ISSN 2079-3316 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ № 3(17), 2013

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

Суперкомпьютеры, иерархия памяти и потоковая модель вычислений

Аннотация. Современные и будущие суперкомпьютеры не могут не быть иерархическими: это и иерархически организованная межпроцессорная сеть, и иерархия памяти внутри одного процессора или их группы. Поэтому среди факторов неэффективности на первый план выходят затраты на перемещения данных, и соответственно растет сложность построения хорошо оптимизированных по этому фактору программ. Возникающие трудности в значительной мере являются следствием традиционной парадигмы программирования, восходящей к фон Нейману. И хотя за нее имеются такие серьезные аргументы как сложившиеся навыки и накопленный софтвер, все же полезно хотя бы в теории понимать альтернативы. Мы объясняем природу проблем фон-неймановского программирования тем, что в ней осуществляется *парадигма сбора*, и предлагаем перейти к использованию модели вычислений с управлением потоком данных, которой свойственна работа в *парадигме раздачи*, и в которой в силу этого проблемы оптимизации перемещения данных решаются значительно проще.

Ключевые слова и фразы: Суперкомпьютер, иерархия памяти, подкачка данных, модель вычислений с управлением потоком данных, парадигма сбора, парадигма раздачи, планирование вычислений.

Введение

Реальная производительность суперкомпьютеров уже сейчас определяется не столько его вычислительной мощностью, выражаемой петафлопсами, сколько мощностью механизмов доставки данных к вычислителям. Соответственно, хорошая программа должна

обеспечивать управление перемещением данных так, чтобы нужные данные доставлялись с наименьшими затратами в нужное время.

В многопроцессорной вычислительной системе есть два вида доставки данных: между процессорными узлами и между процессором и памятью. А в больших и очень больших вычислительных системах имеет место иерархия как в структуре соединений, так и в организации памяти. Для соединений иерархия подразумевает, что связь между элементами внутри частей (некоторого уровня) лучше, дешевле (в смысле затрат времени, энергии и других ресурсов), чем между элементами из разных частей. Для памяти иерархия мыслится как несколько уровней памяти таких, что каждый следующий уровень имеет на порядки больший объем, и большее время доставки. Обычно имеются следующие уровни памяти: регистры, кэши L1, L2, L3, основная память, память на SSD и/или дисковая память, память на лентах. С ростом уровня растет не только задержка (обращения к памяти этого уровня), но также снижается общая пропускная способность канала между этим и предыдущим уровнями, которую для удобства будем относить к одному вычислительному узлу. И если для определенной задачи эта пропускная способность на каком-то из уровней оказывается недостаточной, то именно ею будет ограничиваться производительность, а вовсе не обязательно петафлопсами.

Уровень нагрузки на канал между уровнями памяти обычно характеризует качество программы, хотя часто можно установить оценку снизу для самой решаемой задачи[1] – и тогда можно говорить об оптимальности программы относительно того или иного уровня памяти. Теоретическая оценка трафика (в GB) для задачи с вычислительным графом $G(N)$ между внутренней памятью размера S и неограниченной внешней памятью обычно имеет вид

$L_1(G(N),S) = \mathrm{O}(N^p/S^q)$

[1] Считаем, что задача определяется своим вычислительным графом с точностью до порядка выполнения редукций.

где N – характерный размер задачи, $p,q \geq 0$, причем обычно $q<1$. Теперь, если известна пропускная способность канала B (в GB/s), то можно оценить снизу время работы как L_1/B.

Если используется K одинаковых процессоров с памятью S в каждом, то для суммарного трафика действует та же формула для малых S. А с выходом на уровень большой памяти работает другая формула (при равномерной разбивке задачи на все K процессоров):

$L_2(G(N),K) = \mathrm{O}(N^r K^s)$

Формулы такого вида обычно получаются, когда задача устроена однородно в том смысле, что ее подзадачи имеют такой же граф, но с меньшим N. Показатели p и r обычно таковы, что N^p выражает объем вычислений, а N^r – суммарный размер исходных данных и результата. Например, для умножения матриц будет:

$$L_1(G_{\mathrm{MM}}(N),S) \approx 2N^3/\sqrt{S}$$
$$L_2(G_{\mathrm{MM}}(N),K) \approx 3N^2K^{1/3}$$

При наличии таких ограничений можно говорить о пределах эффективности решения задачи на реальной системе, учитывая характеристики ее памяти на разных уровнях. В данном случае речь идет о пропускной способности каналов между уровнями. С учетом вида зависимости L_1 от S для данной задачи можно указать тот уровень памяти, для которого программу следует оптимизировать в первую очередь.

