УДК 004.021
DOI: 10.55648/1998-6920-2022-16-2-3-11

Иерархический алгоритм барьерной синхронизации для многопроцессорных систем с общей памятью

М. Г. Курносов[1]

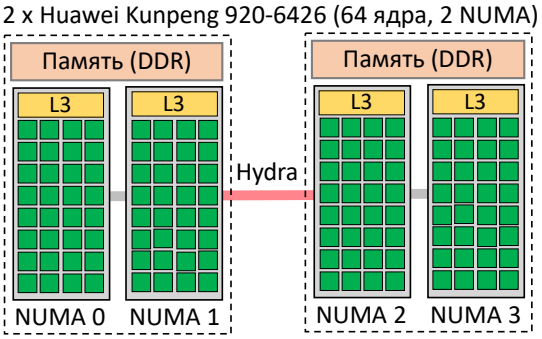
Разработан иерархический алгоритм барьерной синхронизации стандарта MPI, который формирует группы процессов, разделяющие общие ресурсы на уровнях иерархии памяти: кеш-память L2/L3, NUMA-узел, процессор. Синхронизация выполняется параллельно в группах на каждом уровне иерархии, что позволяет локализовать межпроцессные взаимодействия. Реализация выполнена на базе библиотеки Open MPI. Эксперименты на сервере с двумя процессорами Huawei Kunpeng (128 ядер, 4 NUMA-узла) показали, что предложенный алгоритм с группировкой процессов по NUMA-узлам обеспечивает минимальное время выполнения по сравнению с известными методами и устойчив к изменению способа распределения процессов по процессорным ядрам.

Ключевые слова: барьер, синхронизация, MPI.

1. Введение

В работе рассматриваются методы реализации операции барьерной синхронизации стандарта MPI в пределах одного многопроцессорного сервера (вычислительного узла). Такие алгоритмы применяются, если MPI-программа запущена на одном узле или в составе иерархических (многоуровневых) алгоритмов коллективных операций стандарта MPI [1]. Бо́льшая часть современных многопроцессорных серверов имеет архитектуру с неоднородным доступом к памяти. В таких системах множество процессорных ядер разбито на NUMA-узлы, каждый из которых имеет свой интегрированный контроллер доступа к локальной оперативной памяти. Доступ к памяти удаленных NUMA-узлов осуществляется через межпроцессорный канал связи (например, Intel UPI, AMD InfinityFabric, HiSilicon Hydra) и требует больше времени. Количество NUMA-узлов на многопроцессорном сервере зависит от микроархитектуры процессора, числа процессоров и активированного в BIOS режима разделения системы на NUMA-узлы (Intel Sub-NUMA Clustering, AMD EPYC Channel-Interleaving, Huawei Channel Interleaving). На рис. 1 показана система с четырьмя NUMA-узлами – два процессора Huawei Kunpeng 920-6426 (64 ядра с архитектурой ARMv8, 2 NUMA-узла на процессор). Обращение к страницам памяти удаленного процессора требует перехода по шине HiSilicon Hydra.

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



i / j	0	1	2	3
0	10	16	32	33
1	16	10	25	32
2	32	25	10	16
3	33	32	16	10

Рис. 1. NUMA-архитектура сервера на базе двух процессоров Huawei Kunpeng 920-6426 (справа показана матрица расстояний между NUMA-узлами: `numactl -H`)

Операция барьерной синхронизации `MPI_Barrier` блокирует выполнение процесса MPI-программы до тех пор, пока каждый процесс коммуникатора не осуществит её вызов. Процесс выходит из барьерной синхронизации только после того, как все процессы коммуникатора начали её выполнение. Известные алгоритмы барьерной синхронизации для многопроцессорных систем с общей памятью: *глобальный счетчик* (central counter) [1–2], *плоское дерево* [1], плоское дерево с фазами *gather/release* [1], *объединяющее дерево* (combining tree) [3], *MCS-барьер* [4], *турнирный* алгоритм (tournament) [5], *рассеивающий* (dissemination) [6] основаны на использовании счетчиков и флагов в сегменте разделяемой памяти, при помощи которых процессы уведомляют друг друга о достижении барьера или его определенного этапа. Время выполнения барьера зависит от времени доступа к разделяемым переменным, а это, в свою очередь, зависит и от распределения процессов по ядрам системы.

