Визуализация профиля работы программ с памятью[1]

Вад.В. Воеводин

НИВЦ МГУ имени М.В. Ломоносова

Для большинства задач из любой предметной области для получения их эффективной реализации необходимо проводить анализ работы с памятью. В данной работе проводится исследование профиля доступа к памяти на примере некоторых типовых задач. На основе данного исследования с помощью визуализации проводится анализ возможных свойств рассмотренных задач, а также выделяются некоторые общие закономерности. Приводятся примеры визуализации профилей работы с памятью, иллюстрирующие указанные свойства и особенности.

1. Введение

Для многих задач эффективность выполнения программ является очень важным фактором. Чем выше эффективность выбранного решения, тем быстрее программа будет выполнена, тем меньше времени будет потрачено на решение задачи в целом. В области суперкомпьютерных вычислений многие задачи требуют часы, дни, а иногда и недели для решения, поэтому даже незначительное увеличение эффективности может привести к существенному выигрышу в скорости работы программы.

Для повышения эффективности программ на сегодня разработано множество различных средств, однако они всегда нацелены на относительно узкую область: либо на определенный класс вычислительных платформ, либо на предопределенный класс задач, либо на оптимизацию конкретных свойств программ. Для решения проблем общего характера, например, подбора наиболее эффективной платформы для выбранной задачи или эффективной реализации задачи на другой платформе, данные средства плохо подходят. Нужен более общий подход, позволяющий проводить исследование эффективности всего процесса отображения задач на архитектуру вычислительных систем.

Для изучения процесса отображения предлагается на каждом из его этапов выделять набор характеристик, которые могут влиять на эффективность или же помогать в ее исследовании [1, 2]. Все характеристики делятся на три класса: характеристики алгоритма, программы и аппаратуры. Предполагается, что во время отображения задачи и алгоритм, и платформа уже определены, поэтому их характеристики фиксированы. По сути, эти характеристики описывают те рамки, в которых программа должна работать. Для получения эффективной реализации мы можем менять характеристики программы в соответствии с характеристиками алгоритма и аппаратуры. Примерами указанных характеристик из различных классов могут служить следующие параметры:

- Характеристики алгоритма: свойства графа алгоритма [3];
- Характеристики программы: свойства текста программы (структура вхождения циклов, способ индексации массивов и т.д.), коммуникационные свойства (объем передаваемых данных, структура передач, число нитей/процессов);
- Характеристики аппаратуры: строение подсистемы памяти, топология, латентность и пропускная способность сети.

Одним из самых главных факторов, влияющих на эффективность выполнения программы, является эффективность работы с памятью. Эта проблема известна и получила название «стены памяти»: все последние годы скорость работы процессоров растет значительно быстрее скорости работы памяти. Во всех современных вычислительных системах доступ к данным является

[1] Работа выполнена при поддержке грантов РФФИ № 09-07-00168-а и № 10-07-00586-а.

дорогостоящей операцией, поэтому неэффективное использование памяти может очень сильно сказываться на производительности всей выполняемой программы. Вместе с этим, подсистема памяти устроена очень сложно, и ее оптимальное использование является непростой задачей. Кэш-памяти разных уровней, оперативная память, TLB буфер, аппаратный префетчер – каждый из этих компонентов подсистемы памяти обладает целым набором аппаратных характеристик, которые необходимо учитывать для эффективной работы с памятью.

На данный момент разработаны различные программные средства и проводятся исследования, направленные на анализ и повышение эффективности работы с памятью, например, с помощью сбора информации об эффективности использования кэш-памяти [4] или эмуляции работы программы [5]. В данной работе акцент делается на визуальном анализе структуры доступа к памяти, который во многих случаях дает интересную информацию о свойствах программы, и главное – об эффективности ее выполнения. Стоит отметить, что активно визуализация используется и при исследовании других свойств программ, например, коммуникационного профиля параллельных программ [6].

