УДК 004.272

АЛГОРИТМЫ ОПТИМИЗАЦИИ ВЫПОЛНЕНИЯ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ НА ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ СИСТЕМАХ ПРИ РЕШЕНИИ ЗАДАЧ МОДЕЛИРОВАНИЯ ФИЗИЧЕСКИХ ПРОЦЕССОВ

© К. В. Павский[1,2], М. Г. Курносов[1,2], А. В. Ефимов[1,2], К. Е. Крамаренко[1,2], Е. Н. Перышкова[1,2], Поляков А. Ю.[3]

[1]*Институт физики полупроводников им. А. В. Ржанова СО РАН,*
630090, г. Новосибирск, просп. Академика Лаврентьева, 13
[2]*Сибирский государственный университет телекоммуникаций и информатики,*
630102, г. Новосибирск, ул. Кирова, 86
[3]*Networking SW & Sys Arch NVIDIA Corporation,*
2788, San Tomas Expy, Santa Clara, CA 95051, USA
E-mail: pkv@isp.nsc.ru

Предложены алгоритмы, позволяющие повысить эффективность исполнения параллельных программ на высокопроизводительных вычислительных системах, в частности при решении задач моделирования физических процессов. Разработанные алгоритмы ориентированы на оптимизацию выполнения коллективных операций на многопроцессорных SMP/NUMA-узлах в стандарте MPI. Алгоритмы блокировки чтения-записи повышают эффективность синхронизации доступа к общей памяти относительно алгоритмов, используемых в библиотеке Open PMIx.

Ключевые слова: высокопроизводительные системы, параллельные программы, коллективные операции, синхронизация доступа к памяти, блокировка чтения-записи, MPI, PMIx.

DOI: 10.15372/AUT20210515

Введение. Основой для создания перспективных электронных устройств и приборов, в частности твердотельных лазеров, матричных светодиодов и фотоприёмников, служат плотные пространственно-упорядоченные массивы квантово-размерных структур (например, квантовых точек). Такие массивы можно получать путём гетероэпитаксии на структурированных подложках [1]. Для моделирования элементарных процессов, сопровождающих рост гетероструктур, а также учёта вероятностной природы этих процессов на характерных временах роста требуются машинная обработка бóльших объёмов информации и трудоёмкие вычисления на высокопроизводительных вычислительных системах (ВС).

Современные высокопроизводительные системы обработки информации характеризуются мультиархитектурной организацией вычислительных узлов и иерархической структурой коммуникационной сети. В списке Top 500 (57-я редакция, июнь 2021 г.) 92 % высокопроизводительных систем относится к кластерным и имеют высокую масштабируемость. Например, система Fugaku (1 место в 57-й редакции Top 500) состоит из 158 976 вычислительных узлов [2]. Время передачи информации между элементарными машинами (ЭМ) в таких ВС зависит от их взаимного расположения в системе. Накладные расходы на взаимодействие между ЭМ, расположенными в пределах одного узла, заметно меньше времени взаимодействия между ЭМ, лежащими в разных узлах, что следует учитывать при решении задач.

Эффективные решения трудоёмких задач, в частности задач по параллельному моделированию наноструктур с квантовыми точками, на высокопроизводительных масштабируемых вычислительных системах требуют создания нового и развития имеющегося инструментария — математических моделей, средств анализа функционирования, алгоритмов и системного программного обеспечения (ПО), например библиотек стандарта MPI и средств многопоточного программирования. Основное требование к системному ПО заключается в учёте многоуровневого параллелизма ВС и иерархической организации средств доступа к памяти. На всех функциональных уровнях ВС инструментарий должен использовать архитектурно-ориентированные подходы для минимизации времени запуска параллельных программ и передачи данных в ходе их выполнения: использование эффективных алгоритмических и программно-аппаратных методов дифференцированных (point-to-point) и коллективных обменов информацией (collective communication) на уровне вычислительных узлов (MPI, UCX, SHMEM), а при необходимости передачу данных в пределах одного узла — его разделяемую память SMP/NUMA-узлов [3–6].

Цель данной работы — повышение эффективности выполнения параллельных программ на высокопроизводительных системах.

