УДК 004.272.44
DOI 10.53403/9785951505071_2022_193

ОСОБЕННОСТИ РЕАЛИЗАЦИИ ТЕСТА HPCG ДЛЯ ППВС «БУРАН»

Д. Н. Змеев, А. В. Климов, А. С. Окунев, Н. Н. Левченко

Институт проблем проектирования в микроэлектронике РАН, Зеленоград

HPCG (High Performance Conjugate Gradients) –- новый тест для сравнения суперкомпьютерных систем после LINPACK HPC. Считается, что он лучше отражает характеристики рабочей нагрузки реальных научных приложений. Рассматривается реализация его центральной части – сглаживателя Гаусса-Зейделя – на проектируемой параллельной потоковой вычислительной системе (ППВС) «Буран». В отличие от эталонного кода HPCG для MPI, который использует смешанный метод Якоби – Гаусса–Зейделя, мы реализуем чистый алгоритм Гаусса-Зейделя на предлагаемом языке потоков данных UPL, обеспечивающем автоматическое извлечение потокового параллелизма. Таким образом, мы нарушаем условия теста, используя другой вычислительный алгоритм, который частично компенсирует большие задержки сети и памяти.

Ключевые слова: потоковая модель вычислений, архитектура потока данных, параллельные вычисления, высокопроизводительные вычисления, тест HPCG, метод сопряженных градиентов, метод Гаусса–Зейделя.

1. Введение

Тест HPCG (High Performance Conjugate Gradients) [1] является попыткой создать новую метрику для сравнения суперкомпьютерных систем. Он призван дополнить используемый в настоящее время для этой цели тест HPL (High Performance LINPACK), поскольку задачи линейной алгебры, из которых тот состоит, не предъявляют в должной мере требований к свойствам аппаратуры, которые критичны для многих современных задач. Это, прежде всего, требования к подсистеме памяти и коммуникационной сети, их пропускной способности и латентности. А алгоритмам HPL обычно удается обходить эти проблемы.

Авторы теста HPCG и его эталонного кода требуют, чтобы при тестировании новой суперкомпьютерной системы использовался именно их численный метод и принципы его распараллеливания, и чтобы не вносились изменения в код, которые позволяли бы обходить недостатки аппаратуры. В этой статье мы демонстрируем как раз такой альтернативный способ реализации.

Мы рассматриваем возможность реализации данного теста на проектируемой в нашей организации параллельной потоковой вычислительной системе (ППВС) «Буран» [2], реализующей потоковую модель вычислений с динамически формируемым контекстом [3]. От обычных фоннеймановских вычислителей ее отличает использование принципа потока данных, состоящего в активации вычислительных узлов по готовности аргументов, а от классических потоковых архитектур – использование контекста как объекта данных «первого класса», не ограниченного формами, диктуемые стандартными шаблонами программирования (циклы, доступ к элементам массива, рекурсия, и т. п.) в распространенных языках.

Специфика модели вычислений позволяет использовать другой вариант алгоритма предобуславливателя, используемого в тесте, который, с одной стороны, обеспечивает более быструю сходимость, а с другой стороны, при распределенном выполнении является менее требовательным

к латентности памяти и коммуникационной сети. Это особенно актуально, если учесть, что именно латентность сетей и памятей большого размера (в отличие от пропускной способности) с годами улучшается крайне медленно [4]. Также важно отметить, что этот вариант алгоритма в нашей модели вычислений реализуется достаточно просто, тогда как его программирование, например, в MPI настолько сложно, что этот путь даже и не рассматривается.

Статья имеет следующую структуру. В разделе 2 рассмотрен алгоритм теста и проблемы, возникающие при его распараллеливании. В разделе 3 математически изложен фрагмент задачи сглаживателя и его вычислительный граф. В разделе 4 рассмотрены варианты декомпозиции для распараллеливания. В разделе 5 вводится язык UPL, на котором записывается стандартный алгоритм одного прохода по Гауссу–Зейделю, и разъясняется его работа. В разделе 6 изложены принципы работы потокового вычислителя. В разделе 7 определяется функция распределения вычислений по пространству (процессорным ядрам), обеспечивающая хорошее масштабирование. Возникающие при этом блоки могут использоваться для укрупнения узлов (раздел 8). А раздел 9 говорит о том, как распределение по времени помогает решить проблему «стены памяти». Результаты экспериментального моделирования приведены в разделе 10. В Заключении подводятся итоги и перечисляются нерешенные проблемы.

2. Описание задачи теста

В тесте HPCG решается задача итерационного решения системы линейных алгебраических уравнений (СЛАУ) вида $Ax = b$. Основная операция в каждой итерации – это умножение большой разреженной матрицы A на плотный вектор (текущее приближение) x. Матрица A является симметричной и положительно определенной. Портрет (места ненулевых элементов) матрицы A не случаен, он получен из отношения соседства на регулярной прямоугольной трехмерной сетке (рис. 1), где у каждой ячейки не более 26 соседей, а каждое уравнение связывает значение в ячейке со значениями в соседних ячейках. которой содержат элементы искомого вектора x. Ячейке сетки соответствует линейное уравнение, связывающее значение в ячейке со значениями в ее соседях. Задачей теста является решение этой системы уравнений. Если сетка имеет размеры n_x, n_y, n_z, то матрица A имеет порядок $N = n_x n_y n_z$.