2. Профиль работы с памятью и характеристики программ

Для исследования и анализа эффективности программ важно понимать структуру их профиля работы с памятью. Введем понятие “поток обращений”, под которым будем понимать последовательность обращений к используемым в программе переменным, записанную в том порядке, как это происходило при выполнении программы. Если задействован элемент массива или другой составной переменной, то в поток попадает именно этот элемент, а не вся составная переменная. Таким образом, в поток обращений могут входить скалярные переменные, составные переменные (если в некоторой операции адресуется такая переменная целиком) или отдельные элементы составных переменных.

Если для работы с элементами составной переменной используется некоторый итератор, то для каждого конкретного значения итератора в поток обращений записывается отдельное обращение. В случае итерирования массива в потоке обращений появятся отдельные элементы данного массива. Например, если в программе есть код:

```
for (i=0; i<5; i++) A[i]=0;
```

то поток обращений к массиву `A` будет выглядеть следующим образом: `A[0]`, `A[1]`, `A[2]`, `A[3]`,`A[4]`.

Для изучения профиля работы программ с памятью можно опираться на различные подходы. Один из возможных вариантов – это использование специальных характеристик программы, описывающих локальность используемых данных.

В работе [2] для описания пространственной и временно́й локальности данных программы вводится набор характеристик. Первые две характеристики описывают пространственную локальность, которая отражает среднее расстояние между несколькими последовательными обращениями к памяти. Характеристика *L* показывает среднее расстояние по памяти между всеми соседними элементами потока обращений. Это дает оценку, насколько близко в памяти расположен следующий элемент данных по отношению к текущему. Характеристика *SEQ_L* отражает среднюю длину последовательных элементов потока, являющихся непосредственными соседями в памяти. Данная характеристика позволяет оценить самый хороший случай для пространственной локальности, когда используемые данные следуют в памяти подряд, что дает хорошее использование кэш-памяти.

Для описания свойств временно́й локальности выделяются три характеристики. Временна́я локальность данных программы показывает среднее число обращений по одному адресу в памяти за время исполнения всей программы. Характеристика α равна среднему расстоянию (в терминах потока обращений) до следующего обращения к той же переменной, что показывает интенсивность использования данных. Степень влияния этой характеристики, а также отдельных переменных на эффективность работы программы в целом, позволяет оценить характеристика N_α, которая равна среднему числу обращений к переменным (элементам потока). Чем

больше обращений произведено к некоторой переменной, тем важнее с точки зрения эффективности работы с памятью интенсивность обращений к данной переменной. Степень повторного использования переменных программы показывает характеристика *P*, значение которой изменяется на отрезке [0, 1]. Характеристика *P* принимает крайние значения в следующих случаях: если ко всем переменным программы обращение происходит только один раз, то *P*=0, а если ко всем переменным обращение происходит как минимум дважды, то *P*=1.

Более подробное описание и примеры использования данных характеристик приведены в работе [2].

3. Визуализация профиля работы с памятью

Для изучения качественных характеристик профиля работы программ с памятью хорошие результаты дает его визуализация. Конечно, визуализация не дает возможности получить точные оценки или провести аккуратный детальный анализ, однако она позволяет понять общее строение профиля, как в целом устроена программа, что во многих случаях может дать подсказку пользователю о дальнейших действиях.

В данной работе мы будем рассматривать профили работы с памятью для некоторых типовых алгоритмических структур [1, 2]. Для каждого массива строятся отдельные профили. Все профили на рисунках устроены одинаково: по вертикали расположены линеаризованные индексы элементов массива (т.е. смещение элемента относительно начала массива), по горизонтали – номер обращения в потоке (чем больше этот номер, тем позже было произведено данное обращение в программе).

Стоит сразу отметить, что картинка никак не отражает время, затрачиваемое на обращение к различным компонентам подсистемы памяти (например, кэш-памяти или ОП), которое может сильно отличаться для разных обращений. Данное представление является машинно-независимым, и не всегда позволяет сделать точные выводы о работе реальной программы.

