УДК 004.4’242; 004.4’422; 004.432.4 doi 10.26089/NumMet.v20r321

РАСПРЕДЕЛЕНИЕ ВЫЧИСЛЕНИЙ В ГИБРИДНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМАХ ПРИ ТРАНСЛЯЦИИ ПРОГРАММ НА ЯЗЫКЕ НОРМА

А. Н. Андрианов[1], Т. П. Баранова[2], А. Б. Бугеря[3], К. Н. Ефимкин[4]

Рассмотрены методы распределения вычислительной нагрузки при трансляции программ с непроцедурного (декларативного) языка НОРМА в исполняемые программы для различных параллельных архитектур. Приведены краткие характеристики языка НОРМА и основные возможности компилятора программ на языке НОРМА. Описаны способы автоматического распределения вычислительной нагрузки при генерации исполняемых программ следующих типов: OpenMP, NVIDIA CUDA, MPI+OpenMP и MPI+OpenMP+NVIDIA CUDA. Рассмотрена задача динамической балансировки вычислительной нагрузки, возникающая в случае неоднородной вычислительной среды MPI+OpenMP+NVIDIA CUDA, и предложен метод ее решения. Приведены результаты практического применения компилятора программ на языке НОРМА для решения двух различных задач и оценена скорость выполнения получаемых при этом исполняемых программ для различных параллельных архитектур.

Ключевые слова: параллельное программирование, автоматизация программирования, непроцедурные спецификации, гибридные архитектуры, балансировка вычислительной нагрузки, язык НОРМА.

1. Введение. Задача разработки эффективных параллельных программ в настоящее время остается крайне актуальной и имеет важное стратегическое значение, так как в большинстве областей науки и отраслей промышленности используются высокопроизводительные компьютеры и параллельное программирование. Эта задача давно исследуется в мировом научном сообществе, является крайне сложной и пока не имеет удовлетворительного решения. Более того, эта задача постоянно осложняется разработкой все новых архитектур параллельных компьютеров, так как каждая новая архитектура ставит новые проблемы при распараллеливании или осложняет уже существовавшие проблемы.

Существуют различные методы автоматизации разработки параллельных программ. Отметим лишь монографии [1, 2], в которых были строго сформулированы математические основы совместного изучения параллельных численных методов и параллельных вычислительных систем и исследована задача отображения программы на архитектуру параллельного компьютера. В частности, показано, что автоматическое отображение уже написанной последовательной программы на параллельный вычислитель, в общей постановке, является NP-полной задачей. Это и объясняет отсутствие до настоящего времени удовлетворительного с практической точки зрения метода разработки параллельных программ.

Тем не менее, исследования в данной области ведутся весьма активно и часто поддерживаются фирмами-производителями вычислительных систем. Из уже реализованных подходов можно отметить те, которые базируются на вполне разумном симбиозе распараллеливающего компилятора и подсказок со стороны программиста, выполненных в виде специальных программных директив, например [3, 4].

В такой ситуации интерес представляют подходы к построению параллельных программ, точно определяющие границы того, что и как можно автоматически распараллеливать, и предоставляющие возможности для автоматизированного построения эффективных параллельных программ. Одним из таких подходов является непроцедурный язык НОРМА [5].

2. Декларативный подход. Язык НОРМА. Одним из подходов к решению задачи автоматизации разработки параллельных программ для вычислительных задач является подход с использованием непроцедурных (декларативных) языков. Этот подход развивается в нашей стране и за рубежом достаточно давно, например в работах [6–8].

[1] Институт прикладной математики им. М. В. Келдыша РАН (ИПМ РАН), Миусская пл., 4, 125047, Москва; вед. науч. сотр., e-mail: andrianovan52@mail.ru

[2] Институт прикладной математики им. М. В. Келдыша РАН (ИПМ РАН), Миусская пл., 4, 125047, Москва; науч. сотр., e-mail: bart1950@yandex.ru

[3] Институт прикладной математики им. М. В. Келдыша РАН (ИПМ РАН), Миусская пл., 4, 125047, Москва; ст. науч. сотр., e-mail: shurabug@yandex.ru

[4] Институт прикладной математики им. М. В. Келдыша РАН (ИПМ РАН), Миусская пл., 4, 125047, Москва; ст. науч. сотр., e-mail: bigcrocodile@yandex.ru