В данной работе предложен алгоритм, который группирует процессы, совместно использующие ресурсы на каждом уровне иерархии памяти (разделяют кеш-память L2, L3, контроллер доступа к памяти NUMA-узла, принадлежат одному процессору). Это позволяет локализовать синхронизацию в созданных группах и тем самым сократить время операции `MPI_Barrier`.

2. Иерархический алгоритм барьерной синхронизации

2.1. Построение иерархии групп процессов

При создании нового MPI-коммуникатора процессы формируют совместно используемый сегмент разделяемой памяти (вызовом `mmap()` операционной системы GNU/Linux), который затем используется для взаимодействия процессов в ходе выполнения алгоритмов. Часть сегмента отведена под блок – непрерывную область в виртуальном адресном пространстве с глобальными счетчиками и флагами, а также под индивидуальные блоки процессов. Длины блоков всех процессов одинаковы и кратны размеру страницы физической памяти. Зная свой номер *rank* и размер блока, каждый процесс вычисляет начало своего участка в разделяемом сегменте. В блоке каждого процесса хранятся его локальные счетчики и флаги. Чтобы избежать *ложного разделения данных* (false sharing), под каждый счетчик/флаг отводится область, длина которой равна размеру строки кеш-памяти целевой архитектуры (для Intel 64 – 64 байта, для Kunpeng 920-6426 ARMv8 – 128 байт), с соответствующим выравниванием начального адреса. Для сокращения времени доступа к данным своего блока каждый процесс пытается выделить физические страницы памяти с локального NUMA-узла – каждый процесс самостоятельно инициализирует счетчики в своем блоке. В соответствии с политикой *first touch policy* ядра Linux это приводит к выделению страницы физической памяти с NUMA-узла ядра, на котором выполняется процесс, осуществивший первую запись в страницу памяти. По умолчанию блок

с глобальными счетчиками инициализирует процесс 0 (глобальные счетчики и флаги будут размещены в памяти его NUMA-узла).

После создания сегмента памяти каждый процесс осуществляет анализ распределения процессов по ядрам и построение групп процессов, соответствующих иерархии памяти вычислительного узла.

Шаг 1. Анализ иерархии памяти. MPI-процесс определяет число ядер, разделяющих кеш-память уровней L2, L3, число NUMA-узлов и процессоров на вычислительном узле. Вычисляется количество L и формируется список активных уровней иерархии памяти – уровней, которые совместно используются двумя и более ядрами: кеш-память L2, L3, NUMA-узел, процессор (socket, package). В текущей реализации информация получается из библиотеки HWLOC. На рис. 2 приведен пример двухпроцессорного сервера, в котором кеш-память L1 и L2 у каждого ядра своя, поэтому список активных уровней составляют NUMA-узлы (32 ядра) и процессоры (PACKAGE, 2 NUMA-узла, 64 ядра). Уровень кеш-памяти L3 не учтён, так как те же ядра разделяют один NUMA-узел.