Также, поскольку число обращений в потоке часто весьма велико, то визуализация всего профиля может быть обманчива, и для точного анализа, вообще говоря, необходимо рассматривать отдельные, небольшие фрагменты профиля. Возьмем, например, профиль работы с памятью для вектора в задаче умножения разреженной матрицы на плотный вектор (рис. 1а). На первый взгляд, кажется, что данный профиль отражает полностью случайный характер обращения к элементам вектора, однако при увеличении масштаба видно, что обращения к вектору имеют закономерность (рис. 1б).

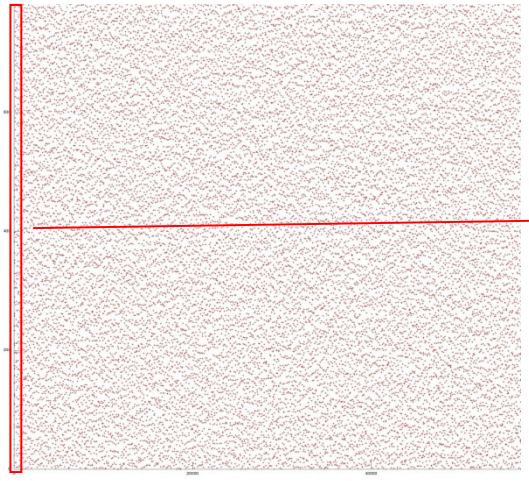

Рис. 1а. Умножение разреженной матрицы на вектор, общий профиль, > 100.000 обращений в память

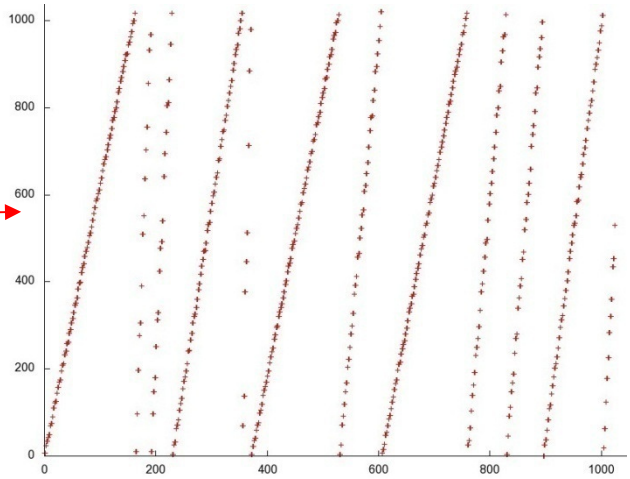

Рис. 1б. Умножение разреженной матрицы на вектор, увеличенный фрагмент, 1024 обращений в память

Несмотря на то, что при рассмотрении общего профиля точный анализ недоступен, некоторые выводы касательно работы с памятью все же можно делать. Например, график на рис. 2а

позволяет предположить, что для любого элемента массива все обращения к нему расположены близко друг к другу, и при этом элементы массива перебираются последовательно, значит, массив обладает достаточно хорошей пространственной локальностью. На рис. 2б видно, что правая часть профиля портит общую локальность, поскольку обращения происходят ко всем элементам массива, причем порядок обращений похож на случайный. Вместе с этим, число обра-