Идеи декларативного программирования были сформулированы в России еще в прошлом веке, теоретические исследования этого подхода для класса вычислительных задач проведены в пионерских работах И.Б. Задыхайло еще в 1963 г. [9]. Предлагалось в качестве языка программирования для описания решения задачи взять сложившуюся в данной предметной области математическую нотацию и использовать ее для автоматического, при помощи компилятора, построения исполняемой программы. На основе этих идей в ИПМ им. М.В. Келдыша РАН разработан непроцедурный язык НОРМА [5, 10–12] и создана система параллельного программирования НОРМА [13–16], предназначенные для разработки параллельных программ для расчетных задач математической физики. Оказалось, что для программы на языке НОРМА можно при определенных условиях автоматически получить исполняемую параллельную программу на основе функции оптимизации, которая может, в частности, учитывать модель параллелизма и особенности архитектуры компьютера.

Для языка НОРМА определены и теоретически обоснованы методы и алгоритмы распараллеливания, условия и ограничения, при которых разрешима задача распараллеливания [16–18]. Эти методы и алгоритмы легли в основы созданной системы программирования НОРМА.

Язык НОРМА и система НОРМА успешно использовались для решения сложных практических задач математической физики [19–25]. При этом весьма важно, что эффективность и скорость выполнения параллельных программ, получаемых компилятором автоматически, обычно сравнима с ручным программированием, а ускорение часто близко к линейному.

Привлекательность рассматриваемого подхода в настоящее время только усиливается, и интерес к непроцедурному (декларативному) программированию растет. Это объясняется тем, что возникновение новых параллельных архитектур требует для традиционных подходов разработки новой программы для новой архитектуры (как было, например, при появлении графических процессоров), существующая программа не может быть перенесена на новую параллельную архитектуру без существенного переписывания и отладки. Для программ на языке НОРМА эти проблемы существенно проще — сама программа остается неизменной, а дорабатывается (один раз) только компилятор.

Некоторые особенности и ограничения языка НОРМА, позволившие решить отмеченные выше проблемы, можно описать следующим образом.

- Декларативность. Программа (и каждый ее оператор) описывает запрос на вычисление. Каким образом этот запрос реализуется, не указывается. Нет понятий памяти, управления (переходов, циклов и тому подобное).

- Однократное присваивание — величины могут принимать значения только один раз, переприсваивание значений невозможно. Язык НОРМА принадлежит к классу SAL-языков (Single Assignment Language). Нет глобальных переменных и побочных эффектов.

- Ограничение на вид индексных выражений у величин имеют вид $(A{*}i{+}C_1)/B{+}C_2$, где i — индексная переменная, A и B — натуральные константы, а C_1 и C_2 — целые константы.

- Области имеют границы, заданные константными выражениями, значения которых известны в момент трансляции. Области также могут иметь переменные границы, определяемые целочисленными формальными параметрами раздела.

При этом язык НОРМА имеет возможности, необходимые для написания практических программ, в частности интерфейсы с Фортраном и Си, позволяющие использовать фрагменты программ на языках Фортран и Си. В системе НОРМА интерфейсы поддерживаются специальной подсистемой — конфигуратором.

3. Компилятор программ на языке НОРМА. В систему программирования НОРМА [13] входит непроцедурный язык НОРМА [5], компилятор программ на языке НОРМА [14], конфигуратор, параллельный диалоговый отладчик в терминах исходной программы [15], интегрированные в едином интерфейсе пользователя. Компилятор программ на языке НОРМА позволяет получать исполняемую программу на заданном процедурном языке программирования для определенной модели параллелизма. На данный момент компилятор умеет создавать программы на языках Си или Фортран для следующих вычислительных архитектур:

— последовательные программы;