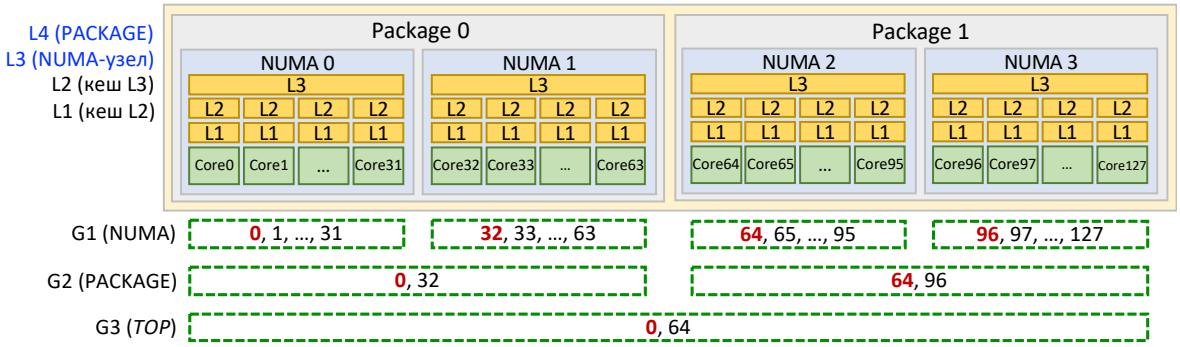


Рис. 2. Двухпроцессорный сервер: 2 NUMA-узла на процессор, 32 ядра на NUMA-узел; 128 MPI-процессов с привязкой к процессорным ядрам (`--bind-to core`) и последовательным распределением (`--map-by core`), два активных уровня иерархии ($L = 2$), три уровня групп процессов

Шаг 2. Определение таблицы размещения процессов в пределах узла. Процесс определяет множество процессорных ядер, к которым он привязан (на каких ядрах планировщик операционной системы GNU/Linux может его выполнять). По умолчанию библиотека Open MPI привязывает процесс к процессорному ядру (`--bind-to core`), однако при запуске программы пользователь может изменить способ привязки – разрешить операционной системе динамически переносить процесс в пределах одного процессора (`--bind-to package`), в пределах NUMA-узла (`--bind-to numa`) или отключить привязку (`--bind-to none`) и разрешить выполнение процесса на любом ядре. При отключенной привязке процессов к процессорным ядрам алгоритм завершает свою работу, так как невозможно однозначно определить, какие уровни иерархии памяти процессы используют совместно. Информация о привязке процесса извлекается через интерфейс PMIx библиотеки Open MPI (`PMIX_LOCALITY_STRING`). Через обращение к интерфейсу PMIx определяется размещение всех P процессов коммуникатора в пределах узла. Для активных уровней иерархии памяти определяется номер функционального модуля $procloc[l][i]$, который используется каждым процессом i на уровне l. Для примера, на рис. 2 PMIx-строка с размещением процесса 32 имеет вид “`SK0:L31:L232:L132:CR32:HT32:NM1`” (номер процессора : номер кеш-памяти L3 : … : номер NUMA-узла). Процесс входит в две группы, таким образом, для процесса 32 $procloc[0][32] = 1$, $procloc[1][32] = 0$. Вычислительная сложность шага – $O(P \cdot L)$.

Шаг 3. Формирование групп процессов. Каждый процесс для всех активных уровней иерархии памяти строит списки процессов, с которыми он разделяет один функциональный модуль (кеш-память L2, NUMA-узел, процессор). Построение начинается с низшего активного уровня. В каждой группе выбирается лидер – процесс с наименьшим номером. Лидеры групп

становятся членами групп следующего активного уровня. Цикл повторяется для всех активных уровней. Вычислительная сложность шага – $O(L(C+P))$, где L – число активных уровней в иерархии, C – число ядер на вычислительном узле, P – число процессов в коммуникаторе. На рис. 2 показана иерархия из трех групп $G1$, $G2$, $G3$ для двух активных уровней (NUMA и PACKAGE). Группа $G3$ включает два процесса – лидеры процессоров 0 и 1. Ниже приведены примеры групп процессов:

- процесс 0: $G1(0, 1, \ldots, 31)$, $G2(0, 32)$, $G3(0, 64)$;
- процесс 1: $G1(0, 1, \ldots, 31)$;
- процесс 32: $G1(32, 33, \ldots, 63)$, $G2(0, 32)$.

На рис. 3 показан пример иерархии групп для случая 14 процессов с циклическим распределением по 4 процесса на NUMA-узел.

