Л9. Простейшие коллективные операции передачи данных. Режимы передачи сообщений

1. Гергель В. П. Теория и практика параллельных вычислений. – М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория Базовых Знаний, 2007. – с. 125..135.

1. Модельный пример

$$S = \sum_{i=1}^{n} x_i \ .$$

Разработка параллельного алгоритма для решения данной задачи не вызывает затруднений: необходимо разделить данные на равные блоки, передать эти блоки процессам, выполнить в процессах суммирование полученных данных, собрать значения вычисленных частных сумм на одном из процессов и сложить значения частичных сумм для получения общего результата решаемой задачи. При последующей разработке демонстрационных программ данный рассмотренный алгоритм будет несколько упрощен: процессам программы будет передаваться весь суммируемый вектор, а не отдельные блоки этого вектора.

2. Передача данных от одного процесса всем остальным процессам программы. Широковещательная рассылка

Использование операций двупроцессного обмена:

```
MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
for (int i = 1; i < ProcNum; i++)
  MPI_Send(&x, n, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
```

неэффективно из-за затрат на подготовку передачи данных, синхронизацию процессов. Кроме того, это может быть выполнено за $\log_2 p$ параллельных итераций. Эффективность достигается при широковещательной рассылке:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype type, int root,
  MPI_Comm comm),
```

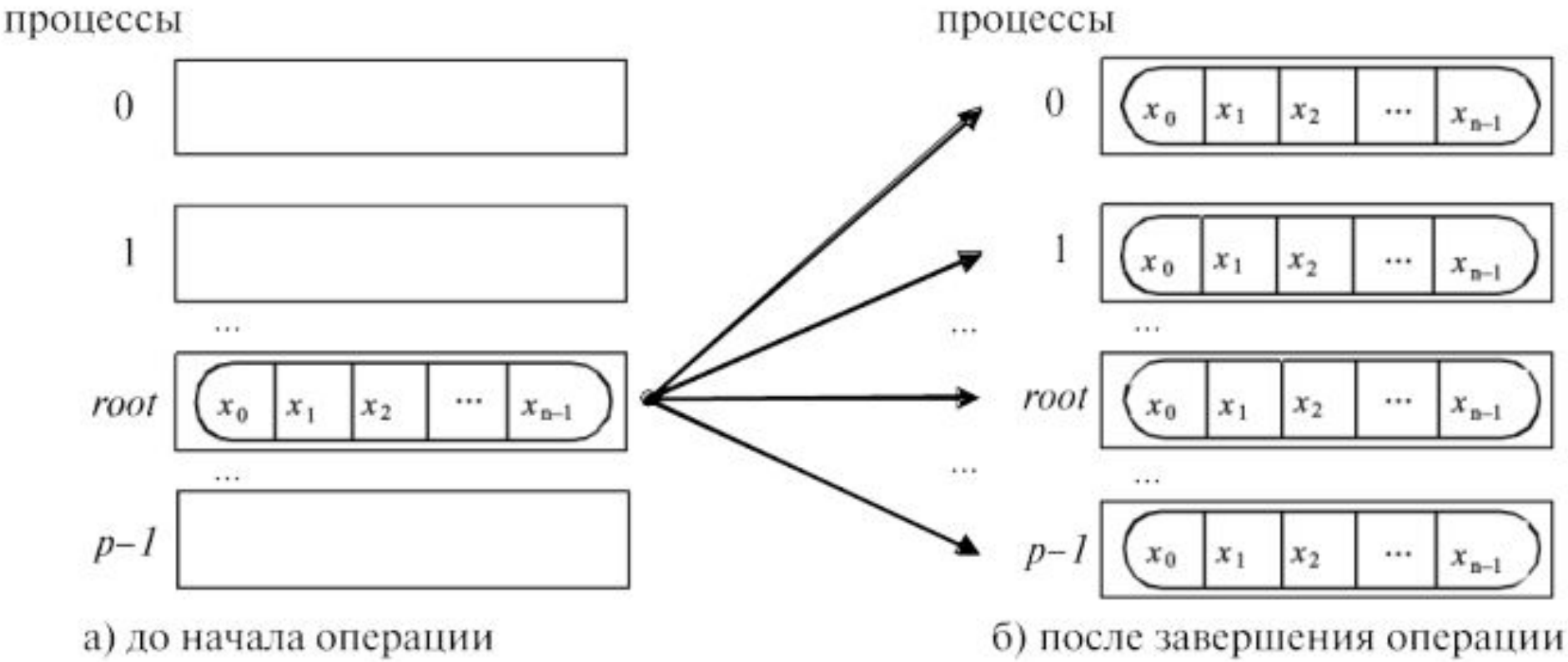