Но даже если требования по пропускной способности удовлетворены наилучшим образом, надо еще озаботиться проблемой ожиданий (простоев) в связи с задержкой доступа к памяти. Для этого управление вычислительным процессом должно обеспечивать заблаговременное перемещение (подкачку) данных, и это представляет собой отдельную проблему, которую приходится решать либо программисту, либо компилятору, либо аппаратуре, либо всем вместе. Аппаратный механизм суперскалярности обеспечивает предвыборку в пределах около 10 тактов, длинная кэш-строка, прогноз последовательности адресов привносят упреждение в сотни тактов, но большее возможно уже только усилиями со стороны компиляторов и/или программистов, причем очень немалыми, особенно в

контексте многих уровней кэш-памяти с различными характеристиками.

По нашему мнению, для успешного решения указанных проблем без перекладывания их на плечи программиста, следует изменить подход к программированию, перейдя к потоковой модели вычислений. Кстати, в этом мы не одиноки, см. например, [1] и [2]. Ниже мы более подробно анализируем природу трудностей и даем свою мотивировку для такого перехода, предлагая свою версию потоковой модели вычислений в парадигме раздачи.

1. Пример: умножение матриц

Рассмотрим проблему оптимизации перемещений на примере задачи умножения плотных матриц. В работе [3] дается оценка снизу для объема межпроцессорных обменов (включая обмены с хостом) C в зависимости от размера задачи N и числа процессоров K: $C = \mathrm{O}(N^2 \cdot \sqrt[3]{K})$. Оптимальный трафик обеспечивает следующая схема: на каждом процессоре выполняется умножение пары блоков порядка $N/\sqrt[3]{K}$, при этом каждый отдельный элемент дублируется в $\sqrt[3]{K}$ процессоров. Однако здесь пока не учитываются ограничения по памяти (считается, что их нет). Если же принять, что в каждом процессоре имеется память объема S, то оценка снизу на внешний трафик процессоров будет $\mathrm{O}(N^3/\sqrt{S})$, то есть в $\mathrm{N}/(\sqrt[3]{\mathrm{K}}\cdot\sqrt{\mathrm{S}})$ раз больше (предполагая, что этот коэффициент больше 1, что равнозначно тому, что памяти S не хватает для размещения одного блока целиком). Аналогичную оценку можно найти в [4]. Мы видим, что с увеличением S трафик убывает обратно пропорционально $\sqrt{\mathrm{S}}$. Например, если размер кэшей L1 и L2, соответственно, 10K и 1M, то (минимально возможный) внешний трафик для L1 будет в 10 раз больше чем для L2. Какой из двух кэшей будет реально «тормозить», зависит от соотношения пропускных способностей их внешних каналов. Но чтобы не возникало напрасных «торможений», программа должна быть оптимизирована по трафику относительно всех кэшей. А для этого умножение должно выполняться блоками, помещающимися в кэш, и хорошо его заполняющими, для

всех уровней кэша. Насколько нам известно, автоматически данная проблема не решается, и стало быть программист должен предоставить блочную программу с правильно выбранными размерами блоков.

2. Парадигмы сбора и раздачи

Обратимся к проблеме задержек. Почему плохо масштабируются механизмы автоматической подкачки, типа суперскаляра? Причина кроется в самой парадигме программирования, восходящей к фон Нейману: программа организует вычисления, запрашивая данные из памяти и помещая результаты в память. При этом обычно программа запрашивает данные ровно тогда, когда они уже нужны для вычислений, и это естественно для данной парадигмы. По этой причине программы обычно чувствительны к задержкам доступа к памяти – «стена памяти». Для ее преодоления используются различные ухищрения:

- многоуровневая кэш память;
- спекулятивное выполнение;
- эмпирическое предсказание;
- поддержка большого количества (сотни) тредов, ожидающих отклика из памяти.

