2023, **24** (4), 339–351. doi 10.26089/NumMet.v24r424

doi 10.26089/NumMet.v24r424 УДК 004.724.3

Алгоритмы редукции и широковещательной рассылки MPI на базе разделяемой памяти многопроцессорных узлов

А. А. Романюта
Сибирский государственный университет телекоммуникаций и информатики, кафедра вычислительных систем, Новосибирск, Российская Федерация
ORCID: 0009-0008-9756-0288, e-mail: alexey_r.98@ngs.ru

М. Г. Курносов
Сибирский государственный университет телекоммуникаций и информатики, кафедра вычислительных систем, Новосибирск, Российская Федерация
Институт физики полупроводников имени А. В. Ржанова Сибирского отделения РАН, Новосибирск, Российская Федерация
ORCID: 0000-0002-7808-1635, e-mail: mkurnosov@sibsutis.ru

Аннотация: Предложены алгоритмы реализации коллективных операций `MPI_Bcast`, `MPI_Reduce`, `MPI_Allreduce` с использованием разделяемой памяти многопроцессорных серверов. Алгоритмы создают сегмент разделяемой памяти и систему очередей в нем, через которые выполняется передача блоков сообщений. Программная реализация выполнена на базе библиотеки Open MPI в виде изолированного компонента `coll/sharm`. В отличие от существующих алгоритмов, взаимодействие с системой очередей организовано через активное ожидание, что сокращает количество барьерных синхронизаций и атомарных операций. При проведении экспериментов на сервере с архитектурой x86–64 для операции `MPI_Bcast` получено наибольшее сокращение времени в 6.5 раз (на 85% меньше) и `MPI_Reduce` в 3.3 раза (на 70% меньше) по сравнению с реализацией в компоненте `coll/tuned` библиотеки Open MPI. Предложены рекомендации по использованию алгоритмов для различных размеров сообщений.

Ключевые слова: Bcast, Reduce, Allreduce, коллективные операции, MPI, вычислительные системы.

Благодарности: Работа выполнена в рамках Государственного задания № 071–03–2023–001 от 19.01.2023.

Для цитирования: Романюта А.А., Курносов М.Г. Алгоритмы редукции и широковещательной рассылки MPI на базе разделяемой памяти многопроцессорных узлов // Вычислительные методы и программирование. 2023. **24**, № 4. 339–351. doi 10.26089/NumMet.v24r424.

2023, **24** (4), 339–351. doi 10.26089/NumMet.v24r424

Shared memory based MPI Reduce and Bcast algorithms

Alexey A. Romanyuta
Siberian State University of Telecommunication and Informatics,
Computer Systems Department, Novosibirsk, Russia
ORCID: 0009-0008-9756-0288, e-mail: alexey_r.98@ngs.ru

Mikhail G. Kurnosov
Siberian State University of Telecommunication and Informatics,
Computer Systems Department, Novosibirsk, Russia
Rzhanov Institute of Semiconductor Physics SB RAS, Novosibirsk, Russia
ORCID: 0000-0002-7808-1635, e-mail: mkurnosov@sibsutis.ru

Abstract: Algorithms for implementing collective operations `MPI_Bcast`, `MPI_Reduce`, `MPI_Allreduce` using shared memory of multiprocessor servers are proposed. The algorithms create a shared memory segment and a system of queues in it, through which message blocks are transmitted. The software implementation is based on the Open MPI library as an isolated `coll/sharm` component. Unlike existing algorithms, interaction with the queuing system is organized with spinlock and focused on reducing the number of barrier synchronizations and atomic operations. When conducting experiments on a server with x86–64 architecture for the `MPI_Bcast` operation, the largest reduction in time was obtained by 6.5 times (85% less) and `MPI_Reduce` by 3.3 times (70% less) compared to the implementation in the `coll/tuned` component of the Open MPI library. Recommendations on the use of algorithms for different message sizes are suggested.

Keywords: Bcast, Reduce, Allreduce, collective operations, MPI, computer systems.

Acknowledgements: The work was supported by research project No. 071–03–2023–001.

For citation: A. A. Romanyuta, M. G. Kurnosov, “Shared memory based MPI Reduce and Bcast algorithms,” Numerical Methods and Programming. **24** (4), 339–351 (2023). doi 10.26089/NumMet.v24r424.

1. Введение. Операции коллективных обменов информацией широко используются в параллельных алгоритмах и программах для рассылки и сбора информации (one–to–all scatter, all–to–one gather), а также обменов типа “каждый-всем” (all–to–all). С развитием систем распределенного машинного обучения и ростом размеров моделей значительную актуальность получила задача создания эффективных по пропускной способности алгоритмов глобальной редукции (all–to–all reduce). Эта операция многократно вызывается в ходе обучения моделей для суммирования градиентов. Коллективные операции составляют значительную часть библиотек стандарта MPI [1] (Open MPI, MPICH, NVIDIA HPCX, Cray MPI), UCX (OpenUCX UCC), а также библиотеки распределенного машинного обучения (NVIDIA NCCL, Huawei HCCL, Alibaba ACCL, Facebook Gloo, Horovod).

Можно выделить несколько основных подходов к созданию алгоритмов коллективных операций.

1. Создание алгоритмов только на основе коммуникационных операций point–to–point (send/recv). Такие алгоритмы обладают наибольшей переносимостью между вычислительными системами (ВС) различных архитектур, но не обеспечивают предельной эффективности.
2. Полная или частичная аппаратная реализация алгоритмов со стороны коммуникационной сети. Как правило, такие алгоритмы реализованы в закрытых библиотеках, поставляемых производителями коммуникационного оборудования, и обеспечивают высокую эффективность (NVIDIA SHARP).
3. Разработка алгоритмов для фиксированного класса систем или учет в алгоритмах определенного архитектурного свойства ВС. Здесь следует выделить иерархические алгоритмы, которые используют разделяемую память вычислительных узлов для обменов между локальными процессами (NVIDIA hcoll, Cheetah, MPICH SMP, NCCL).

2023, **24** (4), 339–351. doi 10.26089/NumMet.v24r424

В данной работе внимание сосредоточено на третьем подходе — алгоритмах коллективных операций в пределах многопроцессорного узла с передачей сообщений через его системную память. Такие алгоритмы используются как отдельный этап иерархических методов [2–4]. В MPI-библиотеках MVAPICH, MPICH, Open MPI, Intel MPI такие алгоритмы коллективных операций получили название “shared memory–based”.

2. Постановка задачи. Имеется вычислительный узел — сервер, укомплектованный одним или несколькими многоядерными процессорами, на котором запущена MPI–программа, p процессов которой распределены по процессорным ядрам и образуют MPI–коммуникатор *comm*. Чтобы избежать переноса операционной системой процессов между ядрами, MPI–библиотеки (Open MPI, MVAPICH, Intel MPI) привязывают процесс к ядру, на котором он запущен.

