УДК 004.272.26

Эвристические алгоритмы оптимизации информационных обменов в параллельных PGAS-программах

И.И. Кулагин, А.А. Пазников, М.Г. Курносов[1]

В работе предложены эвристические алгоритмы оптимизации информационных обменов в параллельных PGAS-программах, обеспечивающие сокращение времени их выполнения. Последнее достигается с помощью учёта иерархической структуры вычислительных систем при выполнении операции редукции и опережающего копирования удалённых массивов на узлы вычислительной системы. Созданные алгоритмы программно реализованы для PGAS-языков Cray Chapel и IBM X10; проведено их экспериментальное исследование на кластерных вычислительных системах.

Ключевые слова: PGAS, параллельное программирование, компиляторная оптимизация, редукция, скалярная замена, Cray Chapel, IBM X10.

1. Введение

Современные распределённые вычислительные системы (ВС) – это композиция множества вычислительных узлов (ВУ) и сети связи [1]. Основным подходом к разработке параллельных программ на таких системах является использование модели передачи сообщений, реализуемой библиотеками стандарта MPI (MPICH2, Open MPI, MVAPICH2, Intel MPI).

Современные ВС являются мультиархитектурными. Например, вычислительный узел «Tianhe-2» (I место списка TOP500) укомплектован двумя процессорами Intel Xeon E5-2692 и тремя многоядерными сопроцессорами Intel Xeon Phi 31S1P. По этой причине для написания эффективных параллельных программ, использующих все архитектурные возможности таких систем, требуется применение целого стека технологий: MPI, OpenMP/Intel TBB/ Intel Cilk Plus, Nvidia CUDA/OpenCL/OpenACC/DVM, инструкций SSE/AVX/AltiVec и др.

Необходимость упрощения процесса разработки параллельных программ вызвала активное развитие высокоуровневых средств параллельного программирования, в частности, языков, реализующих модель разделённого глобального адресного пространства (Partitioned Global Address Space – PGAS), а также инструментария для отладки и оптимизации параллельных программ (TotalView, DDT).

К классу PGAS относятся такие современные языки, как Cray Chapel, IBM X10, Unified Parallel C и др. В отличие от стандарта передачи сообщений MPI, программы в модели PGAS не содержат явных обращений к коммуникационным функциям; вместо этого они оперируют распределёнными структурами данных и языковыми конструкциями управления потоками выполнения (tasks, activities) и их синхронизации. Все информационные обмены планируются компилятором и выполняются runtime-системой, что обеспечивает прозрачный доступ потоков к памяти удалённых ВУ.

[1] Работа выполнена при поддержке РФФИ (гранты № 12-07-00145, 13-07-00160, 12-07-00188, 12-07-00106).

Высокий уровень абстракции модели PGAS позволяет снизить трудоёмкость создания параллельных программ, но в то же время требует разработки эффективных методов оптимизирующей компиляции.

Можно выделить два наиболее часто встречающихся шаблона в параллельных PGAS-программах:

1) итерации по элементам распределённого массива;
2) выполнение заданной операции редукции над его элементами (reduce, reduction).

Существующие на сегодняшний день алгоритмы выполнения операций с распределёнными массивами [2 – 4] не учитывают особенности модели PGAS, такие как большое количество вызовов функций одностороннего доступа к переменным на удалённых ВУ, проблемы организации согласованности памяти, многопоточность и др.

Алгоритмы компиляторной оптимизации [5] в языках IBM X10 [6], UPC [7] не обеспечивают минимума накладных расходов в PGAS-программах, выполняющих циклический доступ к элементам массива, расположенным в памяти удалённых ВУ. В алгоритмах коллективных обменов [8, 9] не учитывается первоначальное распределение элементов массива между вычислительными узлами.

В данной работе предлагаются алгоритмы оптимизации информационных обменов, возникающих при выполнении операций над распределёнными массивами в PGAS-программах. Алгоритмы реализованы программно для языков Cray Chapel и IBM X10.

2. Алгоритмы оптимизации информационных обменов

2.1. Модель PGAS

Пусть $P = \{1, 2, \ldots, N\}$ – множество SMP/NUMA-узлов распределённой ВС. Каждый такой узел $i \in P$ укомплектован n процессорными ядрами (универсальными или ускорителя) и локальной памятью. Узлы связаны сетью связи (Infiniband, Gigabit Ethernet и др.).

В модели PGAS введена абстракция многоядерного ВУ – область (place, locale). За каждой областью закреплён определённый сегмент локальной памяти. В рамках каждой области выполняются динамически создаваемые потоки (activities, tasks, threads). Каждый поток имеет доступ к глобальному адресному пространству, которое формируется из сегментов локальной памяти ВУ (рис. 1). Доступ к локальному сегменту выполняется быстрее, так как доступ к удалённому сегменту требует обращения к коммуникационным функциям.

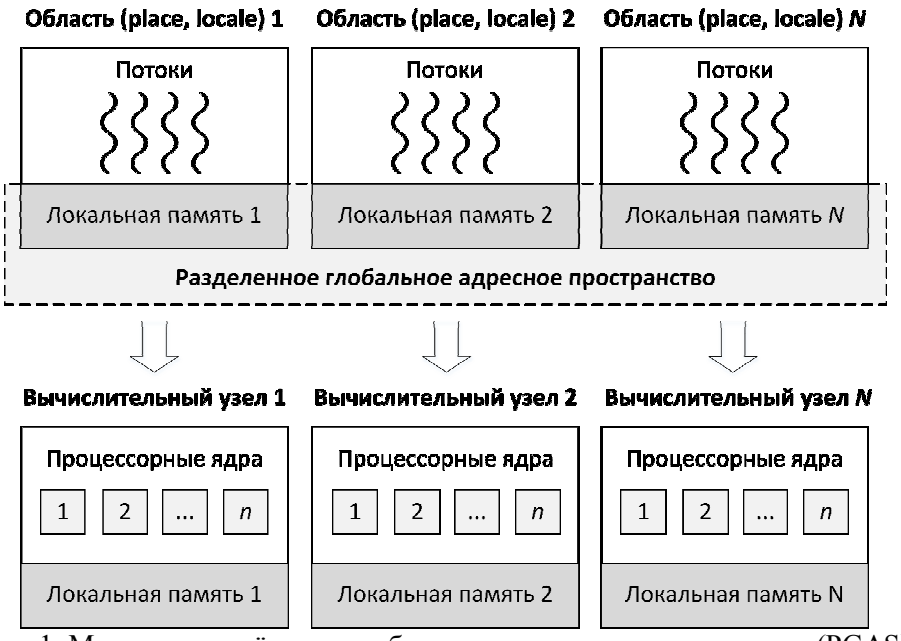


Рис. 1. Модель разделённого глобального адресного пространства (PGAS)

Основные программные конструкции, которые используются при разработке параллельных PGAS-программ:

- *begin S* – асинхронное выполнение инструкций *S* в отдельном потоке на ВУ, с которого вызвана конструкция,
- *on i S* – выполнение инструкций *S* на узле *i*,
- *on x S* – выполнений инструкций *S* на том узле, в памяти которого находится объект *x*,
- *coforall S* – выполнение каждой итерации тела *S* цикла в независимом потоке,
- *sync T* – переменная синхронизации группы *T* потоков.