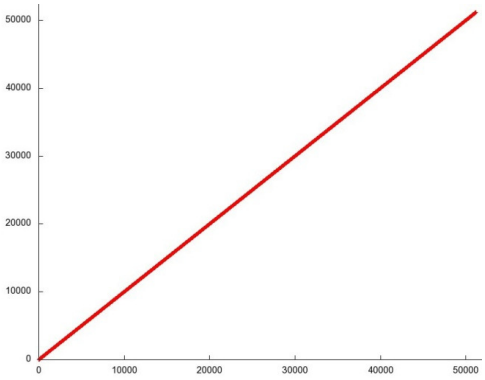

Рис. 2а. Пример профиля с хорошей пространственной локальностью

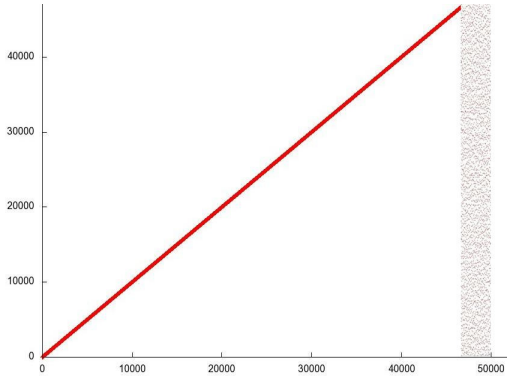

Рис. 2б. Измененный профиль, пространственная локальность ухудшается

щений в этой правой части значительно меньше, чем во всей остальной части, поэтому его влияние на общий профиль будет не так велико.

Рассмотрим более интересный пример – блочное перемножение плотных квадратных матриц (`N` – размерность матриц, `BLOCK` – размер блока):

```
for(i1=0; i1<N; i1+=BLOCK)
   for(j1=0; j1<N; j1+=BLOCK)
      for(k1=0; k1<N; k1+=BLOCK)
         for (i=i1; i<i1+BLOCK; i++)
            for (j=j1; j<j1+BLOCK; j++)
               for(k=k1; k<k1+BLOCK; k++)  {
                  C[i][j]+=A[i][k]*B[k][j];
               }
```

В случае при `N=128, BLOCK=16` профили работы с памятью для массивов `A`, `B` и `C` выглядят следующим образом (рис. 3а,3б,3в).