Алгоритмы коллективных операций для ВС на базе многопроцессорных SMP-систем. Подавляющее большинство современных ВС строится на базе многопроцессорных SMP/NUMA-узлов. Эффективность реализации коммуникационных операций при выполнении параллельных программ на этих подсистемах будет различной в силу иерархической организации коммуникационной сети ВС.

Одной из широко используемых коллективных операций является операция All-to-all, при которой каждый процесс выполняет n операций обмена и передачи (n — число процессов в параллельной программе). В стандарте MPI коллективная операция MPI_Alltoall является реализацией трансляционно-циклического обмена, когда каждый процесс посылает различные данные разным получателям:

$$\mathrm{MPI_Alltoall}\,(\mathrm{sbuf}, \mathrm{scount}, \mathrm{stype}, \mathrm{rbuf}, \mathrm{rcount}, \mathrm{rtype}, \mathrm{comm}).$$

Пример работы коллективной операции All-to-all показан на рис. 1. Функция вызывается всеми процессами группы коммуникатора comm. Процесс i передаёт адрес j-го блока буфера sbuf, содержащего scount элементов типа stype, процессу j. Процесс j помещает полученные данные в i-й блок буфера приёма rbuf. Количество посланных данных должно быть равно количеству полученных данных для каждой пары процессов.

В библиотеках MVAPICH и Open MPI используются три различные реализации данной операции: алгоритм Дж. Брука, блочный алгоритм и алгоритм попарных обменов [6, 7]. Использование различных алгоритмов, реализующих одну операцию в рамках одной библиотеки, напрямую связано с эффективностью данных алгоритмов в зависимости от размеров передаваемых сообщений [8]. В большинстве случаев решение о выборе алгоритма принимается фиксированными правилами на основе размера сообщения.

Далее представлено описание основных алгоритмов реализации коллективной операции All-to-all, выполнен теоретический анализ вычислительной сложности, проведено экспериментальное исследование зависимости времени передачи сообщений от размера передаваемого сообщения, а также предложены рекомендации по выбору алгоритмов.

1. Алгоритм Дж. Брука. На шаге $k = 0, 1, \ldots, [\log_2 n] - 1$ ветвь i передаёт все принятые сообщения ветви $(i - 2^k + n) \bmod n$ и принимает сообщения от ветви $(i + 2^k) \bmod n$. Сообщения размещаются в памяти со смещением, поэтому в конце работы алгоритма каждая ветвь i циклически сдвигает сообщения на i позиций вниз. Вычислительная сложность алгоритма Брука определяется как $O(\log n)$.

2. Блочный алгоритм. Каждая ветвь выполняет n операций передачи и приёма сообщений. При этом операции send/recv группируются в блоки из block коммуникационных

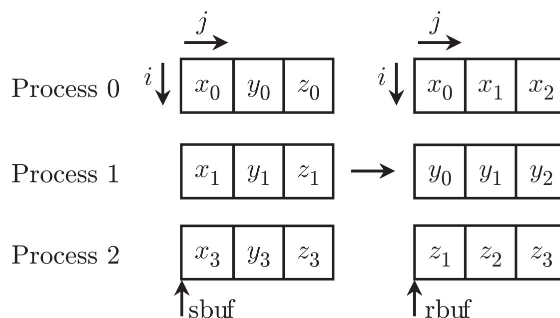

Рис. 1. Коллективная операция MPI_Alltoall

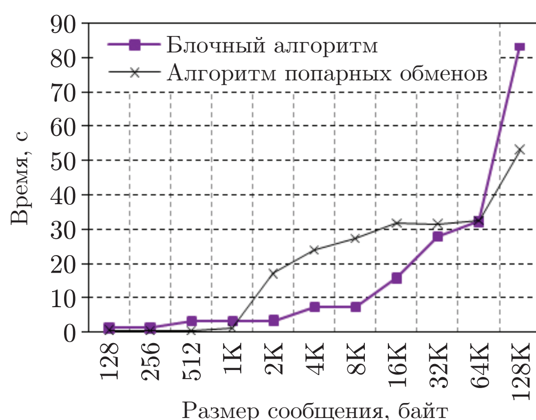

Рис. 2. Среднее время работы блочного алгоритма и алгоритма попарных обменов в зависимости от размера передаваемого сообщения; 16 SMP-узлов сервера Intel SR2520SAF (8 процессов)

операций для сокращения накладных расходов. Блочный алгоритм имеет линейную вычислительную сложность.