где

— **buf, count, type** — буфер памяти с отправляемым сообщением (для процесса с рангом 0) и для приема сообщений (для всех остальных процессов);

— **root** — ранг процесса, выполняющего рассылку данных;

— **comm** — коммуникатор, в рамках которого выполняется передача данных.

3. Схема выполнения широковещательной рассылки



4. Комментарий

Следует отметить:

- функция MPI_Bcast определяет коллективную операцию, и, тем самым, при выполнении необходимых рассылок данных вызов функции MPI_Bcast должен быть осуществлен всеми процессами указываемого коммуникатора (см. далее пример программы);
- указываемый в функции MPI_Bcast буфер памяти имеет различное назначение у разных процессов: для процесса с рангом *root*, которым осуществляется рассылка данных, в этом буфере должно находиться рассылаемое сообщение, а для всех остальных процессов указываемый буфер предназначен для приема передаваемых данных;
- все коллективные операции «несовместимы» с парными операциями — так, например, принять широковещательное сообщение, отосланное с помощью MPI_Bcast, функцией MPI_Recv нельзя, для этого можно задействовать только MPI_Bcast.

5.Пример параллельной программы суммирования чисел

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char* argv[]){
 double x[100], TotalSum, ProcSum = 0.0;
 int  ProcRank, ProcNum, N=100, k, i1, i2;
 MPI_Status Status;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&ProcNum);
MPI_Comm_rank(MPI_COMM_WORLD,&ProcRank);
..  ..  ..
if ( ProcRank == 0 ) DataInitialization(x,N);
```