2.2. Эвристический алгоритм параллельной редукции

Редукция (reduction, reduce) – это коллективная операция, которая выполняет над распределённым массивом *V*[1:*D*] заданную ассоциативную операцию ⊗. Результат *r* операции формируется в памяти потока, который инициировал эту процедуру: $r = V[1] \otimes V[2] \otimes \ldots \otimes V[D]$.

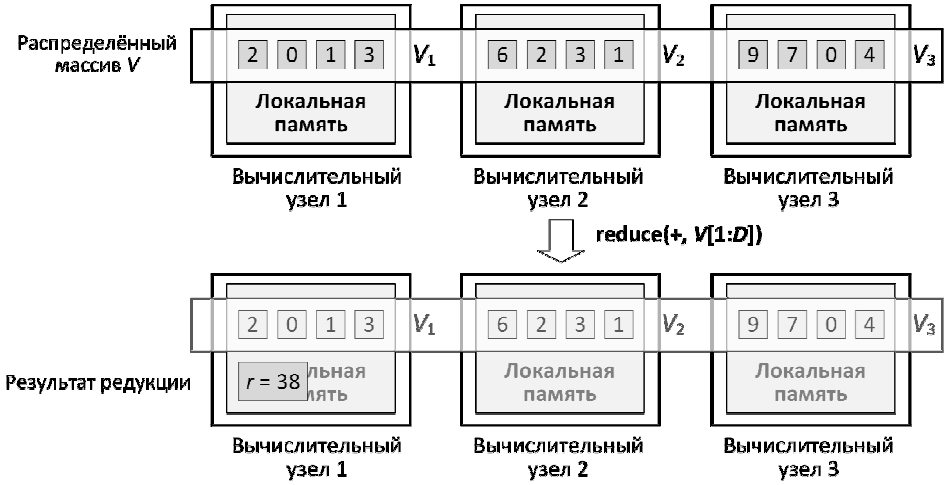


Рис. 2. Пример редукции распределённого массива: операция «+», массив *V*[1:12]

Существует ряд эффективных алгоритмов реализации параллельной редукции [2 – 4], однако они не учитывают особенности разделённого глобального адресного пространства (динамический параллелизм задач и иерархическая память), и их применение в PGAS-языках ограничено. В существующих алгоритмах коллективных обменов [8, 9] для PGAS не учитывается первоначальное распределение элементов массива между вычислительными узлами.

В данной работе предлагается алгоритм *BlockReduce* выполнения редукции в PGAS-программах (рис. 3). В листинге 1 приведён псевдокод алгоритма для языка Cray Chapel [10].

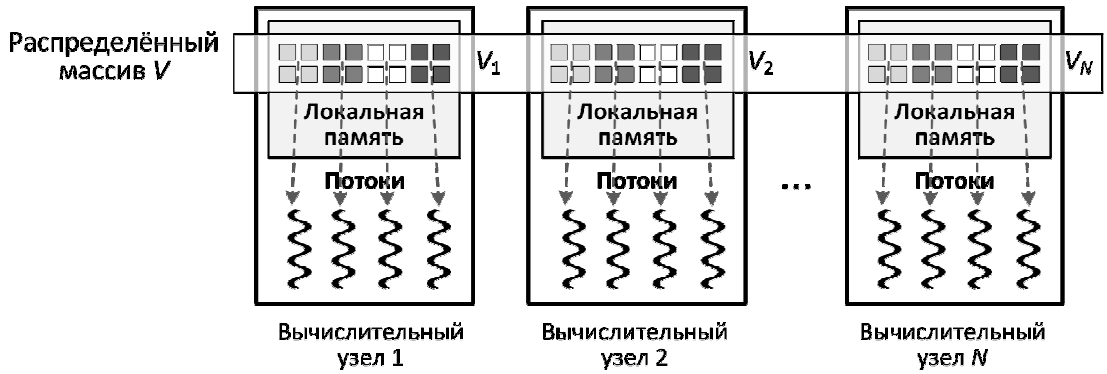


Рис. 3. Распределение элементов массива в алгоритме *BlockReduce*

Листинг 1. Алгоритм *BlockReduce*

Входные данные:	$V[1:D]$ – распределённый массив, $\otimes$ – ассоциативная операция редукции.
Выходные данные:	r – результат применения редукции для массива V.

```
1   function BLOCKREDUCE(V[1:D], ⊗)
2   coforall i in [1, 2, …, N] do  // Параллельное выполнение операции редукции
3                                  // локальных элементов массива в областях
4     on i                         // Выполнение потока на области i
5       SPLITARRAY(V_i, n)         // Разбиение массива V_i на n (число ядер) блоков V_it
6
7       coforall t in [1, 2, …, n] do   // Каждое процессорное ядро в отдельном потоке
8                                       // вычисляет редукцию r_it элементов V_it
9         for each x in V_it do
10          r[i][t] = r[i][t] ⊗ x
11        end for
12      end coforall
13
14      for each t in [1, 2…, n] do
15        r[i] = r[i] ⊗ r[i][t]         // r_i содержит результат редукции для элементов
16                                      // V_i, расположенных в памяти области i
17      end for
18    end on
19  end coforall
20
21  // Получение финального результата с помощью бинарного дерева
22  res = BLOCKREDUCEBINTREE(r[1:N])
23  return res
24  end function
```

Каждый ВУ $i \in P$ располагает информацией о наборе V_i элементов массива V, хранящихся в его локальной памяти. На первом этапе (листинг 1, строки 2 – 19) алгоритма выполняется разбиение областей V_i массива, хранящихся на узлах, на n частей (число процессорных ядер) (листинг 1, строка 5), которые затем обрабатываются параллельно. Потоки $t = 1, 2, \ldots, n$ каждого узла i выполняют редукцию над своей частью массива V_{it} (листинг 1, строки 7 – 12).

Второй этап алгоритма представлен в листинге 2. Вычислительные узлы, зная свой номер, логически организуются в бинарное дерево (рис. 4). Корнем дерева является область 1. Каждая операция $\otimes$ для пары значений $r[first]$, $r[second]$ выполняется в отдельном потоке на ВУ, в памяти которого находится значение $r[first]$ (листинг 2, строки 26 – 29). После редукции всех значений выполняется барьерная синхронизация (строка 42).

Барьерная синхронизация может быть выполнена с использованием различных алгоритмов [11], например, *Dissemination barrier*, который характеризуется временем $O(\log(N))$. В этом случае вычислительная трудоёмкость *BlockReduce* составляет $T = O(|V| / N + \log(N))$. В существующей программной реализации используется алгоритм *Centralized barrier* синхронизации за время $O(N)$ (рис. 5).