Необходимо разработать алгоритмы реализации операций `MPI_Bcast`, `MPI_Reduce`, `MPI_Allreduce`, соответствующие требованиям стандарта MPI и использующие разделяемую память вычислительного узла для передачи информации. Ниже приведены прототипы функций:

`MPI_Bcast` (*buf*, *count*, *dtype*, *root*),

`MPI_Reduce` (*sbuf*, *rbuf*, *count*, *dtype*, *op*, *root*),

`MPI_Allreduce` (*sbuf*, *rbuf*, *count*, *dtype*, *op*),

где *sbuf* — буфер отправки, *buf* — буфер приема/передачи, *rbuf* — буфер приема, *count* — количество элементов типа *dtype*, *root* — корневой процесс операции, *op* — бинарная операция, применяемая к данным в операции *Reduce* (сложение, умножение и др.).

3. Обзор работ. Выделяют два основных подхода для реализации обмена информацией между процессами с использованием разделяемой памяти. Первый — Copy–In–Copy–Out (CICO), использует общую для всех процессов систему очередей и флагов уведомления. Фрагмент сообщения копируется в свободный слот очереди, после чего остальные процессы уведомляются о готовности данных. Затем процессы копируют фрагмент в локальную память. Таким образом, каждый фрагмент копируется дважды. Второй подход предполагает использование специфичных для конкретного ядра операционной системы методов прямого доступа к памяти удаленного процесса: Linux Cross Memory Attach (CMA), KNEM, XPMEM [5–8]. Это позволяет сократить число копирований до одного, поэтому такой подход получил название ZeroCopy (нуль дополнительных копирований). Отметим, 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. В данной работе алгоритмы используют CICO–подход.

Различные программные библиотеки, реализующие стандарт MPI, содержат в себе механизмы межпроцессного взаимодействия, в том числе с использованием разделяемой памяти. Библиотека Open MPI содержит компонент `coll/sm`, который использует сегмент разделяемой памяти и формирует систему очередей в нем для выполнения коллективных операций [9]. Данный компонент описан в работе [10], где предлагается использовать p циклических очередей из s сегментов (слотов) размера f байт, разделенных на $w = 2$ множества, где p — количество процессов. Для каждого процесса создается очередь (по умолчанию $s = 8$, $f = 8192$ B), а также w блоков с управляющей информацией. Процесс-отправитель разбивает исходное сообщение на фрагменты по f байт и копирует фрагмент в следующий свободный буфер своей очереди. После копирования фрагмента процессы уведомляются о готовности данных в буфере очереди путем записи в соответствующий ему управляющий блок дочернего процесса размера скопированного фрагмента. Остальные процессы ожидают, пока в их управляющий блок не будет записан размер отправленного фрагмента сообщения, после чего копируют его в локальную память. При заполнении буферов всех множеств процессы выполняют барьерную синхронизацию. Для уведомления процессов о завершении взаимодействия с множеством фрагментов очереди используется атомарная операция.

При работе с множеством фрагментов процессы используют атомарную операцию для уведомления о завершении взаимодействия с очередью. В библиотеке MVAPICH [11] сегмент разделяемой памяти содержит для каждого процесса циклическую очередь из s сегментов. Каждый слот содержит буфер для хранения фрагмента длины f байт, а также значение *psn* — порядковый номер операции, при выполнении которой слот был использован. Номер *psn* используется для уведомления процессов о готовности данных в слоте очереди. По умолчанию длина очереди $s = 128$ слотов размером $f = 8192$ B. Передаваемое сообщение разбивается на фрагменты по f байт. Процессы локально хранят счетчики *write* и *read*, содержащие порядковый номер операции записи/чтения из очереди. Каждый фрагмент копируется в буфер слота с

2023, **24** (4), 339–351. doi 10.26089/NumMet.v24r424

номером $write \,\%\, w$, а поле psn слота устанавливается в значение счетчика $write$. Перед приемом или передачей следующего фрагмента счетчики $write$ и $read$ всех процессов увеличиваются на единицу. При использовании всех слотов очереди вызывается барьерная синхронизация процессов и слоты начинают заполняться с начала очереди.

В работе [2] предлагается использовать $p + 1$ буферов по 8192 байт. Один общий буфер выделяется для операции широковещательной рассылки, и по одному буферу используется для операций редукции. При передаче сообщения процесс копирует фрагмент в разделяемый буфер и уведомляет об этом процессы (шаг $release$). Другие процессы при получении уведомления о готовности фрагмента копируют его в локальную память. Для повторного использования буфера ожидается уведомление от остальных процессов (шаг $gather$). Уведомление процессов о готовности данных в буфере и его освобождении реализуется через флаги в сегменте разделяемой памяти $sh\_release\_flag[rank]$ и $sh\_gather\_flag[rank]$. Для сокращения времени синхронизации процессы логически выстраиваются в дерево, корнем которого является процесс $root$. Корневой процесс уведомляет дочерние о готовности данных в буфере путем записи в их флаги $sh\_release\_flag[i]$ номера его шага $release$. Дочерние процессы в это время ожидают, пока значение их флага $sh\_release\_flag[rank]$ не станет равно номеру шага $release$. На этапе $gather$ дочерние процессы уведомляют родительский процесс путем записи в свой счетчик $sh\_gather\_flag[rank]$ номера шага $gather$, а родительский процесс ожидает, пока все дочерние процессы установят свои флаги sh_gather_flag.

В работе [12] рассматривается использование сегмента разделяемой памяти, содержащего p циклических очередей из s сегментов размером f байт. Очереди разбиты на $w = 2$ множества фрагментов. По умолчанию длина очереди $s = 8$ сегментов по $f = 8192$ В. Для уведомления процессов о готовности фрагмента данных в слоте очереди для каждой очереди создается массив из s управляющих блоков. Реализована поддержка различных видов деревьев (flat, k–ary, k–nomial) для уведомления процессов. При размещении очередей в сегменте разделяемой памяти, в отличие от компонента `coll/sm`, при выделении памяти учитывается топология NUMA-системы. Очереди процессов располагаются в памяти NUMA-узлов, на которых запущен соответствующий очереди процесс. При работе с множеством фрагментов процессы используют атомарную операцию для уведомления о завершении взаимодействия с очередью.

Рассмотренные реализации содержат алгоритмы операций `MPI_Bcast`, `MPI_Reduce`, `MPI_Allreduce`, `MPI_Barrier`.

Предложенная в данной работе структура сегмента разделяемой памяти ориентирована на сокращение количества барьерных синхронизаций и атомарных операций при выполнении коллективных операций с использованием системы очередей. Разрабатываемый авторами компонент `coll/sharm`, в отличие от рассмотренных, поддерживает реализации коллективных операций `MPI_Scatter`, `MPI_Gather`, `MPI_Allgather`. Компонент не допускает одновременного выполнения нескольких коллективных операций. В отличие от компонента `coll/sm`, фрагменты одной очереди располагаются последовательно в памяти и не используются атомарные операции для уведомления процессов о завершении обмена.