6. Параллельная программа суммирования чисел (продолжение)

```
// Рассылка данных на все процессы
MPI_Bcast(x, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

// Вычисление частичной суммы на каждом из процессов
// на каждом процессе суммируются элементы вектора x от i1 до i2
k = N / ProcNum;
i1 = k *   ProcRank;
i2 = k * ( ProcRank + 1 );
if ( ProcRank == ProcNum-1 ) i2 = N;
for ( int i = i1; i < i2; i++ )
  ProcSum  = ProcSum + x[i];
```

7. Параллельная программа суммирования чисел (окончание)

```
// Сборка частичных сумм на процессе с рангом 0
if ( ProcRank == 0 ) {
  TotalSum = ProcSum;
  for ( int i=1; i < ProcNum; i++ ) {
    MPI_Recv(&ProcSum,1,MPI_DOUBLE,MPI_ANY_SOURCE,0,
      MPI_COMM_WORLD, &Status);
    TotalSum = TotalSum + ProcSum;
  }
}
else // Все процессы отсылают свои частичные суммы
  MPI_Send(&ProcSum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
```

```
 // Вывод результата
 if ( ProcRank == 0 )
   printf("\nTotal Sum = %10.2f",TotalSum);
 MPI_Finalize();
return 0;
}
```

8. Передача данных от всех процессов одному. Операция редукции

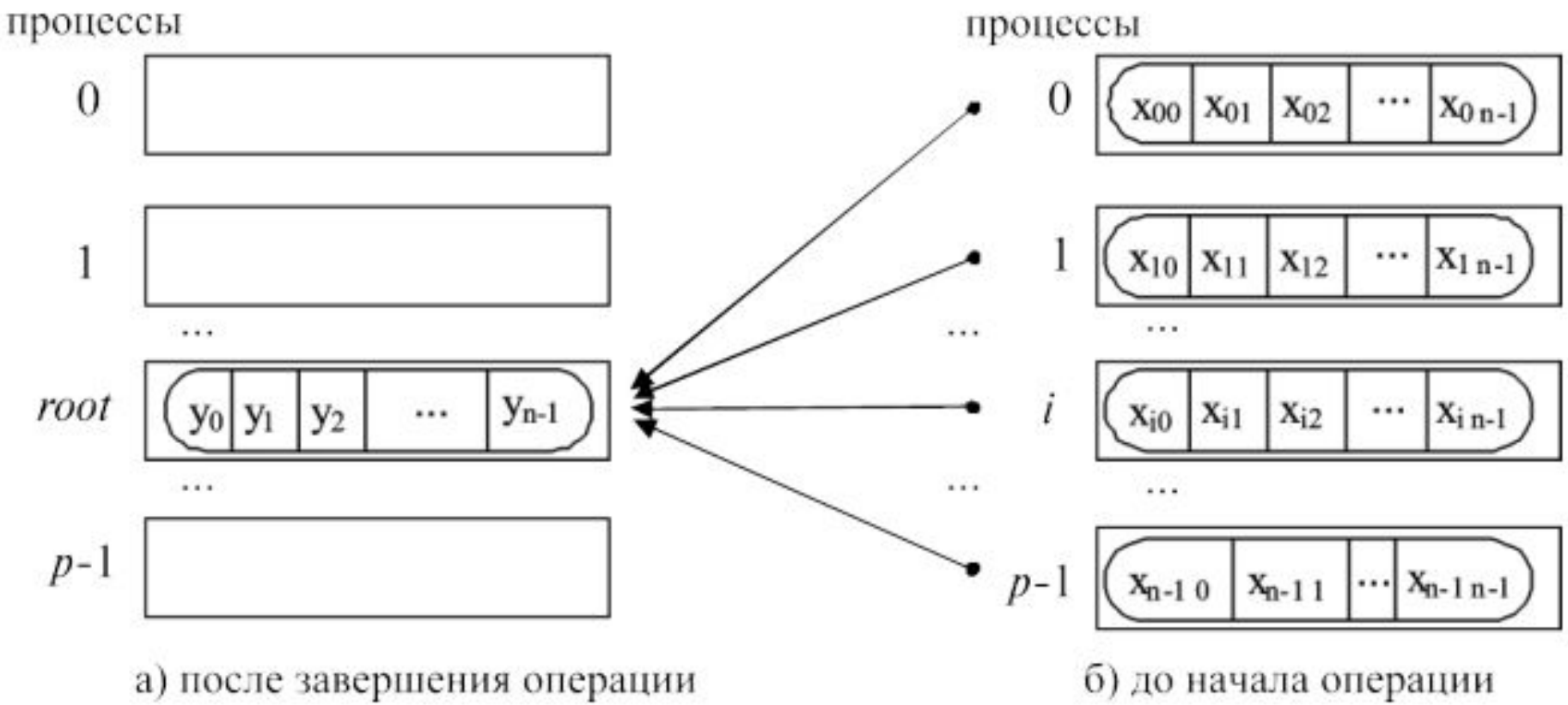
В рассмотренной задаче рассчитанные на каждом из процессов частичные суммы собираются на 0-м процессе. Приведенный код решает эту проблему неэффективно. MPI позволяет решить эту задачу, используя операцию сбора данных или редукцию. Операция редукции, собрав на одном из узлов данные, посланные остальными узлами, выполняет над ними операцию сложения, поиска максимума, минимума, и т.д.

