORLD, &np); /* get total number of processes */

    MPI_Barrier(MPI_COMM_WORLD);            /* synchronize processes */
    t = MPI_Wtime();
    /* ... some code ... */
    MPI_Barrier(MPI_COMM_WORLD);
    t = MPI_Wtime() - t;                    /* get wall-clock time */

    MPI_Finalize();                         /* finalize MPI */
    return 0;
}
```

```
$ mpicc mpi_base.c
$ mpirun -np 4 a.out
```

MPI: практические задания

(x.c) Для одного из примеров `mpi_daxpy.c` и `mpi_norm.c`:

- проанализировать отличия от распараллеливания на OpenMP;
- рассмотреть вариант комбинированного распараллеливания MPI+OpenMP (обозначение `mix`);
- с помощью скрипта `qs_mix` найти оптимальное для данной операции и данного компьютера количество потоков OpenMP и процессов MPI.

(y.c) По аналогии с примером на OpenMP из файла `Notes_omp.txt` для вычисления числа π:

- написать программу с распараллеливанием по MPI;
- найти и распечатать ошибку вычисления числа π;
- объединить в одной программе распараллеливание по OpenMP и MPI, используя скрипт `qs_mix`, найти оптимальное для данного компьютера количество потоков OpenMP и процессов MPI.

(z.c) Найти характеристики компьютера:

- среднее время τ_a одной арифметической операции, например, `y[i] += a * x[i]` в двойной точности;
- среднее время τ_c передачи одного числа в двойной точности на другой процесс (при достаточно длинном сообщении);
- найти отношение $\tau = \tau_c/\tau_a$.

MPI — версии

- 1994: MPI 1.0 (Send+Recv, асинхронные, парные двусторонние, коллективные обмены)
- 1995: MPI 1.1
- 1997: MPI 2.0 (Put+Get = односторонние коммуникации, порождение MPI-процессов и нитей-тредов, расширенные коллективные операции в рамках нескольких коммуникаторов)
- 2012: MPI 3.0 (MPI_Iallreduce, MPI_Ibarrier, ...)
- 2015: MPI 3.1

MPI Forum — http://www.mpi-forum.org/

MPI — реализации

- MPICH — https://ru.wikipedia.org/wiki/MPICH
- OpenMPI — https://www.open-mpi.org/
- Intel MPI — http://software.intel.com/en-us/intel-mpi-library/
- MPJ Express — http://www.mpj-express.org/ MPI на Java

Сейчас OpenMPI 3.1 содержит более 400 функций:

- MPI_Comm* (37)
- MPI_File* (61)
- MPI_Group* (14)
- MPI_Win* (39)
- MPIX_* (23) из будущего MPI 4.0

MPI 1.0: MPI_Allreduce

```
int MPI_Allreduce(
  void *sbuf,
    // address of the beginning of the buffer
  void *rbuf,
    // (OUT) address of the beginning of the buffer
  int count,
    // number of arguments for each process
  MPI_Datatype datatype,
    // type of arguments
  MPI_Op op,
    // global operation identifier
  MPI_Comm comm
    // group id
);
```

MPI 3.0: MPI_Iallreduce

```
#include <mpi.h>

int MPI_Allreduce(
  const void *sendbuf, void *recvbuf,
  int count, MPI_Datatype datatype,
  MPI_Op op, MPI_Comm comm
);

int MPI_Iallreduce(
  const void *sendbuf, void *recvbuf,
  int count, MPI_Datatype datatype,
  MPI_Op op, MPI_Comm comm,
  MPI_Request *request // <---
);
```

MPI 4.0: барьеры

```
// MPI 1.0
    MPI_Barrier(MPI_COMM_WORLD);

// MPI 4.0
    MPI_Request req;
    MPIX_Ibarrier_init(MPI_COMM_WORLD, &req);
    .....
    MPI_Wait(&req, MPI_STATUS_IGNORE);
    MPI_Request_free(&req);

// MPI 4.0
    MPI_Request req;
    MPIX_Barrier_init(MPI_COMM_WORLD, MPI_INFO_NULL, &req);
    MPI_Start(&req);
    .....
    MPI_Wait(&req, MPI_STATUS_IGNORE);
    MPI_Request_free(&req);
```

Немного теории...

Как оценить эффективность MPI программы?

Ведь если считать, что ***все*** операции распараллелены,
то закон Амдала применять бессмысленно.

Где же у нас может быть причина недостаточной эффективности?

¡ Обмены !

Распределенная память (обмены MPI)

$$T_c = \tau_0 + \tau_c L_c$$

τ_0 — время инициализации сообщения
τ_c — скорость передачи сообщения (т.е. время передачи для сообщения единичной длины)
T_c — время передачи сообщения длины L_c

Для простоты положим $\tau_0 = 0$, тогда

$$T_c = \tau_c L_c$$

Аналогично,

$$T_a = \tau_a L_a$$

τ_a — время выполнения одной (характерной) арифметической операции
L_a — общее количество арифметических операций в алгоритме

Оценка //-ной эфф-ти (распределенная память) [ИК, 2012]

Пусть

$$\tau = \frac{\tau_c}{\tau_a}, \qquad L = \frac{L_c}{L_a}$$

являются характеристиками «параллелизма», соответственно, используемого ***компьютера*** и исследуемого ***алгоритма***

Тогда ***ускорение***:

$$S = S(p) = \frac{T(1)}{T(p)} = \frac{T_a}{\frac{T_a}{p} + \frac{T_c}{p}} = \frac{pT_a}{T_a + T_c}$$

$$= \frac{p}{1 + \frac{T_c}{T_a}} = \frac{p}{1 + \frac{\tau_c L_c}{\tau_a L_a}} = \frac{p}{1 + \tau L}$$

и ***//-ная эфф-ть***:

$$E = \frac{S}{p} = \frac{1}{1 + \tau L}$$

Условия наилучшей применимости оценок

- в отличие от формулы закона Амдала, здесь считается, что распараллелены ***все*** вычисления, а полностью последовательная часть алгоритма отсутствует ($\sigma = 0$);
- задержки в вычислениях происходят ***только*** из-за обменов, и алгоритмы с синхронными обменами больше подходят для этой модели;
- процессоры загружаются ***равномерно*** (или, другими словами, вычисления сбалансированы);
- вычислительные узлы являются ***однородными***;
- скорость арифметических вычислений τ_a не зависит от p;
- скорость передачи сообщений τ_c также ***независима*** от p (этот менее очевидный факт означает масштабируемость коммуникационной сети используемого компьютера).

Примеры:

Применение полученных оценок для некоторых алгоритмов ***линейной алгебры***

MPI mvm: $\vec{y} = A \cdot \vec{x}$ (Оценка ускорения)

$$Y_i = \sum_{j=1}^{n} A_{ij} X_j, \quad i = 1, ..., n$$

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

¡ Посылаем всем свою часть x и собираем у себя глобальный X !

Пусть n — размерность матрицы, тогда:

$$L_a = n^2/p, \quad L_c = (n/p)(p-1), \quad L = L_c/L_a = (p-1)/n$$

$$S = p/(1 + (p-1)\tau/n).$$

Если $n = 1000$ и $\tau = 10$, получаем

$$S(p=1) = 1, \qquad S(p=10) \approx 9, \qquad S(p=100) \approx 50.$$

MPI mvm: $\vec{y} = A^T \cdot \vec{x}$ (транспонированная матрица)

$$Y_i = \sum_{j=1}^{n} A_{ij}^T X_j, \quad i = 1, ..., n$$

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

¡ Рассылаем локальные частичные суммы Y и собираем локальный y !

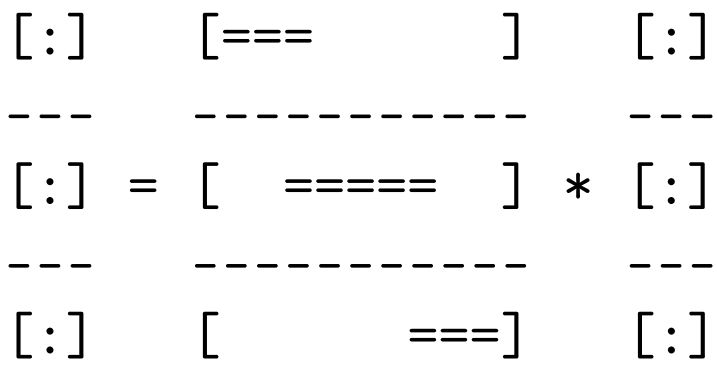
Пусть n — размерность матрицы, тогда:

$$L_a = n^2/p, \quad L_c = (n/p)(p-1), \quad L = L_c/L_a = (p-1)/n$$

$$S = p/(1 + (p-1)\tau/n).$$

¡ Алгоритмы разные, а оценки одинаковые !!

MPI mvm: $\vec{y} = A \cdot \vec{x}$ (ленточная матрица)



¡ Посылаем соседям локальные перекрытия вектора x !

Пусть n — размерность матрицы и r полуширина матрицы, тогда

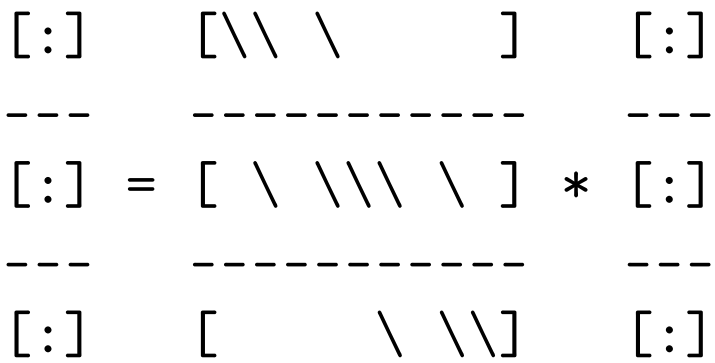
$$L_a = (2r+1)n/p, \quad L_c = 2r(p-1)/p, \quad L = (2r/(2r+1))(p-1)/n \approx (p-1)/n$$

$$S \approx p/(1 + (p-1)\tau/n).$$

¡ То же ускорение что и для плотной матрицы !!

¡ Ускорение не зависит от полуширины ленты матрицы r !!

MPI mvm: $\vec{y} = A \cdot \vec{x}$ (разреженная матрица)



¡ Посылаем соседям локальные перекрытия вектора x !

Пусть n — размерность матрицы, r — полуширина, d — количество диагоналей, тогда

$$L_a = dn/p, \quad L_c = 2r(p-1)/p, \quad L = 2r(p-1)/(dn)$$

$$S = p/(1 + 2r(p-1)\tau/(dn)).$$

¡ Ускорение гораздо меньше чем для ленточной матрицы !!

Примеры:

- daxpy: $\alpha \cdot \vec{x} + \vec{y}$
- norm: $\|\vec{x}\|$
- mvm: $\vec{y} = A \cdot \vec{x}$

Увидим исходные коды и результаты // расчетов на MPI

Кластер ИВМ РАН http://cluster2.inm.ras.ru/

Вычислительные сегменты кластера ИВМ РАН:

- x6core
- x8core
- x10core
- x12core
- x20core

Сегмент **x6core:**

- Compute Node Asus RS704D-E6;
- 12 ядер (два 6-ядерных процессора Intel Xeon X5650@2.67 ГГц);
- Оперативная память: 24 Гб.;
- Операционная система: SUSE Linux Enterprise Server 11 SP1 (x86_64);
- Коммутационная сеть: Mellanox Infiniband QDR 4x.

Для сборки кода использовался компилятор Intel C версии 4.0.1 с библиотекой Intel MPI версии 5.0.3.

Три варианта реализации **daxpy**

1) OMP daxpy: $\alpha \cdot \vec{x} + \vec{y}$