3. Алгоритм попарных обменов. На шаге $k = 0, 1, \ldots, n-1$ ветвь i посылает сообщение ветви $(i + k)$ и принимает сообщение от ветви $(i - k)$. На каждом шаге все ветви имеют входящее и исходящее сообщения, поэтому данные передаются напрямую от источника к получателю. Алгоритм попарных обменов имеет линейную вычислительную сложность.

В рамках библиотеки MVAPICH установлены следующие правила выбора алгоритма реализации коллективной операции All-to-all: для передачи коротких сообщений (менее 256 байт) применяется алгоритм Дж. Брука, для сообщений среднего размера (от 256 байт до 32 Кбайт) рекомендуется использовать блочный алгоритм, а для передачи сообщений больших размеров (от 32 Кбайт) — алгоритм попарных обменов.

Для оценки времени реализации коллективной операции All-to-all на вычислительных системах с многопроцессорными SMP-узлами авторами разработано программное средство, которое в отличие от известных решений учитывает иерархическую организацию коммуникационной сети ВС.

На рис. 2 показано время работы блочного алгоритма и алгоритма попарных обменов в зависимости от размера передаваемого сообщения. Эксперименты были организованы на ВС с SMP-узлами, укомплектованной 18 вычислительными серверами на базе платформы Intel SR2520SAF. В качестве тестовой задачи рассматривается параллельная программа, реализующая вызов каждым процессом коллективной операции All-to-all. Каждый процесс инициализирует буфер приёма-передачи сообщения. Результаты первого вызова

операции All-to-all не учитываются в итоговой оценке времени выполнения операции из-за возможной отложенной инициализации подсистем библиотеки MPI. Время выполнения коллективной операции All-to-all оценивается путём измерения времени count выполнений в цикле (на рис. 2 count = 50). На каждой итерации цикла реализуется барьерная синхронизация всех ветвей параллельной программы и очередной запуск операции All-to-all. За время t выполнения информационного обмена принимается среднее время одного запуска.

Экспериментальные исследования в основном подтверждают официальные рекомендации по выбору алгоритма реализации коллективной операции All-to-all. Однако на основе проведённых экспериментов блочный алгоритм рекомендуется использовать для передачи сообщений среднего размера (от 1 до 64 Кбайт). Алгоритм попарных обменов рекомендуется применять для передачи сообщений большого размера (от 64 Кбайт).

Алгоритмы коллективных операций для ВС на базе многопроцессорных NUMA-систем. Увеличение числа процессоров и процессорных ядер на одном вычислительном узле привело к широкому распространению NUMA-архитектуры, в которой каждый процессор имеет один или несколько интегрированных контроллеров доступа к разделяемой оперативной памяти [9]. На таких ВС алгоритмы коллективных операций MPI, как правило, учитывают иерархию системы — выполняют межмашинные обмены через коммуникационную сеть (InfiniBand, Ethernet, Slingshot и др.), а при необходимости передачи данных в пределах одного узла — обмены через разделяемую память.

Здесь представлены алгоритмы реализации коллективных операций через разделяемую память многопроцессорных NUMA-узлов. Алгоритмы реализованы на базе Open MPI и основаны на создании системы очередей в сегменте разделяемой памяти, через который процессы выполняют конвейерную передачу фрагментов сообщения (pipelining, copy-in/copy-out, CICO). Для сокращения накладных расходов на копирование между NUMA-узлами выполняется динамический анализ топологии MPI-коммуникатора и размещение очередей в памяти локальных NUMA-узлов процессов. В корневых операциях, например Bcast, корневой процесс реализует конвейерную передачу фрагментов сообщения. Текущий фрагмент копируется в следующий свободный буфер очереди корневого процесса, после чего корень уведомляет дочерние процессы о готовности фрагмента через флаги в разделяемой памяти. Некорневые процессы копируют фрагмент, уведомляют корень, цикл повторяется. Выполнен теоретический и экспериментальный анализ эффективности предложенных алгоритмов. Найдены оптимальные значения размера f буферов очереди в сегменте разделяемой памяти и их количество s (длина очереди), которые помещаются в b байт доступной памяти и обеспечивают минимум времени выполнения информационных обменов:

$$f = \sqrt{\frac{m}{\lceil m/b \rceil} \frac{t_W}{t}} \approx \sqrt{\frac{bt_W}{t}}, \qquad s = \frac{b}{f} = \sqrt{\frac{bt}{t_W}},$$

где m — размер сообщения, t_W — время получения уведомления от корня о готовности буфера, t — время копирования одного байта сообщения. Учитывая $t_W > t$ и страничное выделение памяти для очередей с локальных NUMA-узлов, целесообразно использовать буферы размером $f \geqslant \sqrt{b}$, округлённым до ближайшего сверху числа, кратного размеру страницы памяти (4 Кбайт на Intel64).