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
  MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm)
```

где

— **sendbuf** — буфер памяти с отправляемым сообщением;
— **recvbuf** — буфер памяти для результирующего сообщения (только для процесса с рангом root);
— **count** — количество элементов в сообщениях;
— **type** — тип элементов сообщений;
— **op** — операция, которая должна быть выполнена над данными;
— **root** — ранг процесса, на котором должен быть получен результат;
— **comm** — коммуникатор, в рамках которого выполняется операция.

9. Схема выполнения редукции



10. Комментарий

Следует отметить:

- функция `MPI_Reduce` определяет коллективную операцию, и, тем самым, вызов функции должен быть выполнен всеми процессами указываемого коммуникатора. При этом все вызовы функции должны содержать одинаковые значения параметров *count, type, op, root, comm*;
- выполнение операции редукции осуществляется над отдельными элементами передаваемых сообщений. Так, например, если сообщения содержат по два элемента данных и выполняется операция суммирования `MPI_SUM`, то результат также будет состоять из двух значений, первое из которых будет содержать сумму первых элементов всех отправленных сообщений, а второе значение будет равно сумме вторых элементов сообщений соответственно;
- не все сочетания типа данных *type* и операции *op* возможны

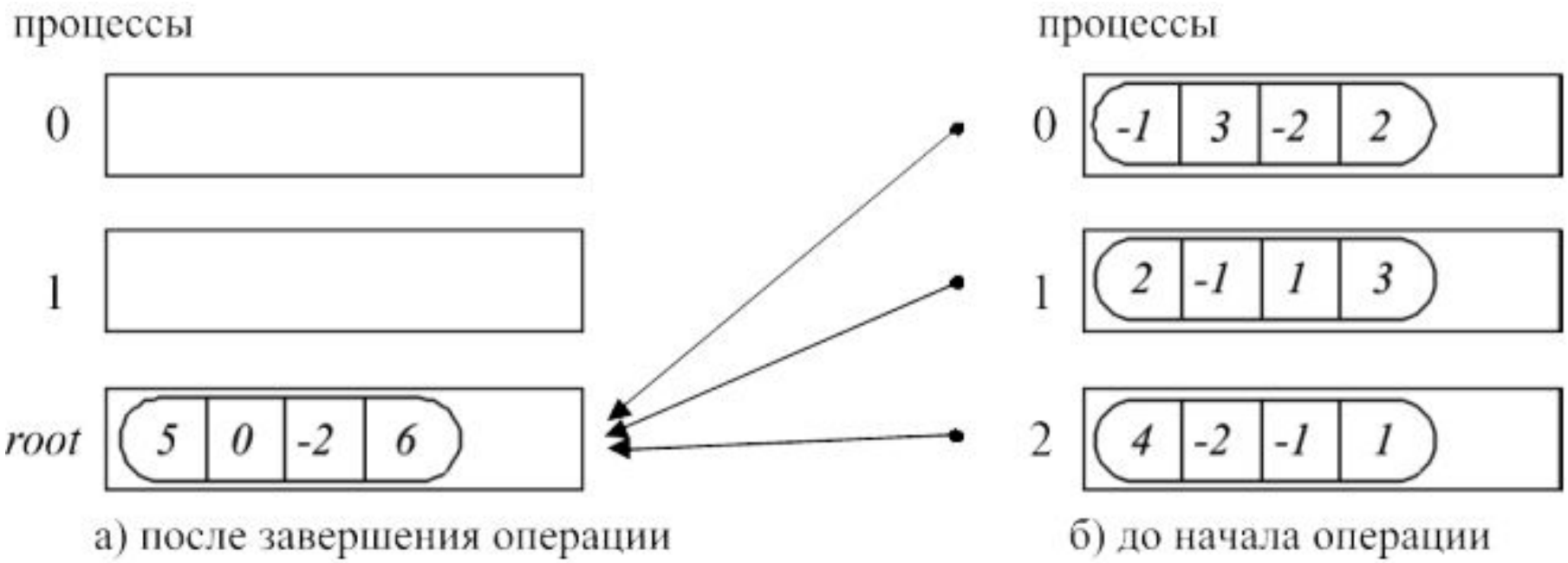
11. Типы операций редукции

Операция	Описание
MPI_MAX	Определение максимального значения
MPI_MIN	Определение минимального значения
MPI_SUM	Определение суммы значений
MPI_PROD	Определение произведения значений
MPI_LAND	Выполнение логической операции «И» над значениями сообщений
MPI_BAND	Выполнение битовой операции «И» над значениями сообщений
MPI_LOR	Выполнение логической операции «ИЛИ» над значениями сообщений
MPI_BOR	Выполнение битовой операции «ИЛИ» над значениями сообщений
MPI_LXOR	Выполнение логической операции исключающего «ИЛИ» над значениями сообщений
MPI_BXOR	Выполнение битовой операции исключающего «ИЛИ» над значениями сообщений
MPI_MAXLOC	Определение максимальных значений и их индексов
MPI_MINLOC	Определение минимальных значений и их индексов

12. Разрешенные сочетания типов

Операция	**Допустимый тип операндов для алгоритмического языка С**
MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD	Целый, вещественный
MPI_LAND, MPI_LOR, MPI_LXOR	Целый
MPI_BAND, MPI_BOR, MPI_BXOR	Целый, байтовый
MPI_MINLOC, MPI_MAXLOC	Целый, вещественный

13. Пример редукции, когда сообщения содержат 4 элемента



14. Сбор данных в задаче суммирования

Применим полученные знания для переработки ранее рассмотренной программы суммирования: как можно увидеть, весь программный код, выделенный рамкой, может быть теперь заменен на вызов одной лишь функции MPI_Reduce:

```
// Сборка частичных сумм на процессе с рангом 0
MPI_Reduce(&ProcSum, &TotalSum, 1, MPI_DOUBLE, MPI_SUM, 0,
 MPI_COMM_WORLD);
```

15. Режимы передачи данных

Рассмотренная ранее функция `MPI_Send` обеспечивает так называемый *стандартный* (*standard*) режим отправки сообщений, при котором

- на время выполнения функции процесс – отправитель сообщения блокируется;
- после завершения функции буфер может быть использован повторно;
- состояние отправленного сообщения может быть различным – сообщение может располагаться на процессе-отправителе, может находиться в состоянии передачи, может храниться на процессе-получателе или же может быть принято процессом-получателем при помощи функции `MPI_Recv`.

16. Дополнительные режимы передачи сообщений

- *синхронный* (*synchronous*) режим состоит в том, что завершение функции отправки сообщения происходит только при получении от процесса-получателя подтверждения о начале приема отправленного сообщения. Отправленное сообщение или полностью принято процессом-получателем, или находится в состоянии приема;
- *буферизованный* (*buffered*) режим предполагает использование дополнительных системных или задаваемых пользователем буферов для копирования в них отправляемых сообщений. Функция отправки сообщения завершается сразу же после копирования сообщения в системный буфер;
- *режим передачи по готовности* (*ready*) может быть использован только, если операция приема сообщения уже инициирована. Буфер сообщения после завершения функции отправки сообщения может быть повторно использован.

17. Обозначения режимов

Для именования функций отправки сообщения для разных режимов выполнения в MPI применяется название функции `MPI_Send`, к которому как префикс добавляется начальный символ названия соответствующего режима работы, т. е.:

— **MPI_Ssend** – функция отправки сообщения в синхронном режиме;

— **MPI_Bsend** – функция отправки сообщения в буферизованном режиме;

— **MPI_Rsend** – функция отправки сообщения в режиме по готовности.

Список параметров всех перечисленных функций совпадает с составом параметров функции `MPI_Send`.

18.

Для применения буферизованного режима передачи может быть создан и передан MPI буфер памяти:

```
int MPI_Buffer_attach(void *buf, int size),
```

где

— **buf** — адрес буфера памяти;

— **size** — размер буфера.

После завершения работы с буфером он должен быть отключен от MPI при помощи функции:

```
int MPI_Buffer_detach(void *buf, int *size),
```

где

— **buf** — адрес буфера памяти;

— **size** — возвращаемый размер буфера.

19. Рекомендации по использованию режимов передачи данных

- стандартный режим обычно реализуется как буферизированный или синхронный, в зависимости от размера передаваемого сообщения, и зачастую является наиболее оптимизированным по производительности;
- режим передачи по готовности формально является наиболее быстрым, но используется достаточно редко, т. к. обычно сложно гарантировать готовность операции приема;
- буферизованный режим также выполняется достаточно быстро, но может приводить к большим расходам ресурсов (памяти), – в целом может быть рекомендован для передачи коротких сообщений;
- синхронный режим является наиболее медленным, т.к. требует подтверждения приема, однако не нуждается в дополнительной памяти для хранения сообщения. Этот режим может быть рекомендован для передачи длинных сообщений.