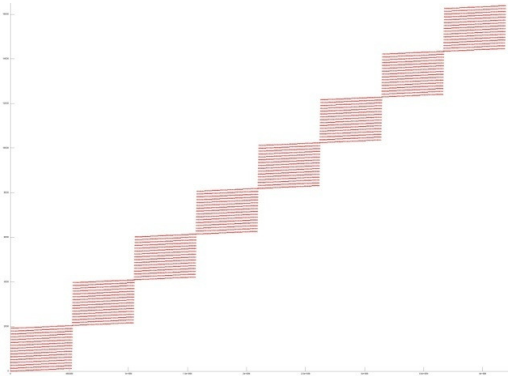

Рис. 3а. Профиль обращений в память к массиву `A`

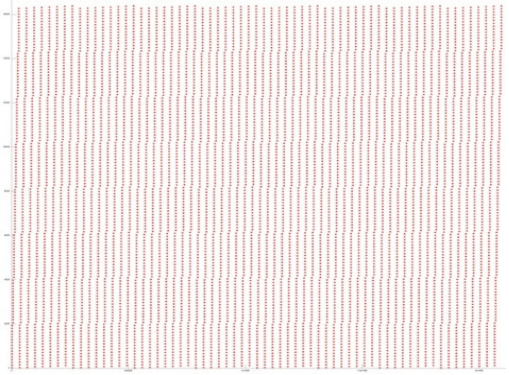

Рис. 3б. Профиль обращений в память к массиву `B`

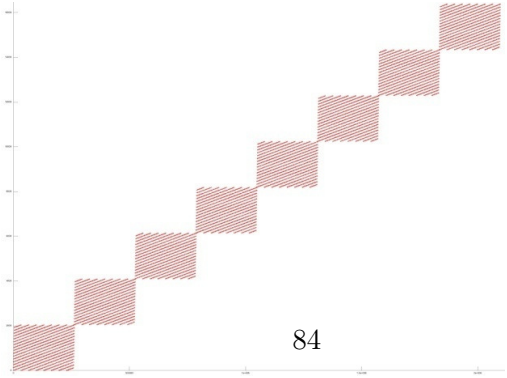

Рис. 3в. Профиль обращений в память к массиву `C`

В данном случае индекс из двумерного линеаризуется в одномерный, с учетом того, что матрица хранится по строкам. По вертикали, как обычно, указан индекс элемента массива, к которому происходит обращение, а по горизонтали – порядковый номер обращения. Данные графики показывают, что профили массивов `A` и `C` похожи – элементы массивов образуют 8 (128/16) блоков, и сначала происходят обращения только к первому блоку, затем – только ко второму и т.д. Это указывает на хорошую локальность обращений, поскольку все обращения к каждому элементу расположены близко друг к другу. В случае массива `B` ситуация иная: из профиля на рис.3б видно, что каждый элемент используется на протяжении всей программы и через большие промежутки времени, а значит, локальность в этом случае, скорее всего, хуже, чем в случае массивов `A` и `C`.

Рассмотрим для более точного анализа небольшие участки профиля. Более подробное изучение профилей массивов `A` и `C` на рис. 4а и рис. 4б позволяет заметить, что локальность данных лучше в случае массива `C`.

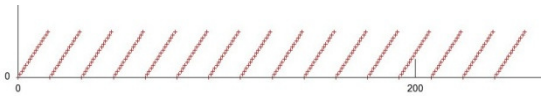

Рис. 4а. Профиль обращений к массиву `A` при выполнении двух внутренних циклов

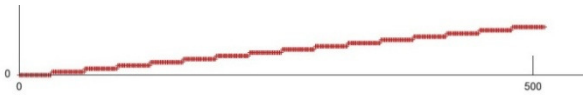

Рис. 4б. Профиль обращений к массиву `C` при выполнении двух внутренних циклов

Сравнивая профили на данных рисунках, полученных при выполнения двух внутренних циклов фрагмента перемножения матриц, видно, что все обращения к элементу массива `C` идут подряд, в то время как обращения к элементам массива `A` происходят с некоторым шагом, что приводит к менее эффективной работе с памятью.

Конечно же, все качественные заключения, полученные при изучении профилей работы с памятью, находят отражение и в соответствующих характеристиках программ. Рассмотрим значения описанных выше характеристик локальности использования данных, и сравним полученные результаты (табл. 1).

Таблица 1. Значения характеристик для задачи блочного перемножения матриц, `N=128`, `BLOCK=16`

Массив	L	SEQ_L	α	N_α	P
A	0.2706	16.0009	0.0006	128	1
B	0.0041	1	0.00007	128	1
C	0.681	512.438	0.0044	256	1

Сейчас не нужно внимательно анализировать абсолютные значения характеристик. Важно, что для каждой из характеристик верно следующее утверждение: чем больше значение характеристики, тем лучше (выше) локальность обращений к данным программы. Иными словами,