На рис. 3, 4 показано время работы предложенного здесь алгоритма SHMBcast и алгоритмов MVAPICH и Open MPI coll/sm. За счёт учёта NUMA-топологии узлов и распределения процессов по ним разработанный алгоритм позволил на 20–40 % сократить время выполнения операции MPI_Bcast по сравнению с алгоритмом MVAPICH и на 20–60 % процентов по сравнению с алгоритмом Open MPI coll/sm.

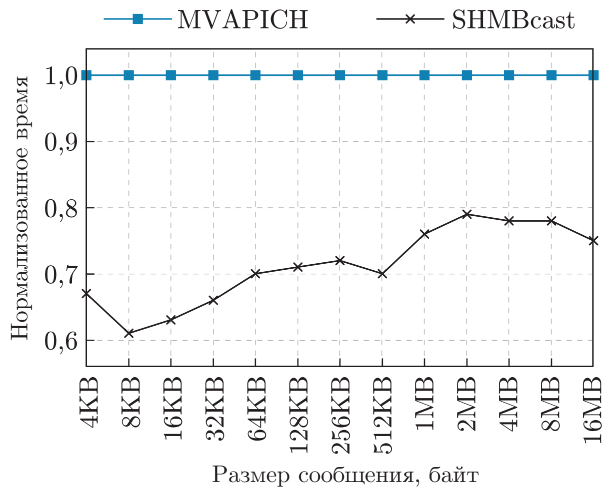

Рис. 3

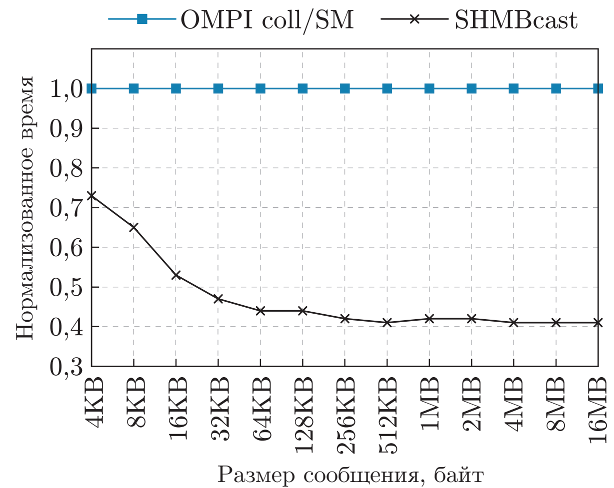

Рис. 4

Рис. 3. Время работы алгоритма SHMBcast и алгоритма MVAPICH на системе из двух NUMA-узлов (8 процессов) с использованием теста IMB (-root_shift on): время нормализовано относительно времени MVAPICH

Рис. 4. Время работы алгоритма SHMBcast и алгоритма Open MPI coll/sm на системе из двух NUMA-узлов Intel Xeon Broadwell (16 процессов) с использованием теста IMB (-root_shift on): время нормализовано относительно времени coll/sm

Основное время выполнения алгоритма SHMBcast составляет копирование корневым процессом фрагментов из входного буфера операции MPI_Bcast. Важно, чтобы буфер пользователя и очередь корневого процесса находились в памяти одного NUMA-узла. Без явного выделения страниц памяти с NUMA-узла текущего процесса буферы очереди могут быть размещены на NUMA-узле процесса 0, который осуществляет начальное формирование сегмента разделяемой памяти и неявно запускает механизм опережающего выделения страниц памяти с текущего NUMA-узла (linux readahead). В предложенном алгоритме для установления корректной привязки страниц памяти к NUMA-узлам на время инициализации сегмента вызовом madvise(MADV_RANDOM) отключается опережающее чтение страниц.