Мы видим корень проблемы фон-неймановского программирования в том, что в нем осуществляется *парадигма сбора*: только потребитель данных знает, какие данные ему нужны и где их взять, и сам их запрашивает, указывая имя переменной или массива с индексами (Рис. 1, слева). В этих условиях аппаратуре трудно предвидеть, какие данные будут нужны. Для более качественной стратегии перемещения данных была бы более продуктивной противоположная *парадигма раздачи*, когда производитель каждого нового значения знает, кому оно потребуется, и обеспечивает рассылку по нужным адресам. А получателю тогда остается просто «пассивно дожидаться» прихода данных, ничего не зная об их источнике (Рис. 1, справа).

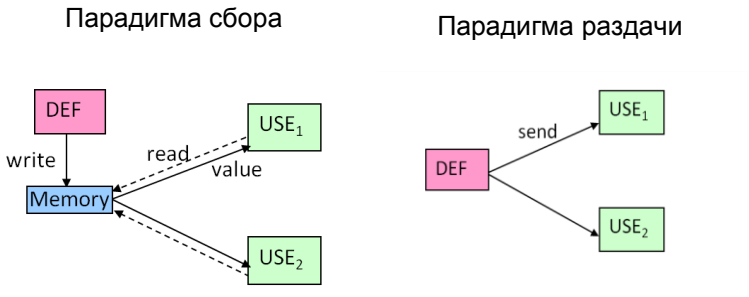

Рис. 1. Парадигмы сбора и раздачи

Парадигма раздачи является и более экономной в плане числа обменов сообщениями: одно сообщение на каждое использование, тогда как в парадигме сбора одно сообщение на запись и еще по два на каждое использование (запрос и ответ). Вдобавок парадигма раздачи имеет еще одно важно преимущество, о котором пойдет речь ниже.

Надо сказать, что парадигма раздачи в какой-то мере воплощается в MPI, где данные упаковываются в сообщения и посылаются в другие процессы. Причем получатель может знать, кто ему посылает, а может и не знать. При этом внутри процесса сохраняется программирование в парадигме сбора. Поскольку обычно MPI-программа может выполняться с любым числом MPI-процессов, начиная с 1, а каждый MPI-процесс это просто фон-неймановская программа, то в ней должны фактически присутствовать оба варианта алгоритма, написанные как в парадигме сбора, так и в парадигме раздачи. А это ведет к дополнительному усложнению программирования.

3. Использование функции распределения по времени

При программировании в парадигме сбора в момент появления новой величины и размещения ее в памяти исполнителю ничего не

известно о времени будущего использования (Рис. 2, вверху). И хотя программист вполне может понимать, где потребуется данное значение, но у него нет способа выразить это понимание в языке. В парадигме раздачи, напротив, программист вынужден при создании величины указать, кем будет использовано значение, задав в какой-либо форме адреса потребителей. Возможно, это осложняет разработку программы, но зато может помочь исполнителю более эффективно справляться с задачей оптимизации подкачек, поскольку в момент создания элемента данных уже известно кое-что о его будущем использовании (Рис. 2, внизу).

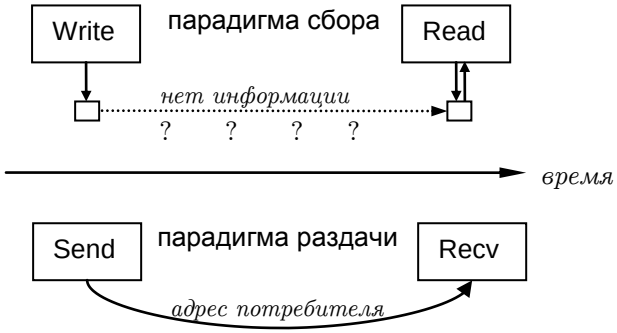

Рис. 2. Доступность информации о будущем использовании в парадигмах сбора и раздачи

Но знать адрес потребителя еще недостаточно, чтобы оценить время будущего использования. Чтобы дать исполнителю более точную информацию о времени, мы добавляем в программу (руками или компилятором) особую функцию – функцию распределения вычислений по времени (в дополнение к функции распределения по пространству, т.е. процессорным узлам). Имея на входе целевой адрес потребителя созданной величины, она вырабатывает условное время использования. Будем считать, что значения данной функции линейно упорядочены: чем больше значение, тем позже использование. Например, часто будет удобно задавать условное время в виде кортежей (tuples) с лексикографическим порядком.