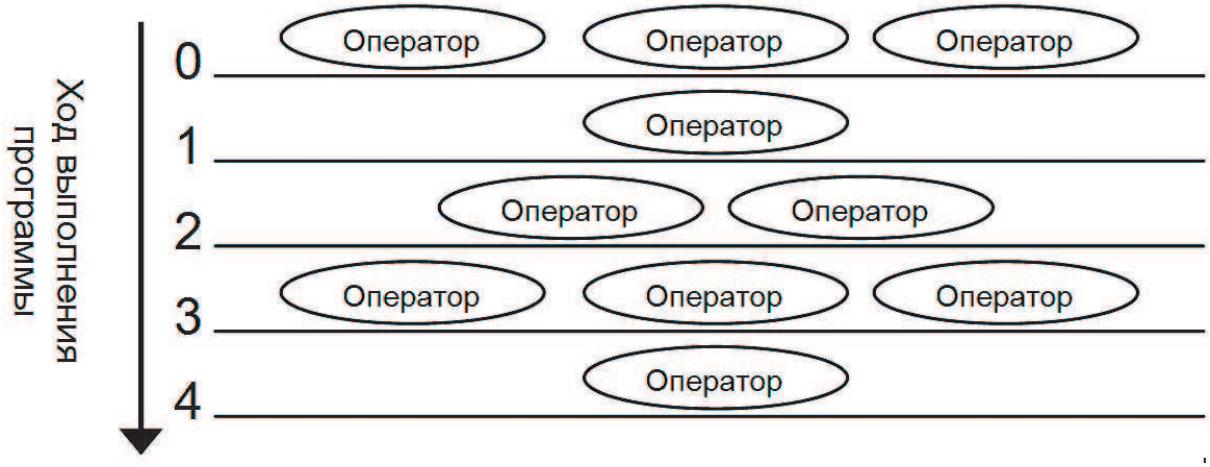
— для многоядерных систем с общей памятью с использованием технологии OpenMP;

— для графических процессоров фирмы NVIDIA с использованием технологии NVIDIA CUDA;

— для распределенных систем с использованием технологии MPI и с возможным одновременным использованием технологии OpenMP для многоядерных узлов.

В настоящее время ведутся работы по созданию версии компилятора для гибридных архитектур. В выходной программе одновременно будут использоваться технологии MPI, OpenMP и NVIDIA CUDA.

4. Распределение вычислений при трансляции программ для различных параллельных архитектур. При трансляции с языка НОРМА решается задача синтеза выходной параллельной программы. В результате анализа зависимостей по данным между операторами программы на языке НОРМА, в случае разрешимости этих зависимостей, строится так называемая "ярусно-параллельная форма" [16, 18] представления выполнения программы. Схематично она изображена на рисунке. На каждом ярусе такой ярусно-параллельной формы располагаются операторы программы, которые не имеют зависимостей друг от друга и могут выполняться независимо и, соответственно, параллельно. В то же время каждый из этих операторов имеет зависимость от одного или более операторов, располагающихся на предыдущем ярусе ярусно-параллельной формы. Ярусно-параллельная форма программы является, фактически, представлением идеального (естественного) параллелизма, определяемого соотношениями и зависимостями между расчетными переменными программы.



Ярусно-параллельная форма представления выполнения программы

Таким образом, группу операторов, располагающихся на одном уровне ярусно-параллельной формы, можно выполнять параллельно в результирующей программе, но только после того, как будут полностью выполнены все операторы предыдущего уровня ярусно-параллельной формы. Это дает первый уровень параллелизма в выходной программе — на уровне операторов одного яруса ярусно-параллельной формы.

В каждом не скалярном (имеющем область применения) операторе, находящемся в ярусно-параллельной форме и не являющемся специальной группой операторов (сильно связной компонентой), каждую точку области применения можно считать независимо от других точек. Это дает возможность организовать второй уровень параллелизма в выходной программе — на уровне точек области применения оператора.

Кроме того, выполнение операций редукции тоже может быть организовано параллельно по точкам области функции редукции. Это дает третий уровень параллелизма в выходной программе.

Далее в качестве примера рассмотрим преобразования, производимые компилятором программ на языке НОРМА, для некоторых операторов небольшой программы, решающей систему линейных уравнений $\sum\limits_{j=1}^{m} A_{i,j}X_j = b_i$, $i = 1, \ldots, m$, разностным методом.

Метод решения:

$$X_i^0 = X0, \quad i = 1, \ldots, m, \qquad X_i^n = \frac{1}{A_{i,i}} \left(b_i - \sum_{j=1, j \neq i}^{m} A_{i,j} X_j^{n-1} \right), \quad i = 1, \ldots, m.$$

Условие выхода из итерации: $\left| X_i^n - X_i^{n-1} \right| < \epsilon$, $i = 1, \ldots, m$.

