УДК 004.021
DOI: 10.55648/1998-6920-2022-16-3-38-47

Алгоритмы нерегулярных коллективных операций стандарта MPI для систем с разделяемой памятью

А. А. Романюта, М. Г. Курносов[1]

Предложены алгоритмы реализации коллективных операций MPI_Scatterv, MPI_Gatherv, MPI_Allgatherv для многопроцессорных SMP/NUMA-систем. Алгоритмы используют подход на основе копирования фрагментов сообщений через очереди в сегменте совместно используемой памяти (Copy-In-Copy-Out). Программная реализация выполнена на базе библитеки Open MPI и в среднем на 20–40 % обеспечивает меньшее время выполнения операций MPI_Scatterv, MPI_Gatherv, MPI_Allgatherv по сравнению с реализацией в компоненте coll/tuned библиотеки Open MPI.

Ключевые слова: Scatterv, Gatherv, Allgatherv, коллективные операции, MPI, вычислительные системы.

1. Введение

Современные высокопроизводительные вычислительные системы (ВС) строятся на базе многопроцессорных узлов с общей памятью. Как правило, каждый узел системы включает от одного до четырех процессоров общего назначения с архитектурой x86-64, POWER или ARM, а также группу сопроцессоров/ускорителей. Процессорные ядра одного узла могут передавать информацию через разделяемую память. Взаимодействие между узлами осуществляется через коммуникационную сеть (InfiniBand, 10X Gigabit Ethernet, Slingshot). Обмен информацией через разделяемую память узла, как правило, требует меньше времени. Поэтому двухуровневая иерархия коммуникационной среды активно эксплуатируется для оптимизации обменов информацией в системах параллельного программирования, распределенного машинного обучения и обработки больших массивов данных. Ключевая идея – выполнять обмены информацией между ядрами узла через разделяемую память и только для межузловых взаимодействий использовать коммуникационную сеть.

К наиболее ресурсоёмким коммуникационным операциям относятся коллективные обмены (глобальные, collective communication), в которых участвуют все или значительная часть процессов (процессорных ядер). По этой причине производители коммуникационного оборудования и библиотек уделяют значительное внимание созданию эффективных алгоритмов коллективных операций. В данной работе рассматриваются алгоритмы реализации коллективных операций стандарта MPI [1] `Scatterv`, `Gatherv`, `Allgatherv` через разделяемую память вычислительного узла. Интерес к указанным операциям обусловлен отсутствием эффективных алгоритмов их реализации в коммерческих и открытых библиотеках Open MPI [2], MPICH, MVAPICH. Ниже приведены прототипы операций:

[1] Работа выполнена в рамках государственного задания № 071-03-2022-001.

```
MPI_Scatterv(sbuf[p], scounts[p], displs[p], stype,
             rbuf, rcount, rtype, root)
MPI_Gatherv(sbuf, scount, stype, rbuf[p], rcounts[p], displs[p],
            rtype, root)
MPI_Allgatherv(sbuf, scount, stype,
               rbuf[p], rcounts[p], displs[p], rtype),
```

Операция `MPI_Scatterv` передаёт *scounts*[*rank*] элементов типа *stype* процессу *rank* = 0, 1, …, *p* – 1 из буфера *sbuf*[*rank*] + *displs*[*rank*] · *sizeof*(*stype*) корневого процесса *root*. Процессы принимают сообщение в свой буфер *rbuf* как *rcount* элементов типа *rtype*. В операции *Gatherv* корневой процесс *root* принимает в буфер *rbuf*[*rank*] + *displs*[*rank*] · *sizeof*(*rtype*) ровно *rcounts*[*rank*] элементов типа *rtype*. В операции *Allgatherv* каждый процесс передаёт своё сообщение, *scount* элементов типа *stype* из буфера *sbuf* и принимает от процесса *rank* в буфер *rbuf*[*rank*] + *displs*[*rank*] · *sizeof*(rtype) ровно *rcounts*[*rank*] элементов.

Основную сложность при разработке алгоритмов представляет нерегулярный размер сообщений, которые передаются в операции в виде векторов *scounts*, *rcounts* длины *p*, где *p* – число процессов.