Теперь в аппаратуре появляется возможность «на лету» сортировать создаваемые значения по их «срочности» и менее «срочные» значения сразу отправлять в более «долгий ящик» и наоборот. Каждому уровню «долготы ящика» приписывается свой «порог отсечения». Самый низкий (внутренний) уровень содержит данные, которые нужны немедленно. Это «активная зона». У каждого пришедшего (или здесь созданного) элемента данных его условное время последовательно сравнивается с порогами разных уровней, начиная с самой внутренней, после чего записывается в память подходящего уровня. Если в какой-то момент порог повышается, то данные, имеющие время использования ниже нового порога, из верхнего уровня спускаются вниз. Для этого на каждом уровне данные должны храниться так, чтобы всегда было легко отобрать подходящее их количество с наименьшим временем.

В системе имеется планировщик, который управляет текущими значениями порогов отсечения. В качестве входных данных планировщик может использовать различные текущие характеристики вычислительного процесса. В Табл. 1 приведены некоторые возможные характеристики и способы их использования для управления порогами.

Табл. 1. Текущие характеристики и их воздействие на пороги

	Повышение порога	Понижение порога
Заполненность (% заполнения памяти уровня)	низкая (менее 50%)	высокая (более 80%)
Вычислительная активность (только для самого низкого уровня)	Низкая	высокая
Интервал (разность между порогом и текущим минимумом времени в активной зоне)	меньше установленного	больше установленного

В зависимости от задачи, может использоваться одна из этих характеристик или все вместе в комбинации. Здесь мы обсуждаем лишь общие принципы. Конкретные стратегии работы планиров-

щика еще предстоит исследовать. Некоторые успешные эксперименты для двухуровневой памяти авторами были осуществлены в рамках системы, о которой пойдет речь ниже [5][6][7].

Заметим, что память не является прямо адресуемой, как и в обычных кэшах. Каждый элемент данных хранится вместе с адресом назначения и условным временем. Освобождение памяти при программировании в парадигме раздачи легко автоматизируется: данные стираются сразу после последнего использования.

4. Потоковая модель вычислений

Теперь хорошая новость: модель вычислений с указанными свойствами существует – это потоковая модель вычислений в парадигме раздачи (ППР) [5]. В реализующем ее языке программирования DFL пишутся узлы, состоящие из заголовка с именем, списком входов, списком атрибутов ключа (контекста) и наконец, код (программный). Активация узла происходит, когда на все входы одного узла с определенными именем и контекстом придут элементы данных – токены. Это принцип управления потоком данных. При активации выполняется код узла, в котором вычисляются новые величины (исключительно на основе значений входов и атрибутов ключа) и посылаются специальными операторами на другие узлы (входы узлов), причем атрибуты ключа адресата вычисляются прямо в этом же коде. А это и означает, что работа производится в парадигме раздачи. Память в этой модели служит для временного хранения токенов, пока они не заполнят все входы своего целевого узла. Тогда и выполняется активация, при которой обычно участвующие в ней токены из памяти удаляются (если только у них нет кратности, которая пока не исчерпана).

Помимо хранения, в памяти также происходит сопоставление (сравнение) ключей токенов, необходимое для формирования групп токенов, направленных на один и тот же узел (с одинаковыми именем и атрибутами ключа). Это требует наличия в ней дорогой ассоциативности (вспомним, что потребление энергии кэша прямо пропорционально степени его ассоциативности, то есть количеству одновременно выполняемых сравнений). Но собственно сравнения с

целью отождествления производятся в небольшой по объему ассоциативной памяти сопоставления (ПС) самого внутреннего уровня, да и в ней часть поиска проводится адресно по хеш-коду. В остальных уровнях производятся только последовательные сравнения значений условных времен с порогами отсечения.