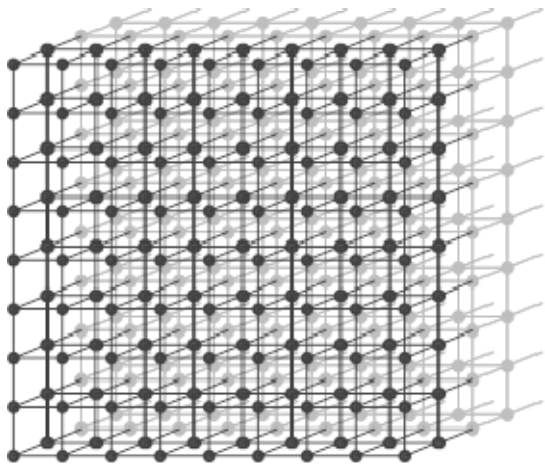

Рис. 1. Структура трехмерной сетки. Показана только часть связей (без «диагональных»)

Общее число итераций метода CG можно уменьшить, если сделать предобусловливание матрицы A. Иначе говоря, умножить ее на матрицу-предобуславливатель M, сближающее ее с единичной, и решать уравнение $MAx = Mb$ вместо исходного. Вместо матричного умножения MA вы-

годнее делать умножение матрицы M на вектор z всякий раз, когда в методе CG выполняется умножение матрицы A на вектор x (и один раз для вектора правых частей b). А такое умножение равносильно приближенному решению уравнения $Az = r$ итерационным методом с применением небольшого фиксированного числа линейных итераций – сглаживателей.

В качестве базового сглаживателя применяется один или несколько шагов итераций Симметричного метода Гаусса–Зейделя (СГЗ) решения уравнения $Az = r$, который сводится к попеременному решению двух треугольных систем уравнений: с нижне-треугольной матрицей $L + D$ и верхне-треугольной матрицей $U + D$, полученными разложением матрицы $A = L + D + U$ (здесь $D = diag(A)$).

Для ускорения сходимости метода CG применяется многосеточный сглаживатель, так называемый V-цикл [5]: строятся 4 уровня все более грубых сеток посредством операций рестрикции и пролонгации, а между ними как раз и применяется СГЗ. Считается, что это сочетание (многосеточность и сглаживатель Гаусса–Зейделя при переходе между уровнями) дает хорошие результаты в плане улучшения сходимости CG [5, 6].

Однако, в общем случае, метод Гаусса–Зейделя трудно распараллеливается: в работе [7], например, автору удается достичь лишь ускорения в 2 раза для 4 процессоров, и дальше оно не растет. Причина этого в том, что для плотных матриц каждая переменная зависит от предыдущей. А в разреженном случае некоторые группы переменных могут определяться независимо, и это дает дополнительное пространство для параллелизма. Тем не менее, считается [6], что в матрице, порожденной трехмерной сеткой, параллелизма немного, и извлечь его – трудная задача.

Поэтому, как правило (в том числе в HPCG), применяют смешанный метод Гаусса–Зейделя–Якоби. Он состоит в разбиении пространства ячеек сетки на прямоугольные блоки, каждый из которых обрабатывается своим отдельным процессором. Каждый процессор выполняет одну итерацию СГЗ локально (при этом для всех «внешних» переменных берутся значения из предыдущей итерации), после чего все процессоры обмениваются значениями вычисленных переменных с соседями (рис. 2). В вычислительном плане это другой метод: он легко программируется как параллельный, но медленнее сходится [6, 8].

Ниже мы реализуем чистый метод СГЗ на потоковом языке UPL [9], извлекая максимум параллелизма из разреженной структуры матрицы A.

3. Симметричный итерационный метод Гаусса–Зейделя

Пусть $A = L + D + U$ – симметричная положительная определенная матрица, z – вектор правой части. Итерационно решается уравнение $Az = r$. Имея приближение $x^{(v)}$, вычисляем приближение $x^{(v+1)}$ из уравнений (верхний индекс в скобках – номер итерации):

$$Lx^{(v+1)} + Dx^{(v+1)} + Ux^{(v)} = r, \text{ если } v \text{ четно, и}$$
$$Lx^{(v)} + Dx^{(v+1)} + Ux^{(v+1)} = r, \text{ если } v \text{ нечетно.}$$

Вычислительный процесс проходит по следующим формулам (матрично-векторным):

$$u^{(0)} = x^{(0)} = 0$$

$$\begin{cases} l^{(2k+1)} = Lx^{(2k+1)} \\ x^{(2k+1)} = D^{-1}\left(r - u^{(2k)} - l^{(2k+1)}\right) \end{cases}, k = 0,1, \ldots$$

$$\begin{cases} u^{(2k)} = Ux^{(2k)} \\ x^{(2k)} = D^{-1}\left(r - l^{(2k-1)} - u^{(2k)}\right) \end{cases}, k = 1,2, \ldots$$

Для удобства введены две (векторные) переменные: l и u. Наличие неявности по x не мешает, поскольку она разрешается благодаря треугольности матриц L и U: для вычисления -го элемента

вектора l достаточно иметь значения компонентов вектора x с меньшими индексами, а для u – с большими. Поэтому итерации с нечетными номерами вычисляются прямым проходом (вдоль векторов), а с четными – обратным.

Выполнение одной итерации, например нечетной (с номером $2k+1$), можно рассматривать как решение системы линейных алгебраический уравнений (СЛАУ) с треугольной матрицей $L+D$ относительно переменных $x^{(2k+1)}$:

Граф зависимостей между узловыми переменными, показан на рис. 2. (Здесь и далее для упрощения и наглядности используется 2-мерная сетка, при этом все сказанное без проблем обобщается на 3-мерный случай). Узлы – это ячейки сетки, а дугам соответствуют ненулевые элементы матрицы A (здесь из 144 строк). Для каждой связи оставлено одно из двух направлений согласно выбору части L или U матрицы. Порядок соответствует лексикографическому по координатам ячеек $\langle \dots, i_2, i_1 \rangle$ (младший индекс i_1, следующий по старшинству i_2, и т. д.)