4. Описание алгоритмов. Разработанные алгоритмы имеют два общих этапа: 1) создание сегмента разделяемой памяти; 2) использование сегмента разделяемой памяти для передачи сообщений между процессами. Реализация выполнена на основе изолированного компонента подсистемы `coll` OpenMPI `coll/sharm`. При создании коммуникатора процессы формируют сегмент разделяемой памяти и систему очередей в нем, которые используются для обмена информацией при обращении к операциям. Для этого процесс с номером 0 при помощи POSIX-совместимого системного вызова `mmap` выделяет память для размещения системы очередей. Остальные процессы присоединяют выделенный сегмент памяти к своему адресному пространству. После инициализации разделяемой памяти вызывается барьерная синхронизация.

Компонент `coll/sharm` учитывает политику ядра Linux *first touch policy*, которая обеспечивает выделение физических страниц из области памяти, ассоциированной с процессорным ядром, на котором выполняется процесс. Это обеспечивает меньшее время доступа к объектам в памяти, так как исключается необходимость обращения через межпроцессорное соединение к страницам удаленного процессора. Обращение через межпроцессорное соединение может увеличивать время доступа к памяти до 10–15% [13].

Важно отметить, что компонент `coll/sharm` предназначен для работы с коммуникаторами, все процессы которых работают в пределах одного вычислительного узла.

2023, **24** (4), 339–351. doi 10.26089/NumMet.v24r424

4.1. Структура сегмента разделяемой памяти. При создании коммуникатора в сегменте разделяемой памяти компонентом `coll/sharm` размещаются: p циклических очередей $q[p][s]$ из s фрагментов по f байт; p управляющих массивов $c[p][s]$ из s блоков. Каждый блок $c[p][s]$ содержит в себе p элементов для индивидуального уведомления процессов о состоянии фрагментов очереди.

Каждый процесс дополнительно в локальной памяти содержит массив указателей $current\_slot[p]$, элементы которого определяют номера фрагментов очереди каждого процесса для выполнения операций чтения/записи в очередь. После каждой операции с очередью соответствующий ей счетчик увеличивается на единицу. Основное отличие от компонента `coll/sm` заключается в организации взаимодействия процессов с системой очередей — каждый процесс может читать данные из произвольной очереди, в то время как запись осуществляется только в очередь с номером равным номеру процесса в коммуникаторе. Такой подход сводит к минимуму необходимость атомарных блокировок и синхронизаций. Поддержка повторного вызова коллективных операций осуществляется за счет проверки, что слот, на который указывает счетчик $current\_slot[rank]$, был прочитан всеми процессами — каждый элемент массива $c[p][s]$ равен 0. Необходимым условием для корректной работы компонента является синхронизация алгоритмами локальных счетчиков $current\_slot$.

Размеры всего сегмента и отдельных его блоков кратны размеру страницы памяти ($a = 4$ kB). Адреса блоков также выравнены по размеру страницы памяти. Суммарный размер сегмента можно определить по формуле $ps(f + a)$. Для $p = 8$, $a = 4$ kB, $s = 8$, $f = 8$ kB размер системы очередей равен 768 kB, а при $p = 16$ — 1.5 MB. Накладные расходы на хранение локальных указателей и счетчиков составляют $80 + 6p$ B. При $p = 8$ накладные расходы составляют 128 B.

4.2. Алгоритм операции MPI_Bcast. При вызове функции `MPI_Bcast` процессам известен номер корневого процесса *root*, размер сообщения m и буфер приема/передачи сообщения *buf*. Псевдокод операции `MPI_Bcast` представлен алгоритмом 1.

Алгоритм 1. Алгоритм операции `MPI_Bcast`

Algorithm 1. `MPI_Bcast` algorithm

а) Корневой процесс a) *root* process	б) Некорневые процессы b) Non-*root* processes
$msgsize = count \cdot sizeof(dtype)$	$msgsize = count \cdot sizeof(dtype)$
$sentsize = 0$	$recvsize = 0$
$k = current\_slot[root]$	$k = current\_slot[root]$
while $sentsize < msgsize$ **do**	**while** $recvsize < msgsize$ **do**
$\quad fsize = \min(f, msgsize - sentsize)$	$\quad$ **wait_for**$(c[root][k] > 0)$
$\quad$ **wait_for**$(c[root][k] == 0)$	$\quad$ **copy**$(buf + recvsize, q[root][k], c[root][k])$
$\quad$ **copy**$(q[root][k], buf + sentsize, fsize)$	$\quad recvsize = recvsize + fsize$
$\quad c[root][k] = fsize$	$\quad c[root][k] = 0$
$\quad sentsize = sentsize + fsize$	$\quad k = (k + 1) \,\%\, s$
$\quad k = (k + 1) \,\%\, s$	**end while**
end while	$current\_slot[root] = k$
$current\_slot[root] = k$	

Корневой процесс *root* выполняет передачу сообщения всем процессам путем копирования $\lceil m/f \rceil$ фрагментов размером f байт в слоты очереди $q[root]$ процесса *root* в разделяемой памяти. Последний передаваемый фрагмент может иметь размер $(m \,\%\, f)$ байт. Уведомление процессов производится при помощи записи размера скопированного фрагмента в соответствующие ячейки управляющего фрагмента $k = 0, 1, \ldots, s - 1$ слота очереди *root* — $c[root][k] = f$. В процессе передачи фрагментов сообщения, если все слоты очереди заполнены, *root* начинает активное ожидание, пока следующий в очереди слот не будет прочитан всеми процессами. Если количество фрагментов превышает размер очереди s, то при заполнении всех слотов очереди запись продолжится с первого фрагмента. Так, при длине сообщения в 10 фрагментов и количестве слотов очереди $s = 8$, первые 8 фрагментов будут записаны в слоты $0, \ldots, 7$. Оставшиеся два фрагмента будут записаны в слоты 0 и 1 по мере их освобождения.

2023, **24** (4), 339–351. doi 10.26089/NumMet.v24r424

Алгоритм 2. Алгоритм операции `MPI_Reduce`
Algorithm 2. `MPI_Reduce` algorithm

а) Корневой процесс a) *root* process	б) Некорневые процессы b) Non-*root* processes
$msgsize = count \cdot sizeof(dtype)$	$msgsize = count \cdot sizeof(dtype)$
$recvsize = 0$	$sentsize = 0$
$g = \mathbf{ceil}(f/sizeof(dtype))$	$k = current\_slot[rank]$
$temprbuf[f]$	$g = \mathbf{ceil}(f/sizeof(dtype))$
$tempbuf[f]$	**while** $sentsize < msgsize$ **do**
while $recvsize < msgsize$ **do**	$fsize = \min(g \cdot sizeof(dtype), sentsize - recvsize)$
$fsize = \min(g \cdot sizeof(dtype), msgsize - recvsize)$	**wait_for**$(c[rank][k] == 0)$
for $r = commsize - 1$ **to** 0 **step** -1 **do**	**copy**$(q[rank][k], buf + sentsize, fsize)$
$k = current\_slot[r]$	$sentsize = sentsize + fsize$
wait_for$(c[r][k] > 0)$	$c[rank][k] = fsize$
copy$(tempbuf, q[r][k], c[r][k])$	$k = (k + 1)\ \%\ s$
$c[r][k] = 0$	**end while**
if $r == commsize - 1$ **then**	$current\_slot[root] = k$
copy$(temprbuf, tempbuf, fsize)$	
else	
op_reduce$(op, tempbuf, temprbuf, fsize)$	
end if	
$current\_slot[r] = (k + 1)\ \%\ s$	
end for	
copy$(rbuf + recvsize, temprbuf, fsize)$	
$recvsize = recvsize + fsize$	
end while	

Листовые процессы ожидают уведомления о готовности очередного фрагмента k и копируют его из очереди корневого процесса $q[root][k]$ в локальный буфер *buf*. Процесс повторяется, пока сообщение не будет получено полностью.