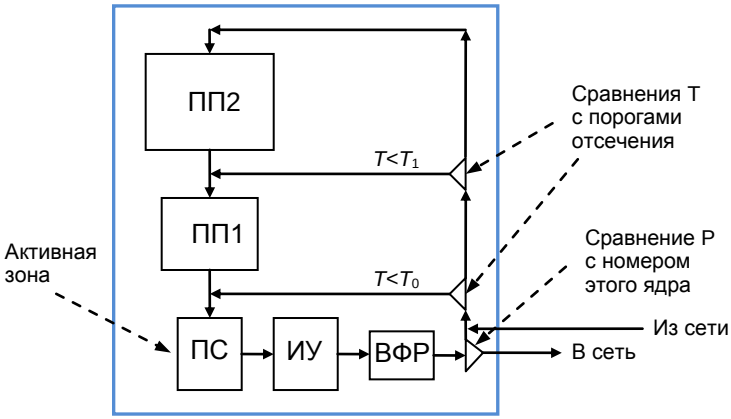

Рис. 3. Схема процессорного ядра. Пути перемещения данных.

На Рис. 3 показаны пути перемещения информации в вычислительном ядре. В исполнительном устройстве ИУ выполняется программа некоторого узла. Порождаемые при этом токены поступают в вычислитель функций распределения (ВФР), где на основе целевого адреса вычисляются функции распределения по пространству (месту) P и времени T. Если величина P отлична от номера данного ядра, токен отправляется в коммуникационную сеть. Иначе он продолжает путь внутри ядра. Далее величина T, которую будем также называть *номером этапа*, сравнивается последовательно с порогами отсечения T_0, T_1 и т.д. всех уровней памяти. Если $T < T_0$, токен отправляется сразу в память сопоставления. Иначе, если $T < T_1$, токен помещается в предварительную память ПП1, и так далее, иначе – в память последнего уровня, здесь ПП2.

На каждом уровне память организована по «корзинам», индексируемым своей частью двоичного кода величины T. Когда память некоторого уровня становится относительно свободной, порог отсечения вышестоящей памяти увеличивается, и токены из ее нижней корзины перекладываются в нижестоящую память, где распределяются по ее корзинам.

Когда токен попадает в память сопоставления (ПС), там либо обнаруживается элемент с тем же ключом, либо нет. В первом случае токен присоединяется к найденному элементу. Во втором – создается новый элемент на основе данного токена. Если в результате образуется элемент с токенами на всех входах, то он преобразуется в пакет и передается в исполнительное устройство (ИУ) на исполнение.

Токены, пришедшие из сети, обрабатываются аналогично.

Описанный механизм планирования состоит из двух частей: базовой и управляющей. На базовом уровне происходит обработка отдельных токенов в соответствии с приписанными им номерами этапа и текущими порогами, приписанными уровням памяти (свои у каждого ядра). Правила работы базовой части простые и однозначные. На уровне управления происходит медленное изменение порогов, направленное на поддержание достаточного уровня активности в ядре. Здесь правила более сложные, представлены здесь очень приблизительно, и требуется еще много экспериментальной работы для выбора хорошо работающих правил. Возможно, будет требоваться особая настройка данного механизма для каждой конкретной задачи.

Для работы данного механизма в любом случае от программиста требуется хорошая (оптимальная) функция распределения по времени. При ее выборе следует руководствоваться следующими принципами:

токены, порождаемые некоторым узлом, должны иметь время (номер этапа), которое равно или больше времени этого узла;

каждый этап должен с хорошим запасом умещаться в активной зоне; более того, несколько соседних этапов должны умещаться в активной зоне;

функция распределения должна быть вычислительно простой: она будет вычисляться при каждом токене, и ее должно быть легко реализовать в виде быстрой схемы в ПЛИС.

5. Реализация умножения матриц

Рассмотрим программирование в парадигме раздачи задачи умножения матриц, используя язык потока данных DFL.