Листинг 2. Алгоритм *BlockReduce*: второй этап

Входные данные:	$r[1{:}N]$ – значения редукции, полученные на областях.
Выходные данные:	Результат применения редукции для массива V

```
1   function BLOCKREDUCEBINTREE (r[1:N])
2   newIndexes = [1, 2, …, N]
3   indCntr = N
4
5   while indCntr > 1 do
6      indexes = newIndexes          // Индексы элементов в массиве r[1:N]
7      numElems = indCntr            // Число элементов в массиве r[1:N], оставшихся
                                     // после попарного выполнения операции ⊗
8      oddFlag = 0                   // Количество элементов является нечётным
9      indCntr = 0
10
11     if numElems mod 2 ≠ 0 then
12        oddFlag = 1
13        newIndexes[numElems / 2 + 1] = indexes[numElems]
14        numElems = numElems – 1
15     end if
16     numPairs = numElems / 2       // Переменная синхронизации count – число пар
17                                   // областей, завершивших выполнение операции
18     release = false               // Переменная синхронизации release блокирует
19                                   // выполнение потока до завершения всех операций
20     i = 1
21
22     do
23        first = indexes[i]
24        second = indexes[i + 1]
25        begin
26           on r[first]             // Выполнение инструкции на той области,
27                                   // на которой находится переменная r_i
28              r[first] = r[first] ⊗ r[second]
29           end on
30           if numPairs ≠ 1 then
31              numPairs = numPairs – 1
32           else
33              release = true
34           end if
35        end begin
36        indCntr = indCntr + 1
37        newIndexes[indCntr] = first
38        i = i + 2
39     while i + 1 ≤ numElems
40     indCntr = indCntr + oddFlag
41
42     wait while release = false    // Основной поток заблокирован до тех пор,
43                                   // пока не выполнится условие release = true
43  end while
44  return r[1]
45  end function
```

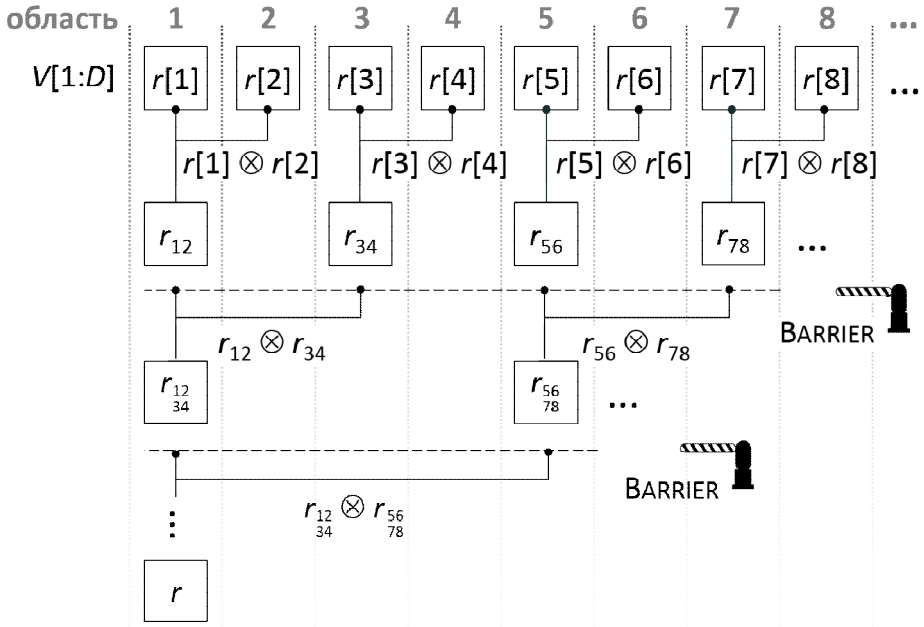


Рис. 4. Второй этап алгоритма *BlockReduce*

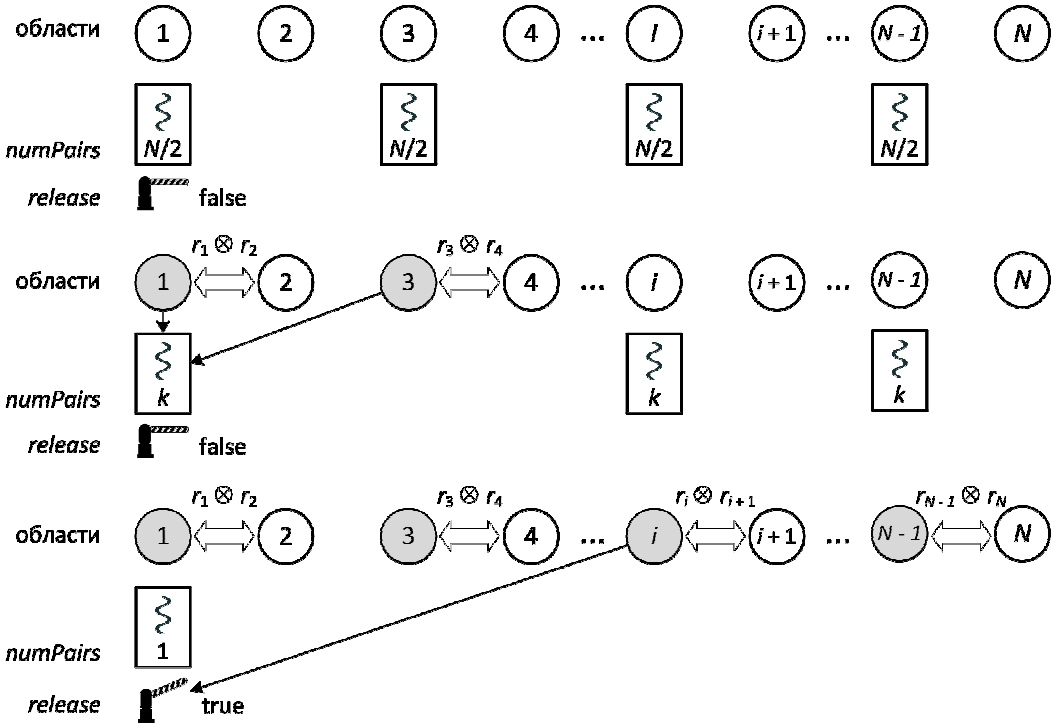


Рис. 5. Барьерная синхронизация на основе алгоритма *Centralized barrier*

2.3. Алгоритмы оптимизации доступа к массивам

Ещё одним часто встречающимся шаблоном в параллельных PGAS-программах является обращение в цикле к элементам массива, причём потоки могут обращаться к элементам, размещённым в памяти других ВУ (листинг 3*а*). В этом случае на runtime-систему языка ложится задача по доставке требуемых элементов на текущий узел.

На данном этапе развития языков семейства PGAS в компиляторах используются относительно простые эвристические алгоритмы. При доступе к одному элементу массива, расположенного в памяти удалённого ВУ, происходит избыточное копирование всего массива

в память локального ВУ (рис. 6*а*). Это создаёт большие накладные расходы, для сокращения которых используется алгоритм (*Scalar replacement* [5, 6, 12]), который предполагает использование временных скаляров вместо элементов удалённых массивов в определённых участках кода. Временные скаляры являются копией элементов удалённого массива, к которому осуществляется доступ. Таким образом, при обращении к созданным на удалённом ВУ скалярам будет выполняться копирование только используемых элементов массива, точнее, соответствующих им скаляров (рис. 6*б*). В случае циклического обращения к элементам удалённого массива, runtime-система языка копирует весь массив на каждой итерации в память локального ВУ (рис. 7*а*), что является неприемлемым. Использование алгоритма *Scalar Replacement* в этой ситуации может привести к избыточному копированию: суммарное количество переданных элементов может оказаться больше размера всего массива.