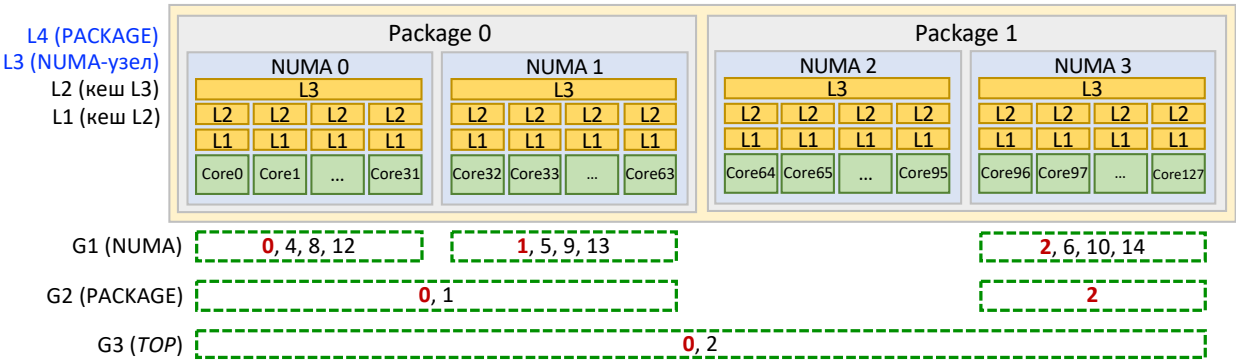


Рис. 3. Двухпроцессорный сервер: 2 NUMA-узла на процессор, 32 ядра на NUMA-узел; 14 процессов с привязкой к процессорным ядрам (`--bind-to core`) и циклическим распределением процессов по NUMA-узлам (`--map-by numa`), два активных уровня иерархии ($L = 2$), три уровня групп процессов

2.2. Алгоритм барьерной синхронизации

В момент вызова операции `MPI_Barrier` каждому процессу известен свой исходный номер *rank* в коммуникаторе, а также номер *group*[*l*].*group_rank* в каждой его группе уровня *l*. В сегменте разделяемой памяти процесс хранит счетчики *group*[*l*].*state*[*group_rank*] числа обращений к барьеру на уровне иерархии $l \in \{0, 1, \ldots, L\}$, а также имеет доступ к общему флагу *sense* уведомления о достижении барьера каждым процессом. При входе в барьер каждый процесс *rank* меняет значение своего флага *sense_local*[*rank*] на противоположное. Далее выполняется синхронизация процессов в группах уровня 0, 1, …, L. Синхронизация в группе выполняется алгоритмом плоского дерева – лидер ожидает, пока члены группы не переведут счетчики состояния в необходимое значение. После этого лидер переходит к синхронизации на уровне выше (`BarrierGroupLevel(level + 1)`), а остальные процессы переходят в ожидание, пока значение локального флага *sense_local*[*rank*] не станет равным значению глобального флага *sense*. Инвертирование счетчиков *sense_local*[*rank*] и *sense* обеспечивает корректную работу многократных вызовов барьера (схема *sense reversing* [2, 4]). Начальные состояния: *sense* = 1, *sense_local*[*rank*] = 1. Ниже приведен псевдокод алгоритма.

```
algorithm MPI_Barrier()
  sense_local[rank] = sense_local[rank] xor 1
  BarrierGroupLevel(0)
end algorithm

algorithm BarrierGroupLevel(level)
  grank = group[level].group_rank           // Индекс процесса в группе
  gleader = group[level].group_rank_leader  // Индекс лидера в группе
  group[level].state[grank]++               // Уведомление лидера о входе
```

```
  if grank = gleader then
    while true do       // Лидер ожидает уведомления от процессов группы
      narrived = 0
      for child = 0 to group[level].size - 1 do
        if group[level].state[child] >= group[level].state[gleader] then
          narrived++
      end for
      if narrived = group[level].size then break
    end while
    if level + 1 < ngroups then      // Лидер переходит на следующий уровень
      BarrierGroupLevel(level + 1)
    else if rank = 0 then
      sense = sense_local[rank]      // Лидер уведомляет о выходе из барьера
    end if
  else
    while sense != sense_local[rank] do  // Ожидание уведомления от лидера 0
  end if
end algorithm
```