```
#include <omp.h>

int main()
{
    t = omp_get_wtime();
#pragma omp parallel for
    for (i=0; i<N; i++) y[i] += a * x[i];
    t = omp_get_wtime() - t;
}

$ gcc -fopenmp omp_daxpy.c
$ export OMP_NUM_THREADS=4 ; a.out
```

2) MPI daxpy: $\alpha \cdot \vec{x} + \vec{y}$

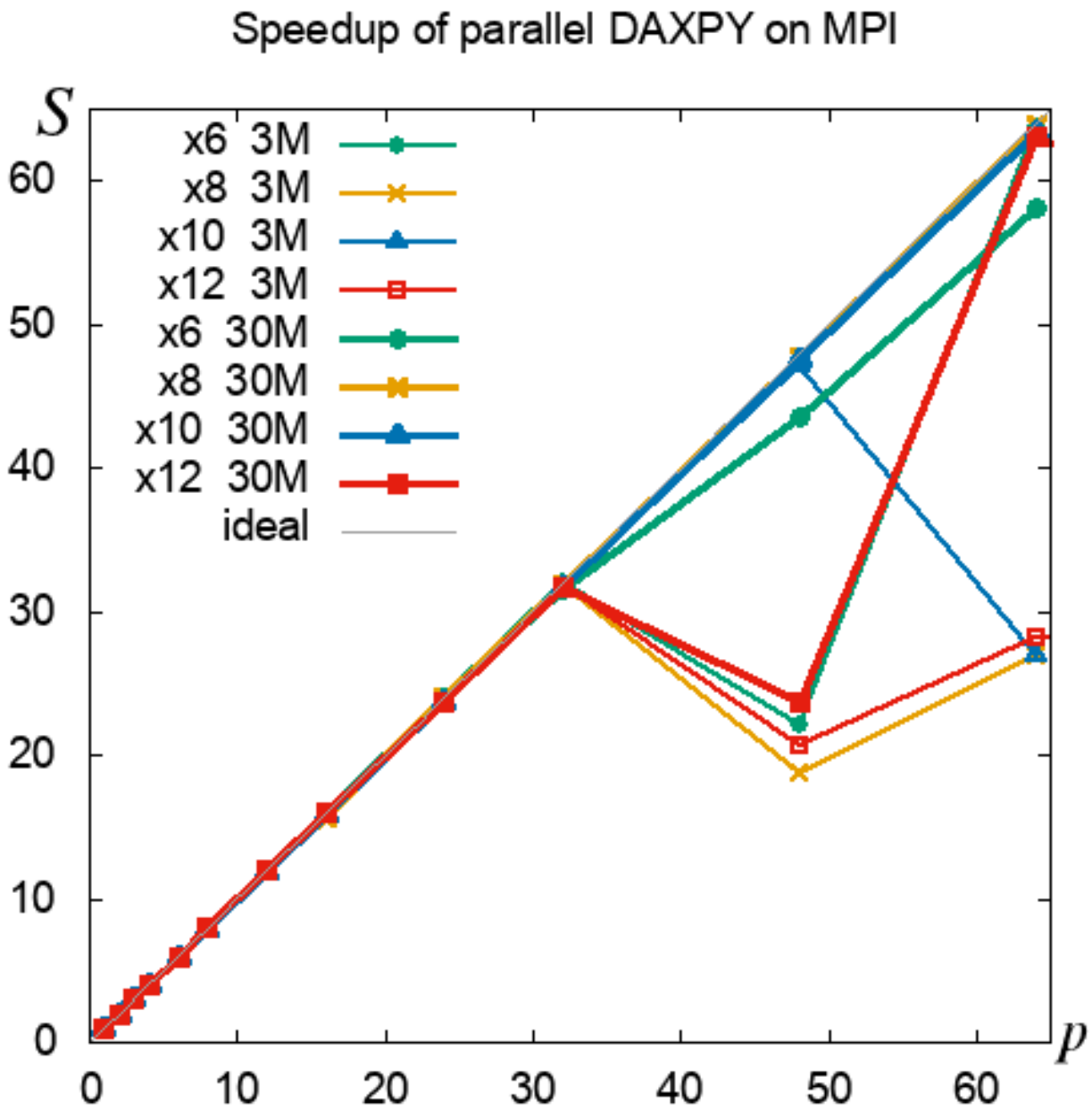
```
#define N 1000000 /* global vector dimension */

    double a, x[N/np], y[N/np], t, perf;

    MPI_Barrier(MPI_COMM_WORLD);
    t = MPI_Wtime();
    for (i=0; i<N/np; i++)
        y[i] += a * x[i];
    MPI_Barrier(MPI_COMM_WORLD);
    t = MPI_Wtime() - t;
    perf = N / t * 1e-6;
    if (id == 0)
        printf("MPI daxpy: N=%d np=%d time=%lf perf=%lf MFLOPS
            \n", N, np, t, perf);


$ mpicc -O mpi_daxpy.c
$ mpirun -np 10 a.out
N=1000000 np=10 time=0.01 perf=100.0 MFLOPS
```

3) MPI+OMP daxpy: $\alpha \cdot \vec{x} + \vec{y}$

```
#define N 1000000 /* global vector dimension */

    double a, x[N/np], y[N/np], t, perf;

    MPI_Barrier(MPI_COMM_WORLD);
    t = MPI_Wtime();
#pragma omp parallel for
    for (i=0; i<N/np; i++)
        y[i] += a * x[i];
    MPI_Barrier(MPI_COMM_WORLD);
    t = MPI_Wtime() - t;
    perf = N / t * 1e-6;