Оптимизация блокировок чтения-записи. Параллельные задачи содержат метаданные с описанием ресурсов ВС, необходимых для их решения, и могут состоять из одной и более параллельных программ. Интерфейс управления процессами — Process Management Interface (PMI) [10–12] — абстрагирует детали реализации среды исполнения (СИ) от прикладных процессов, формирующих параллельную задачу. Информация о СИ в PMI хранится в виде базы данных ключ-значение (Key-Value Database, KVDb). Интенсивность чтения из базы данных KVDb значительно превышает интенсивность записи прикладными процессами. Размещение базы данных KVDb в общей памяти вычислительного узла обладает следующими преимуществами: масштабируемое потребление памяти без дублирования информации на каждом прикладном процессе; параллельный независимый доступ прикладных процессов к чтению информации из общей памяти без участия процесса-сервера. Выполнение операции записи требует остановки всех процессов на чтении, что влечёт за собой накладные расходы на синхронизацию доступа к общей памяти. Инструментом синхронизации доступа к общей памяти является блокировка чтения-записи [13].

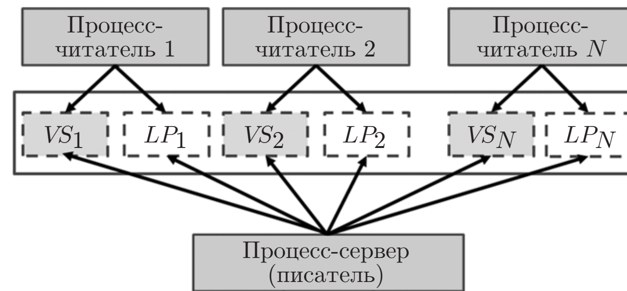

Рис. 5. Алгоритм блокировки чтения-записи $N(\text{mutex} + \text{signal})$

Далее представлены алгоритмы блокировки чтения-записи N(mutex+signal) и N-MCS, где N — число процессов-читателей общей памяти, в пределе соответствующее максимальному числу процессорных ядер на вычислительном узле. Алгоритмы повышают эффективность доступа к общей памяти для PMI Exascale (PMIx — наиболее актуальная и стандартизированная версия PMI, разработанная для организации выполнения прикладных процессов параллельных программ на ВС экзафлопсного уровня производительности) [11, 12].

Алгоритм блокировки чтения-записи. Данный алгоритм реализует (рис. 5) оптимизацию доступа к общей памяти в конкурентном режиме относительно алгоритма $2N$-mutex [10], используемого в настоящий момент в библиотеке Open PMIx.

Алгоритмы блокировки чтения-записи $2N$-mutex и N(mutex + signal) используют сигнальные примитивы синхронизации. Оптимизация заключается в замене примитива POSIX/mutex в алгоритме $2N$-mutex, сигнальной переменной VS_x, применяемой в алгоритме $N(\text{mutex} + \text{signal})$ и не требующей выполнения атомарных операций.

Использование сигнальных примитивов обусловлено тем, что стандарт POSIX Threads не требует соблюдения очерёдности доступа [12], а реализация примитива POSIX/mutex в библиотеке GNU C Library таким свойством не обладает. Блокировка сигнальных примитивов гарантирует, что новые попытки доступа для чтения не будут создавать дополнительной конкуренции для захвата основных примитивов (LP_1–LP_N) доступа к общей памяти.

Захватив основной примитив LPx, сигнальный примитив LSx освобождается. Таким образом, исключается возможность взаимоблокировки. При доступе для записи сначала осуществляется захват всех сигнальных примитивов ($LS_1 - LS_N$). Далее последовательно захватываются основные примитивы ($LP_1 - LP_N$).

Эффективность конкурентного доступа к общей памяти на запись оценивалась как накладные расходы времени на ожидание блокировки от момента инициализации операции до момента захвата всех основных примитивов ($LP_1 - LP_N$).

Алгоритм блокировки чтения-записи *N*-MCS разработан на основе примитива MCS, используемого вместо пары примитивов синхронизации LS_i, LP_i, и гарантирует очерёдность доступа.