Время выполнения алгоритма $O(L \cdot P)$ линейно зависит от числа процессов и активных уровней иерархии памяти. В программном компоненте реализована возможность отключения учета отдельных уровней иерархии памяти при формировании групп процессов. Например, алгоритм *shm-topo-NUMA-SOCKET* для системы на рис. 2 учитывает все активные уровни иерархии памяти. Алгоритм *shm-topo-NUMA* получен путем отключения учета уровня процессора, он группирует процессы только по NUMA-узлам (32 ядра в группе). Алгоритм *shm-topo-SOCKET* игнорирует NUMA-узлы и группирует 64 ядра каждого процессора в отдельную группу.

3. Результаты экспериментов

Эксперименты выполнялись на двухпроцессорном сервере: два процессора Huawei Kunpeng 920-6426 ARMv8 (64 ядра, 2 NUMA-узла на процессор, размер строки кеш-памяти L1/L2 – 64 байта, L3 – 128 байт), 255 Гб оперативной памяти, по 64 Гб на каждый NUMA-узел. Процессоры соединены каналами HiSilicon Hydra. На сервере установлена операционная система CentOS 7.6.1810 (ядро linux 4.14.0-115, aarch64, размер страницы физической памяти – 64 Кб), gcc 4.8.5. Матрица расстояний между NUMA-узлами показана на рис. 2.

Для реализации и тестирования алгоритмов использовалась библиотека Open MPI 5.0.0rc2 (параметры сборки: `CFLAGS="-O3 -g" --disable-debug`). В качестве теста взят пакет OSU Micro-Benchmarks 5.8. Для каждого числа процессов операция `MPI_Barrier` запускалась 1000 раз (параметры запуска теста: `osu_barrier -x1 -i1000 -f`). В каждом эксперименте тест запускался 5 раз, отбрасывались наименьшее и наибольшее значения максимальной латентности, среди оставшихся значений отыскивалось среднее время работы операции [1, 2]. При запуске теста каждый процесс привязывался к процессорному ядру (`mpiexec --bind-to core`).

Выполнялось сравнение разработанного алгоритма операции `MPI_Barrier` с алгоритмами на базе операций send/recv библиотеки Open MPI (coll/base): *base-liner* (линейный алгоритм, время выполнения $O(P)$), *base-double-ring* (двухэтапный кольцевой алгоритм, $O(P)$), *base-recursive-doubling* (алгоритм рекурсивного удваивания, время $O(\log P)$), *base-bruck* (алгоритм Брука, $O(\log P)$), *base-tree* (двухэтапный алгоритм рекурсивного удваивания, $O(\log P)$). Также сравнение выполнялось с алгоритмами барьерной синхронизации, основанными на взаимодействии процессов через разделяемую память (shm-barrier) [2]: *shm-sense-reversing* (алгоритм с глобальным счетчиком на базе атомарных операций, время $O(P)$), *shm-sense-reversing-counter* (алгоритм плоского дерева, $O(P)$), *shm-sense-reversing-gr* (двухэтапный алгоритм плоского дерева, $O(P)$) [1], *shm-combining-tree* (алгоритм бинарного объединяющего дерева,

O(log*P*)), *shm-mcs* (MCS-барьер), *shm-tournament* (турнирный алгоритм, *O*(log*P*)), *shm-dissemination* (рассеивающий алгоритм, *O*(log*P*)).