$ mpicc -openmp -O mix_daxpy.c
$ export OMP_NUM_THREADS=4; mpirun -np 4 ./a.out
```

Результаты расчетов для MPI

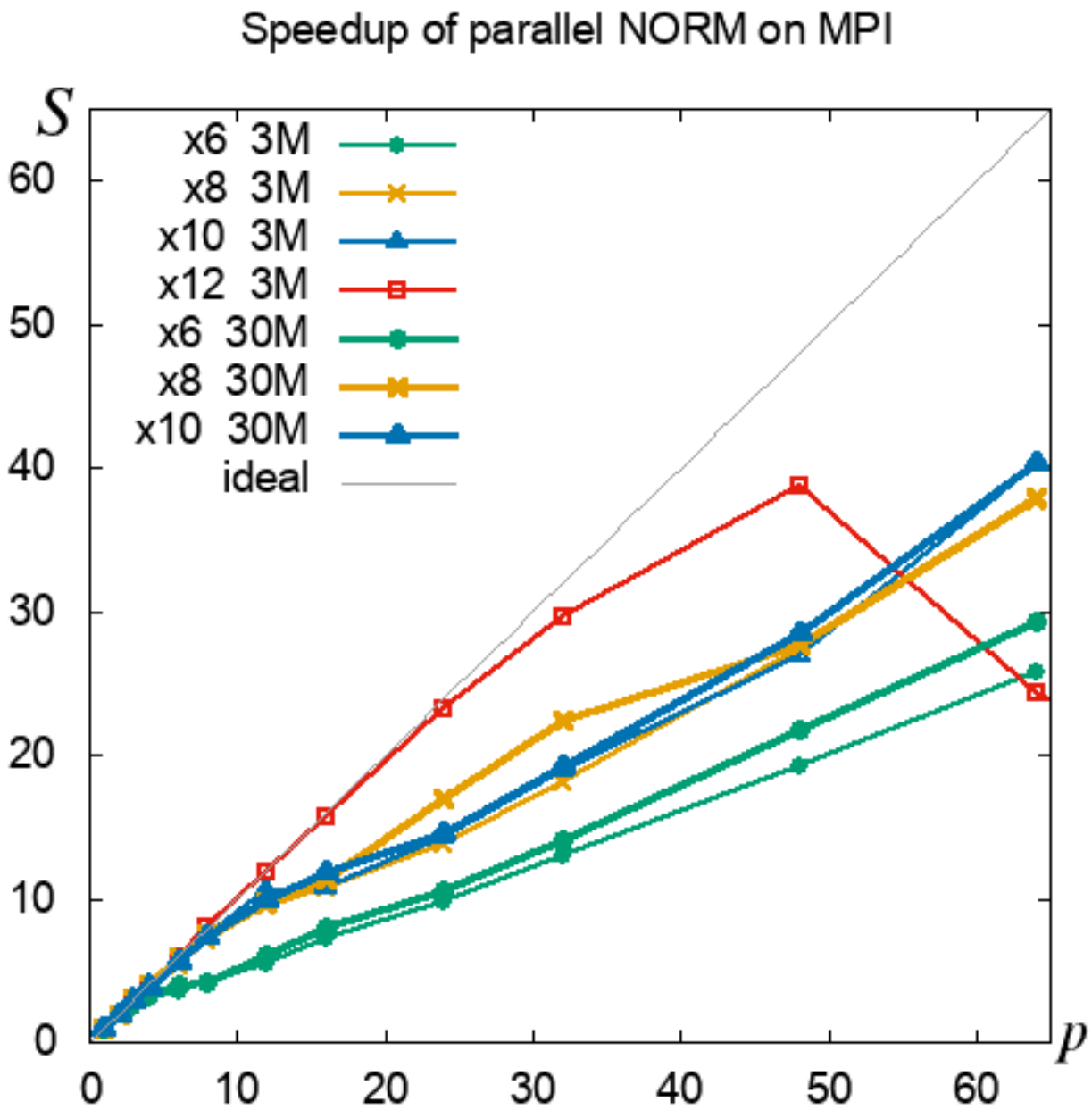
MPI daxpy: $\alpha \cdot \vec{x} + \vec{y}$



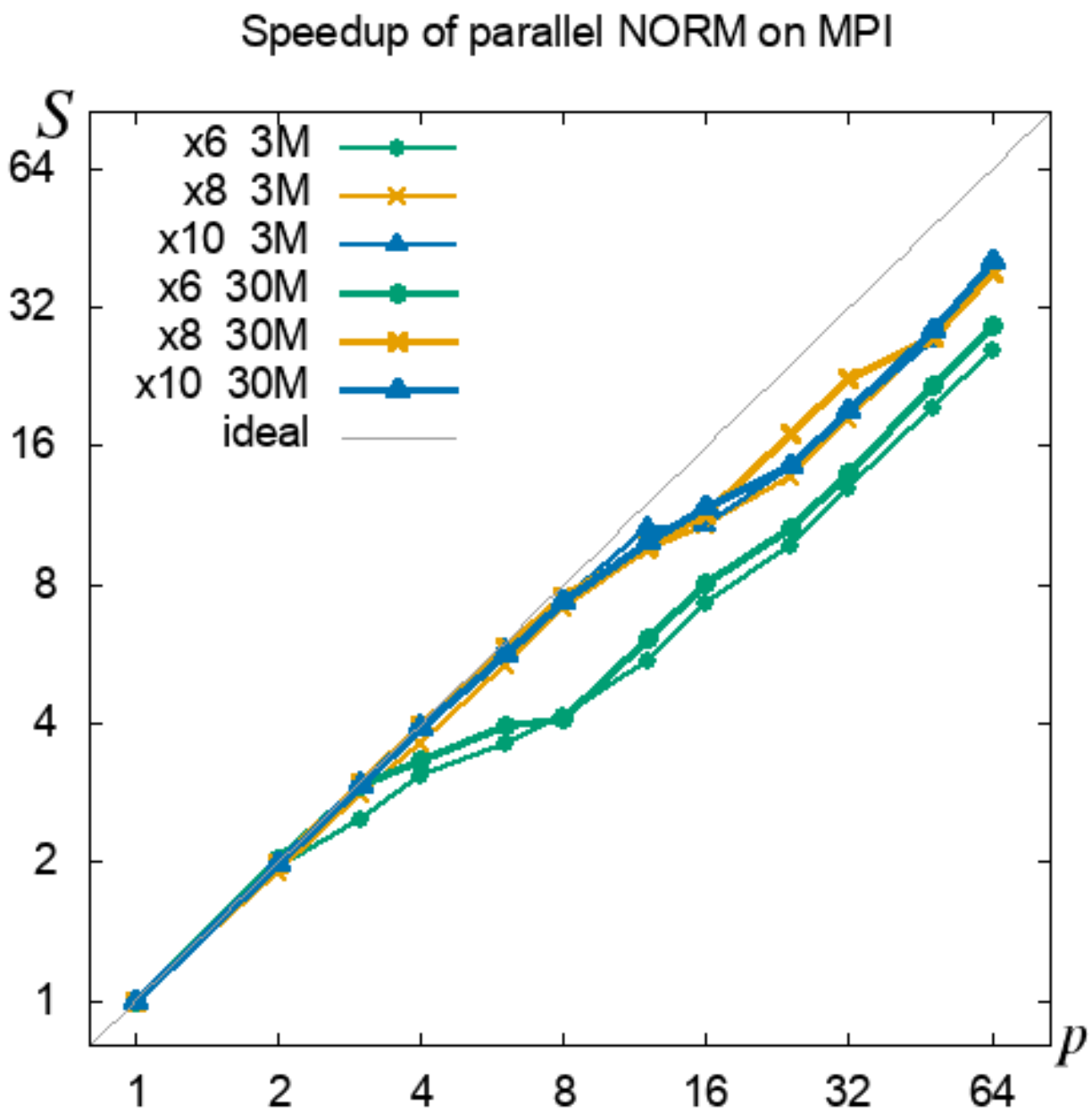
MPI norm: $\|\vec{x}\|$

```
#define N 1000000 /* global vector dimension */

    double sum, tmp, x[N/np], t, perf;

    MPI_Barrier(MPI_COMM_WORLD);
    t = MPI_Wtime();
    sum = 0.0;
    for (i=0; i<N/np; i++)
        sum += x[i] * x[i];
    tmp = sum;
    MPI_Allreduce(&tmp, &sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    sum = sqrt(sum);
    MPI_Barrier(MPI_COMM_WORLD);
    t = MPI_Wtime() - t;
    perf = 1e-6 * N / t;
    if (id == 0)
        printf("MPI norm: N=%d np=%d norm=%f time=%lf perf=%lf MFLOPS\n",
               N, np, sum, t, perf);


$ mpicc -O mpi_norm.c
$ mpirun -np 10 a.out
N=1000000 np=10 norm=100.0 time=0.01 perf=100.0 MFLOPS
```

MPI norm: $\|\vec{x}\|$



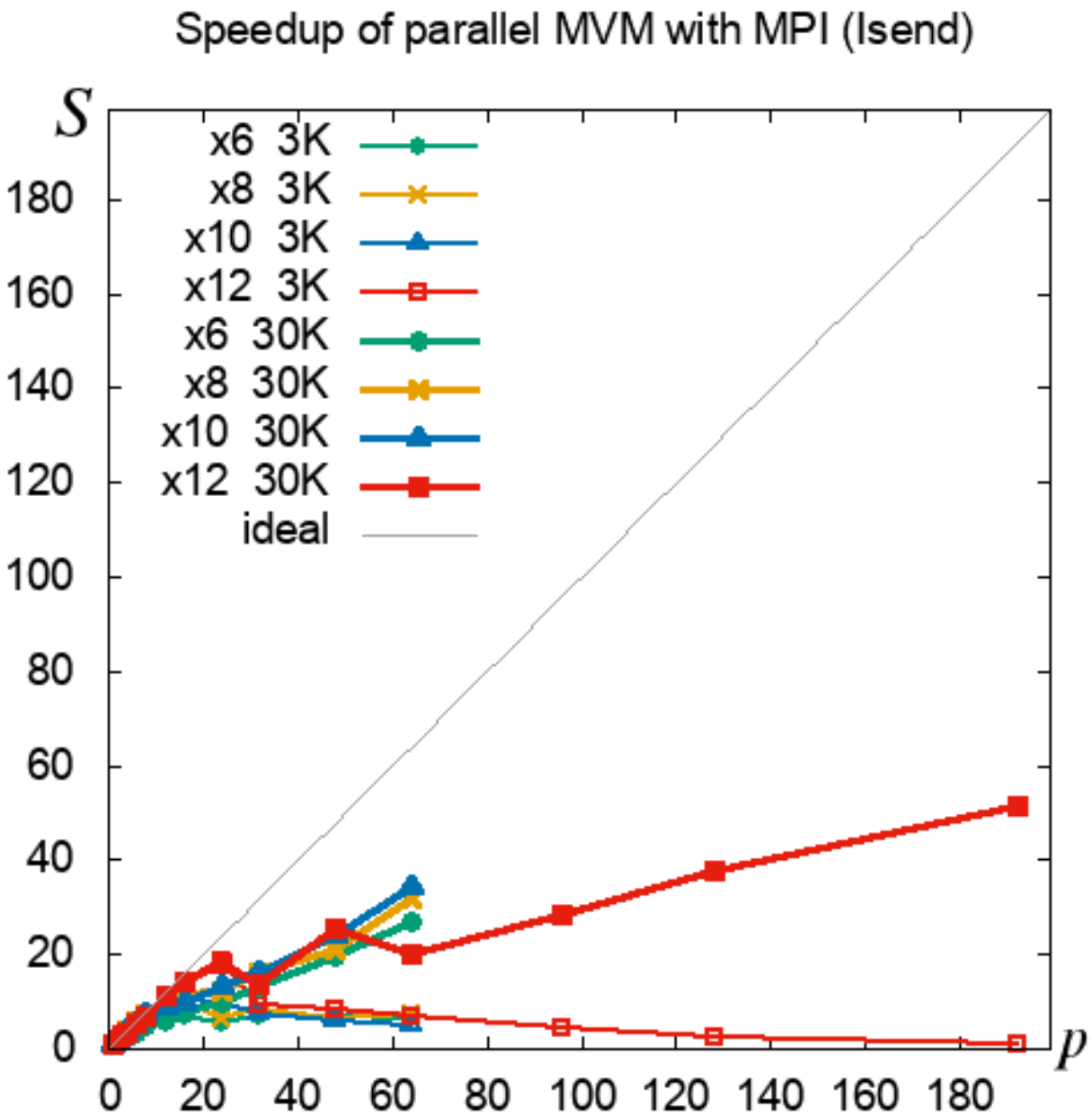
MPI norm: $\|\vec{x}\|$ в log-шкале



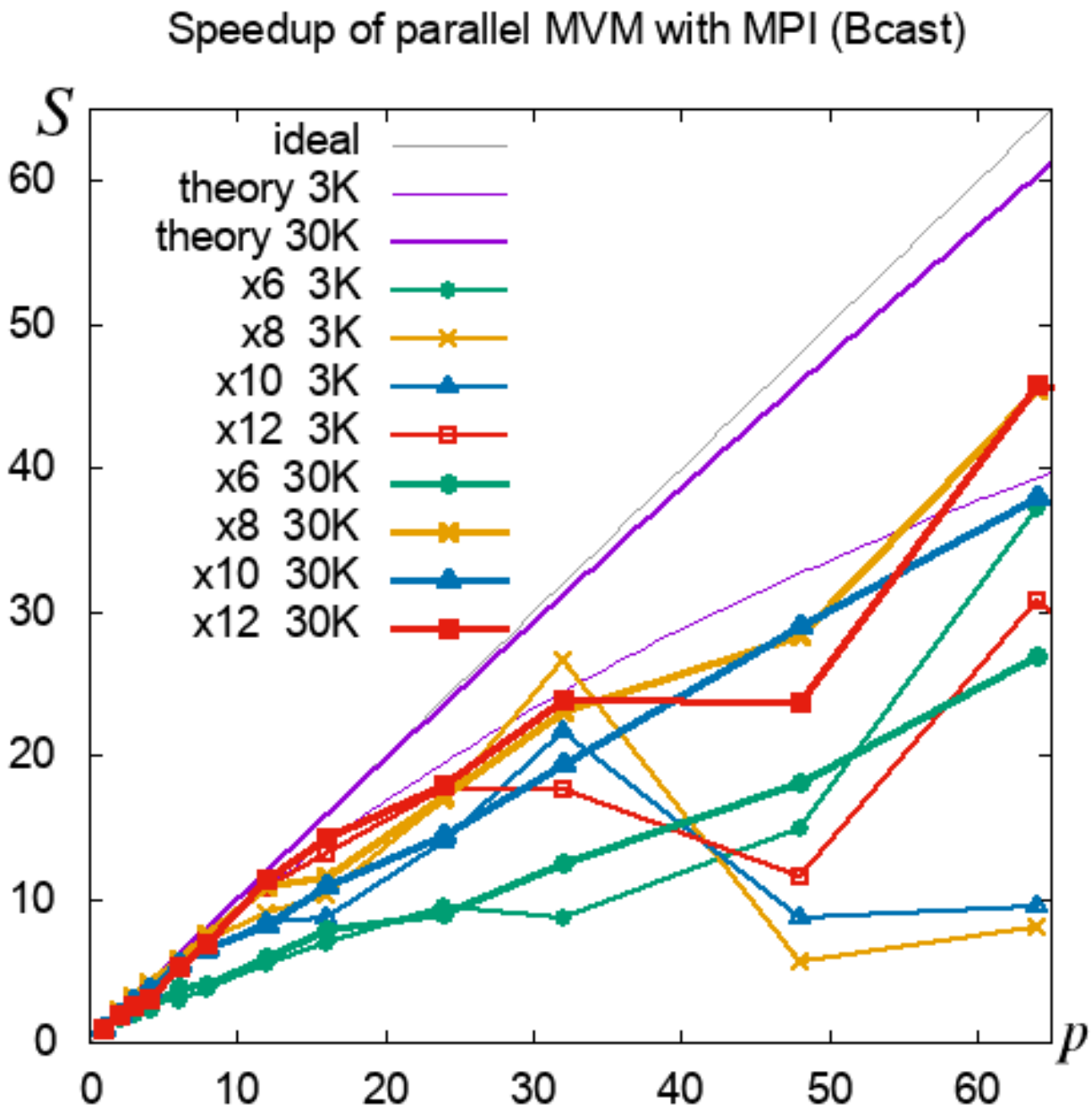
MPI mvm: $\vec{y} = A \cdot \vec{x}$