Можно выделить два основных подхода к реализации обмена информацией между процессами с использованием разделяемой памяти многопроцессорного узла. Подход Copy-In-Copy-Out (CICO) основан на использовании сегмента памяти с общей для всех процессов очередью и системой флагов уведомления. Процесс-отправитель копирует в очередь фрагмент сообщения и уведомляет через флаги остальные процессы, после чего они копируют фрагмент в буфер пользователя в своем адресном пространстве. Таким образом, каждый фрагмент копируется дважды. Второй подход подразумевает использование специфичных для конкретного ядра операционной системы методов прямого доступа к памяти удаленного процесса: Linux Cross Memory Attach (CMA), KNEM, XPMEM [3–5]. Это позволяет сократить число копирований до одного, поэтому такой подход получил название ZeroCopy (нуль дополнительных копирований). Заметим, что ZeroCopy-подход эффективен, однако требует дополнительных модулей ядра или же повышения привилегий процессов, в то время как CICO-подход обеспечивает переносимость алгоритма на разные GNU/Linux-системы [6].

В данной работе рассматриваются алгоритмы на основе CICO-подхода. Предложены алгоритмы коллективных операции `MPI_Scatterv`, `MPI_Gatherv` и `MPI_Allgatherv`, использующие сегмент разделяемой памяти и систему очередей в нём. Приводится описание реализации алгоритмов на базе библиотеки Open MPI для операционной системы GNU/Linux и результаты экспериментов.

2. Описание алгоритма

Разработанные алгоритмы выполняются в два этапа:

1. Создание сегмента разделяемой памяти на этапе формирования нового MPI-коммуникатора и системы очередей и флагов в нём.
2. Реализация коллективной коммуникационной операции через очереди сегмента разделяемой памяти (при каждом вызове операции).

2.1. Структура сегмента разделяемой памяти

Нулевой процесс коммуникатора создает сегмент разделяемой памяти, используя системный вызов `mmap`. Остальные процессы подключаются к нему. В сегменте разделяемой памяти размещаются очереди для передачи фрагментов сообщений, массивы со счётчиками для уведомления процессов о готовности фрагментов в очереди, а также дополнительные флаги, обеспечивающие синхронизацию доступа к очередям при многократных вызовах

коллективных операций и смене корневых процессов. Для каждого из p процессов в сегменте хранятся:

- очередь $q[s]$ из s фрагментов (слотов) по f байт;
- массив $ctrl[s]$ для хранения размеров фрагментов, записанных в слоты очереди при выполнении корневых операций (one-to-all, all-to-one);
- массив $ctrlall[s \cdot p]$ для хранения размеров фрагментов, записанных в слоты очереди при выполнении операций all-to-all.

По умолчанию $s = 8$, $f = 8192$ байт, размеры и адреса массивов q, $ctrl$, $ctrlall$ выравнены на границу размера страницы памяти.

Для сокращения накладных расходов на ожидание освобождения очереди для повторного использования её слотов она разбивается на w множеств [7]. Для синхронизации доступа к множествам (частям очередей) в сегменте хранятся счётчики и флаги:

- op – счётчик количества обращений к коллективной операции;
- $nproc$ – количество процессов, читающих/записывающих слоты очереди.

По умолчанию каждая очередь логически разбита на $w = 2$ множества (множество – это s/w фрагментов очереди).

Для поддержки операций `MPI_Scatterv`, `MPI_Gatherv` с нерегулярными размерами сообщений корневой процесс уведомляет остальные процессы о размерах сообщений через разделяемые массивы:

- $typesize$ – размер типа данных в байтах;
- $counts[p]$ – массив с количеством элементов;
- $rootready[p]$ – флаги готовности данных в массиве $counts$.

Каждому процессу известно расположение в сегменте как своих блоков, так и блоков других процессов. У каждого процесса определена локальная для него переменная $local\_op$, использующаяся как счётчик количества обращений к коллективной операции. Счётчик $local\_op$ используется для синхронизации выполнения операции между процессами и должен иметь одинаковое значение у каждого процесса для корректного начала и завершения выполнения коллективной операции. В начале работы со слотами множества i корневой процесс устанавливает значение $ws[i].op = local\_op$, остальные процессы дожидаются, пока $ws[i].op$ не станет равно их локальному значению $local\_op$. После этого все процессы могут начать чтение/запись в слоты множества i. При задействовании процессом множества локальный счётчик его операций $local\_op$ увеличивается на единицу.