На рис. 4 показано время выполнения алгоритмов барьерной синхронизации на одном процессоре – 64 процесса в пределах одного NUMA-узла 0. Наименьшим временем выполнения характеризуется алгоритм бинарного объединяющего дерева (*shm-combining-tree*), который на 40 % быстрее турнирного алгоритма *shm-tournament* и на 55 % быстрее MCS-барьера. Производительность многоуровневого алгоритма на одном процессоре не исследовалась, так как в этом случае ядра разделяют единственный уровень иерархии – контроллер доступа к памяти NUMA-узла. Алгоритмы барьерной синхронизации на основе явного использования операций send/recv (*base*) значительно уступают по производительности алгоритмам, использующим для взаимодействий сегмент разделяемой памяти (*shm*). Алгоритм рекурсивного удваивания *base-tree* почти в 2.8 раз медленнее алгоритма объединяющего дерева. Исключение составляет алгоритм с *глобальным счетчиком* на базе атомарных операций (*shm-sense-reversing*), который демонстрирует низкую масштабируемость.

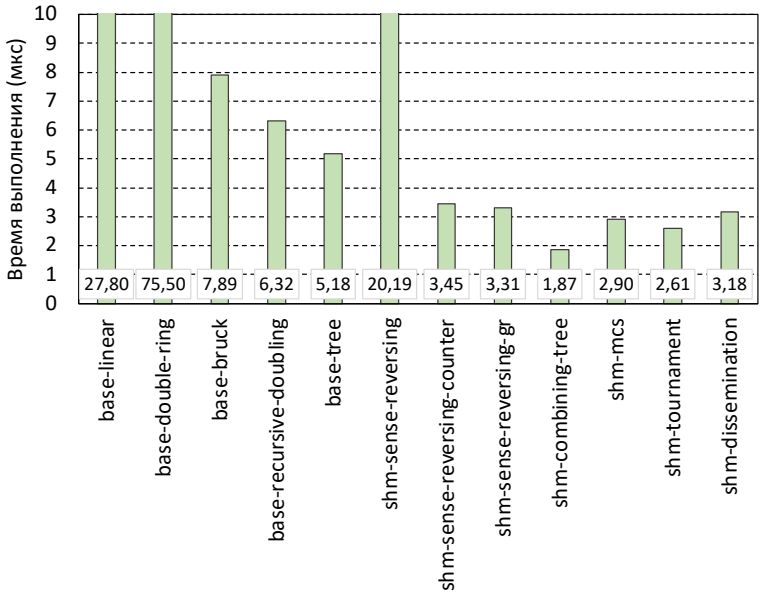


Рис. 4. Время выполнения операции `MPI_Barrier` на 64 ядрах одного процессора

На рис. 5 приведены результаты для двух процессоров (128 ядер, четыре NUMA-узла). Наилучшие результаты демонстрирует иерархический алгоритм с группировкой процессов по NUMA-узлам (*shm-topo-NUMA*), незначительно ему уступает алгоритм объединяющего дерева. Иерархический алгоритм с группировкой по NUMA-узлам и процессорам (*shm-topo-NUMA-SOCKET*) опережает на 14 % алгоритм с группировкой только по сокетам (*shm-topo-SOCKET*). Алгоритм рекурсивного удваивания на базе операций send/recv *base-tree* в 2.5 раз медленнее иерархического алгоритма *shm-topo-NUMA*.