```
#define N 1000

    int i, j, n, id, np;
    double *a, *x, *y, *X, *ai, t;

    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &np);

    n = N / np;
    a = (double *) malloc(n*N * sizeof(double));
    x = (double *) malloc(n   * sizeof(double));
    y = (double *) malloc(n   * sizeof(double));
    X = (double *) malloc(N   * sizeof(double));

    MPI_Barrier(MPI_COMM_WORLD);
    t = MPI_Wtime();

... A code fragment from your practical work ...

    MPI_Barrier(MPI_COMM_WORLD);
    t = MPI_Wtime() - t;
    if (id == 0)
        printf("MPI mvm: N=%d np=%d time=%lf\n", N, np, t);


$ mpicc -O mpi_mvm.c
$ mpirun -np 4 ./a.out
MPI mvm: N=1000 np=4 time=0.1
```

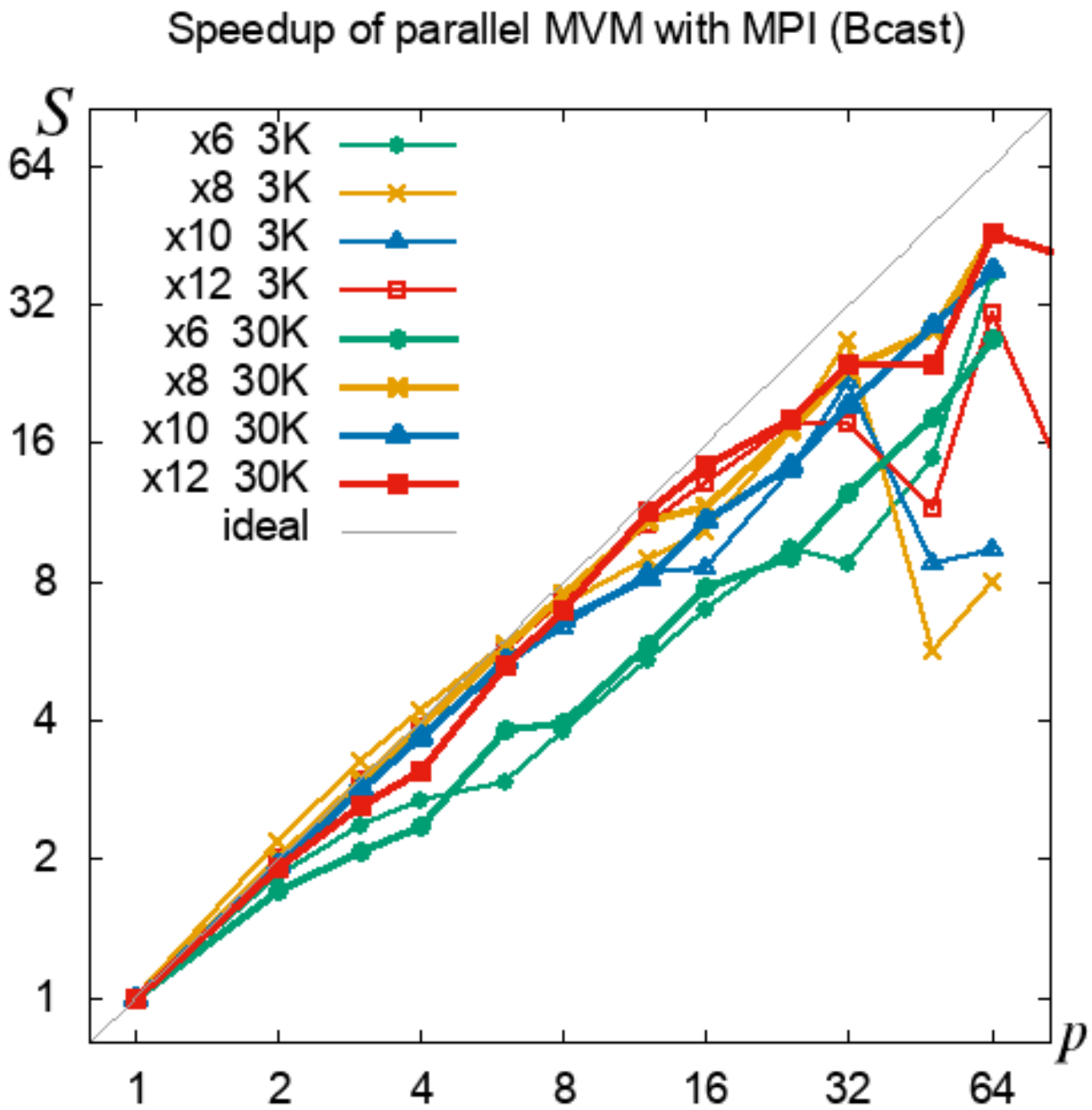
MPI mvm: $\vec{y} = A \cdot \vec{x}$



MPI mvm: $\vec{y} = A \cdot \vec{x}$

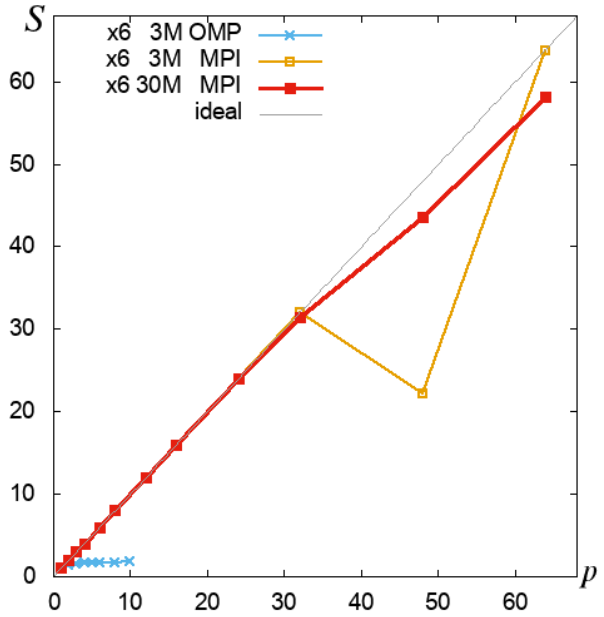


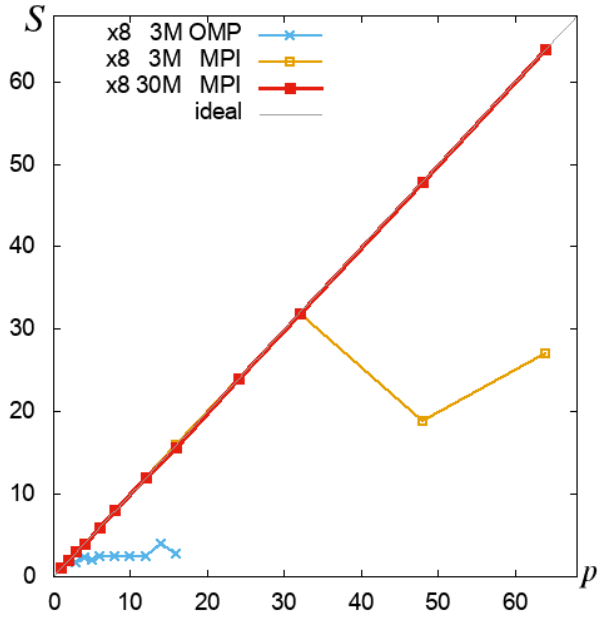
MPI mvm: $\vec{y} = A \cdot \vec{x}$