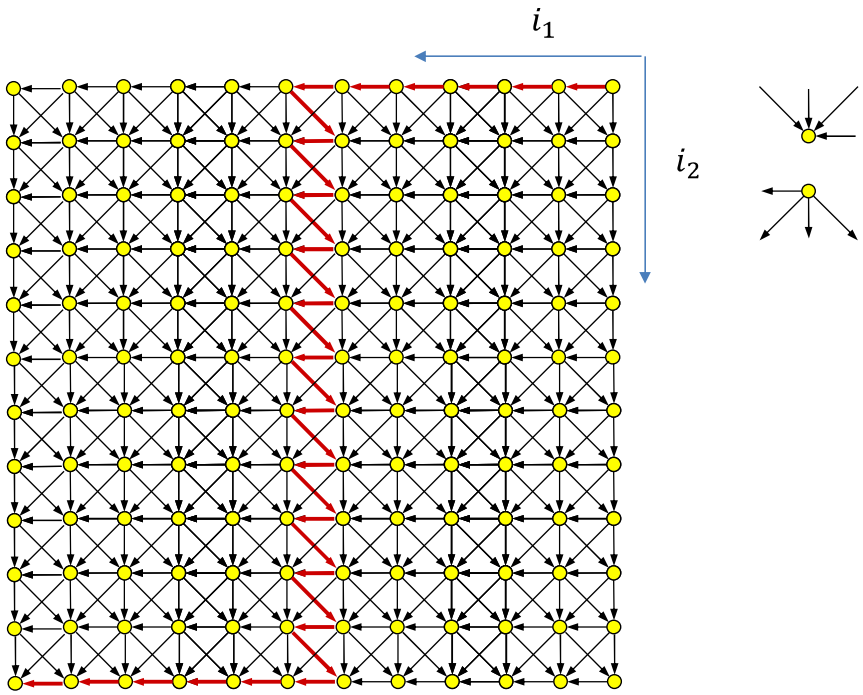

Рис. 2. Граф зависимостей одного прохода сглаживателя по Гауссу–Зейделю.
Красным цветом выделен один из критических путей

4. Варианты декомпозиции для распареллеливания

В эталонной реализации HPCG на C++ для MPI+OpenMP [10] для распараллеливания используется декомпозиция сетки на прямоугольные области, каждая из которых помещается в свой процесс. При этом умножение матрицы A на текущее приближение может быть «честно» выполнено автономно в каждой области при условии, что значения внешних переменных доставлены из других процессов. Однако, сглаживатель Гаусса-Зейделя при такой декомпозиции выполняется неточно: одна итерация метода выполняется локально в каждом прямоугольнике (см. рис. 3), используя всегда старые значения переменных из соседних процессов (прямоугольников), после чего новые значения переменных вдоль границ областей пересылаются соседям. Фактически это смешанная итерация Гаусса–Зейделя–Якоби: локально по Гауссу–Зейделю, глобально по Якоби. Такой вариант легко программируется, но требует большего числа итераций для достижения того же качества в смысле сходимости [6, 8].

Мы добиваемся распараллеливания вычислительного графа «чистого Гаусса–Зейделя», показанного на рис. 2. В этом мы отходим от условий теста HPCG. Наша цель – показать, что в модели вычислений ППВС можно легко осуществлять способы распараллеливания, которые трудны для MPI и OpenMP, и потому обычно отвергаются.

Множество входящих и исходящих дуг каждой ячейки показано на рис. 2 справа (в трехмерном случае их число равно 13). Положительные линейные комбинации этих векторов в системе координат сетки образуют конус зависимостей.

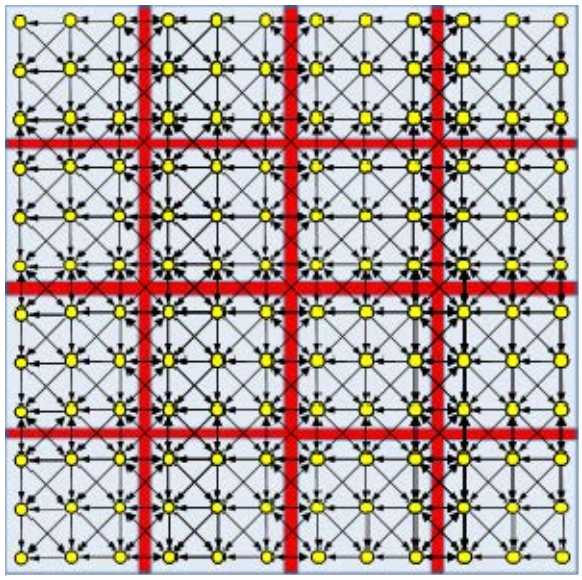

Рис. 3. Обычная декомпозиция на прямоугольные области, применяемая в HPCG. Внутри областей итерации по Гауссу–Зейделю, между областями – по Якоби

Согласно общим принципам, описанным в [11], такие зависимости допускают распараллеливание путем разбиения на блоки-параллелепипеды с гранями, параллельными граням конуса зависимостей (в котором следует оставить k граней, где k – размерность сетки). В данном случае для $k = 2$ это стороны параллелограмма на векторах (0,1) и (1,–1). Для $k = 3$ – параллелепипед на векторах (0,0,1), (0,1,–1), (1,–1,–1). По каждому направлению (вектору) можно выбрать свой размер блока. На рис. 4 показаны блоки 3×3.

Индуцируемый граф зависимостей между блоками ациклический, поскольку в блочной системе координат, основанной на указанных векторах, направления всех связей неотрицательные. Каждый блок готов к выполнению, когда завершены блоки сверху и справа от него. Поэтому здесь уместен потоковый параллелизм. Он создает фронт волны вычислений, параллельный большой диагонали блока, движущийся в направлении вектора (2,1).

Для другой части матрицы будет тот же граф, но с обратными стрелками.

5. Программа в потоковой модели вычислений

Потоковая программа решателя треугольной СЛАУ вида $\boldsymbol{Lx = b}$, представленная в графической версии языка UPL [9], показана на рис. 5. Эта программа применима к произвольной разреженной треугольной матрице L (здесь она содержит диагональ, которая должна быть целиком ненулевой), с любым «портретом» (включая плотный), а не только построенным из регулярной сетки.