4.3. Алгоритм операции MPI_Reduce. Коллективная операция редукции `MPI_Reduce`, выполняя ассоциативную операцию *op* над элементами буферов отправки *sbuf* всех процессов, записывает результат в буфер приема *rbuf* корневого процесса *root*. Псевдокод операции `MPI_Reduce` представлен алгоритмом 2.

Каждый процесс выполняет пофрагментную передачу буфера *sbuf* процессу *root* так, что фрагмент сообщения содержит целое количество элементов типа *dtype*, такое, что их суммарный размер g не превышает f. В случаях, когда один элемент типа *dtype* превышает размер одного фрагмента очереди, алгоритм с использованием системы очередей не может выполнить обмен сообщениями и вызывает компонент `coll/tuned`. Передача сообщений процессом i выполняется посредством записи g байт в фрагмент k очереди $q[i][k]$. Уведомление корневого процесса осуществляется путем записи в соответствующее процессу *root* поле управляющего блока $c[i][k][root]$. Цикл передачи выполняется каждым процессом пока все фрагменты сообщения не будут полностью записаны в очередь.

Корневой процесс *root* выполняет прием фрагментов от процессов в порядке обратной нумерации $p-1, p-2, \ldots, 0$. Порядок процессов определяется необходимостью поддержки некоммутативных операций и специфичным для Open MPI выполнением редукции. После приема каждого фрагмента сообщения, кроме фрагментов от процесса $p-1$, выполняется определенная операция редукции *op*. Фрагмент, полученный от процесса $p-1$, полностью копируется в буфер приема *rbuf* и является основой для последующего выполнения операции *op*.

Операция *op* обладает свойством ассоциативности и может быть как коммутативной, так и некоммутативной. В случае, когда операция обладает свойством коммутативности, корневой процесс может изменить порядок приема сообщений.

4.4. Алгоритм операции MPI_Reduce с использованием k–номиального дерева. Для оптимизации производительности операций `MPI_Reduce` применяются древовидные структуры данных, такие как плоское дерево, бинарное дерево, биномиальное, k–номиальное дерево.

2023, **24** (4), 339–351. doi 10.26089/NumMet.v24r424

В работе [14] дается оценка эффективности использования таких структур данных для операции `MPI_Reduce`. В алгоритме с организацией процессов в плоское дерево все процессы передают сообщения корневому процессу. В плоском дереве корневой процесс взаимодействует со всеми процессами. Алгоритм биномиального дерева (binomial tree) является частным случаем k–номиального дерева при $k = 2$. При таком подходе процессы организуют биномиальное дерево, представляющее собой совокупность биномиальных деревьев различных степеней. На рис. 1 приведен пример биномиального дерева для $p = 16$ процессов.

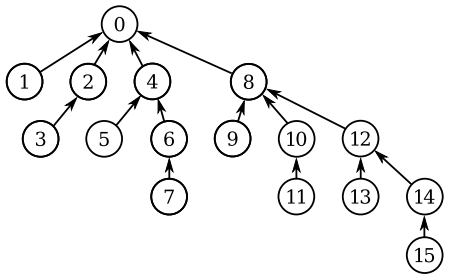

Рис. 1. Биномиальное дерево процессов для операции MPI_Reduce, $p = 16$

Fig. 1. MPI_Reduce binomial tree, $p = 16$

При выполнении операции `MPI_Reduce` процессы могут быть логически организованы в k–номиальное дерево. Частным случаем такого дерева является биномиальное дерево ($k = 2$). Каждый процесс на основе известного ему количества процессов и своего номера вычисляет свое расположение в дереве — количество дочерних процессов и их номера, а также номер родительского процесса.

При отсутствии дочерних процессов выполняется пофрагментная передача родительскому процессу сообщения *sbuf*. Остальные процессы, за исключением процесса 0, выполняют следующие этапы: 1) получение фрагмента; 2) формирование во временном буфере результата частичной редукции (выполнение операции *op*) полученных данных с буфером *sbuf*; 3) передача фрагмента родительскому процессу.

Процесс с номером 0 ожидает от дочерних процессов в порядке обратной нумерации уведомления о готовности фрагментов, после чего начинает копирование фрагментов из очереди в локальную память, формируя результат редукции во временном буфере, равном размеру целого числа элементов типа *dtype*, такого, что их суммарный размер g не превышает f. В случае, когда корневой процесс $root \neq 0$, процесс выполняет передачу результата редукции процессу *root*. Процесс *root* при выполнении цикла приема-передачи сообщения осуществляет проверку готовности результата редукции от процесса 0 и копирует его из разделяемой очереди в свой буфер *rbuf*. Если процесс 0 является корневым, то результат редукции сразу копируется в буфер *rbuf* процесса *root*.

Поддержка операций, не обладающих свойством коммутативности, обеспечивается за счет формирования дерева, корневым узлом которого является процесс с номером 0. Таким образом, всегда соблюдается порядок операндов операции *op*.

4.5. Алгоритм операции MPI_Allreduce. Операция `MPI_Allreduce` не имеет корневого процесса, и результат редукции формируется в каждом процессе. Для обеспечения поддержки некоммутативных операций, как следует из реализации операции корневой редукции, необходимо соблюдать порядок операндов. Таким образом, алгоритм состоит из следующих этапов: 1) `MPI_Reduce` формирует результат редукции в процессе с номером 0; 2) с помощью широковещательной рассылки `MPI_Bcast` буфер *rbuf* отправляется остальным процессам.

5. Организация экспериментов. Алгоритмы реализованы в виде отдельного компонента `coll/sharm` библиотеки Open MPI 5.0. Для предотвращения влияния возможного внеочередного выполнения инструкций процессором на процесс передачи сообщений через разделяемую память и уведомления после записи в буферы очередей и в управляющие блоки вызывается операция барьера записи в память (write memory barrier). Архитектурно-зависимые функции используются из подсистемы Open MPI OPAL (Open Portable Access Layer) [12]. Копирование фрагментов реализовано с использованием упаковки и распаковки буферов в памяти для поддержки производных типов данных. Экспериментальная часть выполнялась на сервере со следующей конфигурацией:

- двухпроцессорный сервер Intel Xeon Broadwell: 2 × Intel Xeon E5–2620 v4 (8 ядер, HyperThreading отключен, кэш-память L1 32 kB, L2 256 kB, L3 20 MB);
- оперативная память: 64 GB;
- ядро linux 5.18.11–100.fc35.x86_64 (ОС Fedora), gcc 11.3.1;
- MPI: Open MPI 5.0 (branch: v5.0.x).