Для реализации POSIX/mutex в библиотеке GNU C Library характерно неограниченное количество попыток захвата примитива синхронизации и, следовательно, в среднем выполнение атомарной операции требуется более одного раза. В реализации MCS как для захвата, так и для освобождения примитива синхронизации атомарная операция выполняется ровно один раз. Другой особенностью MCS по сравнению с mutex является процедура передачи права владения примитивом синхронизации, которая осуществляется владельцем доступа и не требует дополнительных действий со стороны претендента на блокировку (доступ). Таким образом, структура и логика MCS гарантируют очерёдность.

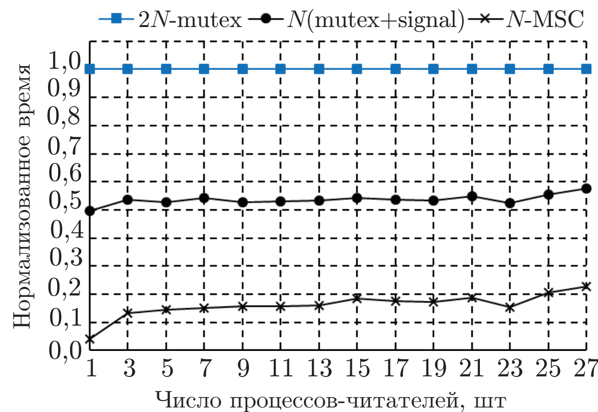

Рис. 6. Время работы алгоритмов $2N$-mutex, N(mutex+signal) и N-MCS: время нормализовано относительно времени $2N$-mutex

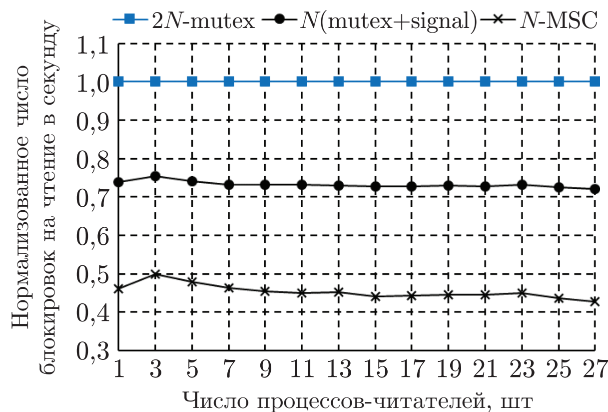

Рис. 7. Интенсивность чтения алгоритмов $2N$-mutex, N(mutex + signal) и N-MCS: значения нормализованы относительно $2N$-mutex

Это позволяет обойтись без сигнального элемента, необходимого в алгоритмах $2N$-mutex и N(mutex + signal).

Эффективность алгоритмов блокировок чтения-записи оценивалась в конкурентном режиме доступа к общей памяти двумя показателями: временем захвата блокировки на запись и суммарной интенсивностью захвата блокировок на чтение всеми процессами-читателями. Результаты оценки эффективности, представленные на рис. 6, 7, получены с помощью микротеста [14], в рамках которого все процессы-читатели непрерывно обращаются к общей памяти для чтения до тех пор, пока процесс-писатель не выполнит заданное число операций записи. Реализации алгоритмов выполнялись на системе NUMA, состоящей из 2 процессоров Intel Xeon E5-2680 v4 при 28 процессах-читателях (- -bind-to-core).

На рисунках видно, что предложенные алгоритмы блокировки чтения-записи обеспечивают на 20–80 % более эффективный доступ к общей памяти NUMA-узлов по сравнению с блокировками, используемыми в библиотеке Open PMIx.

Анализ алгоритмов доступа к данным в KVDb. Важной особенностью PMIx является параллельное и интенсивное чтение данных KVDb, выполняемое PMIx-клиентами, в то время как только один процесс (PMIx-сервер) осуществляет доступ для записи и такая операция производится относительно редко. Описаны блокировки, повышающие эффективность выполнения примитивов чтения и записи в KVDb. Необходимо произвести анализ методов доступа к данным в KVDb, чтобы определить наиболее вычислительно

Таблица 1

Количество клиентов	$P_7 - P_1$	$P_2 - P_1$	$P_3 - P_2$	$P_4 - P_3$	$P_5 - P_4$	$P_6 - P_5$	$P_7 - P_6$
$N = 2$	4182	931	284	1511	437	86	930
$N = 8$	4231	759	175	1832	404	113	946
$N = 16$	4493	803	271	1769	435	209	1002
$N = 28$	4318	775	240	1735	441	143	981