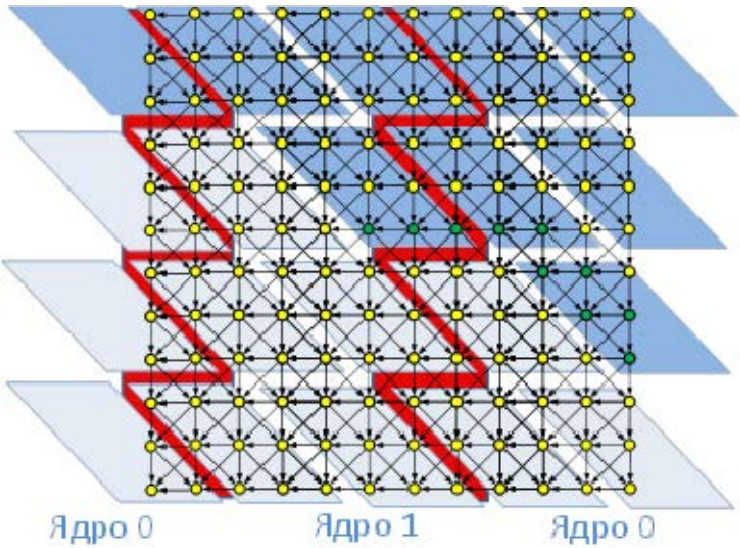

Рис. 4. Разбиение на блоки-параллелограммы. Каждый блок получает значения переменных от блоков справа и сверху и выполняется по приходу этих данных. Вычисленные блоки – темно-синие. Фронт движется «по диагонали» сверху вниз справа-налево. Красный зигзаг – распределение блоков по вычислительным ядрам. По пересекающим его стрелкам данные передаются между ядрами

Каждый узел, изображенный прямоугольным блоком, выполняет одну скалярную операцию. Все узлы имеют один или два индекса, *i,j*, стоящие на фоне шестиугольников: *i* – номер уравнения, *j* – номер переменной. (Этот номер в нашем случае будет иметь структуру $\langle i_3, i_2, i_1 \rangle$, но в программе это никак не проявляется). По стрелкам передаются токены со значениями переменных на входы узлов, изображенные в виде желтых овалов с именами. Токен со значением *x*, идущий на вход a узла *N* с индексом *I* обозначается как $x \to N.a\langle I \rangle$.

Значение *x* определяется либо формулой, записанной внутри узла, либо формулой, стоящей на стрелке (на фоне маленького прямоугольника), либо значением первого входа. Индекс узла-получателя *I* стоит на стрелке на фоне розового шестиугольника (а по умолчанию формируется из одноименных индексов источника). У начала стрелки может стоять условие выдачи токена (например, из узла *L* токен выдается на узел Mul при $i > j$, а на узел Div при $i = j$). Узел (с конкретным индексом) активируется, когда на всех его входах присутствуют токены. В момент активации токены удаляются, а их значения (с общим индексом) становятся параметрами процедуры узла, которая будет исполнена и породит новые токены.

Входные узлы слева в форме больших желтых овалов – это источники входных данных. Аналогичный узел справа принимает выходной результат. Каждый скалярный элемент (коэффициент или значение переменной) передается или принимается отдельным токеном вместе с индексом. Над входными узлами записаны области допустимых значений индексов, но это лишь неформальные комментарии.

Матрица *L* разреженная в том смысле, что она представляется как множество токенов с ненулевыми элементами матрицы, каждый со своей парой индексов $\langle i, j \rangle$. Кроме матрицы *L* и правых частей *b* также на вход подаются число ненулевых элементов в -й строке, $k\langle i \rangle$, и число ненулевых элементов в *j*-м столбце, $m\langle j \rangle$. В отличие от матрицы *L*, вектора b, k, m плотные, т. е. содержат в виде токенов все свои элементы, включая нулевые.

Узел Sum – это особый, суммирующий узел. Он имеет суммирующий вход *s*, заданный с префиксом (+), указывающим операцию редукции, и суффиксом [*k*], указывающим, что число слагаемых придет на другой одноименный вход *k*. Узел исполняется, когда на вход s придет *k* токенов, а значение *k* придет на другой вход, *k*. Результатом становится сумма значений этих *k* токенов. Значение *k* может быть нулевым, тогда по его приходу узел Sum сразу сработает с выдачей нулевого

результата, не ожидая токенов на входе *s* (и не используя его, даже если он и придет). Здесь в качестве одного из слагаемых всегда приходит правая часть $b\langle i\rangle$. Поэтому k должно быть числом ненулевых элементов, включая диагональный.

Узел Div⟨j⟩ получает сумму *s* от узла Sum⟨j⟩ и значение диагонального элемента *d* от входного узла L⟨j,j⟩, а посылает частное s/d на узел X⟨j⟩ и на выход Xout⟨j⟩.

Еще одна особенность – токен, посылаемый узлом X⟨j⟩. Он содержит кратность #m и множественный индекс вида ⟨*,j⟩. Такой токен будет взаимодействовать.

Дана: нижне-треугольная матрица L (с диагональю) и вектор b.
Матрица L разрежена, все диагональные элементы отличны от 0.
k_i – число ненулевых элементов в строке i (включая диагональный).
m_j – число ненулевых элементов в столбце j (исключая диагональный).
Найти: вектор x, такой что $Lx=b$.
Формула решения:

$$x_i = (b_i - \sum_{j=1}^{i-1} x_j L_{ij}) / L_{ii}, \text{ для } i \text{ от } 1 \text{ до } N$$

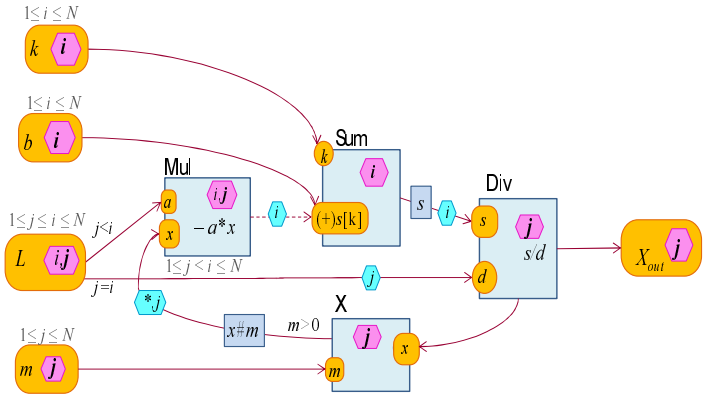

Рис. 5. Программа решения треугольной СЛАУ на графическом потоковом языке UPL

Данная программа будет хорошо и правильно работать, только если исходные данные правильные, т. е. *k* и *m* это действительно числа ненулевых элементов. В противном случае может возникнуть зависание или по окончании работы на входах некоторых узлов останется «грязь».