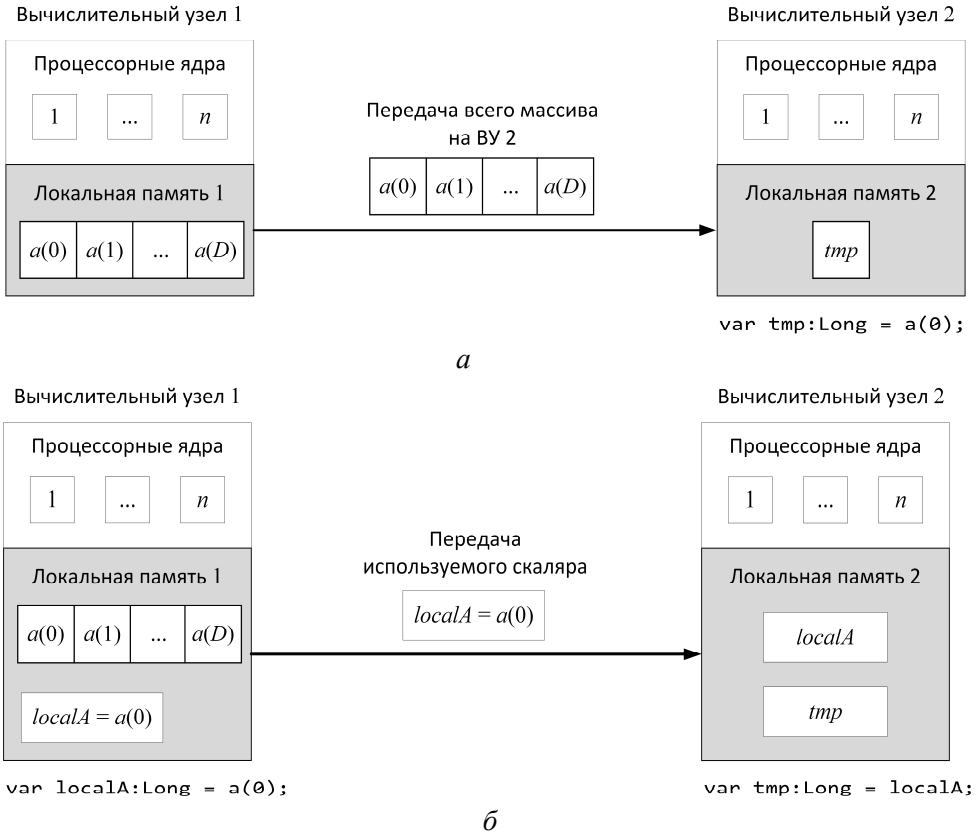
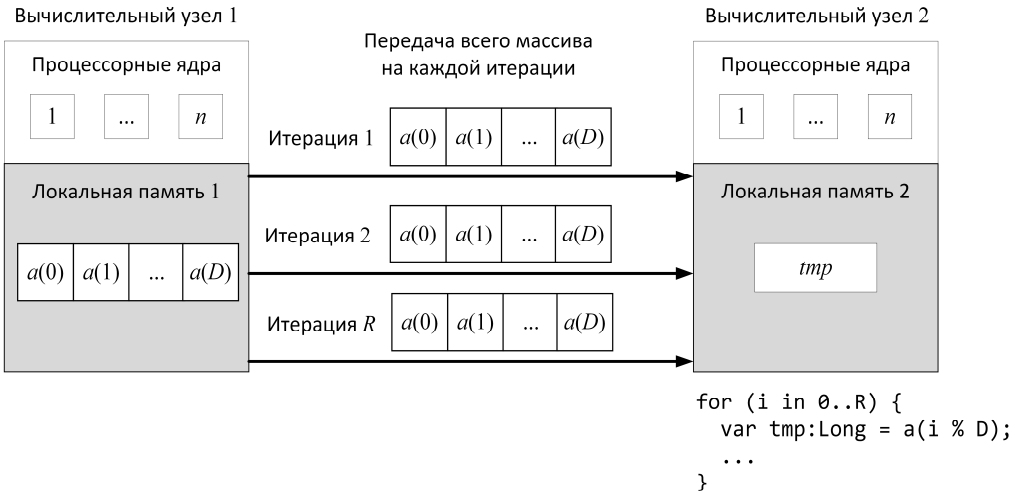


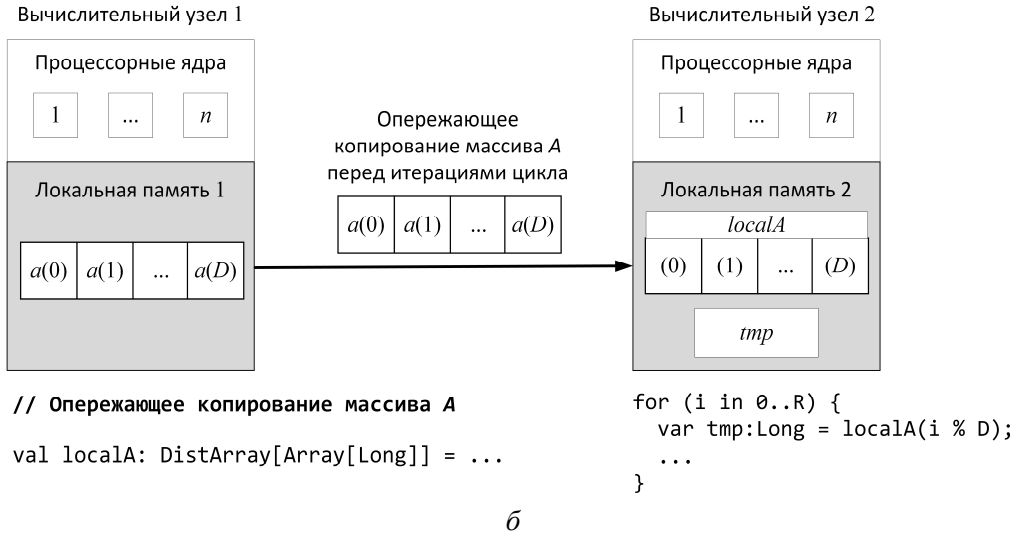
Рис. 6. Обращение потока, выполняющегося на ВУ 2, к нулевому элементу массива, расположенного в памяти ВУ 1:
а – без применения оптимизации, *б* – применён алгоритм *Scalar Replacement*

Авторами предложен алгоритм *ArrayPreload* оптимизации циклического доступа к удалённым массивам, минимизирующий время информационных обменов. *ArrayPreload* предотвращает многократное копирование массивов, размещённых в памяти других ВУ, выполняя опережающее копирование один раз перед итерациями цикла (рис. 7*б*).

В листинге 3 показан пример оптимизации передачи массива *A* для языка IBM X10. В случае неоптимизированной версии (листинг 3*а*) на каждой итерации цикла массив *A* передаётся узлу с номером *id*. В оптимизированной версии (листинг 3*б*) производится предварительное копирование массива *A* на каждый узел один раз, сохранив его в распределённом массиве *localA*. Используемая языке IBM X10 конструкция *at* семантически соответствует конструкции *on*.