Умножение матриц A и B определено формулой

$$C_{ij} = \sum^{n}_{k=1} A_{ik} B_{kj}$$

Код приведен на Рис. 4. Центральный узел M умножает два элемента A_{ik} и B_{kj} и передает результат на суммирование для получения суммы C_{ij}. Как предполагает парадигма раздачи, узел M умножает значения двух своих входов a и b, не зная, откуда они берутся. О том, что это именно A_{ik} и B_{kj}, сообщает первая строка-комментарий. Индекс «*» это «джокер», который говорит о том, что здесь может быть любое значение. Фактически каждый элемент должен дойти до n разных узлов M. Любые два токена, направленные на входы M.a и M.b с общим индексом k создадут активацию узла M(a,b)[i,j,k]. Тело узла M говорит о том, что произведение a*b посылается на (единственный) вход узла S. Описатель входа (+)s[n] в заголовке узла S говорит, что это суммирующий (+) вход s, на который должно прийти n слагаемых. Здесь n – константа задачи, посылать ее не нужно. Когда придут все n токенов-слагаемых, направленных на S.s[i,j], узел S[i,j] будет активирован со значением на входе s, равным их сумме, которая будет выдана «наружу» в качестве элемента [i,j] выходной матрицы C.

Однако, такая программа с «джокерами» может правильно работать, только когда все токены поступают в одно-единственное ядро с ассоциативной памятью, в которую могут поместиться обе матрицы, и где должно произойти N^4 сравнений. Поскольку нас это не устраивает, надо избавиться от джокеров в посылаемых токенах. Будем использовать метод удвоений по дереву. Суммирование тоже лучше сделать явным сдваиванием, иначе оно должно будет

выполняться последовательно для каждой пары [i,j] (а суммарный трафик к этим узлам будет почти N^3 вместо желаемых $N^2 \cdot \sqrt[3]{K}$).

```
// Inputs: Aik → M.a[i,k,*]; Bkj → M.b[*,k,j];
node M(a,b) [i,k,j]; a*b → S.s[i,j];
node S((+)s[n])[i,j]; s → C[i,j] // Output: Cij
```

Рис. 4. Программа умножения матриц на языке DFL. Наивный вариант с «джокерами».

При написании хорошо масштабируемой распределенной программы на DFL полезно представлять каждый экземпляр узла как отдельный виртуальный процессор в многопроцессорной системе. В дальнейшем множества таких виртуальных узлов отображаются – посредством функции распределения (по пространству) – на процессоры (ядра) реальной системы с прицелом на равномерность загрузки и минимизацию коммуникаций. Пространство виртуальных узлов-умножителей заполняет куб NxNxN. Элементы матрицы A_{ik} (B_{kj}) размножаются вдоль измерения j (i), суммы собираются вдоль измерения k. Учитывая, что реальные ядра будут содержать блоки LxLxL при числе ядер K=$(N/L)^3$, будет полезно размножение и сбор сумм организовать по двоичному дереву так, чтобы каждый элемент входной матрицы входил в физическое ядро не более чем по одному разу, а также чтобы каждое слагаемое выходной матрицы выходило из физического ядра не более чем по одному разу.

В распределенном варианте программы, представленном на Рис. 5, были добавлены узлы AA и BB, осуществляющие рассылку элементов методом удвоений. В результате создается по N копий для каждого входного элемента, так что каждый умножитель получает свою собственную копию обоих входных элементов. Для удобства кодирования удвоений элемент M[i,k,j] имеет индексы, увеличенные на N, то есть каждый индекс лежит в диапазоне [N..2N-1]. Вдоль измерения k производится суммирование методом сдваивания.

```
// Inputs: A_ik → AA[i+N,k+N,  1];
//         B_kj → BB[1  ,k+N,j+N];
node AA(a)[i,k,m];
   if (m<N) a → AA[i,k,2*m], AA[i,k,2*m+1];
   else a → M.a[i,k,m];
node BB(b)[m,k,j];
   if (m<N) b → BB[2*m,k,j], BB[2*m+1,k,j];
   else b → M.b[m,k,j];
node M(a,b) [i,k,j]; a*b → SS.s[i,k/2,j];
node SS((+)s[2])[i,m,j];
   if (m=1) s → C[i-N,j-N] // Output: C_ij
   else s → SS[i,m/2,j];
```

Рис. 5. Распределенная программа умножения матриц

5.1. Функции распределения

При наличии $K{=}2^k$ процессорных ядер оптимальное распределение по ядрам должно быть блочным по 3-м измерениям с числом блоков по измерениям $K_1{\approx}K_2{\approx}K_3$, всего $K{=}K_1{\cdot}K_2{\cdot}K_3$ блоков, где $K_i{=}2^{k_i}$. Такое распределение обеспечит любая функцияи от {i,k,j}, которая у индексов i,k,j использует только старшие k_1,k_2,k_3 разрядов соответственно, а младшие игнорирует. Однако, внутри ядер желательно, чтобы блоки меньшего размера хорошо помещались внутрь уровней памяти.