2023, **24** (4), 339–351. doi 10.26089/NumMet.v24r424

В качестве теста производительности использовался пакет Intel MPI Benchmarks 2021 Update 3 (IMB–v2021.3). В работе была применена методика оценки эффективности коллективных операций, описанная в [10]. Сравнение проводилось с компонентом `coll/tuned` Open MPI. Для каждого размера сообщения производился запуск алгоритма с использованием плоского дерева (sharm linear) и k-номиального (sharm k-nomial) для $k = 2$. Параметры системы очередей использовались по умолчанию: размер фрагмента $f = 8192$ B, количество фрагментов $s = 8$.

При сравнении производительности запуск выполнялся для следующего количества процессов: $p = 8$ и $p = 16$.

Параметры запуска теста:

```
IMB-MPI1 reduce -off_cache 20,64 -iter 5000,250 -msglog 6:24 -sync 1
         -imb_barrier 1 -root_shift 0 -time 600.0
```

За время выполнения функции в каждом процессе принимается среднее время одного ее запуска (среднее время одной итерации цикла измерений). Псевдокод измерения среднего времени представлен алгоритмом 3. Пакет IMB вычисляет на основе среднего времени каждого процесса максимальное $t_{\max}$, среднее t_{avg} и минимальное $t_{\min}$ время выполнения операции. В каждом эксперименте тест IMB запускался 5 раз, значение $t_{\max}$ усреднялось по результатам трех запусков, исключая минимальный и максимальный результат $t_{\max}$. За время выполнения коллективной операции принимается максимальное время выполнения среди процессов.

Компонент `coll/tuned` реализует выбор алгоритма коллективной операции на основе размера коммуникатора, количества передаваемых сообщений и размера типа данных. При запуске Open MPI использовались следующие модули:

- PML: компонент ob1;
- BTL: компонент sm;
- SMSC: CMA.

На рис. 2 представлена зависимость нормализованного времени операции `MPI_Bcast` относительно компонента `coll/sharm` от размера передаваемого сообщения. Наибольшее сокращение времени работы в 5 раз достигнуто при запуске $p = 16$ процессов и размере сообщения равного размеру страницы памяти $m = a = 4$ kB.

Алгоритм 3. Алгоритм измерения времени выполнения коллективной операции
Algorithm 3. Algorithm for measuring the execution time of a collective operation

1: $time = 0$
2: **MPI_Barrier**($comm$)
3: **for** $i = 1$ **to** $nsamples$ **do**
4: $\quad t1 =$ **MPI_Wtime**()
5: $\quad$ **Collective**($parameters, \ldots, comm$)
6: $\quad t2 =$ **MPI_Wtime**()
7: $\quad time+ = (t2 - t1)$
8: **end for**
9: $time = time/nsamples$

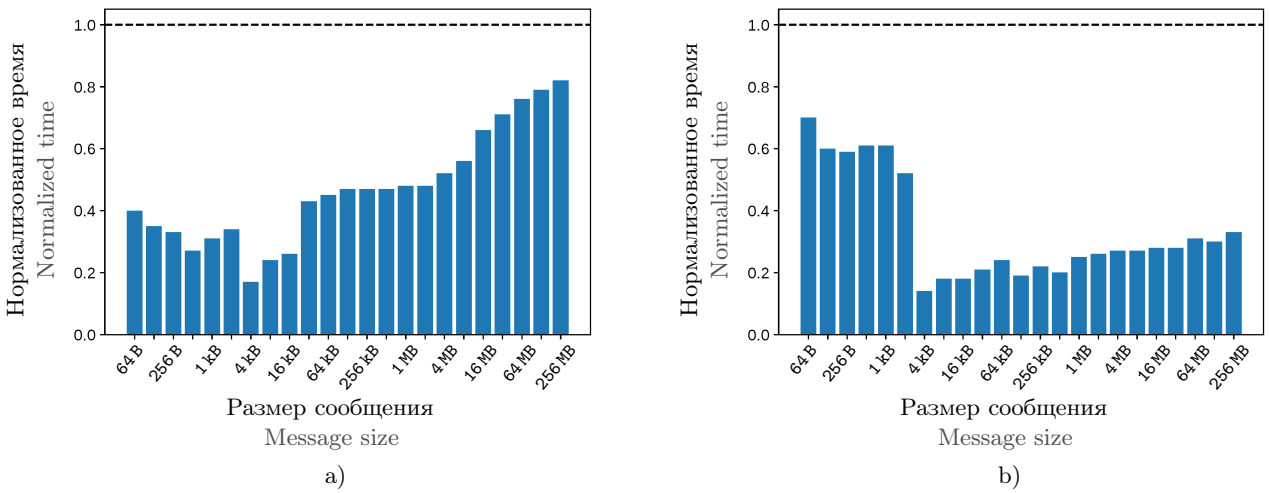

Рис. 2. Нормализованное время алгоритма `MPI_Bcast` компонента `coll/sharm` при запуске a) $p = 8$; b) $p = 16$ процессов. Время нормализовано относительно компонента coll/tuned

Fig. 2. `MPI_Bcast` normalized time of `coll/sharm` component: a) $p = 8$; b) $p = 16$ processes. Time is normalized to the `coll/tuned` component