а



б

Рис. 7. Циклическое обращение потока, выполняющегося на ВУ 2, к элементам массива, расположенного в памяти ВУ 1:
а – без применения оптимизации, *б* – применён алгоритм *ArrayPreload*

Листинг 3. Пример оптимизации передачи массива *A* в параллельной программе на языке IBM X10:
а – неоптимизированный код, *б* – оптимизированный код алгоритмом *ArrayPrelod*

```
for (i in 0..R) {
  val id: Long = i % Places.MAX_PLACES;
  at (Place.place(id)) {
    // Использование одного элемента A
    // приводит к копированию всего массива
    // на ВУ с номером placeId
    var a: Long = A(i % Size);
  }
}
```

а

```
// Пролог цикла копирует A на каждый узел
// один раз, сохраняя его в распределённом
// массиве localA
val localA: DistArray[Array[Long]] = ...
for (i in 0..R) {
  val id: Long = i % Places.MAX_PLACES;
  at (Place.place(id)) {
    // Использование локальной копии localA
    // массива A
    var a: Long = localA(id)(i % Size);
  }
}
```

б

Алгоритм *ArrayPreload* основан на статическом анализе PGAS-программы, представленной абстрактным синтаксическим деревом (АСД). Каждый узел в АСД имеет определённый тип в зависимости от выполняемой инструкции (например, операторы цикла, ветвления, конструкция передачи управления другому ВУ, объявление переменных и т.д.). На рис. 8 приведён фрагмент АСД неоптимизированной программы из листинга 3*а* на языке IBM X10 [13]. На вход *ArrayPreload* поступает указатель на корень *Root* АСД параллельной PGAS-программы. Результатом работы алгоритма является модифицированное АСД-программы.

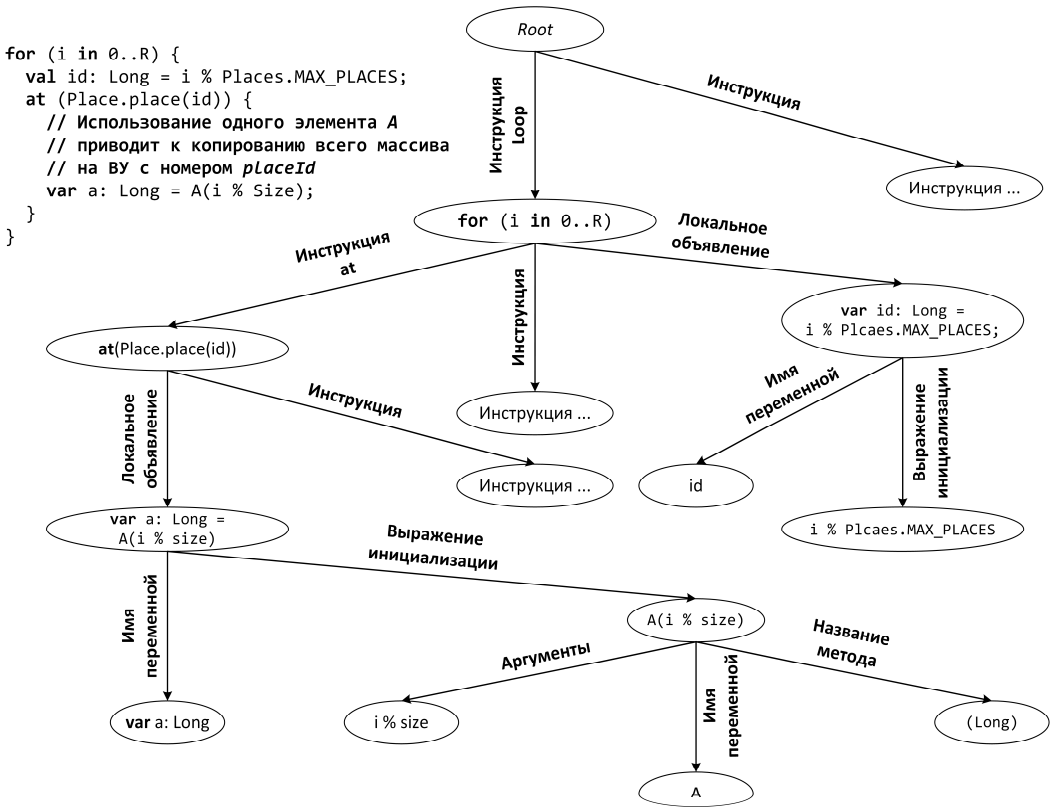


Рис 8. Фрагмент АСД параллельной PGAS программы на языке IBM X10, представленной на листинге 3а

Все основные действия алгоритма производятся во время рекурсивного обхода АСД в глубину. На первом этапе выполняется поиск циклов, содержащих обращения к элементам удалённых массивов. Рассмотрим подробнее этот этап. В листинге 4 приведён псевдокод функции *Visit*, выполняющая рекурсивный обход АСД в глубину. В процессе обхода отыскиваются узлы АСД с типом *Loop*, которые соответствуют оператору *for* (листинг 4, строка 6), и запускается функция анализа циклов (листинг 4, строка 7).

Функция *ProcessLoop* (листинг 5) выполняет проход по всем инструкциям цикла (листинг 5, строка 3) с целью поиска узла АСД типа *AtStmt*, который соответствует конструкции *at* (синоним конструкции *on*) передачи управления другому ВУ (листинг 5, строка 5). Для получения массива поддеревьев, соответствующих дочерним инструкциям, используется функция *GetStatements*. В случае обнаружения конструкции *at* вызывается функция *ProcessAtStmt* (листинг 5, строка 6), реализующая поиск операции чтения элементов массива типа *Array* (листинг 6). Для упрощения псевдокода в листинге 6 рассматривается случай использования удалённых массивов только при объявлении переменных в выражениях инициализации (*LocalDeclaration*). В общем случае последовательность действий анализа кода, выполняющегося на удалённом ВУ, будет зависеть от конкретной программной реализации модели PGAS. В процессе анализа инструкции *at* перебираются все её инструкции до тех пор, пока не будет найден узел АСД с типом *LocalDeclaration* (листинг 6, строка 5), указывающий на локальное объявление переменных. Затем необходимо проверить наличие операций чтения элементов массива в выражении инициализации. В описании алгоритма предполагается, что обращение к элементу массива выполняется при помощи вызова перегруженного оператора круглых скобок. При помощи функции *GetInitialization* получается узел АСД, соответствующий выражению инициализации объявляемой переменной (листинг 6, строка 7). Если полученный узел имеет тип *Call* (указывает на вызов метода объекта, листинг 6, строка 10), а объект – тип *Array* (листинг 6, строка 10), то выражение содержит обращение к массиву. На этом завершается первый этап работы алгоритма.