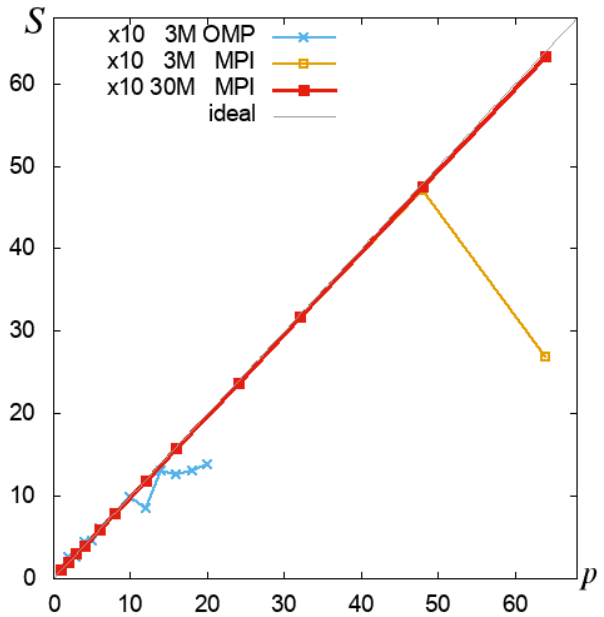


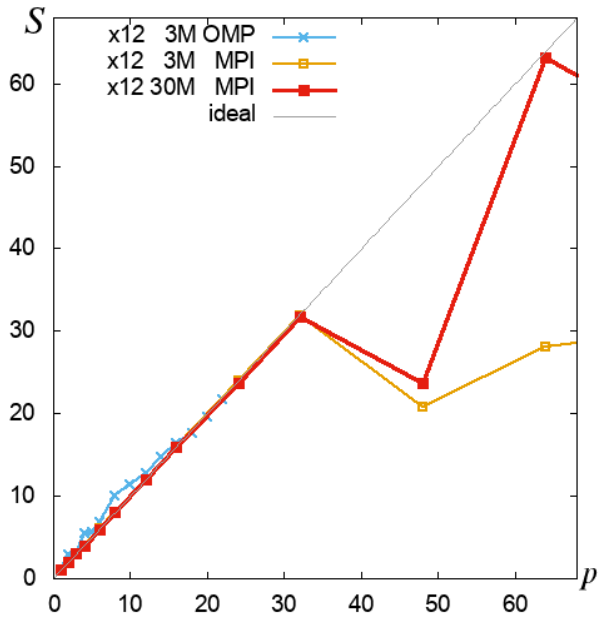
Сравнение результатов для OpenMP и MPI

Сравнение: OMP vs. MPI daxpy: $\alpha \cdot \vec{x} + \vec{y}$

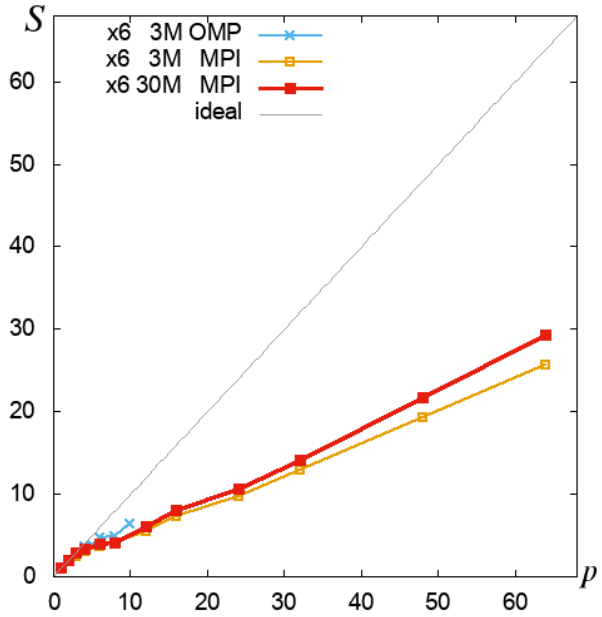


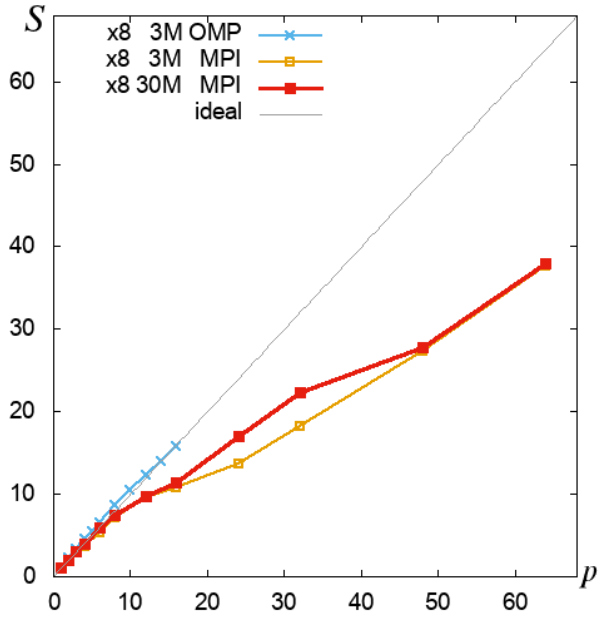


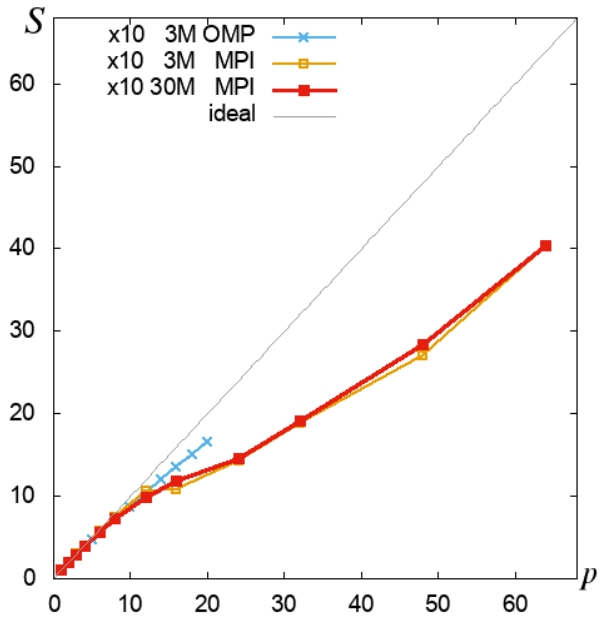


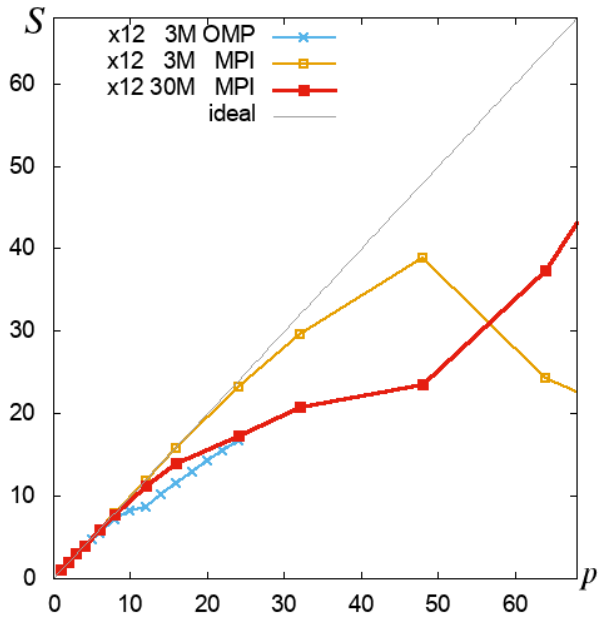


Сравнение: OMP vs. MPI norm: $\|\vec{x}\|$

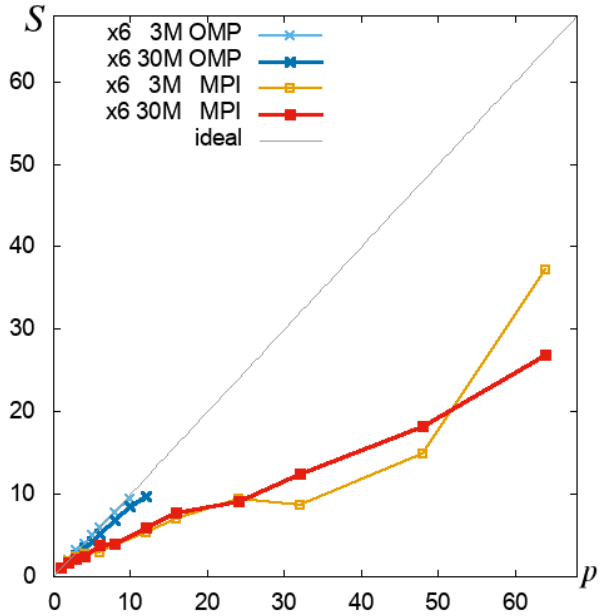


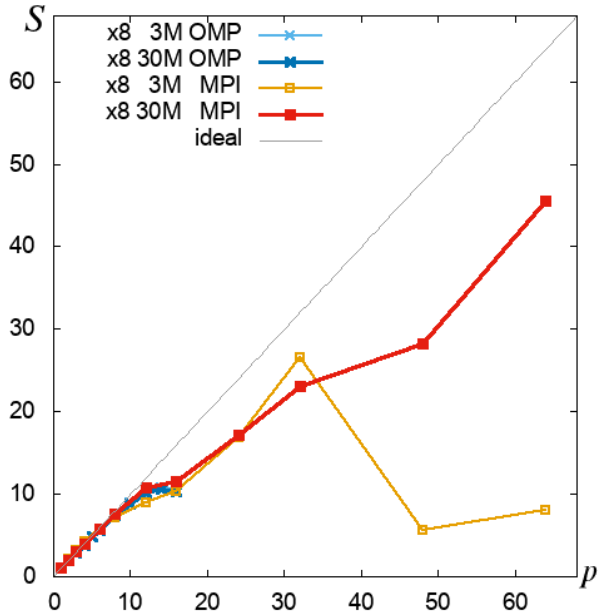


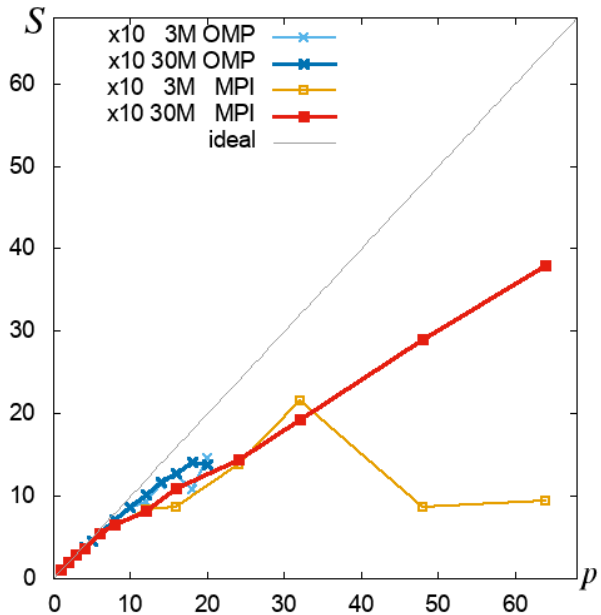


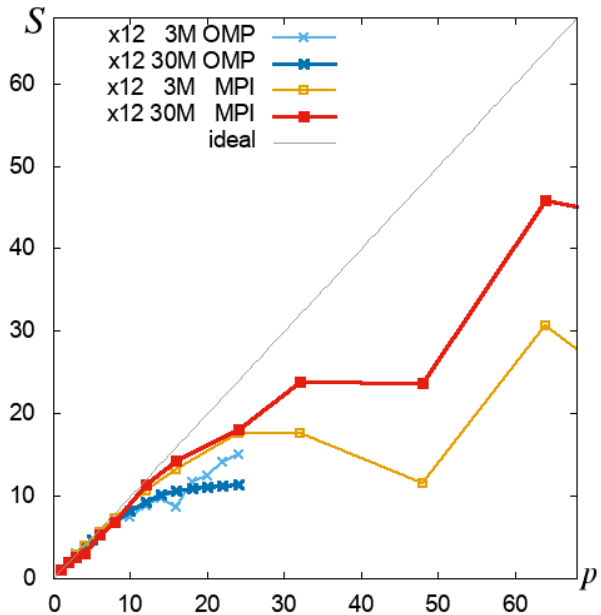


Сравнение: OMP vs. MPI mvm: $\vec{y} = A \cdot \vec{x}$

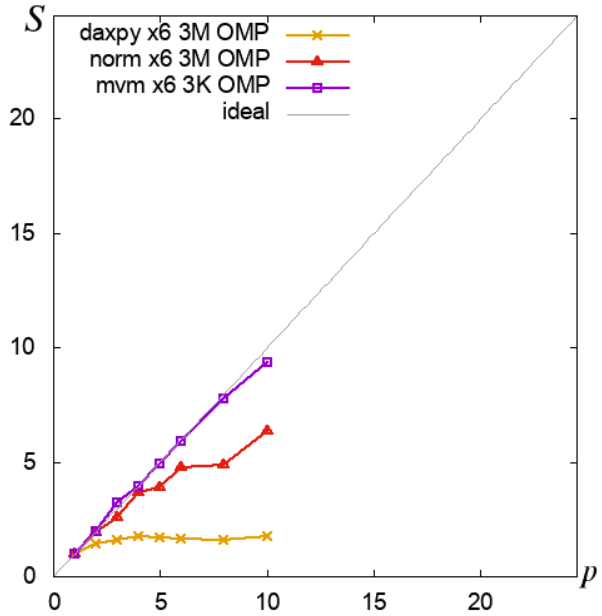


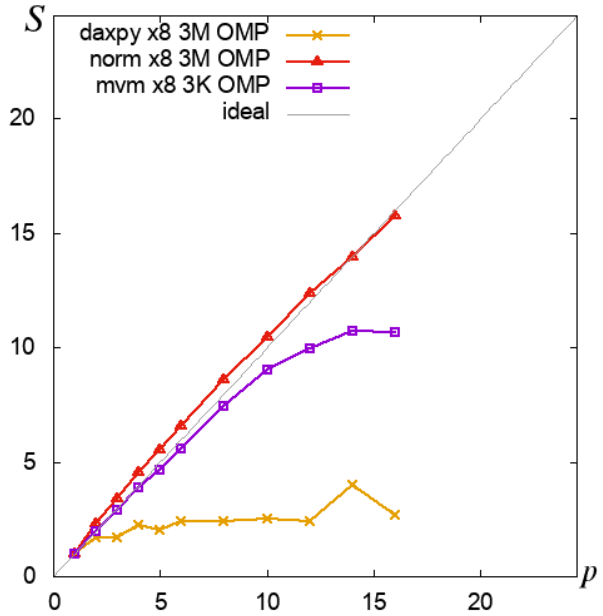


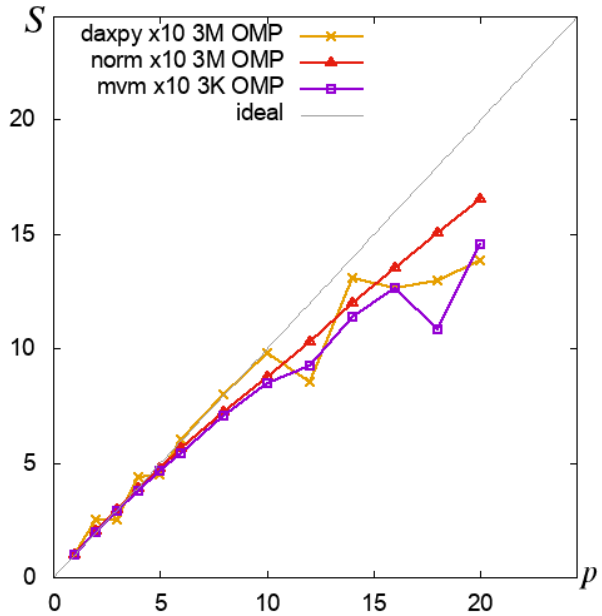


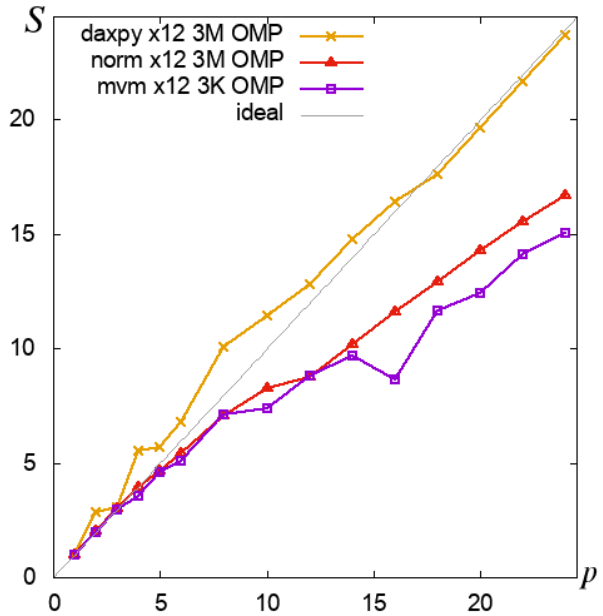


Сравнение на OMP: операции (daxpy, norm, mvm)

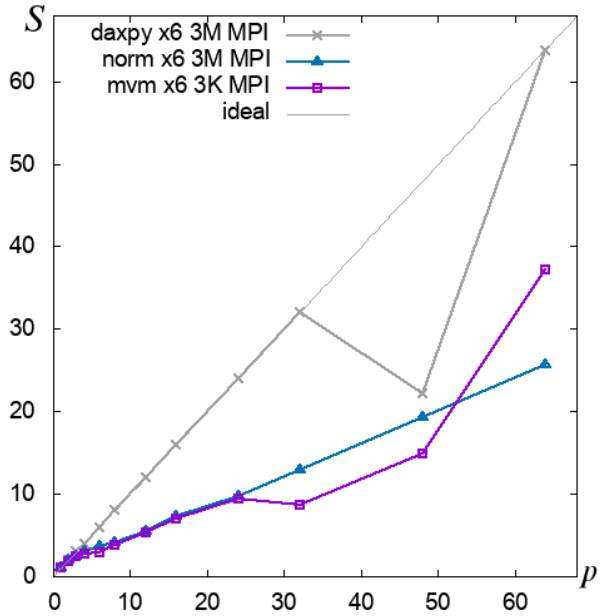


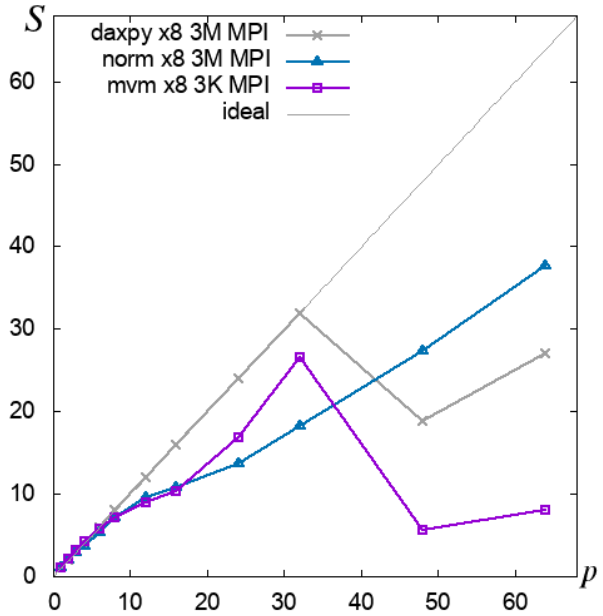