Применительно к нашей «сеточной» матрице эта программа «ничего не знает» ни про блоки, ни про фактические зависимости. Если матрица треугольная (при каком-либо порядке строк), то индуцируемый ею граф зависимостей ациклический, что необходимо и достаточно для завершения вычисления.

Процесс выполнения программы происходит следующим образом. Первым сработает узел Sum⟨i⟩, на который придет токен $1 \to \sum.k\langle i\rangle$ (для нижнетреугольной матрицы $i = 1$) и значение правой части $b_i \longrightarrow \sum.s\langle i\rangle$.

В результате будет выдан токен $b_i \to \div.s\langle i\rangle$, который «спарится» с токеном $L_{ii} \to \div.d\langle i\rangle$. Активированный узел Div⟨j⟩ (для $j = i$) вычислит значение переменной $x_j = s/d$, выдав токен $x_j \to X.x\langle j\rangle$. Узел X⟨j⟩ (получив также $m_j \to X.m\langle j\rangle$) выдаст токен $x_j \to Mul.x\langle ,j\rangle$. Этот токен «спарится» со всеми имеющимися токенами вида $L_{ij} \to Mul.a\langle i,j\rangle$ (с различными *i*, которых должно быть ровно m_j), всякий раз порождая активацию узла Mul, выдающую произведение $(-l_{ij} * x_j) \to \sum.s\langle i\rangle$. Оно сложится с такими же слагаемыми в узле $\sum\langle i\rangle$. И если число слагаемых достигло k_i, процесс продолжится для всех таких *i*. Процесс завершится вычислением значений всех переменных, если и только если граф зависимостей согласно матрице *L* является ациклическим.

6. Принципы работы потокового вычислителя

Схема устройства, способного вычислять потоковую программу, показана на рис. 6

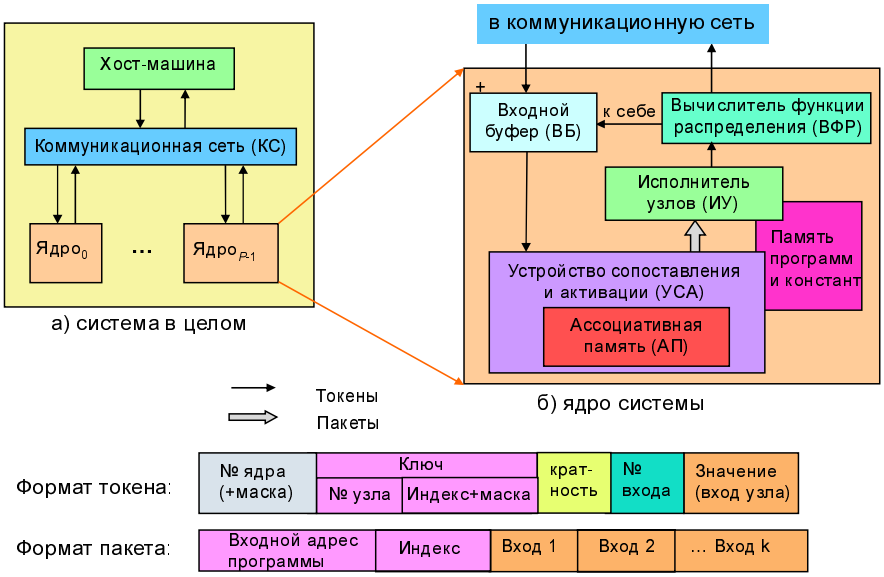

Рис. 6. Структура исполняющей системы

В каждом ядре имеется полнофункциональный вычислитель, отвечающий за свою часть адресного пространства задачи, где роль виртуальных адресов играют ключи вида ⟨имя узла, индекс⟩. (Маска ключа отмечает позиции со звездочками, что позволяет такому ключу «спариваться» с другими ключами с различными значениями в этой позиции). На основе ключа в ВФР определяется значение номера ядра, в который будет направлен токен. (При непустой маске ключа, если ФР существенно зависит от замаскированных полей, этот номер также будет иметь ненулевую побитовую маску, и тогда токен будет направлен в некоторое множество ядер [12]).

Со входа ядра (через ВБ) токены поступают в УСА, в АП которого хранятся ранее поступившие токены. Там обнаруживаются наборы из токенов, поступивших на все входы виртуального узла с конкретным индексом. Тогда этот узел активируется: токены набора изымаются из АП (при наличии кратности в токене, она уменьшается на 1 и, если получается не 0, токен остается в АП) и одновременно формируется пакет с данными для всех входов, который передается (через буфер пакетов) в ИУ, где он будет исполнен позднее. Результатом выполнения пакета будет образование одного или нескольких (возможно, ни одного) токенов, поступающих через ВФР в КС (или в свой ВБ).

7. Распределение узлов по ядрам ППВС

Каждой ячейке на рис. 2 отвечает одна (j-я) переменная и уравнение с тем же номером, из которого она определяется. В программе ей соответствует группа узлов с индексом j (или i, когда j отсутствует). На рисунке эти индексы выделены жирным шрифтом. Будет удобно (и экономно), если все узлы одной группы выполняются в одном ядре, иначе говоря, если функция распределения зависит только от выделенного индекса. Тогда только одна дуга на схеме переходит (возможно) из одного ядра в другое: это дуга из узла Mul в узел Sum, она выделена пунктиром. По ней передаются произведения j-й переменной на элементы j-го столбца матрицы.

Таким образом, одно ядро отвечает за некоторое множество таких групп, или ячеек. Будет так же экономно, если одно множество содержит тесно связанные ячейки, а связей между ними мало. Учитывая, что время передачи токена между ядрами (латентность) может быть заметно больше, чем время передачи «к себе», важно также, чтобы критические пути (на рис. 2 один такой путь вы-

делен красным) пересекали границу между ядрами малое число раз. Это критическое число обозначим как CN.