На втором этапе выполняется проверка найденного массива. Для применения оптимизации необходимо убедиться в том, что массив не изменяется на протяжении всех итераций

цикла, иначе применение оптимизации нарушит оригинальную семантику программы. Способ проверки неизменяемости массива зависит от реализации компилятора языка, например, данная проверка может быть реализована на основе заранее построенного контекста каждой инструкции цикла, или на основе контекста цикла целиком. Поэтому алгоритм проверки подробно не рассматривается, а используется функция *IsReadOnlyArray*, выполняющая её. Функция *IsReadOnlyArray* возвращает истину, если массив не изменяется на протяжении всего цикла, ложь – в противном случае. Функция *GetCallVarName* возвращает имя объекта, чей метод вызывается.

Листинг 4. Псевдокод *ArrayPreload*:
рекурсивный обход АСД с поиском циклов и добавлением пролога цикла

Входные данные:	*Root* – указатель на корень АСД параллельной PGAS-программы
Выходные данные:	*Root* – указатель на корень АСД модифицированной PGAS-программы

```
1  function VISIT(Root)
2  for each v in GETCHILDREN(Root) do // Перебор всех дочерних узлов
3     VISIT(v)                        // Рекурсивно посещаем вершину v
4  end for
5
6  if NODETYPE(Root) = Loop then
7     PROCESSLOOP(Root)
8  end if
9  end function
```

Листинг 5. Псевдокод *ArrayPreload*: функция анализа циклов

Входные данные:	*Node* – фрагмент АСД, соответствующий началу цикла
Выходные данные:	*Node* – модифицированный фрагмент АСД цикла

```
1  function PROCESSLOOP(Node)
2  // Проход по всем инструкциям цикла
3  for each S in GETSTATEMENTS(Node) do
4     // Поиск конструкции передачи управления другому ВУ
5     if NODETYPE(S) = AtStmt then
6        PROCESSATSTMT(S, Node)
7     end if
8  end for
9  end function
```

На третьем этапе выполняется трансформация АСД параллельной программы. В случае успешной проверки найденного массива (листинг 6, строки 10 – 11) необходимо создать пролог цикла, копирующий удалённый массив в память ВУ один раз перед итерациями, а также произвести замену чтения удалённого массива на скопированную прологом цикла локальную копию. Опережающее копирование можно выполнить различными способами, в зависимости от реализации модели PGAS. В данной работе опережающее копирование реализуется с использованием распределённого массива. Достоинством данной реализации является то, что она не требует дополнительного расширения модели PGAS. Программист вручную может выполнять данные преобразования после тщательного анализа кода программы, но это трудоёмко.

Предлагаемая реализация заключается в создании распределённого массива длиной по количеству ВУ, элементы которого распределены по одному на каждый ВУ. Каждый элемент будет представлять копию удалённого массива. Таким образом, каждый ВУ будет хранить локальную копию используемого массива.

Для создания пролога цикла необходимо:

1) Определить позицию оптимизируемого цикла в исходном коде (листинг 6, строка 12). Это необходимо для того, чтобы знать, куда необходимо вставить пролог цикла.
2) Генерация имени локальной копии массива (листинг 6, строка 15).
3) Формирование фрагмента АСД, который соответствует созданию распределённого массива для организации опережающего копирования согласно выше описанному способу (листинг 6, строка 17).
4) Вставка сформированного фрагмента в заданную позицию АСД программы, а именно, перед рассматриваемым циклом (листинг 6, строка 25). Позиция цикла определялась в строке 12.

После добавления пролога цикла необходимо выполнить замену чтения элементов удалённого массива на чтение элементов локальной копии. Для этого необходимо:

1) Получить имя локальной копии массива (листинг 6, строка 15).
2) Получить список аргументов (листинг 6, строка 13). Так как обращение к элементу массива – это вызов перегруженного оператора круглых скобок, то аргументом является индекс элемента массива. Обращение к локальной копии массива будет производиться по тем же самым индексам, что и обращение к удалённому массиву.
3) Создать фрагмент АСД, выполняющий чтение используемых элементов локального массива (листинг 6, строка 21).
4) Выполнить замену фрагментов АСД (листинг 6, строка 23).

Листинг 6. Псевдокод ArrayPreload: функция анализа конструкции at

Входные данные: *Node* – фрагмент АСД, соответствующий началу конструкции *at*
NodeLoop – фрагмент АСД, соответствующий началу цикла
Выходные данные: *Node* – модифицированный фрагмент АСД конструкции *at*

```
1   function ProcessAtStmt(Node, NodeLoop)
2   // Перебор всех инструкций конструкции at
3   for each S in GetStatements(Node) do
4     // Обработка локальных объявлений
5     if NodeType(S) = LocalDeclaration then
6       // Получение выражения инициализации
7       I = GetInitialization(S)
8       // Поиск обращения к элементам Array-массива,
9       // независимого от итерации к итерации
10      if NodeType(I) = Call && typeof(GetTypeOfTarget(I)) = Array &&
11                                IsReadOnlyArray(GetCallVarName(I)) then
12        pos = GetSrcPosition(NodeLoop) // Позиция цикла в исходном коде
13        args = GetCallArguments(I)
14        // Генерация имени локальной копии массива
15        localName = GenerateNewName()
16        // Создание кода пролога цикла
17        loopProlog = CreateDistrebutedArray(localName, GetCallVarName(I))
18        // Определение номера ВУ, на котором выполняется код
19        localPlace = CreateLocalPlace()
20        // Создание кода обращения к локальному массиву
21        newCall = CreateLocalCall(localName, localPlace, args)
22        // Замена обращения к удалённому массиву обращением к локальной копии
23        ReplaceNewCall(I, newCall)
24        // Добавление пролога цикла
25        InsertStatementsToPos(loopProlog, pos)
26      end if
27    end if
28  end for
29  end function
```

Рассмотрим описанные действия на фрагменте АСД из рис. 8. В процессе рекурсивного обхода будет обнаружен узел цикла **`for (i in 0..R)`**, имеющий тип *Loop*. Поддерево с корнем, соответствующее циклу **`for`**, передаётся в функцию *ProcessLoop* для поиска в теле цикла инструкции *at* передачи управления другому ВУ. После обнаружения инструкции *at* она передаётся в функцию *ProcessAtStmt* для поиска в ней чтения элементов массива, расположенных в памяти удалённого ВУ. Таким массивом является *A*. Массив *A* не изменяется на протяжении всего цикла, значит, можно произвести оптимизацию данного участка программы. Для опережающего копирования создаётся распределённый массив *localA*, элементами которого является массив *A*. После чего в конструкции *at* будет использоваться массив *localA* вместо удалённого массива *A*.

На рис. 9 приведён фрагмент АСД программы на языке IBM X10 после применения оптимизирующей трансформации. Жирным выделены узлы, подвергшиеся модификации. Был добавлен пролог цикла **`val localA: DistArray[Array[Long]]`**, а также заменено обращение к массиву *A* на обращение к массиву *localA*.