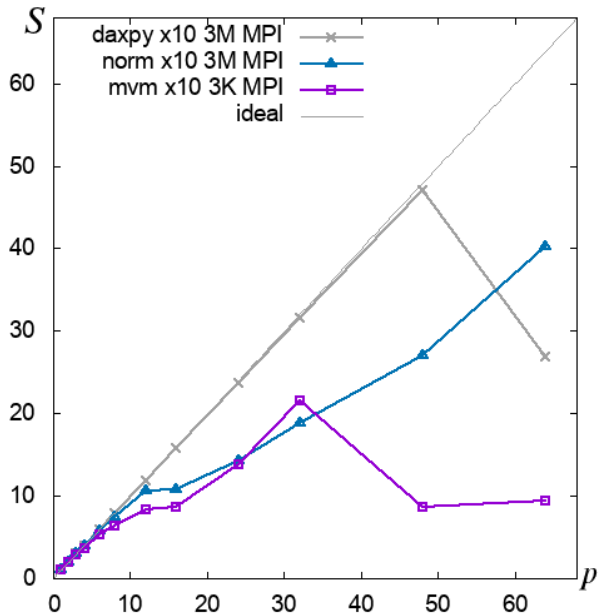


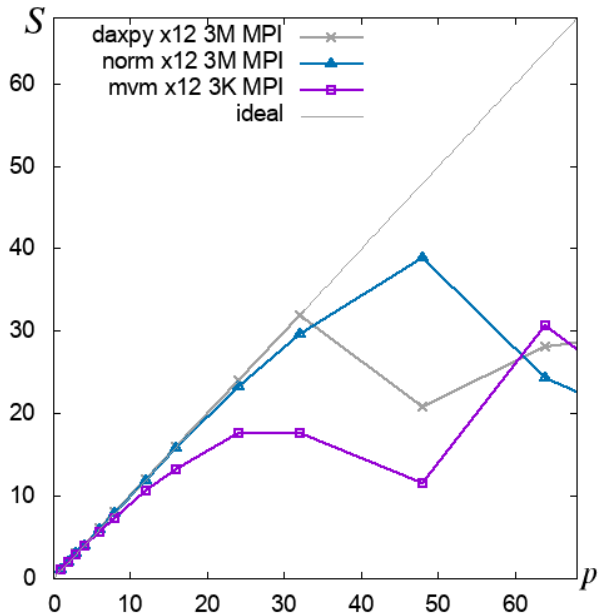


Сравнение на MPI (малые задачи): (daxpy, norm, mvm)

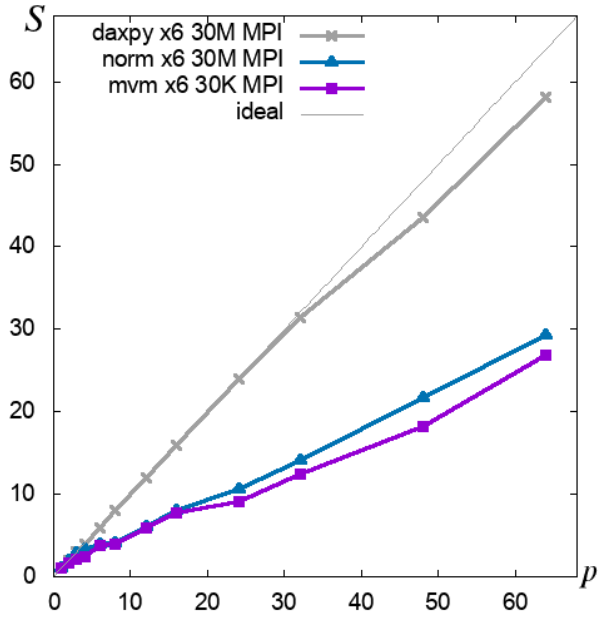


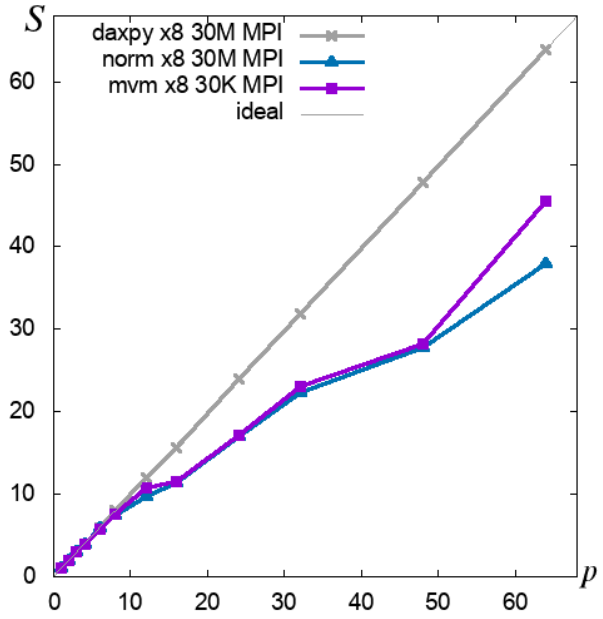


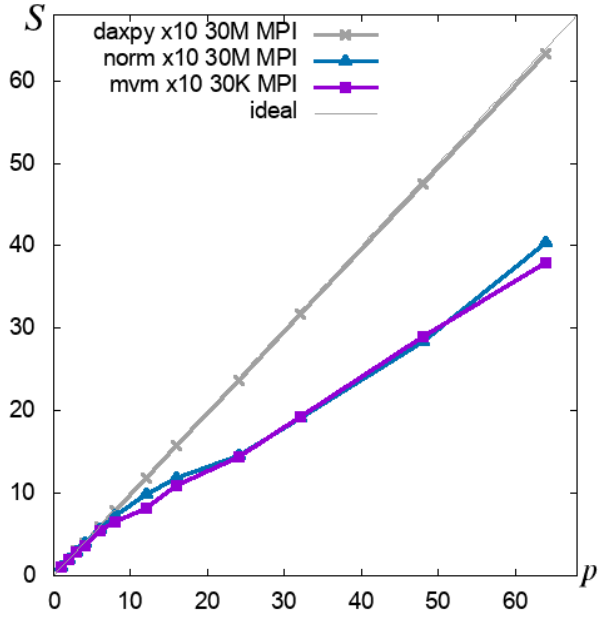


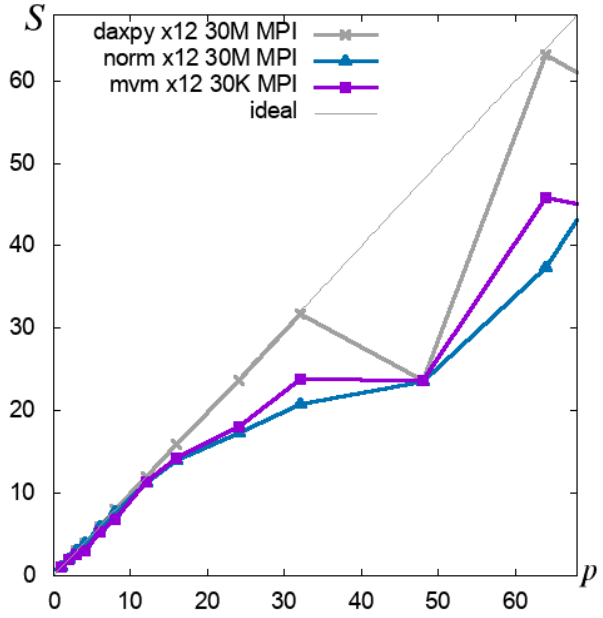


Сравнение на MPI (большие задачи): (daxpy, norm, mvm)









Продвинутая линейная алгебра

Линейная алгебра: метод сопряженных градиентов (PCG)

$$r_0 = b - Ax_0, \quad p_0 = Hr_0$$
$$i = 0, 1, \ldots$$
$$\alpha_i = (r_i^T Hr_i)/(p_i^T Ap_i)$$
$$x_{i+1} = x_i + p_i\alpha_i$$
$$r_{i+1} = r_i - Ap_i\alpha_i$$
$$\beta_i = (r_{i+1}^T Hr_{i+1})/(r_i^T Hr_i)$$
$$p_{i+1} = Hr_{i+1} + p_i\beta_i$$

где:

- A — симметричная положительно определенная $N \times N$ матрица
- H — переобуславливатель
- b — правая часть
- x_0 — начальное приближение (обычно $x_0 = 0$)
- x — искомый вектор решения

Линейная алгебра: модель для IC0 + AS(0) + PCG