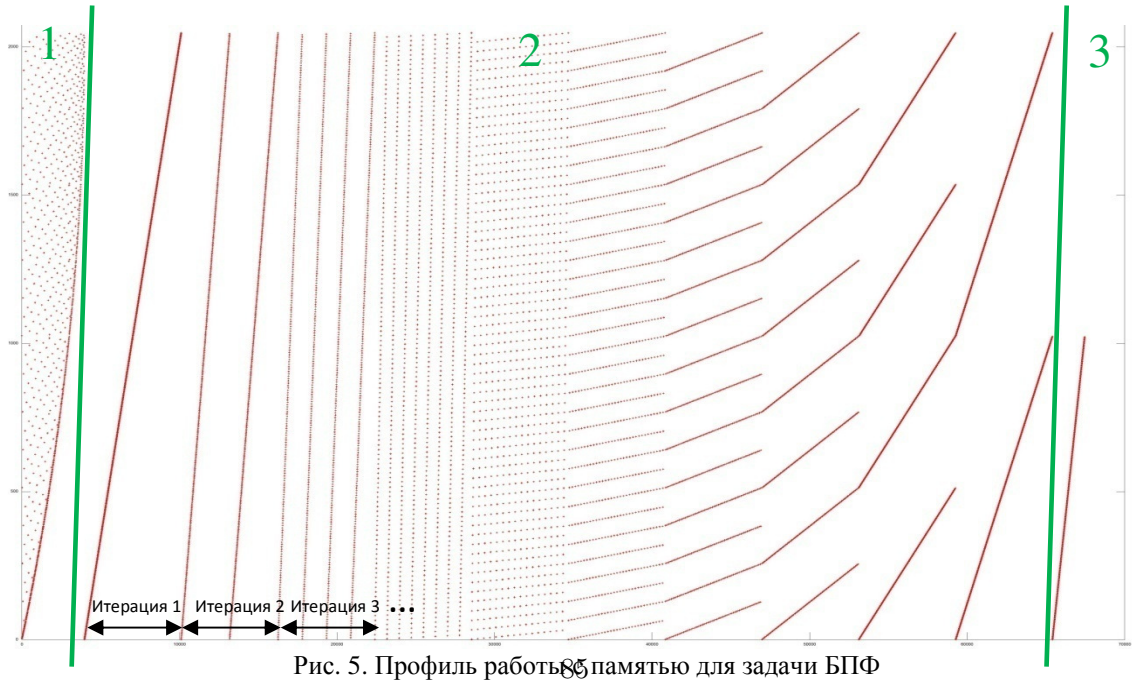

Рис. 5. Профиль работы с памятью для задачи БПФ

выводы, сделанные по профилям на рисунках, подтверждаются значениями характеристик.

Следует еще раз подчеркнуть возможную обманчивость первого восприятия всего профиля целиком. С этой точки зрения, изначальное предположение о том, что, согласно рисункам, работа с массивом `B` организована менее эффективно, чем работа с массивами `A` и `C`, могло оказаться и неверным. Например, программа могла быть устроена так, что к элементам массива `B` происходят последовательные обращения на протяжении многих итераций, в то время, как в каждом блоке обращений к массиву `A` профиль работы с памятью мог носить случайный характер. В такой ситуации локальность обращений к массиву `B` оказалась бы лучше, чем к массиву `A`. Это еще раз подчеркивает, что общая картина профиля не всегда адекватно отражает реальные свойства программы.

Нужно отметить и тот факт, что далеко не всегда обращения к памяти имеют “линейный” характер, как могло показаться из приведенных выше рисунков. Рассмотрим профиль работы с памятью для другой типовой алгоритмической структуры – быстрого преобразования Фурье (рис. 5). Для решения использовался простой нерекурсивный алгоритм. Несмотря на достаточно сложную структуру обращений к памяти, на данном графике легко можно увидеть 3 этапа алгоритма. На этапе 1 происходит начальное преобразование входного массива для последующего расчета. На этапе 2 выполняются основные вычисления, причем отчетливо видно увеличение шага по элементам массива в 2 раза на каждой последующей итерации. Этап 3 соответствует записи результатов в выходной массив.

В заключение рассмотрим еще один интересный пример: умножение разреженной матрицы размером `NxN` на вектор с диагональной схемой хранения. В такой схеме хранения матрицы `i`–й диагональю называется совокупность ненулевых элементов, расположенных на `i`-м слева месте в каждой строке. То есть нулевая диагональ – это совокупность первых в строке ненулевых элементов. В случае если в строке нет `i`-го элемента, в `i`-й диагонали не будет присутствовать ни одного элемента из данной строки. Данная форма хранения матриц используется в тех случаях, когда нужно работать с длинными векторами, а диагональная схема, в данном смысле, имеет большое преимущество перед популярными столбцовой или строчной формами. Алгоритм выглядит следующим образом:

```
for (j = 0; j <= num_diag-1; j++)
   for (i = ip[j]; i <= ip[j+1]-1; i++)
      b[ia[i]] = b[ia[i]]+a[i]*x[ic[i]];
```

В массиве `a` записана матрица по диагоналям (всего в матрице `num_diag` диагоналей), `x` – вектор, `b` – результирующий вектор. Массив `ip` хранит номера строк, отмечающих начало и конец каждой диагонали, а массивы `ic` и `ia` содержат соответственно номера столбцов и строк элементов матрицы.