На языке НОРМА программа, решающая такую систему линейных уравнений, выглядит так:

```
MAIN PART Linear.
BEGIN
Oi: (I=1..M). Oj: (J=1..M). Oij: (Oi; Oj). O1,O2: Oj / J<>I.
VARIABLE XO, X, b DEFINED ON Oi. VARIABLE A DEFINED ON Oij.
VARIABLE E.
DOMAIN PARAMETERS M = 20000.
DISTRIBUTION INDEX I = 10, J = 10.
INPUT XO, b ON Oi. INPUT A ON Oij. INPUT E.
OUTPUT X ON Oi.
ITERATION X ON N.
  INITIAL N = 0:
  FOR Oi ASSUME X = XO.
  END INITIAL
  FOR Oi ASSUME X = 1/A[J=I] * (b - SUM((O1)A * X[I=J,N-1])).
  EXIT WHEN MAX((Oi) ABS(X[N] - X[N-1]) ) < E.
END ITERATION N.
END PART.
```

4.1. Распределение вычислений для программ с использованием OpenMP. При генерации выходной программы с использованием технологии OpenMP в первую очередь параллельно вычисляются точки области применения оператора. Если же оператор скалярный, но в нем имеется функция редукции, то организуется параллельное вычисление операции редукции. В обоих случаях применяются соответствующие OpenMP-директивы для цикла по внешнему индексу: либо области применения оператора, либо области редукции.

Информация о параллелизме на уровне операторов также используется в выходной программе, хотя дополнительно никаких параллельных вычислений и не организуется, следующим образом. Если оператор заканчивает очередной уровень ярусно-параллельной формы и после него начнутся операторы следующего яруса, то в конце этого оператора необходима синхронизация OpenMP-потоков. В ином случае, если за оператором следует оператор того же уровня ярусно-параллельной формы, синхронизация OpenMP-потоков не требуется.

В качестве примера рассмотрим генерацию кода для реализации следующих операторов приведенной выше программы.

1. Оператор ASSUME:

```
FOR Oi ASSUME X = 1/A[J=I] * (b - SUM((O1)A * X[I=J,N-1])).
```

Соответствующий код на языке Си, построенный компилятором с использованием технологии OpenMP, выглядит следующим образом:

```
#pragma omp for private(J, Total)
for(I = 0; I <= M-1; I++) {
  Total = 0.0;
  for(J = 0; J <= M-1; J++) {
    if(J+1 != I+1)
      Total = Total + A[I][J] * XShadow[J];
  }
  X[I] = 1 / A[I][I] * (b[I] - Total);
}
```

В данном случае параллельно вычисляются все точки области применения оператора, а вычисление функции редукции, требуемое в каждой точке, производится без применения OpenMP-директив. При этом, однако, вводимые локальные временные переменные (`Total`) объявляются приватными.

2. Скалярный оператор в функции редукции в условии выхода из итерации:

```
EXIT WHEN MAX((Oi) ABS(X[N] - X[N-1]) ) < E.
```

Соответствующий код на языке Си, построенный компилятором с использованием технологии OpenMP, выглядит следующим образом:

```
#pragma omp parallel private(Priv)
```

```
#pragma omp single
{
  Total1 = abs(X[1-1] - (XShadow[1-1]));
}
Priv = abs(X[1-1] - (XShadow[1-1]));
#pragma omp for private(Tmp)
for(I = 1; I <= M; I++) {
  Tmp = abs(X[I-1] - (XShadow[I-1]));
  if(Priv < Tmp)
    Priv = Tmp;
}
#pragma omp critical
{
  if(Total1 < Priv)
    Total1 = Priv;
}
```

Функция редукции вычисляется параллельно. Каждая OpenMP-нить вычисляет свое значение в переменной `Priv`, а затем с использованием критической секции формируется общий результат `Total1`, который затем сравнивается с `E` для проверки условия выхода из итерации.

4.2. Распределение вычислений для программ с использованием NVIDIA CUDA. В результате ряда исследований по трансформации описанной ярусно-параллельной формы представления выполнения программы на языке НОРМА в программу с использованием технологии NVIDIA CUDA для графических процессоров предлагается использовать следующий подход для автоматического построения исполняемой программы. Исполняемая программа стартует и завершается на центральном процессоре, на нем же выполняется ввод-вывод данных и итерационные циклы. Очевидно также, что вся логика по управлению вычислениями, выделению памяти на графическом процессоре, организации обменов данными между памятью графического и центрального процессора, тоже создается в программном коде, выполняющемся на центральном процессоре. Вызовы других процедур и функций должны выполняться из программы на центральном процессоре, если это “обычные” пользовательские или библиотечные функции, имеющие реализации для центрального процессора, или же из вычислительного ядра, выполняющегося на графическом процессоре, если это пользовательские или библиотечные функции, имеющие реализации для графического процессора.