2.2. Алгоритм операции MPI_Scatterv

В алгоритме операции `MPI_Scatterv` только корневому процессу $root$ известны размеры сообщений $scount[i] \cdot sizeof(stype)$ для передачи процессам. Корневой процесс $root$ реализует передачу фрагментов сообщения через разделяемую память, используя фрагменты (слоты) очереди. Перед началом использования множества set процесс $root$ дожидается, когда значение $ws[set].nproc$ станет равным 0 (0 процессов работают со слотами). Синхронизация доступа к множеству осуществляется путем записи в $ws[set].op$ локального счётчика $local\_op$ и количества процессов, участвующих в обмене $ws[set].nproc = p$. Далее процесс $root$ размещает информацию о размерах сообщений $scounts$ в блоке разделяемой памяти $ws[set].counts$ и записывает размер типа данных $stype$ в блок $ws[set].typesize$, после чего уведомляет все процессы о готовности массива $scounts$: $ws[set].rootready[0..p-1] = 1$. После того как все процессы прочитают информацию о размерах сообщений, $root$ начинает выполнять передачу первого фрагмента длиной f байт каждому процессу в порядке их нумерации 0, 1, …, $p - 1$. Передача выполняется путем записи первых f байт для i-го процесса в фрагмент s очереди $q[s][i]$ процесса i. Уведомление процесса-получателя осуществляется путем записи в управляющий блок $ctrl[s][i]$ процесса i количества байт, записанных во фрагмент очереди. Цикл передачи повторяется, пока сообщения не будут переданы для всех процессов. После того как процесс запишет все данные в слоты множества, вызывается атомарная операция

уменьшения счётчика *ws*[*set*].*nproc* для уведомления о завершении работы процесса с множеством *set*. Если одного множества фрагментов недостаточно для обеспечения передачи всего сообщения, корневой процесс использует следующее множество фрагментов.

```
function MPI_Scatterv(sbuf, scounts[p], displs[p], stype, rbuf, rcount, rtype, root)
  is_first_iter = true
  if rank = root then
    nops = ceil(max(scounts, p) * sizeof(stype) / (f * s / w))
    // Копирование сообщения из sbuf в буфер корневого процесса
    copy(sbuf + displs[root] * sizeof(stype), rbuf, scounts[root] * sizeof(stype))
    nsend[root] = scounts[root] * sizeof(stype)
    while sum(nsend, p) < sum(scounts, p) * sizeof(stype) and nops > 0 do
      set = local_op % w          // Номер текущего множества фрагментов
      nops = nops - 1
      while ws[set].nproc > 0 do  // Ожидание освобождения множества очереди
      end while
      ws[set].nproc = p
      ws[set].op = local_op
      local_op = local_op + 1
      slot = set * s / w
      if is_first_iter then
        ws[set].typesize = sizeof(stype)   // Заполнение scounts[] размерами сообщений
        copy(scounts, ws[set].counts, p * sizeof(scounts))
        for i = 0 to p - 1 do            // Уведомления процессов о готовности scounts[]
          if root == i then continue
          ws[set].rootready[i] = 1
        end for
        is_first_iter = false
      end if
      while sum(nsend, p) < sum(scounts, p) * sizeof(stype) and
            slot < (set + 1) * s / w do
        for i = 0 to p - 1 do                       // Передача фрагмента каждому процессу
          if nsend[i] = scounts[i] * sizeof(stype) then continue
          fragsize = min(f, scounts[i] * sizeof(stype) - nsend[i])
          copy(sbuf + displs[i]* size(stype) + nsend[i], q[slot][i],
               fragsize)                     // Копирование фрагмента в очередь процесса i
          nsend[i] += fragsize
          ctrl[slot][i] = fragsize
        end for
        slot = slot + 1
      end while
      atomic_dec(ws[set].nproc)
    end while
  else // Некорневые процессы
    nrecv = 0
    nops = ceil(max(scounts, p) * sizeof(stype) / (f * s / w))
    while nrecv < rcount * sizeof(rtype) and nops > 0
      set = local_op % w                                  // Номер множества фрагментов
      while ws[set].op != local_op do  // Ожидание использования множества процессом root
      end while
      if is_first_iter then                              // Чтение размеров сообщений
        while ws[set].rootready[rank] != 0 do
        end while
        copy(ws[set].counts, scounts, p * size(scounts))
        ws[set].counts[rank] = 0                         // Уведомление о получении counts
        nops = ceil(max(scounts, p) * ws[set].typesize / (f * s / w))
        is_first_iter = false
      end if
      local_op = local_op + 1
      nops = nops - 1
      slot = set * s / w
      while nrecv < rcount * size(rtype) and slot < (set + 1) * s / w do
        while ctrl[slot][rank] != 0 do                    // Ожидание уведомления от root
        end while
        // Копирование фрагмента i из очереди процесса rank в буфер пользователя
        copy(q[slot][rank], rbuf + nrecv, ctrl[slot][rank])
        nrecv = nrecv + ctrl[slot][rank]
        ctrl[slot][rank] = 0                      // Уведомление о завершении копирования
        slot = slot + 1
      end while
```

```
      atomic_dec(ws[set].nproc)
    end while
  end if
end function
```