2023, **24** (4), 339–351. doi 10.26089/NumMet.v24r424

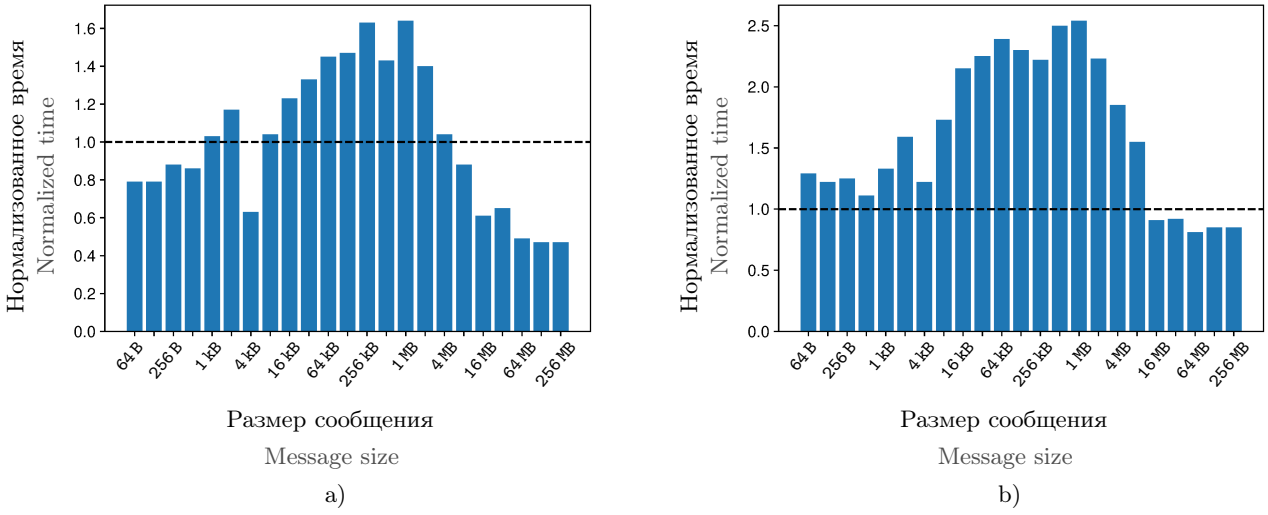

Рис. 3. Нормализованное время алгоритма `MPI_Reduce` компонента `coll/sharm` при запуске a) $p = 8$; b) $p = 16$ процессов. Время нормализовано относительно компонента coll/tuned

Fig. 3. `MPI_Reduce` normalized time of `coll/sharm` component: a) $p = 8$; b) $p = 16$ processes. Time is normalized to the `coll/tuned` component

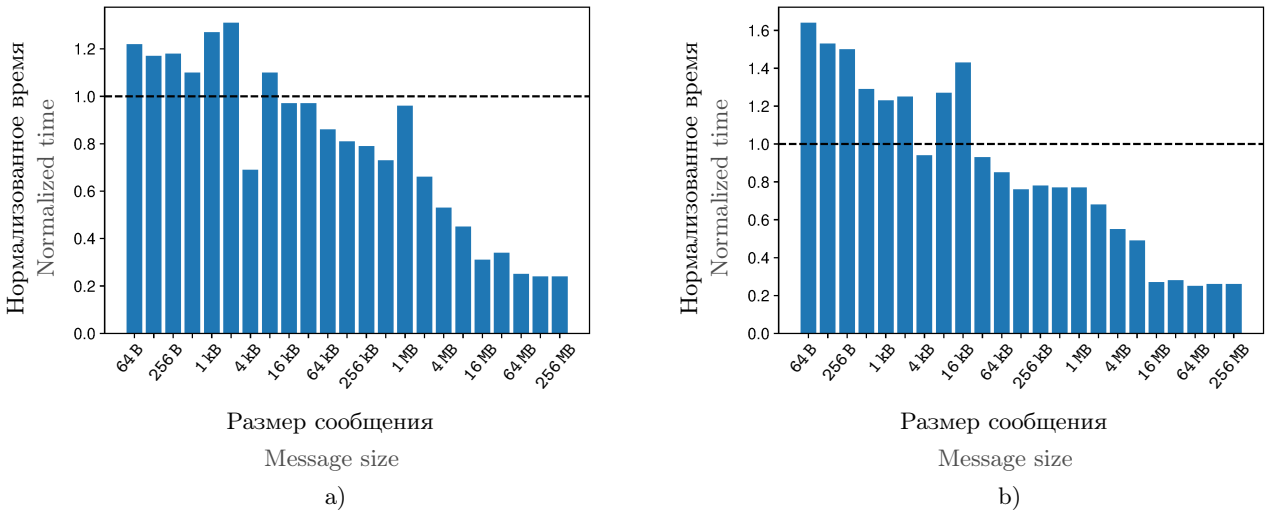

Рис. 4. Нормализованное время алгоритма `MPI_Reduce` компонента `coll/sharm` с использованием k–номиального дерева при запуске a) $p = 8$; b) $p = 16$ процессов для $k = 2$. Время нормализовано относительно компонента coll/tuned

Fig. 4. k–nomial `MPI_Reduce` normalized time of `coll/sharm` component: a) $p = 8$; b) $p = 16$ processes for $k = 2$. Time is normalized to the `coll/tuned` component

Таблица 1. Рекомендации выбора алгоритма MPI_Reduce в зависимости от количества процессов

Table 1. MPI_Reduce algorithm selection recommendations

Размер сообщения Message size	$p = 8$	$p = 16$
64 B–512 B	`coll/sharm` linear	`coll/tuned`
1 kB–2 kB	`coll/tuned`	`coll/tuned`
4 kB	`coll/sharm` linear	`coll/sharm` k–nomial
8 kB	`coll/tuned`	`coll/tuned`
16 kB	`coll/sharm` k–nomial	`coll/tuned`
32 kB–256 MB	`coll/sharm` k–nomial	`coll/sharm` k–nomial

2023, **24** (4), 339–351. doi 10.26089/NumMet.v24r424

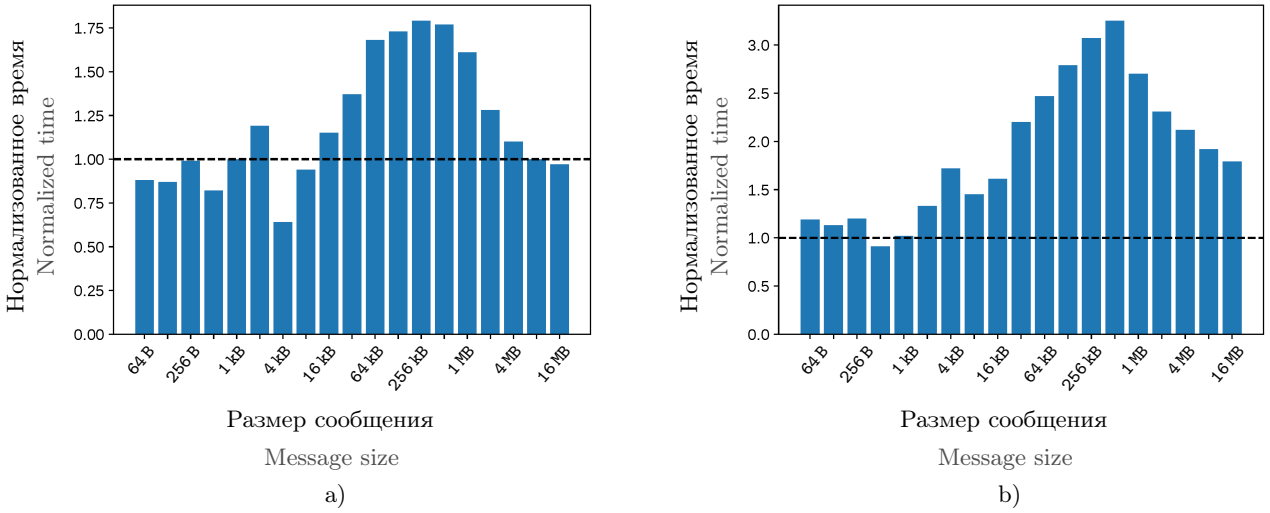

Рис. 5. Нормализованное время алгоритма `MPI_Allreduce` компонента `coll/sharm` при запуске a) $p = 8$; b) $p = 16$ процессов. Время нормализовано относительно компонента coll/tuned