Сами вычислительные операторы выполняются, по возможности, на графическом процессоре с использованием параллелизма на уровне точек области применения оператора или на уровне точек области функции редукции. Вычисления для каждой точки области выполняются своей нитью графического процессора. Для этого для области выбирается распределение блоков и нитей графического процессора, покрывающее область распараллеливания.

Такие вычислительные операторы группируются в определенные наборы, каждый из которых может выполняться в пределах одного ядра программы с использованием технологии NVIDIA CUDA. С одной стороны, чем больше операторов будет содержать такое ядро, т.е. чем крупнее будут исполняемые ядра в программе, тем эффективнее она будет работать, так как будет меньше запусков ядер, меньше передачи параметров, в том числе и через память центрального процессора, и т.п. Но, с другой стороны, все операторы, сгруппированные в одно ядро, должны работать с одним и тем же распределением рабочих областей на конфигурацию блоков и нитей, определяемую при запуске ядра. И может оказаться, что поиск такого распределения для большой группы операторов, объединенных в одно ядро, может дать худший результат, чем разбиение этой группы операторов на более мелкие отдельные ядра и поиск распределения для каждой группы в отдельности. Так что задача группировки вычислительных операторов в ядра весьма нетривиальна.

В качестве примера рассмотрим генерацию кода для реализации того же оператора ASSUME приведенной выше программы:

```
FOR Oi ASSUME X = 1/A[J=I] * (b - SUM((O1)A * X[I=J,N-1])).
```

В соответствующем коде на языке Си с использованием технологии NVIDIA CUDA компилятором организуется 2 вычислительных ядра с разным распределением индексов по блокам и нитям. Первое ядро начинает вычисление функции редукции и получает ее частичный результат. При этом индекс `I` представлен столбцами матрицы блоков, а индекс `J` представлен произведением строк матрицы блоков на нити.

Вызов первого ядра из кода, выполняемого центральным процессором, выглядит следующим образом:

```
#define J_kernel1 512
#define J_kernel1_Blocks (M + J_kernel1 - 1) / J_kernel1
kernel1<<<dim3(J_kernel1_Blocks,M),J_kernel1>>>(X_ptr,XShadow_p tr);
```

Код первого ядра приведен ниже:

```
static __global__ void kernel1(float X[M], float XShadow[M])
{
  __shared__ float shared[J_kernel1];
  int Red;
  int I = blockIdx.y;
  int J = threadIdx.x + blockIdx.x*J_kernel1;
  if(J < M)
  {
    if(J != I)
      shared[threadIdx.x] = A_dev[I][J] * XShadow[J];
    else
      shared[threadIdx.x] = 0;
    for(Red = J_kernel1 >> 1; Red > 0; Red = Red >> 1)
    {
      __syncthreads();
      if((threadIdx.x < Red) && (J + Red < M))
        shared[threadIdx.x] = shared[threadIdx.x] + shared[threadIdx.x + Red];
    }
    if(threadIdx.x == 0)
      SUM_block[J] = shared[0];
  }
}
```

Второе ядро заканчивает вычисление значения функции редукции в каждой точке индекса I, используя нити для редукции по частичному результату, полученному первым ядром. При этом индекс I представлен линейкой блоков. Вызов второго ядра из кода, выполняемого центральным процессором, выглядит следующим образом:

```
#define J_kernel2 64
kernel2<<< M, J_kernel2 >>>(X_ptr,XShadow_ptr);
```

Код второго ядра приведен ниже:

```
static __global__ void kernel2(float X[M], float XShadow[M])
{
  __shared__ float shared1[J_kernel2];
  int Red1;
  int I = blockIdx.x;
  int J = threadIdx.x;
  if(J < (M + 512 - 1) / 512 + 1 - 1)
  {
    int JSUM = J*512;
    shared1[threadIdx.x] = SUM_block[JSUM];
    for(Red1 = J_kernel2 >> 1; Red1 > 0; Red1 = Red1 >> 1)
    {
      __syncthreads();
      if((threadIdx.x < Red1) && (J + Red1 < (M + 512 - 1) / 512 + 1 - 1))
        shared1[threadIdx.x] = shared1[threadIdx.x] + shared1[threadIdx.x + Red1];
```

```
    }
    if(threadIdx.x == 0)
      X[I] = 1 / A_dev[I][I] * (b_dev[I] - shared1[0]);
    }
}
```