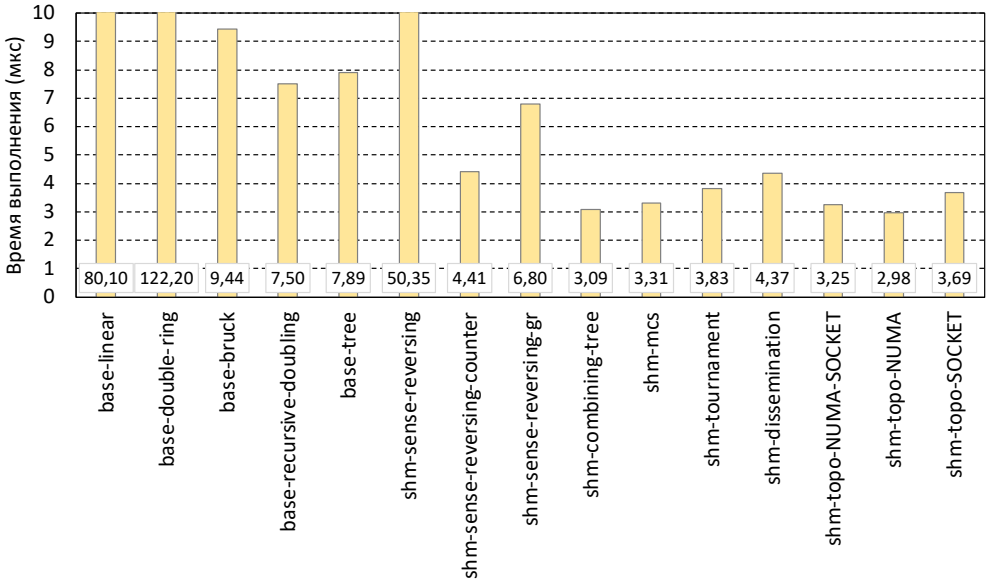


Рис. 5. Время выполнения операции `MPI_Barrier` на 128 ядрах двух процессоров (четыре NUMA-узла по 32 ядра)

До момента запуска MPI-приложения неизвестно, процессы с какими номерами на каких ядрах будут выполняться. На практике пользователи указывают одну из трех схем распределения процессов по ядрам:

- `map-by core`: процессы программы назначаются последовательно с шагом через одно ядро (процесс 0 на ядро 0, процесс 1 на ядро 1, …, процесс 127 на ядро 127);
- `map-by NUMA`: процессы циклически распределяются с шагом через NUMA-узел (для рис. 1 – процесс 0 на ядро 0, процесс 1 на ядро 32, процесс 2 на ядро 64, процесс 3 на ядро 96, …, процесс 127 на ядро 127);
- `map-by SOCKET`: процессы циклически распределяются по ядрам двух процессоров (шаг 64 ядра, на рис. 1 – процесс 0 на ядро 0, процесс 1 на ядро 64, процесс 2 на ядро 1, процесс 3 на ядро 65, …, процесс 127 на ядро 127).

Способ распределения процессов может значительно влиять на время выполнения одного и того же алгоритма, так как в процессе синхронизации процессы могут читать и записывать флаги и счетчики, находящиеся в памяти удаленных NUMA-узлов, в L2/L3 кеш-памяти других ядер или процессора. В то же время иерархический алгоритм гарантирует локализацию взаимодействий через локальную (общую) для группы процессов L2/L3 кеш-память, контроллер памяти NUMA-узла. На рис. 6 показано время выполнения алгоритмов для трех схем назначения процессов на ядра. Иерархический алгоритм демонстрирует хорошую устойчивость к изменению способа распределения процессов – коэффициент вариации времени выполнения алгоритма не превышает 7 % для трех рассмотренных схем, при этом алгоритм обеспечивает минимум времени реализации барьерной синхронизации на данной системе.

Алгоритм MPI_Barrier	Времы выполнения для заданной схемы распределения процессов (мкс)			Коэффициент вариации времени выполнения (%)
	map-by core	map-by NUMA	map-by SOCKET	
shm-sense-reversing	50,35	45,55	52,8	7,4
shm-sense-reversing-counter	4,41	5,1	4,64	7,4
shm-sense-reversing-gr	6,80	7,18	6,98	2,7
shm-combining-tree	3,09	3,92	3,63	11,9
shm-mcs	3,31	5,47	4,9	24,5
shm-tournament	3,83	3,59	7,38	43,0
shm-dissemination	4,37	3,45	3,85	11,9
shm-topo-NUMA-SOCKET	3,25	2,86	3,16	6,6
shm-topo-NUMA	**2,98**	**2,68**	**3,03**	6,5
shm-topo-SOCKET	3,69	3,34	3,16	7,9

Рис. 6. Время выполнения алгоритмов операции `MPI_Barrier` для трех схем распределения процессов по 128 ядрам двух процессоров (4 NUMA-узла по 32 ядра)

4. Заключение

Разработанный иерархический алгоритм барьерной синхронизации обеспечивает многоуровневую синхронизацию между процессами, объединёнными в группы в соответствии с разделяемыми ими уровнями иерархии памяти (кеш-память L2/L3, NUMA-узел, процессор). Алгоритм реализован на базе библиотеки Open MPI. Эксперименты на сервере с двумя процессорами Huawei Kunpeng (128 ядер, 4 NUMA-узла) показали, что разработанный алгоритм с группировкой процессов по NUMA-узлам обеспечивает минимальное время выполнения по сравнению с реализованными в библиотеке Open MPI и устойчив к изменению способа распределения процессов по процессорным ядрам.