Рис. 1. Псевдокод алгоритма операции MPI_Scatterv

Операция `MPI_Scatter` является частным случаем операции `MPI_Scatterv`, в которой блоки данных, посылаемые каждому процессу, имеют одинаковый размер. На рис. 1 приведён псевдокод алгоритма операции `MPI_Scatterv`, в котором массив *nsend*[*i*] хранит суммарный размер фрагментов, переданных *root* процессу *i*.

Сложность алгоритма линейно зависит от количества p процессов и размера сообщения. Сложность по памяти определяется размером сегмента разделяемой памяти, который зависит от числа процессов p и длин очередей s и размеров фрагментов f.

2.3. Алгоритм операции MPI_Gatherv

Операция коллективного сбора данных `MPI_Gatherv` является обратной по отношению к `MPI_Scatterv`. Корневому процессу *root* известны размеры получаемых от процессов сообщений $rcounts[i] \cdot sizeof(rtype)$. В отличие от операции рассылки, алгоритм реализует сбор блоков данных различной длины, посылаемых всеми процессами группы, в один буфер *rbuf* процесса с номером *root*. Корневой процесс, как и в операции `MPI_Scatterv`, размещает информацию о размере сообщений *rcounts* в блок разделяемой памяти, после чего *root* ожидает уведомления о готовности фрагмента сообщения от всех некорневых процессов. Некорневой процесс i записывает первые f байт во фрагмент s очереди $q[s][i]$, после чего записывает в свой управляющий блок $ctrl[s][i]$ размер скопированного фрагмента, тем самым уведомляя *root* о завершении процесса записи в $q[s][i]$. После записи всех фрагментов в слоты множества вызывается атомарная операция уменьшения счётчика $ws[set].nproc$. Если одного множества фрагментов недостаточно для обеспечения передачи процессами всего сообщения, корневой процесс использует следующее множество фрагментов.

Операция `MPI_Gather` является частным случаем операции `MPI_Gatherv`, в которой блоки данных, посылаемые каждым процессом, имеют одинаковый размер.

2.4. Алгоритм операции MPI_Allgatherv

Функция MPI_Allgatherv является расширенной версией `MPI_Gatherv`, в которой получателями являются все процессы коммуникатора. Каждому процессу известны размеры получаемых сообщений $rcounts[i] \cdot sizeof(rtype)$. Для работы с множеством фрагментов процессы логически разделены – процесс с номером 0 обеспечивает синхронизацию для доступа к множеству, что в операции `MPI_Gatherv` выполняет процесс *root*.

Алгоритм операции `MPI_Allgatherv` не имеет корневого процесса, поэтому все процессы выполняют одинаковую последовательность действий. Каждый процесс i записывает первые f байт во фрагмент s очереди $q[s][i]$. После чего процесс i уведомляет остальные процессы о готовности данных в слоте очереди путем записи в управляющий блок $ctrlall[s][i \cdot j]$ количества байт для каждого процесса $j = 0, 1, \ldots, p - 1$, записанных во фрагмент очереди. Цикл записи в слоты очереди повторяется, пока не закончатся свободные слоты множества или сообщение не будет скопировано полностью. После завершения копирования фрагментов для передачи каждый процесс начинает активное ожидание уведомления от других процессов в порядке их нумерации $0, 1, \ldots, p - 1$ и копирует фрагмент сообщения в локальный буфер. Копирование фрагментов из очередей процессов повторяется, пока не закончатся слоты множества или сообщения от процессов не будут полностью получены. После того как процесс запишет и прочитает все данные из слотов множества, вызывается атомарная операция для уведомления о завершении работы процесса с множеством *set*. Если одного множества фрагментов

недостаточно для обеспечения передачи процессами всего сообщения, корневой процесс использует следующее множество фрагментов.