При разрезании по вертикалям $CN = n_x/d + 2n_y$. (Здесь и ниже d – ширина «полосы»). Это много. Если резать по горизонталям, $CN = n_y/d$, а если по диагоналям $i_1 + i_2 = const$, то $CN = (n_x + n_y)/d$. Уже лучше, но тогда очень медленными будут «разгон» и «торможение», и загрузка ядер будет сильно неполной в начале и конце прогона. Оптимальным будет комбинированное разрезание по красным зигзагам, показанным на рис. 4, для которого $CN = K + 2n_y/d$, где d – размер «зуба», K – число зигзагов (или число ядер).

При традиционном параллельном программировании (на OpenMP или MPI) построить такое разрезание было бы очень непросто (особенно в 3-мерном случае), и притом его пришлось бы кодировать прямо в программе – в виде сложных границ циклов и кода формирования и передачи сообщений. В нашей вычислительной модели мы должны только задать функцию распределения, вычисляющую номер ядра по индексу j. Нужная нам функция F на функциональном языке может быть записана так:

```
fun F2D(j) = let
     val j1 = j mod n
     val j2 = j / n
     val B1 = (i1+i2) / d
     val B2 = i2 / d
in
    (K*d*(B1–B2)/nx) mod K
```

Сначала переходим к двумерному индексу ⟨j1,j2⟩, затем к «блочным» координатам ⟨B1,B2⟩. Далее проекция блока на горизонтальную ось (B1–B2) нормируется к интервалу [0..K]. Здесь K – количество ядер, nx –размер сетки по горизонтали. Заключительное mod K нужно, чтобы подклеить неполные блоки слева к неполным блокам справа.

В трехмерном случае аналогичная функция будет немного сложнее:

```
fun F3D(j) = let
     val j1 = j mod nx
     val j2 = (j / nx) mod ny
     val j3 = j / (nx * ny)
     val B1 = (i1+i2+2*i3) / d
     val B2 = (i2+i3) / d
     val B3 = i3 / d
  in
    ( (K1*d*(B1–B2-B3)/nx) mod K1,
      (K2*d*(B2-B3)/ny) mod K2 )
```

Функция F3D выдает номер ядра в виде пары $\langle p_1, p_2 \rangle$, предполагая, что ядра образуют двумерную решетку. В результате пространство ячеек (с координатами $\langle j_3, j_2, j_1 \rangle$) разбивается на трехмерные блоки, имеющие форму параллелепипедов и блочные координаты $\langle B_3, B_2, B_1 \rangle$. На рис. 7 изображен блок $\langle 0,0,0 \rangle$. Блоки с $B_3 = 0$ своей нижней горизонтальной гранью лежат в горизонтальной плоскости (i2,i1), их верхняя горизонтальная грань смещена в горизонтальном направлении на вектор $\langle 0, -d, -d \rangle$.

Выходные формулы отображают блоки на ядра $\langle k_1, k_2 \rangle$ двумерной решетки. При этом нижняя (горизонтальная) плоскость 0,j2,j1 разрезается на K_1 частей по оси j_1 и K_2 частей по оси j_2, и в одну часть попадают целиком блоки, начальный угол которых проектируется (вдоль оси j_3) в эту часть. Таким образом, ячейки, выполняющиеся в одном ядре, образуют вертикальные колонны с зубчатыми гранями (в полной аналогии с двумерным случаем, изображенным на рис. 4).

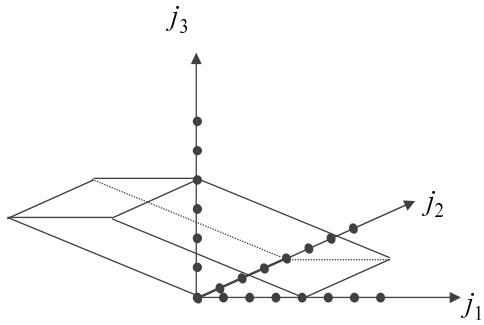

Рис. 7. Один блок $\{0,0,0\}$ разбиения трехмерной сетки на блоки ($d = 4$)

Важно отметить, что функция распределения задается вне программы, и при соблюдении определенных принципов [12] ее выбор не может нарушить правильность программы, а может только повлиять на скорость ее работы.

8. Блоки и укрупнение узлов

Смысл блоков пока состоял только в том, чтобы все узлы одного блока попали целиком в одно ядро, и чтобы близкие блоки тоже по возможности попали в то же ядро. Но есть и другой смысл: расчет всех ячеек одного блока можно задать как один узел потоковой программы, который активируется, когда для него готовы все входные аргументы. Это, во-первых, группы значений переменных, вычисленных предыдущими смежными блоками (на рис. 4 зеленым цветом выделены входные переменные для очередного блока в ядре 0) и, во-вторых, строки матрицы системы, связанные с ячейками блока. Значения, передаваемые из одного источника целесообразно объединить в один «векторный» токен. Здесь их размеры будут 3,2,5 и 36 чисел с плавающей запятой.

Такое укрупнение узлов может существенно улучшить быстродействие за счет значительного снижения накладных расходов на передачи мелких токенов и активации мелких узлов. Но за это придется заплатить резким усложнением программы узла. Мы предполагаем, что такого рода преобразования программы должны производиться автоматически [11]. Пользователю надо будет только составить несложное задание на преобразование, указав младшую часть функции распределения, характеризующую узлы, подлежащие группировке. Чтобы минимизировать передачи между ядрами, несколько смежных блочных узлов могут направляться в одно ядро – посредством старшей части той же функции распределения.

9. Распределение по времени

Строки матрицы подаются многократно для каждой итерации. Их подкачка является основным фактором, ограничивающим производительность. В ППВС можно будет задать функцию распределения по времени (этапам), согласно которой эти блоки будут по скоростному каналу заблаговременно подкачиваться в ассоциативную память ограниченного объема (аналог кэша) [13]. Быстродействие будет ограничиваться только пропускной способностью канала процессор-память.