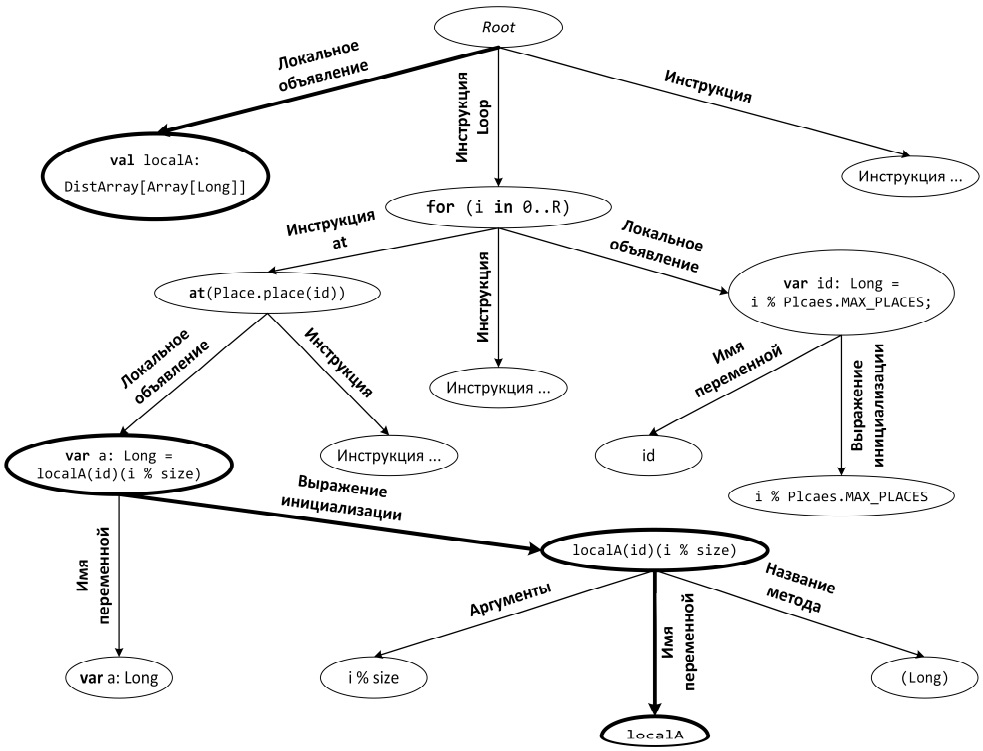


Рис. 9. Фрагмент АСД-программы на языке IBM X10
после применения оптимизирующей трансформации (алгоритм *ArrayPreload*)

Вычислительная сложность алгоритма *ArrayPreload* определяется высотой АСД-программы, все трансформации выполняются за один проход АСД.

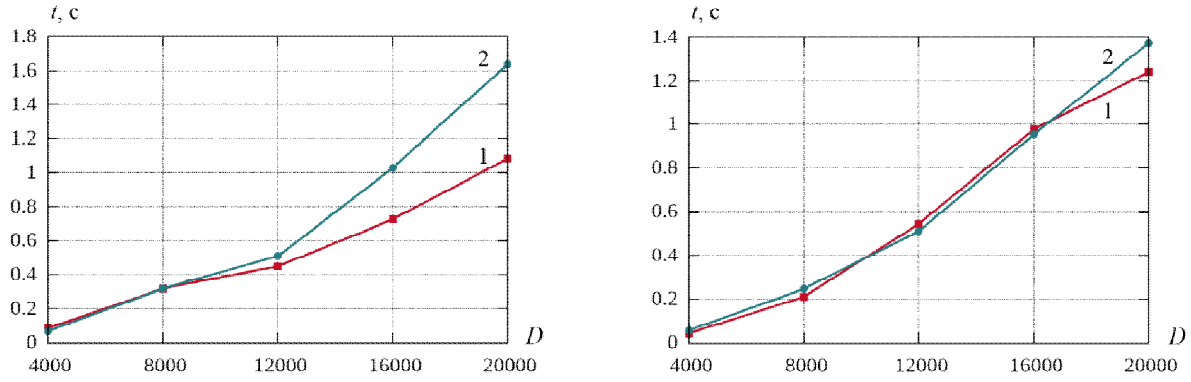
3. Исследование эффективности алгоритмов

3.1. Исследование эффективности алгоритмов

Исследование алгоритмов проводилось на вычислительных кластерах Jet (16 ВУ на базе двух четырёхъядерных процессоров Intel Xeon E5420, соединённых сетью Gigabit Ethernet) и Oak (6 ВУ на базе двух четырёхъядерных процессоров Intel Xeon E5420, соединённых сетью InfiniBand QDR) Центра параллельных вычислительных технологий ФГОБУ ВПО «Сибирский государственный университет телекоммуникаций и информатики» и Института физики полупроводников им. А.В. Ржанова СО РАН. Созданные алгоритмы реализованы для языков Cray Chapel (*BlockReduce*) и IBM X10 (*ArrayPreload*).

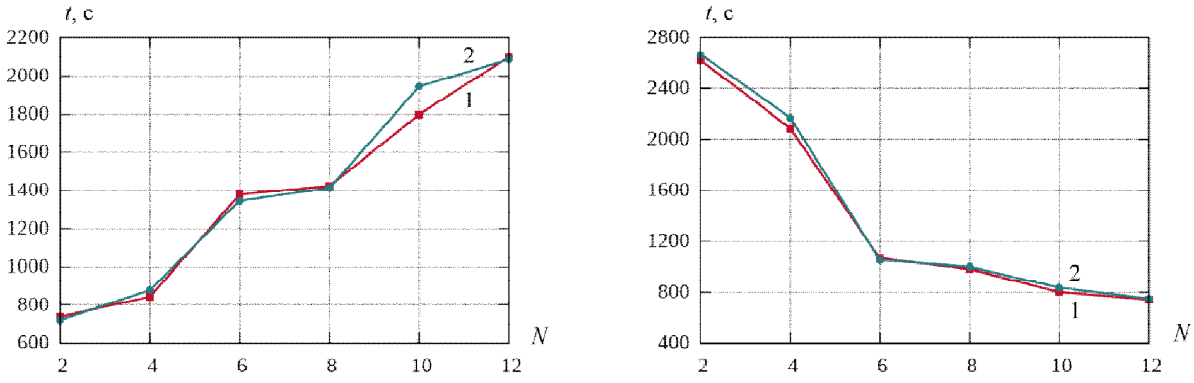
Сравнительный анализ алгоритмов редукции выполнялся на основе синтетических тестов (редукция распределённых массивов длины $D = 4000, \ldots, 20000$) и тестовых Chapel-программ PTRANS (транспонирование распределённых матриц) и miniMD (молекулярная динамика). Варьировалось количество ВУ $N = 1, 2, \ldots, 16$.

Эффективность алгоритма *BlockReduce* (рис. 10, 11) зависит от числа N вычислительных узлов и размера массива. Применение алгоритма *BlockReduce* позволяет в среднем сократить на 10 – 30% время выполнения редукции по сравнению со стандартным алгоритмом *DefaultReduce* (алгоритм с линейной сложностью). Незначительное сокращение времени выполнения реальных программ (рис. 11) по сравнению с алгоритмом *DefaultReduce* объясняется тем, что время реализации отдельных операции редукции намного меньше суммарного времени выполнения программы.