Операция `MPI_Allgather` является частным случаем операции `MPI_Allgatherv`, в которой блоки данных, посылаемые каждым процессом, имеют одинаковый размер.

3. Организация экспериментов

Алгоритмы реализованы на базе компонента `coll/sm` библиотеки Open MPI 4.1.2rc3. Компонент создает сегмент разделяемой памяти и очереди в нём, которые используют разработанные алгоритмы для обмена сообщениями. Для предотвращения некорректного уведомления процессами друг друга, в силу возможного внеочередного выполнения инструкций процессором, после записи в буферы очередей и управляющие блоки вызывается операция барьера записи в память (write memory barrier). Архитектурно-зависимые функции используются из подсистемы Open MPI OPAL (Open Portable Access Layer) [6].

Экспериментальная часть проводилась на сервере со следующей конфигурацией:

- двухпроцессорный сервер Intel Xeon Broadwell: 2 x Intel Xeon E5-2620 v4 (8 ядер, HyperThreading отключен, кеш-память L1 32 Кб, L2 256 Кб, L3 20 Мб);
- оперативная память: 64 Гб (2 NUMA-узла);
- ядро linux 5.14.10-300.fc35.x86_64 (ОС Fedora), gcc 11.2.1;
- MPI: Open MPI 4.1.2rc3.

В качестве теста производительности использовался пакет Intel MPI Benchmarks 2021 Update 3 (IMB-v2021.3). В работе использовалась методика оценки эффективности коллективных операций, описанная в работе [8]. Сравнение производительности осуществлялось с компонентом Open MPI `coll/tuned`. Для каждого размера сообщения операции запускались 5000 раз. Отключалось использование кеш-памяти (на каждом вызове использовался новый буфер, `-off_cache 20,64`), циклическое изменение номера корневого процесса контролировалось параметром `-root_shift 1`. При запуске разработанного алгоритма использовались параметры по умолчанию: количество множеств $w = 2$, количество фрагментов в очереди каждого процесса $s = 8$, размер одного фрагмента $f = 8192$ байт [7]. После каждого обращения к операции выполнялась барьерная синхронизация встроенным алгоритмом IMB. Параметры запуска теста:

```
IMB-MPI1 scatterv -off_cache 20,64 -iter 5000,250 -msglog 6:24 -sync 1
                  -imb_barrier 1 -root_shift 1 -time 600.0
```

В каждом эксперименте тест IMB запускался 5 раз, для каждого размера сообщения отбрасывались минимальное и максимальное значения t_max времени выполнения алгоритма, далее значение t_max усреднялось по результатам трех запусков.

3.1. Операция MPI_Scatterv

На рис. 2 и 3 показаны зависимости времени выполнения разработанного алгоритма операции `MPI_Scatterv` от размера сообщения при запуске 8 процессов и 16 процессов с различным распределением по ядрам NUMA-узлов. Время нормализовано относительно времени выполнения операции компонентом `coll/tuned`.

Разработанный алгоритм позволяет сократить время выполнения операции `MPI_Scatterv` на 60 % по сравнению с `coll/tuned` на сообщениях до 64 Кб. Пик времени выполнения при размере сообщения $m = 32$ Мб и количестве процессов $p = 16$ обусловлен накладными расходами на передачу сообщения и синхронизацию процессов на разных NUMA-узлах.

В среднем разработанный алгоритм показал сокращение времени на 30 % по сравнению с `coll/tuned`.