Fig. 5. `MPI_Allreduce` normalized time of `coll/sharm` component: a) $p = 8$; b) $p = 16$ processes. Time is normalized to the `coll/tuned` component

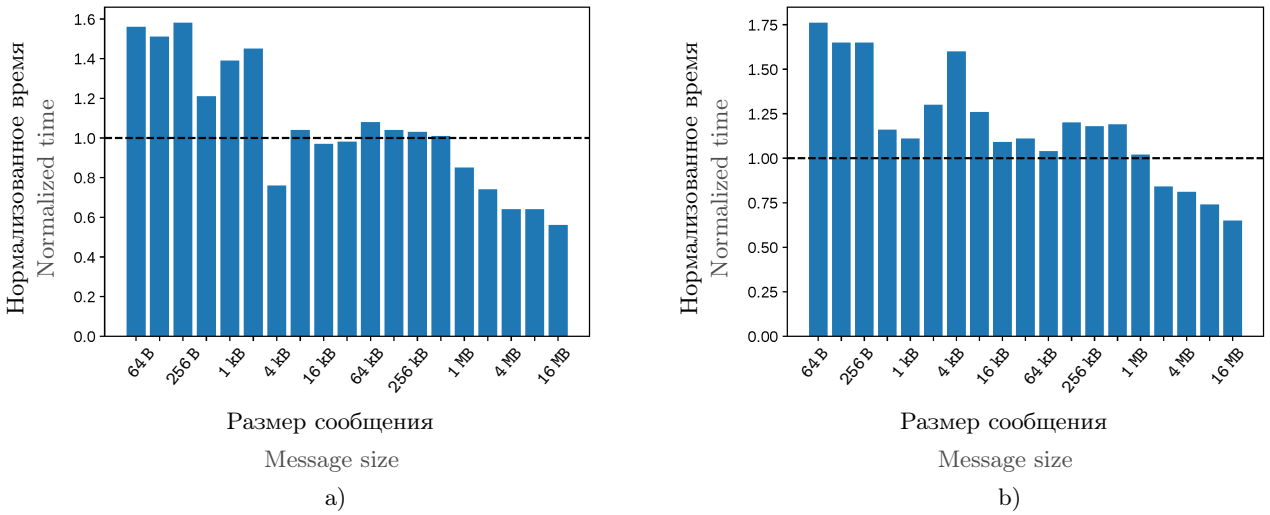

Рис. 6. Нормализованное время алгоритма `MPI_Allreduce` компонента `coll/sharm` с использованием k–номиального дерева при запуске a) $p = 8$; b) $p = 16$ процессов для $k = 2$. Время нормализовано относительно компонента coll/tuned

Fig. 6. k–nomial `MPI_Allreduce` normalized time of `coll/sharm` component: a) $p = 8$; b) $p = 16$ processes for $k = 2$. Time is normalized to the `coll/tuned` component

Для операции `MPI_Reduce` на рис. 3, 4 изображена зависимость времени алгоритмов плоского (`linear`) и k–номиального деревьев. Наибольшее сокращение времени выполнения достигнуто алгоритмом с использованием k–номиального дерева для $p = 16$ и размеров сообщений $m > 16$ MB. Табл. 1 содержит рекомендации использования алгоритмов компонентов `coll/sharm` и `coll/tuned` в зависимости от количества процессов и размера передаваемого сообщения.

Реализация операции `MPI_Allreduce`, как показано на рис. 5, в сравнении с компонентом `coll/tuned` при запуске 8 процессов показала сокращение времени в 1.4 раза (на 30% меньше) для сообщений равных по размеру странице памяти ($m = a = 4$ kB). Результат для алгоритма биномиального дерева ($k = 2$) показан на рис. 6. Алгоритм показал сокращение времени выполнения в 2 раза для больших размеров сообщения ($m > 16$ MB) при запуске $p = 16$ процессов. Табл. 2 содержит рекомендации использования алгоритмов операции `MPI_Allreduce` компонентов `coll/sharm` и `coll/tuned` в зависимости от количества процессов и размера передаваемого сообщения.

2023, **24** (4), 339–351. doi 10.26089/NumMet.v24r424

Таблица 2. Рекомендации выбора алгоритма MPI_Allreduce в зависимости от количества процессов

Table 2. MPI_Allreduce algorithm selection recommendations

Размер сообщения Message size	$p = 8$	$p = 16$
64 B–512 B	`coll/sharm` linear	`coll/tuned`
1 kB–2 kB	`coll/tuned`	`coll/tuned`
4 kB–8 kB	`coll/sharm` linear	`coll/tuned`
16 kB–32 kB	`coll/sharm` k–nomial	`coll/tuned`
64 kB–512 kB	`coll/tuned`	`coll/tuned`
1 MB	`coll/sharm` k–nomial	`coll/tuned`
2 MB–16 MB	`coll/sharm` k–nomial	`coll/sharm` k–nomial

6. Заключение. Разработанные алгоритмы реализуют операции `MPI_Bcast`, `MPI_Reduce` и `MPI_Allreduce` с использованием системы очередей в разделяемой памяти. В лучшем случае, алгоритмы показывают сокращение времени работы в 5 раз по сравнению с алгоритмами компонента `coll/tuned` библиотеки Open MPI.

По результатам проведенных экспериментов рекомендуется использовать систему очередей в разделяемой памяти для операции `MPI_Reduce` при размерах сообщений $m < 64$ kB. Также алгоритм показывает лучшее сокращение времени выполнения при размерах сообщений $m > 16$ MB. Использование алгоритма k–номиального дерева при $k = 2$ рекомендуется при больших размерах сообщений.

Алгоритм `MPI_Allreduce` целесообразно использовать для обмена сообщениями, размер которых $m > 1$ MB. В случае запуска $p = 8$ процессов и сообщения размером $m = a = 4$ kB получено сокращение времени работы в 1.6 раз (на 40% меньше).

В дальнейшем планируется доработать компонент `coll/sharm`, реализовав полный набор блокирующих коллективных операций, и предложить реализацию в проект Open MPI. Также необходимо оценить влияние параметров очереди (размер фрагмента f, количество фрагментов s) на время выполнения алгоритмов и поддержать в алгоритмах использование технологий CMA, KNEM, XPMEM.