Литература

1. *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.
2. *Курносов М. Г., Токмашева Е. И.* Оптимизация барьерной синхронизации на асимметричных NUMA-подсистемах процессорных ядер // Вестник СибГУТИ. 2021. № 1. С. 36–49.
3. *Yew P. C., Tzeng N. F., Lawrie D. H.* Distributing Hot Spot Addressing in Large Scale Multiprocessors // IEEE Transactions on Computers. 1987. V. C-36, Is. 4. P. 388–395.
4. *Mellor-Crummey J. M., Scott M. L.* Algorithms for Scalable Synchronization on Shared-memory Multiprocessors // ACM Transactions on Computer Systems. 1991. № 9 (1). P. 21–65.
5. *Hengsen D., Finkel R., Manber U.* Two Algorithms for Barrier Synchronization // Int. Journal of Parallel Programming. 1988. V. 17, Is. 1. P. 1–17.
6. *Brooks E.* The butterfly barrier // Journal of Parallel Programming. 1986. V. 15, Is. 4. P. 295–307.

Статья поступила в редакцию 21.03.2022.

Курносов Михаил Георгиевич

д.т.н., профессор, профессор кафедры вычислительных систем Сибирского государственного университета телекоммуникаций и информатики (630102, Новосибирск, ул. Кирова, 86), тел. (383) 269-83-82, e-mail: mkurnosov@sibguti.ru; старший научный сотрудник Института физики полупроводников им. А. В. Ржанова СО РАН (630090, Новосибирск, просп. Академика Лаврентьева, 13), тел. (383) 330-56-26, e-mail: mkurnosov@isp.nsc.ru.

Barrier synchronization hierarchical algorithm for multicore shared-memory systems

Mikhail G. Kurnosov

Doctor of technical sciences, Professor, Siberian State University of Telecommunications and Information Science (SibSUTIS, Novosibirsk, Russia), mkurnosov@sibguti.ru.

A hierarchical MPI barrier synchronization algorithm creating groups of processes that share common resources at the memory hierarchy levels (L2/L3 caches, NUMA node, socket) is proposed. Synchronization is performed in groups at each level of the hierarchy. Experiments on a dual-socket server with two Huawei Kunpeng processors (128 cores, 4 NUMA nodes) showed that the proposed algorithm with NUMA nodes process grouping provides the minimum execution time compared to known methods and is resistant to different schemes of process placement.

Keywords: barrier, shared memory, MPI, NUMA.

References

1. 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.
2. Kurnosov M.G., Tokmasheva E.I. Optimizacija bar'ernoj sinhronizacii na asimmetrichnyh NUMA-podsistemah processornyh jader [Barrier Optimization on Asymmetrical NUMA Subsystems]. *Vestnik SibGUTI*, 2021, no. 1, pp. 36-49.
3. Yew P.C., Tzeng N.F., Lawrie D.H. Distributing Hot Spot Addressing in Large Scale Multiprocessors. *IEEE Transactions on Computers*, 1987, vol. C-36, iss. 4, pp. 388-395.
4. Mellor-Crummey J.M., Scott M.L. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Transactions on Computer Systems*, 1991, vol. 9(1), pp. 21-65.
5. Hengsen D., Finkel R., Manber U. Two Algorithms for Barrier Synchronization. *Int. Journal of Parallel Programming*, 1988, vol. 17, iss. 1, pp. 1-17.
6. Brooks E. The butterfly barrier. *Journal of Parallel Programming*, 1986, vol. 15, iss. 4, pp. 295-307.