сложные участки кода в методах для дальнейшей их оптимизации. Для проведения измерений разработана функция Counter на основе машинной инструкции rdtsc, которая возвращает количество тактов, выполненных процессором с момента последней перезагрузки системы. В качестве схемы оценки эффективности методов доступа к базе данных предлагается следующая последовательность действий:

1) исходный код метода доступа разбивается на части (инициализация, синхронизация, поиск, распаковка ключей, завершение и т. п.);

2) в начале и конце каждой части кода вставляется функция Counter;

3) для каждой части производится оценка вычислительной сложности.

Доступ к базе данных осуществляется через примитивы синхронизации (Commit, Fence), чтения (Get) и записи (Put). Приведём пример использования предложенной схемы на основе анализа метода Get.

Пример анализа метода Get (с применением блокировки $N(\mathrm{mutex} + \mathrm{signal})$. Функция Counter, представленная метками (P_1) – (P_7), расставлялась в следующих фрагментах исходного кода исследуемого метода: P_1 — при входе в метод Get; P_2 — перед блокировкой примитива синхронизации; P_3 — после блокировки примитива синхронизации; P_4 — после нахождения ключа; P_5 — после распаковки ключа; P_6 — после разблокировки примитива синхронизации; P_7 — при выходе из метода Get.

В качестве экспериментальной базы использовались вычислительные NUMA-узлы с процессорами Intel (2 x Xeon E5-2680 v4) с числом ядер 28. Для вызова метода Get применялась утилита pmix_intra_perf (входит в состав исходного кода библиотеки Open PMIx). Длина ключа — 50 символов. Количество клиентов N, считывающих ключи, варьировалось от 2 до 28. В табл. 1 приводятся усреднённые значения 100 экспериментов.

Экспериментальные данные показывают, что при выполнении метода Get на инициализацию данных и их очистку ($P_2 - P_1$, $P_7 - P_6$) приходится примерно 35–45 %, а на примитивы синхронизации ($P_3 - P_2$, $P_6 - P_5$) — 5–10 % от общего числа тактов процессора при использовании блокировки $N(\mathrm{mutex} + \mathrm{signal})$. В [10] показано, что бóльшая доля тактов процессора при выполнении метода Get до применения блокировок приходилась на примитивы синхронизации.

Заключение. В данной работе предложены алгоритмы, позволяющие повысить эффективность исполнения параллельных программ на высокопроизводительных вычислительных системах, в частности при решении задач моделирования физических процессов. Повышение эффективности исполнения программ выполнено за счёт оптимизации обмена информацией между параллельными процессами и оптимизации доступа к данным общей памяти.

Разработано программное средство оценки времени реализации коллективных операций на многопроцессорных SMP-узлах в стандарте MPI. Выполнен теоретический анализ вычислительной сложности алгоритмов реализации коллективной операции All-to-all, проведено экспериментальное исследование зависимости времени передачи сообщений от

размера передаваемого сообщения, а также определены рекомендации по выбору представленных алгоритмов (алгоритм Дж. Брука, блочный алгоритм и алгоритм попарных обменов). Показано, что блочный алгоритм в сравнении с рассматриваемыми алгоритмами более эффективен для передачи сообщений среднего размера (от 1 до 64 Кбайт), а алгоритм попарных обменов даёт лучшие результаты при передаче сообщений от 64 Кбайт.

Разработаны алгоритмы, ориентированные на оптимизацию выполнения коллективных операций на многопроцессорных NUMA-узлах в стандарте MPI. В рамках разработанной модели найдены оптимальные параметры очередей для конвейерной передачи фрагментов сообщения из корневого процесса остальным. Полученные результаты хорошо согласуются с практикой — рекомендуемые значения обеспечивают близкие к оптимальным значения размера буфера очереди и её длины. Представленный в алгоритме метод размещения очередей процессов в памяти локальных NUMA-узлов обеспечивает на 20–60 % меньшее время выполнения операции по сравнению с MVAPICH и Open MPI coll/sm.