Рассмотрим профиль обращений к вектору `x`. Данный профиль зависит от того, каким образом формируется разреженная матрица. Для начала рассмотрим такую стратегию выбора не-

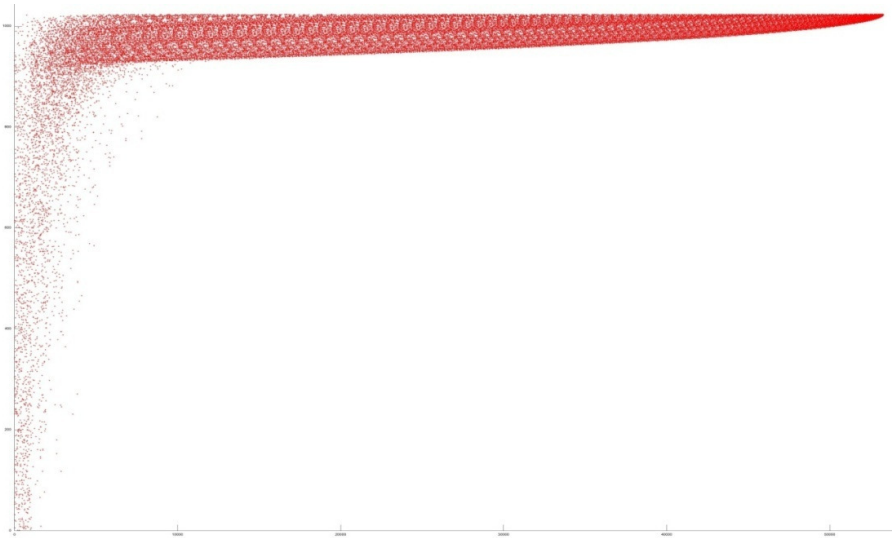

Рис. 6. Профиль работы с памятью для задачи умножения разреженной матрицы на вектор, первый вариант заполнения матрицы

нулевых элементов – для каждой строки случайным образом выбирается число ненулевых элементов в ней. Для каждой строки случайным образом выбирается первый элемент в диапазоне от `0` до `N-K-1`, где `K` – число ненулевых элементов в данной строке; затем выбирается второй элемент в диапазоне от `A1+1` до `N-K-2`, где `A1` – номер столбца первого ненулевого элемента; и т.д. В результате образуется следующий профиль (рис. 6).