В традиционных архитектурах аналогичный эффект можно пытаться достичь за счет сложного механизма умного предсказания будущих запросов на чтение. А в архитектуре ППВС «Буран» мы опираемся на то, что отправитель токена «знает», кому (какому узлу, с какими индексами) будут нужны передаваемые данные. И при наличии функции распределения по времени становится известным приблизительное время их потребления. Можно построить относительно несложный механизм, который будет обеспечивать своевременную подкачку токенов в рабочую ассоциативную память [13].

10. Эксперименты

Измерения проводились на параллельной блочно-регистровой модели ППВС «Буран» – ПБРМ, которая моделирует работу системы по событийному принципу с точностью до такта. «Параллельной» является моделируемая ППВС, а сама ПБРМ однопоточная, т. е. довольно медленная, и поэтому позволяет в разумное время прогонять лишь небольшие задачи.

Мы провели моделирование на двумерной задаче с сеткой 80×80 ячеек (1600 переменных). Моделирование одного прохода занимало около 2 минут. Целью моделирования было показать масштабируемость задачи с увеличением числа ядер с 2 до 8. Дальше ее нет из-за слишком малого объема работы, приходящегося на одно ядро, и растущих затрат на коммуникации. Если здесь масштабируемость есть, то можно говорить, по принципу подобия, о такой же возможной масштабируемости задачи размером 800×800 при росте количества ядер от 20 до 80.

Основным фактором и источником хорошей масштабируемости является возможность выполнять работу на фоне передач токенов между ядрами, особенно при высокой латентности (задержках) в коммуникационной сети. Поэтому мы проводим моделирование для разных значений параметра латентности сети, определяющего задержку на передачи токенов между ядрами: от 1000 до 8000 тактов. При этом пропускную способность сети, определяемую темпом передач, считаем достаточно высокой и постоянной: один токен каждые 100 тактов (на канал). Если производительность с ростом задержек не убывает, или убывает малозаметно, то это обычно называют толерантностью (нечувствительностью) к задержкам, и будет свидетельством хорошей потенциальной масштабируемости.

На рис. 8 приведены графики зависимости времени работы (в тактах) от величины задержки при передаче токенов между ядрами (тоже в тактах). Мы провели две серии экспериментов с двумя разными функциями распределения:

· Light: $P(i) = i\,1/(nx/K)$,

· Hard: $P(i) = (K * d * (B1 - B2)/nx)\%K$, где $B1 = (i1 + i2)/d$, $B2 - i\,2/d$.

Функция Light дает разрезание области на прямые вертикальные полосы, а функция Hard соответствует вертикальным зигзагам на рис. 4. Здесь K – число ядер, $d = 5$. Для каждой функции имеется несколько кривых, соответственно для 2, 4, 6 и 8 ядер.

С ростом задержки общее время растет, сначала медленно, потом быстрее. Участок медленного роста – это и есть область толерантности к задержке. Для функции Light она почти сразу исчезает (для 2х ядер от 2000, а для 4 и более – сразу от 1000). Для функции Hard толерантность заметно лучше: для 2-х ядер вплоть до задержки в 7000 тактов, для 4, 6 и 8 ядер, соответственно, до 5000, 3000 и 1000). Стартовые позиции (при нулевых задержках) у обеих функций примерно одинаковы, но затем сложная Hard заметно опережает простую Light.

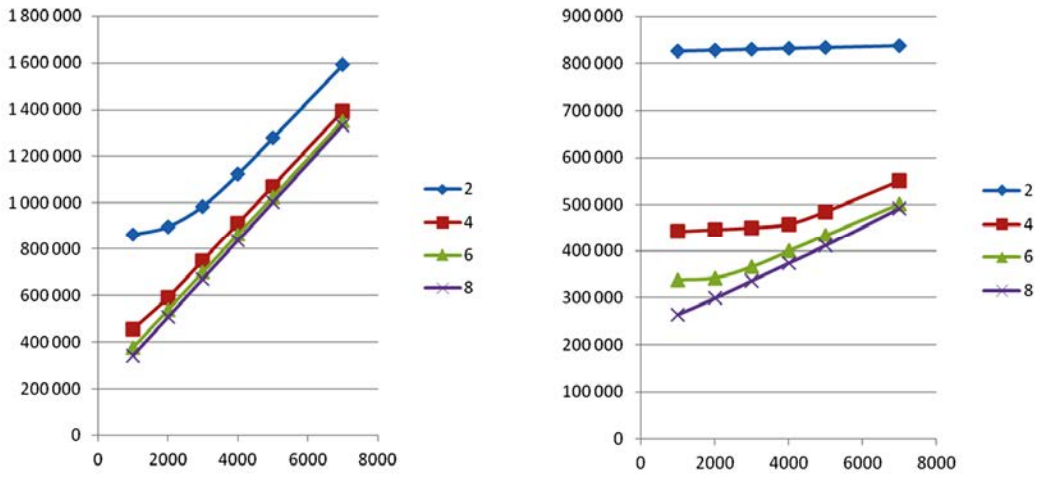

Рис. 8. Время работы в тактах в зависимости от задержки L при передаче между ядрами для различных конфигураций по числу ядер K (меньше – лучше). Варианты Light и Hard

На рисунке также можно видеть для функции Hard заметный прирост быстродействия с ростом числа ядер до 6, а при малых задержках до 8 (там, где велико расстояние между графиками). В то же время при функции Light прирост заметен только до 4 ядер. Таким образом, мы здесь видим не только пределы масштабируемости (роста быстродействия с увеличением числа ядер), но и основную причину ее раннего прекращения – латентность коммуникационной сети (зависящую от выбранной функции распределения).