Многоуровневое «блочное» распределение по времени обеспечит функция $F{=}\mathrm{zip}(i,k,j)$, которая в двоичном представлении реализуется как «скрещивание» разрядов аргументов: для получения трех самых младших разрядов значения F берем по одному младшему разряду каждого аргумента, для следующих трех берем по одному следующему разряду в том же порядке и т.д. В качестве номера процессора P берем k старших (из $3n$) разрядов F. Оставшиеся младшие разряды будем использовать в качестве условного времени S внутри каждого процессорного ядра. Однако, поскольку внутри ядра старшие разряды неизменны, в качестве условного времени мы можем просто использовать значение F.

В размножающих и суммирующих узлах для индекса m желательно обеспечить соблюдение условия: одна из двух копий (соответственно, одно из двух сдваиваемых слагаемых) попадают в то же ядро, где было исходное значение (соответственно, где будет сумма). Поэтому потребуем, чтобы в интервале [1..N-1] выполнялось свойство $P(..., m, ...) = P(..., 2m,...)$ для любого аргумента функции распределения P. Для этого при обращении к функции zip применим к каждому ее аргументу v функцию «нормализации» $\mathrm{norm}(v,l)$, где l – параметр, которая находит в двоичном представлении v старшую единицу и затем выбирает следующие за ней l разрядов (при необходимости дополняя их нулями). Окончательный вид функции распределения приведен на Рис. 6.

5.2. Работа программы

Рассмотрим процесс работы для случая, когда в процессорном узле имеется 2 уровня памяти, причем внешней памяти хватает для размещения всех токенов, а внутренней нет. Пусть объема внутренней хватает с необходимым запасом для размещения одного блока размера LxLxL, L=2^l. Тогда наиболее успешным будет такой режим работы планировщика, когда в активной зоне будут размещаться все токены из диапазона этапов длиной около L^3. (Можно у вышеописанной функции распределения отбросить несколько младших разрядов и работать с более крупными этапами).

Можно показать, что при таком планировании в активной зоне будет всегда не более $2L^3$ токенов. А без планирования внутренняя память быстро переполнилась бы, скажем, токенами с элементами матрицы A, и процесс был бы заблокирован из-за отсутствия токенов матрицы B.

Вначале в систему «впрыскиваются» элементы Aik и Bkj, которые активируют узлы AA{i+N,k+N,1} и BB{1,k+N,j+N}. Каждый узел удваивает элемент, оставляя одну копию (с индексом 2m) в том же ядре. Другая копия (с индексом 2m+1) на верхних уровнях дерева удвоений пересылается в другие ядра, а на нижних остается в том же. При этом первая копия принадлежит тому же этапу, а вторая – тому же или более позднему.

Поскольку есть порог отсечения, то в активную зону пойдут только токены ниже порога. Планировщик, управляющий текущим значением порога, должен быть настроен так, чтобы активная зона всегда была достаточно заполнена, но не переполнялась.

При суммировании процесс идет в обратном порядке, поэтому следует подправить функцию распределения по этапам для аргумента k, чтобы исключить порождение токена с меньшим временем. Например, можно инвертировать (путем вычитания из N-1) все разряды значения функции norm, относящиеся к этому аргументу. На Рис. 6 изображено окончательное описание функций распределения по пространству (place) и времени (stage).

```
const n=…;  // N=2^n
F=zip(norm(i,n),N-1-norm(k,n),norm(j,n));
place = shr(F,3*n-k); // K=2^k
stage = F;
```

Рис. 6. Функции распределения по пространству (place) и времени (stage) для умножения квадратных матриц.

Заключение

Мы утверждаем, что можно достичь существенного облегчения разработки оптимизированных приложений для суперкомпьютеров, если перейти от программирования в парадигме сбора, характерной для стандартных фон-неймановских языков, к программированию в парадигме раздачи, свойственной потоковой модели вычислений.