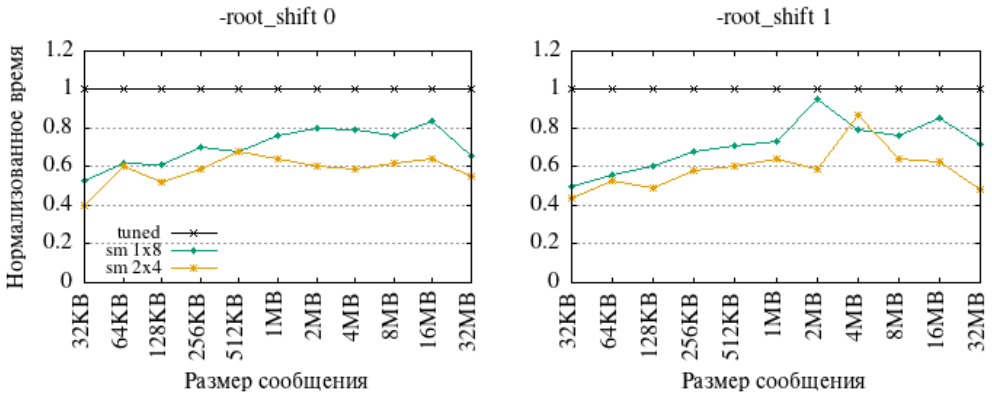

Рис. 2. Нормализованное время алгоритма MPI_Scatterv при запуске $p = 8$ процессов на одном NUMA-узле (1x8) и двух NUMA-узлах (2x4) с циклической сменой корневого процесса (-root_shift 1) и без (-root_shift 0). Время нормализовано относительно времени coll/tuned

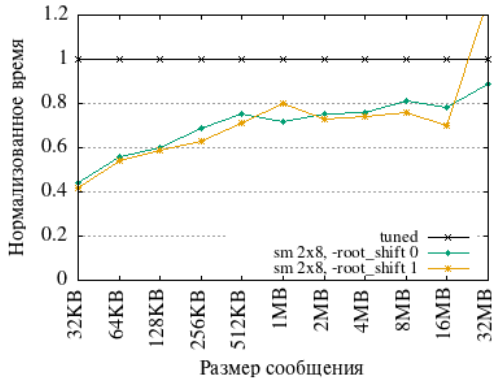

Рис. 3. Нормализованное время алгоритма `MPI_Scatterv` при запуске $p = 16$ процессов на двух NUMA-узлах (2x8) с циклической сменой корневого процесса (`-root_shift 1`) и без (`-root_shift 0`). Время нормализовано относительно времени компонента `coll/tuned`

3.2. Операция MPI_Gatherv

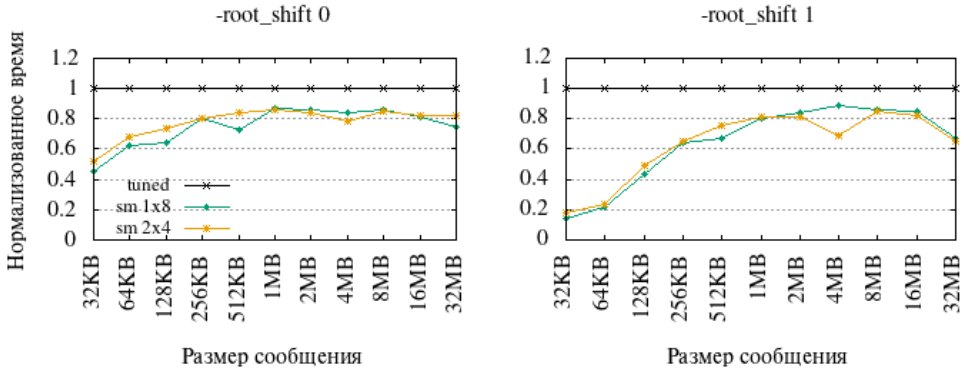

Рис. 4. Нормализованное время алгоритма `MPI_Gatherv` при запуске $p = 8$ процессов на одном NUMA-узле (1x8) и двух NUMA-узлах (2x4) с циклической сменой корневого процесса (`-root_shift 1`) и без (`-root_shift 0`). Время нормализовано относительно времени `coll/tuned`

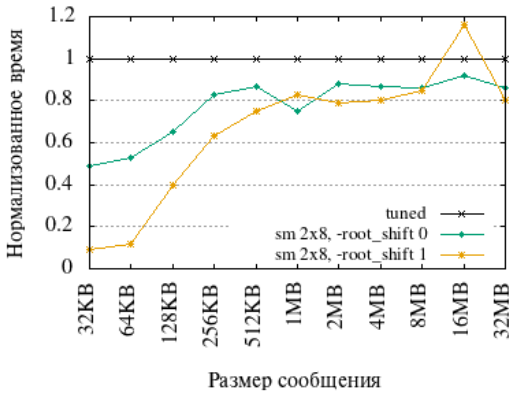

Рис. 5. Нормализованное время алгоритма `MPI_Gatherv` при запуске $p = 16$ процессов на двух NUMA-узлах (2x8) с циклической сменой корневого процесса (`-root_shift 1`) и без (`-root_shift 0`). Время нормализовано относительно времени компонента `coll/tuned`

На рис. 4 и 5 показаны зависимости времени выполнения алгоритма операции `MPI_Gatherv` от размера сообщения при запуске 8 и 16 процессов с различным распределением по ядрам NUMA-узлов. Время нормализовано относительно времени выполнения операции компонентом `coll/tuned`.