- 3 x DAXPY
- 2 x DDOT
- 1 x MVM
- 2 x SOL с блочно-диагональной треугольной матрицей

Матрица системы получена из дискретизации некоторой задачи:

n — размерность в одном направлении
$N = n \times n \times n$ — количество неизвестных (размерность всей системы)
$r = n^2$ — полуширина ленты матрицы
$(2d + 1)$-точечный d-мерный шаблон дискретизации $(d = 3)$

$$L = L_c / L_a = (p - 1)(r + 2)/((2d + 3)N)$$

$$S = \frac{p}{1 + \tau L} = \frac{p}{1 + \frac{\tau(p-1)(r+2)}{(2d+3)N}}$$

Линейная алгебра: PCG эксперимент

- Сегмент x6core кластера ИВМ РАН
- $\tau_a = 3.14 \cdot 10^{-10}, \quad \tau_c = 3.06 \cdot 10^{-8}, \quad \tau = \tau_c/\tau_a = 100$
- Программная платформа INMOST: http://www.inmost.org
- Встроенный тест: solver_test002 (3D: $n = m \times m \times m$)
- $r = m^2$, $n = m^3$, $d = 7$, $\tau = 100$
- $m = 64, 96, 128, 160, \quad p = 1, 2, 4, 8, 16, 32, 64$
- Линейный решатель PETSc: https://www.mcs.anl.gov/petsc