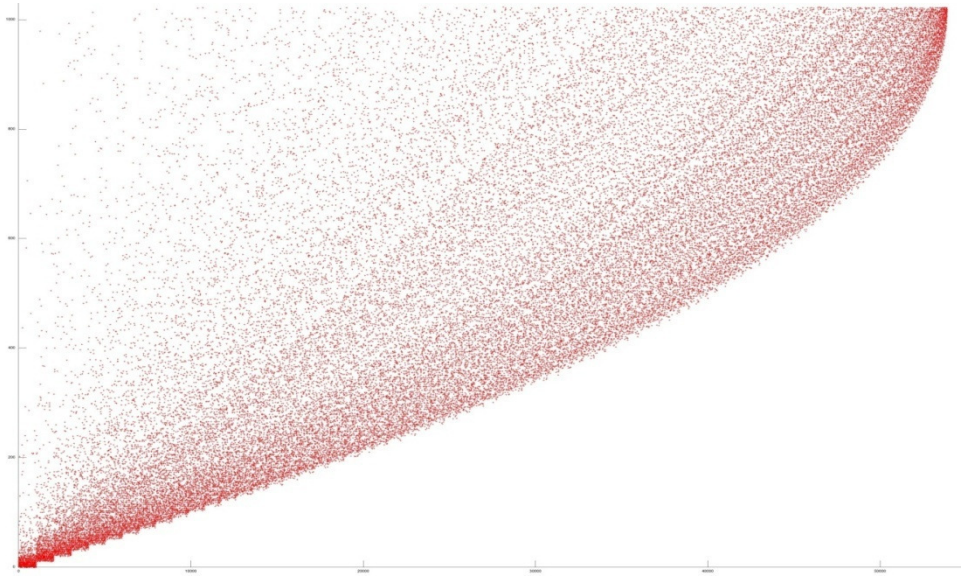

Рис. 7. Профиль работы с памятью для задачи умножения разреженной матрицы на вектор, второй вариант заполнения матрицы

Данный профиль позволяет увидеть, что большинство обращений происходит к элементам, расположенным в конце массива `x`. Следовательно, выбранный способ формирования матрицы приводит к тому, что большинство ненулевых элементов скапливается в последних столбцах матрицы. В случае если желательно получить более равномерное распределение ненулевых элементов в матрице, можно использовать другой способ их выбора: разделим строку на `K` равных частей, и в каждой части случайным образом выберем один элемент. В этом случае обращения к элементам массива `x` будут более равномерными, что и отражает профиль, показанный на рис. 7.

4. Заключение

Все профили программ, приведенных в данной работе, построены с помощью простого подхода: сначала фиксируется поток обращений анализируемой программы к памяти, который затем визуализируется с помощью системы gnuplot. Формирование потока обращений делается путем сохранения после каждой операции с массивом индекса задействованного элемента.

Безусловно, для детального изучения профиля работы программ с памятью необходим полноценный инструментарий, объединяющий глубокий аналитический анализ с развитыми возможностями по визуализации. Данный инструмент должен позволять выбирать для анализа только необходимые переменные и фрагменты кода, легко управлять объемом собираемых данных, иметь гибкие средства фильтрации, агрегирования, позволять менять масштаб и выбирать область для детального исследования. Разработка подобного инструмента стоит в плане работ на будущее, но все текущие исследования выполняются более простыми средствами.

Приведенное исследование показывает, что для обеспечения эффективной работы программ стоит изучать и анализировать профиль работы с памятью. Данный анализ можно проводить различными способами, одним из которых является исследование результатов визуализации профиля. Несмотря на то, что данный способ не позволяет проводить точный анализ, он помогает понять на общем уровне, каким образом происходит работа с памятью в программе, а также сделать некоторые выводы касательно общей эффективности использования памяти.

Список литературы

1. Воеводин Вад.В. Характеристики типовых алгоритмических структур// ПАВТ'2010, Уфа, принято к публикации в журнал ВАК «Вестник ННГУ». Март-апрель 2011, выпуск №2.
2. Воеводин Вад.В. Характеристики работы с памятью и эффективность работы программ // Высокопроизводительные параллельные вычисления на кластерных системах (HPC-2010): материалы X Международной конференции. Пермь, 2010. С. 114-118.
3. Воеводин В.В. Вычислительная математика и структура алгоритмов // М: Изд-во МГУ, 2006. 112 с.
4. Документация по программному инструменту ThreadSpotter компании Acumem. URL: http://acumem.com. (дата обращения 9.02.2011)
5. Корж А.А. Оценочное тестирование современных систем и постановка задачи разработки суперкомпьютеров с перспективной архитектурой // доклад на семинаре «Научный семинар Parallel.ru», НИВЦ МГУ, 2008. URL: http://agora.guru.ru/parallel/files/MemoryWall-1_22-12-08.zip (дата обращения 9.02.2011)
6. Документация по программному средству Vampir. URL: http://www.vampir.eu/ (дата обращения 9.02.2011)