Предложенный алгоритм позволяет уменьшить время выполнения операции `MPI_Gatherv` на 90 % по сравнению с `coll/tuned` при передаче сообщений размером до 64 КБ. В проведенных экспериментах алгоритм показал время в среднем на 20–30 % меньше по сравнению с реализацией в компоненте `coll/tuned`.

3.3. Операция MPI_Allgatherv

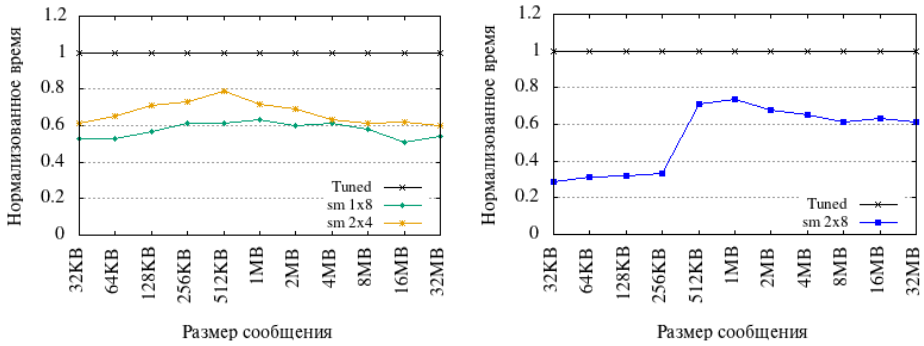

Рис. 6. Нормализованное время алгоритма `MPI_Algatherv` при запуске $p = 8$ процессов на одном NUMA-узле (1x8) и двух NUMA-узлах (2x4) и $p = 16$ процессов на двух NUMA-узлах (2x8). Время нормализовано относительно времени компонента `coll/tuned`

На рис. 6 представлены зависимости времени работы алгоритма операции `MPI_Allgatherv` от размера передаваемого сообщения при запуске 8 и 16 процессов с различным распределением по ядрам NUMA-узлов.

В проведенных экспериментах алгоритм показал время при передаче сообщений размером до $m = 256$ КБ и количестве процессов $p = 16$ на 70 % меньше, чем при использовании компонента `coll/tuned`. При запуске $p = 8$ процессов алгоритм показал стабильное сокращение времени выполнения на 30–40 % в сравнении с реализацией в компоненте `coll/tuned`.

4. Заключение

В работе предложены алгоритмы операции MPI_Scatterv, MPI_Gatherv и MPI_Allgatherv на многопроцессорных вычислительных SMP/NUMA-систем с использованием разделяемой памяти. Выполнена программная реализация алгоритмов на базе библиотеки Open MPI. Эксперименты на двухпроцессорном сервере показали: в среднем предложенные алгоритмы выполняют коллективные операции MPI_Scatterv, MPI_Gatherv и MPI_Allgatherv на 20–40 % быстрее алгоритмов компонента `coll/tuned`. Наибольшее сокращение времени достигается при передаче сообщений до 64 Кб для операций `MPI_Scatterv` и `MPI_Gather` и до 256 Кб для операции `MPI_Allgatherv`.

В дальнейшем планируется реализовать полный набор блокирующих коллективных операций, а также учесть возможности ZeroCopy-подхода (GNU/Linux CMA, XPMEM, KNEM).

Литература

1. MPI: A Message-Passing Interface Standard Version 4.0 [Электронный ресурс]. URL: `http://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf` (дата обращения: 15.05.2022).
2. Open MPI: Open Source High Performance Computing [Электронный ресурс]. URL: `http://www.open-mpi.org` (дата обращения: 15.05.2022).
3. CMA: Cross Memory Attach [Электронный ресурс]. URL: `https://lwn.net/Articles/405284/` (дата обращения: 20.05.2022).
4. KNEM High-Performance Intra-Node MPI Communication [Электронный ресурс]. URL: `https://knem.gitlabpages.inria.fr` (дата обращения: 20.05.2022).
5. XPMEM: Linux Cross-Memory Attach [Электронный ресурс]. URL: `https://github.com/hjelmn/xpmem` (дата обращения: 20.05.2022).
6. *Курносов М. Г., Токмашева Е. И.* Алгоритм широковещательной передачи стандарта MPI на базе разделяемой памяти многопроцессорных NUMA-узлов // Вестник СибГУТИ. 2020. № 1 (49). С. 42–59.
7. *Graham R. L., Shipman G.* MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives // Proc. of the 15th European PVM/MPI Users' Group Meeting, 2008. P. 130–140.
8. *Jain S., Kaleem R., Balmana M., Langer A., Durnov D., Sannikov A. and Garzaran M.* Framework for Scalable Intra-Node Collective Operations using Shared Memory // Proc. of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC-2018), 2018. P. 374–385.