```
-ksp_type cg
-pc_type asm
-pc_asm_overlap 0
-sub_pc_type ilu
-sub_pc_factor_levels 0
```

Ускорение (теория и эксперимент): IC0 + AS(0) + PCG

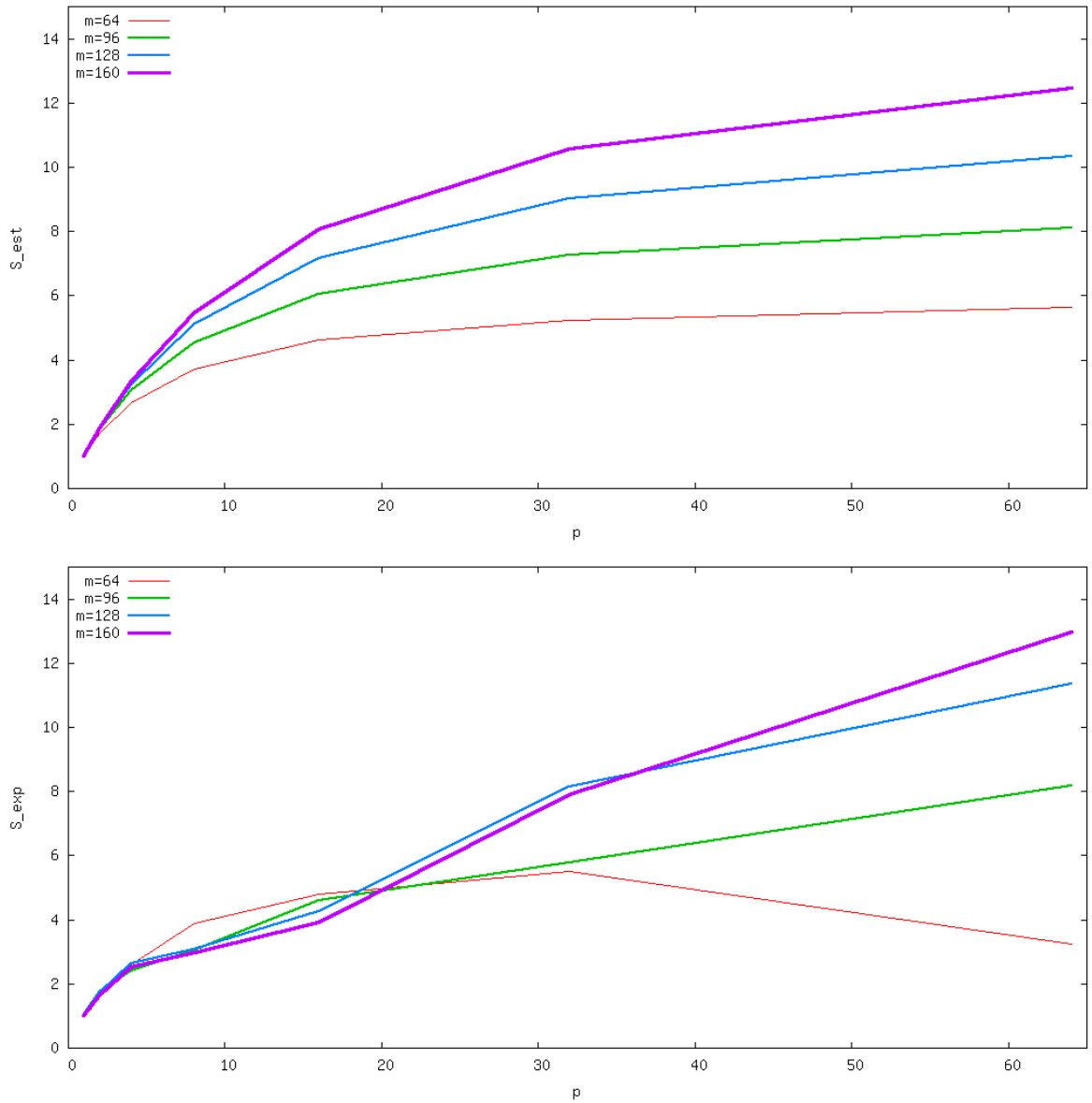


Рис.: Ускорение (теория и эксперимент) для $n = 64, 96, 128, 160$

Ускорение (теория и эксперимент): IC0 + AS(0) + PCG

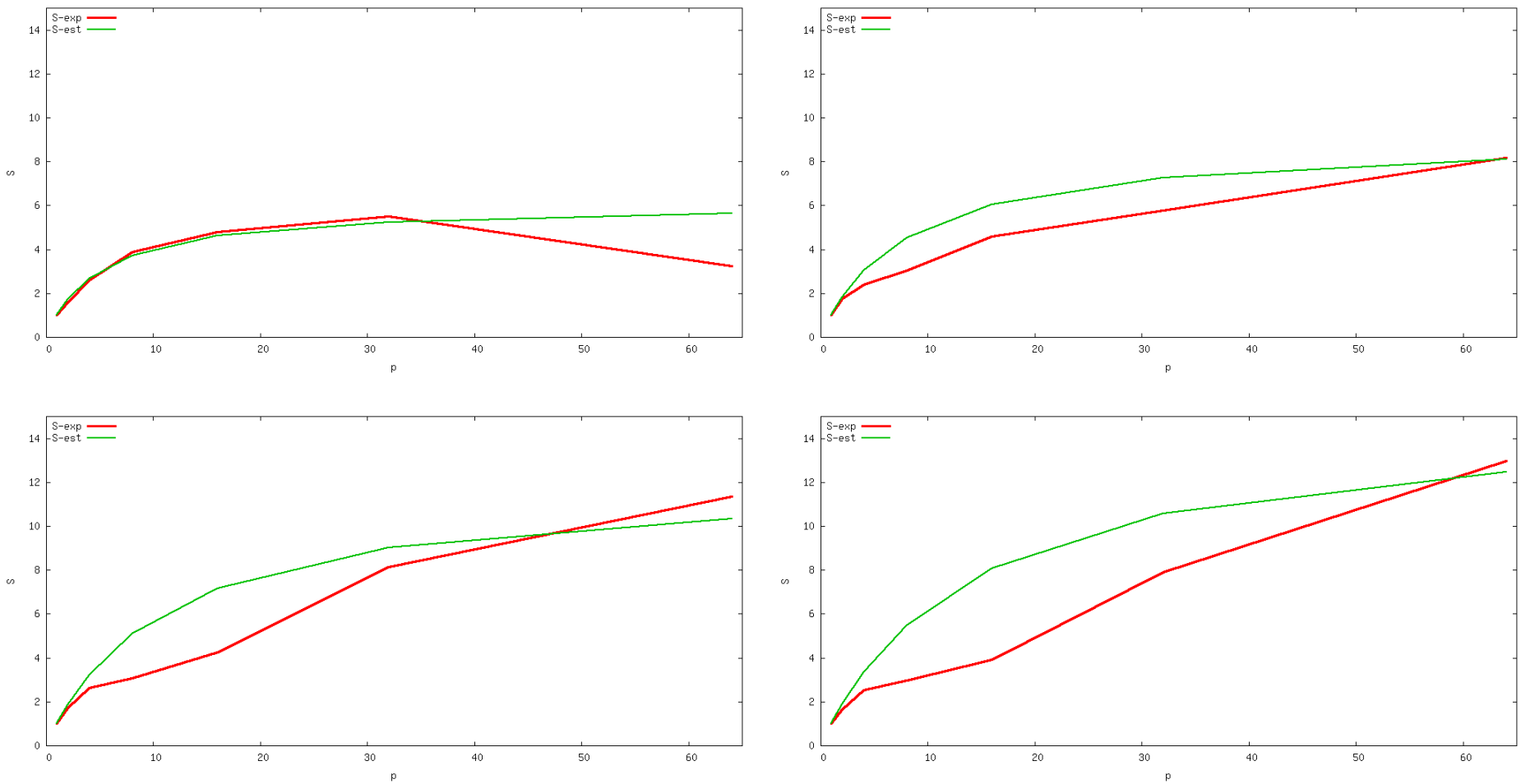


Рис.: Ускорение (теория и эксперимент) для $n = 64, 96, 128, 160$

MPI: выводы

MPI — распараллеливание *данных*

Главные особенности:

- распределенная память (все компьютеры из Top500 относятся к этому типу)
- библиотечные функции для межпроцессорных связей
- *более сложное распараллеливание чем на OpenMP*

MPI: примеры для самостоятельного решения

1. При каком количестве процессоров для каждой из трех готовых реализаций daxpy (OMP, MPI, MPI+OMP) время счета в сегменте “normal” будет минимальным?
2. Аналогично daxpy реализовать вычисление нормы вектора тремя способами (OMP, MPI, MPI+OMP), после чего определить оптимальное количество процессов для каждого варианта (результаты могут сильно отличаться от daxpy)
3. Посчитать время выполнения одной арифметической операции (double) τ_a, передачи одного числа (double) τ_c, и найти их отношение $\tau = \tau_c/\tau_a$ — основную характеристику параллельности компьютера
4. Найти время инициализации сообщения τ_0 из формулы $T_c = \tau_0 + \tau_c L_c$ и сравнить его со временем τ_c, вычислив τ_0/τ_c

Литература

- Вл.В. Воеводин, Технологии параллельного программирования, MPI: https://parallel.ru/tech/tech_dev/mpi.html
- А.С. Антонов, Вычислительный практикум по технологии MPI: https://parallel.ru/tech/tech_dev/MPIcourse
- А.С. Антонов, Параллельное программирование с использованием технологии MPI, МГУ, Москва, 2004 http://dodo.inm.ras.ru/konshin/HPC/bib/MPI-Antonov.pdf
- И.Н.Коньшин, Модели параллельных вычислений для оценки реального ускорения исследуемого алгоритма. In: Proc. of the Int. Conf. Russian Supercomputing Days, Moscow State University, Moscow, 2016, 269-280. http://dodo.inm.ras.ru/konshin/HPC/bib/INK-LinAlg-2016.pdf