Предложенные алгоритмы блокировки чтения-записи повышают эффективность доступа к базе данных KVDb, размещённой в общей памяти вычислительного узла. В режиме конкурентного доступа алгоритм $N(\text{mutex} + \text{signal})$ позволяет снизить накладные расходы на захват блокировки на запись примерно на 20 % относительно алгоритма, используемого в библиотеке Open PMIx. В то же время N-MCS обеспечивает снижение накладных расходов в 2 раза для NUMA-узлов на базе процессоров Intel (28 ядер). При этом интенсивность захвата блокировок на чтение увеличивается на 30–50 %. Показано, что дальнейшая оптимизация должна быть направлена на примитивы инициализации данных и их очистку.

Финансирование. Работа выполнена в рамках государственного задания ИФП СО РАН (№ 0242-2021-0011) и при поддержке Российского фонда фундаментальных исследований (грант № 20-07-00039).

СПИСОК ЛИТЕРАТУРЫ

1. **Dvurechenskii A. V., Yakimov A. I.** Silicon Based Nanoheterostructures with Quantum Dots // Advances in Semiconductor Nanostructures. Eds. A. V. Latyshev, A. V. Dvurechenskii, A. L. Aseev. Amsterdam — Boston — Heidelberg — London — New York: Elsevier Inc, 2017. P. 59–100.
2. **Supercomputer Fugaku.** URL: https://www.fujitsu.com/global/about/innovation/fugaku/ (дата обращения: 29.06.2021).
3. **Graham R. L., Shipman G.** MPI support for multi-core architectures: Optimized shared memory collectives // Proc. of the 15th Europ. PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface. Dublin, Ireland, Sept. 7–10, 2008. P. 130–140.
4. **Jain S., Kaleem R., Balmana M. et al.** Framework for scalable intra-node collective operations using shared memory // Proc. of the Intern. Conference for High Performance Computing, Networking, Storage, and Analysis (SC-2018). Dallas, USA, Nov. 11–16, 2018. P. 374–385.
5. **Wu M., Kendall R., Aluru S.** Exploring collective communications on a cluster of SMPs // Proc. of the Intern. Conference on High Performance Computing and Grid in Asia Pacific Region. Omiya Sonic City, Tokyo, Japan, July 20–22, 2004. P. 114–117.
6. **Bruck J., Ho Ch.-T., Kipnis Sh. et al.** Efficient algorithms for all-to-all communications in multiport message passing systems // IEEE Trans. Parallel Distrib. Syst. 1997. **8**, N 11. P. 1143–1156.
7. **Thakur R., Rabenseifner R., William G.** Optimization of collective communication operations in MPICH // Int. Journ. High Performance Comput. Appl. 2005. **19**, N 1. P. 49–66.

8. **Balaji P., Buntinas D., Goodell D. et al.** MPI on millions of cores // Parallel Processing Lett. 2011. **21**, Iss. 1. P. 45–60.
9. **Li S., Hoefler T., Snir M.** NUMA-aware shared memory collective communication for MPI // Proc. of the 22nd Int. Symposium on High-Performance Parallel and Distributed Computing. New York, USA, June 17–21, 2013. P. 85–96.
10. **Polyakov A., Karasev B., Hursey J.** A performance analysis and optimization of PMIx-based HPC software stacks // Proc. of the 26th Europ. MPI Users' Group Meeting. Zurich, Switzerland, Sept. 10–13, 2019. N 9. P. 1–10.
11. **PMIx Consortium.** 2017–2018. PMIx-based Reference RunTime Environment (PRRTE). URL: https://github.com/pmix/prrte (дата обращения: 25.06.2021).
12. **Castain R. H., Solt D., Hursey J., Bouteiller A.** PMIx: Process management for exascale environments // Proc. of the 24th Europ. MPI Users' Group Meeting. New York, USA, Sept. 25–28, 2017. N 14. P. 1–10.
13. **IEEE Standard** for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications. Iss. 7. IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008) (Jan. 2018). P. 1–3951. URL: https://doi.org/10.1109/IEEESTD.2018.8277153 (дата обращения: 25.06.2021).
14. **Микробенчмарк.** URL: https://github.com/artpol84/poc/tree/master/arch/concurrency/locking/shmem_locking (дата обращения: 25.06.2021).

Поступила в редакцию 04.08.2021
После доработки 21.08.2021
Принята к публикации 23.08.2021