а *б*

Рис. 10. Среднее время выполнения операции редукции n элементов
1 – *BlockReduce*, 2 – *DefaultReduce*
а – $N = 4$, *б* – $N = 8$

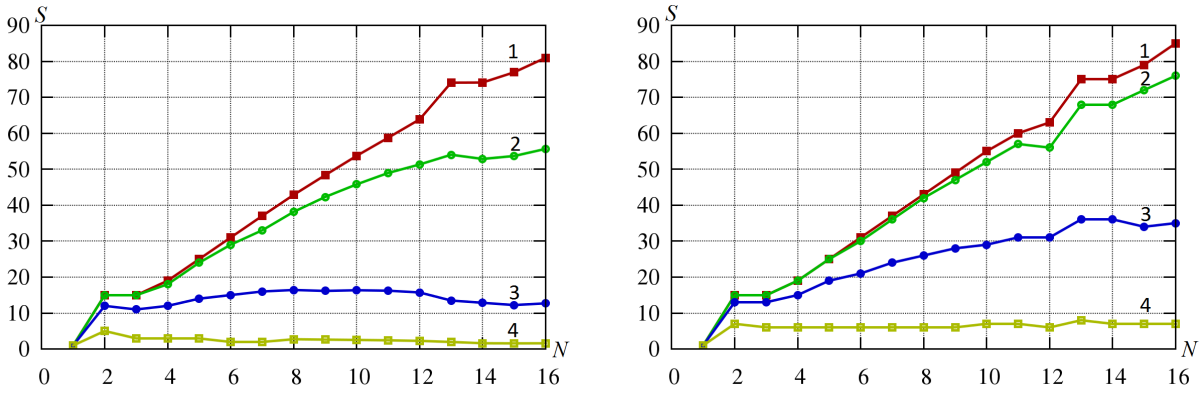


а *б*

Рис. 11. Время выполнения параллельных программ
1 – *BlockReduce*, 2 – *DefaultReduce*
а – программа miniMD, *б* – программа ptrans

Для оценки эффективности алгоритмов *ArrayPreload* и *Scalar replacement* использовался синтетический тест – циклический доступ к элементам массива, расположенным в памяти удалённых ВУ. Компилятор IBM X10 был собран с библиотеками MPICH2 3.0.4 (Jet) и MVAPICH2 2.0 (Oak).

На рис. 12 представлены графики зависимости коэффициента ускорения выполнения синтетического теста после применения алгоритмов оптимизации (*ArrayPreload*, *Scalar replacement*) от количества *N* ВУ и числа *R* обращений к массиву. На данном тесте оба алгоритма демонстрируют ускорение от 5 до 82 раз. В общем случае ускорение зависит от производительности коммуникационной сети, количества ВУ, размера массива, количества итераций в цикле (в теле которого организован циклический доступ).



а *б*

Рис. 12. Ускорение тестовой программы (кластер Jet):
1 – $D = 40000$, $R = 40000$; 2 – $D = 40000$, $R = 4000$;
3 – $D = 40000$, $R = 400$; 4 – $D = 40000$, $R = 40$;
а – алгоритм *ArrayPreload*, *б* – алгоритм *Scalar replacement*

8. Заключение

Предложенные эвристические алгоритмы позволяют уменьшить время выполнения параллельных PGAS-программ за счёт сокращения времени реализации информационных обменов. Последнее достигается средствами опережающего копирования удалённых массивов и учётом иерархической структуры ВС. Созданные алгоритмы применимы для широкого спектра языков семейства PGAS.

Литература

1. *Хорошевский В.Г.* Распределённые вычислительные системы с программируемой структурой // Вестник СибГУТИ. 2010. № 2 (10). С. 3–41.
2. *Rabenseifner R.* Optimization of Collective Reduction Operations // Computational Science - ICCS 2004 – Lecture Notes in Computer Science. 2004. Vol. 3036. P. 1–9.
3. *Li S., Hoefler T., Snir M.* NUMA-Aware Shared-Memory Collective Communication for MPI // HPDC 2013. 2013. P. 85–96.

4. *Курносов М.Г.* Алгоритмы трансляционно-циклических информационных обменов в иерархических распределённых вычислительных системах // Вестник компьютерных и информационных технологий. 2011. № 5. С. 27 – 34.
5. *Kennedy K.*, *Allen John R.* Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA 2002. 834 p.
6. *Barik R.*, *Zhao J.*, *Grove D.*, *Peshansky I.*, *Budimlic Z.*, *Sarkar V.* Communication Optimizations for Distirbuted – Memory X10 Programs // IEEE International Parallel and Distributed Processing Symposium. 2011. P. 1 – 13.
7. *Chen W.*, *Iancu C.*, *Yelick K.* Communication Optimizations for Fine-grained UPC Applications // 14th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2005. Tech Report LBNL-58382.
8. *Mallón D.A., Taboada G.L., Teijeiro C., Domínguez J.G., Gómez A., Wibecan B.* Scalable PGAS collective operations in NUMA clusters // Cluster Computing. 2014. 23 p.
9. *Nishtalab R., Zhenga Y., Hargrovea P.H., Yelick K.A.* Tuning collective communication for Partitioned Global Address Space programming models // Parallel Computing. 2011. Vol. 37. P. 576 – 591.
10. *Callahan D., Chamberlain B.L., Zima H.P.* The Cascade High Productivity Language // HIPS 2004. 2004. P. 52 – 60.
11. *Nanjegowda R., Hernandez O., Chapman B., Jin H.* Scalability Evaluation of Barrier Algorithms for OpenMP // IWOMP '09 Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism. 2009. Vol. 5568. P. 42 – 52.
12. *Callahan D.*, *Carr S.*, *Kennedy K.* Improving register allocation for subscripted variables // PLDI '90, New York, NY, USA, 1990. ACM. P. 53 – 65.
13. *Charles P., Donawa C., Ebcioglu K., Grothoff C., Kielstra A., Praun C., Saraswat V., Sarkar V.* X10: An Object-oriented approach to non-uniform Clustered Computing // OOPSLA 2005. P. 519 – 538.

Статья поступила в редакцию 30.06.2014

Кулагин Иван Иванович
аспирант кафедры вычислительных систем СибГУТИ (630102, Новосибирск, ул. Кирова, 86) тел. 8-913-772-09-19, e-mail: ivan.i.kulagin@gmail.com.

Пазников Алексей Александрович
к.т.н., старший преподаватель кафедры вычислительных систем СибГУТИ (630102, Новосибирск, ул. Кирова, 86) тел. (383) 2-698-293, e-mail: apaznikov@gmail.com.

Курносов Михаил Георгиевич
к.т.н., директор Центра параллельных вычислительных технологий СибГУТИ (630102, Новосибирск, ул. Кирова, 86) тел. (383) 2-698-275, e-mail: mkurnosov@gmail.com.

Heuristic algorithms of communication optimization in parallel PGAS-programs

I. Kulagin, A. Paznikov, M. Kurnosov

This paper represents the heuristic algorithms of communication optimization in parallel PGAS-programs providing time minimization of its execution. This is achieved by accounting of hierarchical structure of computer systems while executing reduction and preloading of remote arrays to computer system's nodes. Developed algorithms are implemented for PGAS languages: Cray Chapel and IBM X10 and simulated on cluster computer systems.

Keywords: PGAS, compiler optimization, reduction, scalar replace, Cray Chapel, IBM X10.