В предлагаемой парадигме программирования для управления распределением вычислений по пространству и времени используется метод указания функций распределения. Пользователь (или компилятор) задает формулы для вычисления двух величин: place и stage по адресу виртуального вычислительного узла, в котором будет данный элемент использован. Этот адрес, в соответствии с парадигмой раздачи, вырабатывается при порождении каждого нового элемента данных. Функция place задает номер процессорно-

го ядра, stage – номер этапа в рамках каждого ядра. Данные с ближайшим номером этапа подаются в «активную зону». Остальные – в зависимости от ожидаемого времени наступления этапа – перенаправляются в буферную память «отложенных» токенов соответствующего уровня.

Важно отметить, что в этом подходе реализуется четкое разделение аспектов: математическая правильность алгоритма определяется программой на DFL, тогда как функции распределения влияют только на скорость работы программы и, возможно, ее завершаемость. Функция place влияет непосредственно на равномерность распределения по ядрам и на объем коммуникаций между ними, а функция stage – на то, чтобы данные поступали вовремя, обеспечивая работой исполнительные устройства без переполнения памяти разных уровней при минимизации обменов между уровнями.

Конечно, об эффективности предложенного решения можно говорить только в том случае, если накладные расходы на передачу токена внутри ядра «к себе», включая вычисление функции распределения, будут заметно меньше затрат на выполнение отдельного умножения. В принципе это всегда достижимо за счет укрупнения зернистости, когда умножаемыми и складываемыми элементами являются матричные блоки подходящего размера. Кроме того, мы предполагаем, что функции распределения компилируются в эффективную прошивку для специальной ПЛИС, стоящей на выходе из ИУ. И в любом случае данная модель полезна для анализа эффективности разрабатываемых алгоритмов, когда на первый план выходят затраты на взаимодействие и перемещения в иерархической памяти.

Список литературы

[1] Guang R. Gao, Thomas Sterling, Rick Stevens, Mark Hereld, Weirong Zhu, "ParalleX: A Study of A New Parallel Computation Model," ipdps, pp.294, 2007 IEEE International Parallel and Distributed Processing Symposium, 2007

[2] Левшин И. *Свежий взгляд из-за океана. Беседа с Джеком Донгаррой* // Суперкомпьютеры, 14, 2013.

[3] Климов А.В. *Умножение плотных матриц на неоднородных высокопараллельных вычислительных системах (анализ коммуникационной нагрузки).* Журнал «Информационные технологии», №3, 2008, с.24-31.

[4] John E.Savage, Mohammad Zubair, A Unified Model for Multicore Architecture, In Proc. 1st International Forum on Next-Generation Multicore/Manycore Technologies, ACM, NewYork, NY, 2008.

[5] 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" №10, 2008, с.2–7.

[6] Levchenko N.N., Okunev A.S., Zmejev D.N., Klimov A.V. Effective planning of calculations on the PDCS "Buran" architecture. In: Proc. of the International IEEE EAST-WEST DESIGN & TEST SYMPOSIUM (EWDTS'2013), Rostov-on-Don, Russia, September 2013.

[7] Levchenko N.N., Okunev A.S., Zmejev D.N., Klimov A.V. Management methods of computational processes in the PDCS "Buran". In: Proc. of the International IEEE EAST-WEST DESIGN & TEST SYMPOSIUM (EWDTS'2013), Rostov-on-Don, Russia, September 2013.

A.V.Klimov, N.N. Levchenko, A.S. Okunev. Supercomputers, memory hierarchy and dataflow computation model.

Abstract. Modern and future supercomputers are necessarily hierarchical; such is the interconnecting network as well as the memory inside a node or a node group. Therefore, data movement overhead become the most significant factor of inefficiency, and thus the task of optimizing programs in this respect gets more and more difficult. We claim that these difficulties are largely a consequence of traditional programming paradigm that goes back to von Neumann. And although it has such a strong case as the acquirements and the legacy software, it is still useful at least in theory to understand the alternatives. We explain that the problem of the von Neumann programming model is in implementing the so-called *gather* paradigm, and propose the usage of the dataflow computation model, which implements the *scatter* paradigm and thus provides much easier solution to the data movement optimization problem.

Key Words and Phrases: (Supercomputer, memory hierarchy, data preloading, dataflow computation model, gather paradigm, scatter paradigm, computation scheduling).