Список литературы

1. MPI: A Message-Passing Interface Standard. Version 4.0. http://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf. Дата обращения: 21 сентября 2023.
2. *Jain S., Kaleem R., Balmana M.G., Langer A., Durnov D., Sannikov A., Garzaran M.* Framework for scalable intra-node collective operations using shared memory // Proc. Int. Conf. on High Performance Computing, Networking, Storage, and Analysis. Vol. 1. Piscataway: IEEE Press, 2018. 374–385. doi 10.1109/SC.2018.00032.
3. *Ladd J.S., Venkata M.G., Shamis P., Graham R.L.* Collective framework and performance optimizations to open MPI for Cray XT platforms // Proc. 53rd Cray User Group Meeting. https://www.ornl.gov/publication/collective-framework-and-performance-optimizations-open-mpi-cray-xt-platforms. Дата обращения: 22 сентября 2023.
4. High-Performance Portable MPI. https://www.mpich.org/. Дата обращения: 22 сентября 2023.
5. Cross Memory Attach. https://lwn.net/Articles/405284/. Дата обращения: 22 сентября 2023.
6. High-Performance Intra-Node MPI Communication. https://knem.gitlabpages.inria.fr. Дата обращения: 22 сентября 2023.
7. *Goglin B., Moreaud S.* KNEM: a generic and scalable kernel-assisted intra-node MPI communication framework // Journal of Parallel and Distributed Computing. 2013. **73**, N 2. 176–188. doi 10.1016/j.jpdc.2012.09.016.
8. Linux Cross-Memory Attach. https://github.com/hjelmn/xpmem. Дата обращения: 22 сентября 2023.
9. Open Source High Performance Computing. http://www.open-mpi.org. Дата обращения: 22 сентября 2023.
10. *Graham R.L., Shipman G.* MPI support for multi-core architectures: optimized shared memory collectives // Lecture Notes in Computer Science. Vol. 5205. Heidelberg: Springer, 2008. 130–140. doi 10.1007/978-3-540-87475-1_21.
11. MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, RoCE, and Slingshot. https://mvapich.cse.ohio-state.edu/. Дата обращения: 22 сентября 2023.

2023, **24** (4), 339–351. doi 10.26089/NumMet.v24r424

12. *Курносов М.Г., Токмашева Е.И.* Алгоритм широковещательной передачи стандарта MPI на базе разделяемой памяти многопроцессорных NUMA-узлов // Вестник СибГУТИ. 2020. **49**, № 1. 42–59.
13. *Li S., Hoefler T., Snir M.* NUMA-aware shared-memory collective communication for MPI // Proc. 22nd Int. Symposium on High-Performance Parallel and Distributed Computing. New York: ACM Press, 2013. 85–96. doi 10.1145/2462902.2462903.
14. *Курносов М.Г.* Анализ масштабируемости алгоритмов коллективных обменов на распределенных вычислительных системах // Материалы 4-й Всероссийской научно-технической конференции "Суперкомпьютерные технологии". Том 2. Ростов-на-Дону: Изд-во Южного Федерального Университета, 2016. 48–52.

Поступила в редакцию
24 июля 2023 г.

Принята к публикации
12 сентября 2023 г.

Информация об авторах

Алексей Андреевич Романюта — аспирант; Сибирский государственный университет телекоммуникаций и информатики, кафедра вычислительных систем, ул. Кирова, 86, 630102, Новосибирск, Российская Федерация.

Михаил Георгиевич Курносов — д.т.н, профессор; 1) Сибирский государственный университет телекоммуникаций и информатики, кафедра вычислительных систем, ул. Кирова, 86, 630102, Новосибирск, Российская Федерация; 2) Институт физики полупроводников имени А. В. Ржанова Сибирского отделения РАН, пр-кт Лаврентьева, 13, 630090, Новосибирск, Российская Федерация.

References

1. MPI: A Message-Passing Interface Standard. Version 4.0. http://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf. Cited September 21, 2023.
2. S. Jain, R. Kaleem, M. G. Balmana, et al., "Framework for Scalable Intra-Node Collective Operations Using Shared Memory," in *Proc. Int. Conf. on High Performance Computing, Networking, Storage, and Analysis, Dallas, USA, November 11–16, 2018* (IEEE Press, Piscataway, 2018), Vol. 1, pp. 374–385. doi 10.1109/SC.2018.00032.
3. J. S. Ladd, M. G. Venkata, P. Shamis, and R. L. Graham, "Collective Framework and Performance Optimizations to Open MPI for Cray XT Platforms," in *Proc. 53rd Cray User Group Meeting, Fairbanks, Alaska, USA, May 23–26, 2011.* https://www.ornl.gov/publication/collective-framework-and-performance-optimizations-open-mpi-cray-xt-platforms. Cited September 22, 2023.
4. High-Performance Portable MPI. https://www.mpich.org/. Cited September 22, 2023.
5. Cross Memory Attach. https://lwn.net/Articles/405284/. Cited September 22, 2023.
6. High-Performance Intra-Node MPI Communication. https://knem.gitlabpages.inria.fr. Cited September 22, 2023.
7. B. Goglin and S. Moreaud, "KNEM: a Generic and Scalable Kernel-Assisted Intra-Node MPI Communication Framework," J. Parallel Distrib. Comput. **73** (2), 176–188 (2013). doi 10.1016/j.jpdc.2012.09.016.
8. Linux Cross-Memory Attach. https://github.com/hjelmn/xpmem. Cited September 22, 2023.
9. Open Source High Performance Computing. http://www.open-mpi.org. Cited September 22, 2023.
10. R. L. Graham and G. Shipman, "MPI Support for Multi-Core Architectures: Optimized Shared Memory Collectives," in *Lecture Notes in Computer Science* (Springer, Heidelberg, 2008), Vol. 5205, pp. 130–140. doi 10.1007/978-3-540-87475-1_21.
11. MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, RoCE, and Slingshot. https://mvapich.cse.ohio-state.edu/. Cited September 22, 2023.
12. M. Kurnosov and E. Tokmasheva, "Shared Memory Based MPI Broadcast Algorithms for NUMA Systems," in *Communications in Computer and Information Science* (Springer, Cham, 2020), Vol. 1331, pp. 473–485. doi 10.1007/978-3-030-64616-5_41.

2023, **24** (4), 339–351. doi 10.26089/NumMet.v24r424

13. S. Li, T. Hoefler, and M. Snir, "NUMA-Aware Shared-Memory Collective Communication for MPI," in *Proc. 22nd Int. Symposium on High-Performance Parallel and Distributed Computing, New York, USA, June 17–21, 2013* (ACM Press, New York, 2013), pp. 85–96. doi 10.1145/2462902.2462903.
14. M. G. Kurnosov, "Analysis of the Scalability of Collective Exchange Algorithms on Distributed Computing Systems," in *Proc. 4th All-Russian Scientific and Technical Conf. on Supercomputer Technologies, Rostov-on-Don, Russia, September 19–24, 2016* (Southern Federal Univ. Press, Rostov-on-Don, 2016), Vol. 2, pp. 48–52 [in Russian].

Received
July 24, 2023

Accepted for publication
September 12, 2023

Information about the authors

Alexey A. Romanyuta — PhD Student; Siberian State University of Telecommunication and Informatics, Computer Systems Department, ulitsa Kirova, 86, 630102, Novosibirsk, Russia.

Mikhail G. Kurnosov — Dr. Sci., Professor; 1) Siberian State University of Telecommunication and Informatics, Computer Systems Department, ulitsa Kirova, 86, 630102, Novosibirsk, Russia; 2) Rzhanov Institute of Semiconductor Physics SB RAS, Lavrentyev prospekt, 13, 630090, Novosibirsk, Russia.