Статья поступила в редакцию 06.07.2022.

Романюта Алексей Андреевич
аспирант 1 курса, ассистент кафедры вычислительных систем Сибирского государственного университета телекоммуникаций и информатики (630102, Новосибирск, ул. Кирова, 86), e-mail: `alexey_r.98@ngs.ru`.

Курносов Михаил Георгиевич
д.т.н., профессор, профессор кафедры вычислительных систем Сибирского государственного университета телекоммуникаций и информатики, тел. (383) 269-83-82, e-mail: `mkurnosov@sibsutis.ru`;

старший научный сотрудник федерального государственного бюджетного учреждения науки Института физики полупроводников им. А. В. Ржанова СО РАН (630090, Новосибирск, пр. Ак. Лаврентьева, 13), тел. (383) 330-56-26, e-mail: mkurnosov@isp.nsc.ru.

Shared Memory-based Algorithms for Vector Collective Operations of the MPI Standard

Alexey A. Romanyuta
Computer systems department, Siberian State University of Telecommunications and Information Science (SibSUTIS, Novosibirsk, Russia), alexey_r.98@ngs.ru.

Mikhail G. Kurnosov
Doctor of Technical Science, Professor, Siberian State University of Telecommunications and Information Science (SibSUTIS, Novosibirsk, Russia), mkurnosov@sibsutis.ru.
Senior Research Scientist, Computer Systems Laboratory, Rzhanov Institute of Semiconductor Physics of the Siberian Branch of the RAS (Novosibirsk, Russia), mkurnosov@isp.nsc.ru.

Algorithms for MPI_Scatterv, MPI_Gatherv, MPI_Allgatherv collective operations using the shared memory of multiprocessor SMP/NUMA systems are proposed. The algorithms use an approach based on copying message fragments via queues in a shared memory segment (Copy-In-Copy-Out). The algorithms were implemented based on Open MPI and the execution time is reduced on average by 20-40% for the MPI_Scatterv, MPI_Gatherv, MPI_Allgatherv operations compared to the coll/tuned component of the Open MPI library.

Keywords: Scatterv, Gatherv, Allgatherv, collective operations, MPI, computer systems.

References

1. *MPI: A Message-Passing Interface Standard Version 4.0,* available at: http://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf (accessed: 15.05.2022).
2. *Open MPI: Open Source High Performance Computing*, available at: http://www.open-mpi.org (accessed: 15.05.2022)
3. *CMA: Cross Memory Attach*, available at: https://lwn.net/Articles/405284/ (accessed: 20.05.2022)
4. *KNEM High-Performance Intra-Node MPI Communication*, available at: https://knem.gitlabpages.inria.fr (accessed: 20.05.2022)
5. *XPMEM: Linux Cross-Memory Attach*, available at: https://github.com/hjelmn/xpmem (accessed: 20.05.2022)
6. Kurnosov, M. G., Tokmasheva E. I. Algoritm shirokoveshchatel'noj peredachi standarta MPI na baze razdelyaemoj pamyati mnogoprocessornyh NUMA-uzlov [MPI Broadcast Algorithm Based on Shared Memory of Multiprocessor NUMA Nodes]. *Vestnik SibGUTI*, 2020, no. 1, pp. 42-59.
7. Graham R. L., Shipman G. MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives. *Proc. of the 15th European PVM/MPI Users' Group Meeting*, 2008, pp. 130–140.
8. Jain S., Kaleem R., Balmana M., Langer A., Durnov D., Sannikov A. and Garzaran M. Framework for Scalable Intra-Node Collective Operations using Shared Memory. *Proc. of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC-2018)*, 2018, pp. 374–385.