11. Заключение

Рассмотрена задача распараллеливания симметричного сглаживателя Гаусса-Зейделя, применяемого в тесте HPCG. В отличие от обычного явного распараллеливания с разбиением на прямоугольные блоки и вычислениями по методу Якоби на уровне блоков (а по методу Гаусса–Зейделя только внутри блоков) мы добиваемся распараллеливания вычислений целиком по методу Гаусса-Зейделя (что ведет к лучшей сходимости метода) просто за счет выполнения на принципах потока данных. Для улучшения масштабируемости мы применяем разбиение области на косые параллелограммы (параллелепипеды). Такая декомпозиция слишком сложна для традиционного программирования средствами MPI. Но в потоковой модели вычислений, свойственной архитектуре разрабатываемой системе ППВС «Буран», такая нетривиальная декомпозиция, достигается просто путем написания функции распределения в несколько строк. И важно, что это дополнение не является частью основной программы и не может испортить ее корректность. Ошибки в нем могут ухудшить время работы, но не могут привести к неправильному результату.

Пока мы испытали этот подход только на потактовой модели на небольшой задаче для двумерного случая и только для одной итерации решения треугольной системы. Результаты обнадеживают. И теперь надо провести эксперименты с трехмерной сеткой.

Однако, для практического применения остается еще много нерешенных проблем. Вот главные:

Для устранения избыточных накладных расходов, связанных с мелкозернистым параллелизмом, разработать компилятор, который сможет автоматически породить укрупненные узлы, соответствующие блокам-параллелепипедам.

Доработать механизм распределения по времени, обеспечивающий высокоскоростную своевременную подкачку отложенных токенов из большой обычной памяти (на кристалле или вне его) в рабочую ассоциативную память небольшого размера.

Перейти от потактовой программной модели к реальной системе ППВС «Буран» в кремнии.

Благодарности

Авторы признательны Ю. Г. Бартеневу за обсуждение и полезные советы.

Литература

1. Dongarra J., Heroux M.A., Luszczek P. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems // The International Journal of High Performance Computing Applications. 2015. Volume: 30 issue: 1. P. 3–10.

2. Стемпковский А. Л., Левченко Н. Н., Окунев А. С, Цветков В. В. Параллельная потоковая вычислительная система – дальнейшее развитие архитектуры и структурной организации вычислительной системы с автоматическим распределением ресурсов // Информационные технологии 2008. № 10. С. 2–7.

3. Климов А. В., Левченко Н. Н., Окунев А. С., Стемпковский А. Л. Вопросы применения и реализации потоковой модели вычислений // Проблемы разработки перспективных микро- и наноэлектронных систем (МЭС). 2016. Вып. 2. С. 100–106. URL: http://www.mes-conference.ru/data /year2016/pdf/D114.pdf.

4. Chang K. Understanding and Improving the Latency of DRAM-Based Memory Systems. – PhD thesis. – Carnegie Mellon University, Pittsburgh, PA – May, 2017. URL: https://arxiv.org/pdf/1712.08304.pdf .

5. Brezina M., Hu J., Tuminaro R. Algebraic multigrid // Encyclopedia of parallel computing. David Padua, ed. – Springer, 2011. P. 23–33.

6. Ahmadi A., Khademi A., Smith M.C. A parallel Jacobi-embedded Gauss-Seidel method // IEEE Transactions on Parallel and Distributed Systems. – Vol. 32, No 2 – June, 2021. P. 1452–1464.

7. Shang Y. A distributed memory parallel Gauss-Seidel algorithm for linear algebraic systems // Computers & Mathematics with Applications. Vol. 57, Issue 8. – April, 2009. P. 1369–1376.

8. Adams M., Brezina M., Hu J., Tuminaro R. Parallel multigrid smoothing: polynomial versus Gauss-Seidel // J. Comput. Phys. – 188 (2003). P. 593–610.

9. Климов А. В., Окунев А. С. Графический потоковый метаязык для асинхронного распределенного программирования // Проблемы разработки перспективных микро- и наноэлектронных систем (МЭС). 2016. № 2. С. 151–158. URL: http://www.mes-conference.ru/data/year2016/ pdf/D149.pdf .

10. HPCG Benchmark [Electronic resource]: URL: https://www.hpcg-benchmark.org/software.

11. Klimov Ark.V. On the Possibility of Parallel Programs Synthesis from Algorithm Graph Specification // Proceedings of the 21st Conference on Scientific Services & Internet (SSI-2019), Novorossiysk-Abrau, Russia, September 23-28, 2019. – CEUR Conf. Proc. – Vol. 2543, 2020. P. 219–233. URL: http://ceur-ws.org/Vol-2543/rpaper20.pdf.

12. Климов Арк.В. Средства верификации распределения вычислений в потоковой архитектуре ППВС «Буран» // Проблемы разработки перспективных микро- и наноэлектронных систем (МЭС) – 2020. Вып. 4. С.236–243. URL: http://www.mes-conference.ru/data/year2020/pdf/D107.pdf.

13. Климов А. В., Левченко Н. Н., Окунев А. С., Стемпковский А. Л. Суперкомпьютеры, иерархия памяти и потоковая модель вычислений // Программные системы: теория и приложения: электрон. научн. журн. – 2014. Т. 5. № 1(19). С. 15–36. URL: http://psta.psiras.ru/read/ psta2014_1_15-36.pdf.

FEATURES OF HPCG BENCHMARK IMPLEMENTATION FOR THE “BURAN” PDCS

D. N. Zmejev, A. V. Klimov, A. S. Okunev, N. N. Levchenko

Institute for Design Problems in Microelectronics RAS, Zelenograd

HPCG (High Performance Conjugate Gradients) is a new benchmark for comparing supercomputer systems after LINPACK HPC. It is believed to better represent the workload properties of real scientific applications. We consider the implementation of its central part, the Gauss-Seidel smoother, on our projected parallel dataflow computing system (PDCS) "Buran". Unlike the HPCG reference code for MPI that uses a mixed Jacobi – Gauss-Seidel method, we implement the pure Gauss-Seidel method in our proposed dataflow language UPL providing automatic extraction of dataflow parallelism. Therefore, we violate the benchmark conditions by using a different computation algorithm that partially compensates high network and memory latencies.

Key words and phrases: dataflow computation model, dataflow architecture, parallel computing, high-performance computing, HPCG benchmark, conjugate gradients method, Gauss-Seidel method.