4.3. Распределение вычислений для программ с использованием MPI. При использовании технологии MPI в разделе программы на языке НОРМА можно задать распределяющие индексы (один или более) или указать, что индексы уже являются распределенными (если этот раздел вызывается из раздела с распределяющими индексами). Все переменные, определенные на областях с такими индексами, равномерно распределяются между MPI-процессами. Вернее, равномерное распределение производится для максимальной по такому индексу области. Остальные области, содержащие распределяющие индексы, распределяются по принципу соответствия индекса точек в каждом MPI-процессе индексам точек максимальной области, а переменные распределяются в соответствии с распределением области их определения.

Вычисления для таких областей производятся каждым MPI-процессом только по той части области, которая распределена в данный MPI-процесс. Если какому-то MPI-процессу для вычислений требуются данные, распределенные в другой MPI-процесс, то организуется передача данных путем посылки и принятия сообщения. Ввод/вывод данных осуществляется нулевым MPI-процессом, который рассылает/собирает данные со всех остальных MPI-процессов. При вычислении функций редукции используются соответствующие функции MPI.

Рассмотрим генерацию компилятором кода для реализации все того же оператора ASSUME приведенной выше программы:

```
FOR Oi ASSUME X = 1/A[J=I] * (b - SUM((O1)A * X[I=J,N-1])).
```

В начале программы каждый процесс определяет свою позицию в сетке процессов программы и создает необходимые для обмена данными коммуникационные группы:

```
MPI_Comm_rank(MPI_COMM_WORLD, &IPROC_ID);
IMATR_J = (IPROC_ID/10)%10;
IMATR_I = (IPROC_ID)%10;
MPI_Comm_split(MPI_COMM_WORLD, IMATR_I, IPROC_ID, &Comm);
```

Все эти данные затем используются при организации вычислений. Вначале каждый MPI-процесс вычисляет частичный результат суммирования по своей части подобласти по J-индексу в каждой точке I, это значение сохраняется в переменной `Total`. Затем с помощью функции `MPI_Reduce` получается окончательное значение суммирования для точки I (в переменной `Global`), которая затем используется для вычисления `X[I]`).

```
for(I = 0; I <= 1999; I++)
{
  Total = 0.0;
  IPROC_J3 = IMATR_J*2000;
  IPROC_I4 = IMATR_I*2000;
  for(J = 0; J <= 1999; J++)
  {
    if(J+1+IPROC_J3 != I+1+IPROC_I4)
      Total = Total + A[I][J] * XShadow[J];
  }
  MPI_Reduce(&Total, &Global, 1, MPI_FLOAT, MPI_SUM, 0, Comm);
  if(IMATR_J == 0)
    X[I] = 1 / A[I][I] * (b[I] - Global);
}
MPI_Bcast(X, 2000, MPI_FLOAT, 0, Comm);
```

4.4. Распределение вычислений для программ с использованием MPI, OpenMP и NVIDIA CUDA. Если при использовании технологии MPI задано также и использование технологии OpenMP, то

каждый MPI-процесс при вычислении распределенных в него данных использует директивы OpenMP точно так же, как описано выше для программы с использованием только OpenMP. При этом операции приема/передачи данных выполняются одним из OpenMP-потоков, а все остальные потоки при этом дополнительно синхронизируются: при приеме данных — после операции приема данных, а при передаче данных — перед операцией передачи данных.

При одновременном использовании технологий MPI, OpenMP и NVIDIA CUDA области с распределяющими индексами и определенные на них переменные распределяются, во-первых, точно так же, как описано в предыдущем подразделе для технологии MPI. Однако затем в каждом MPI-процессе попавшая в него подобласть делится далее на зону, вычисляемую на центральном процессоре (возможно, с применением технологии OpenMP), и на зону, вычисляемую на графическом процессоре (процессорах). Если графических процессоров в вычислительной системе несколько, то их зона распределяется между ними поровну.

Таким образом, в каждом MPI-процессе имеется одна зона подобласти, вычисляемая центральным процессором, и одна или более зон подобласти, каждая из которых вычисляется своим графическим процессором.

При организации вычислений на центральном процессоре компилятором используется схема вычислений, описанная выше в подразделе MPI и OpenMP, а при организации вычислений на графическом процессоре используется схема вычислений, описанная выше в подразделе NVIDIA CUDA. При этом дополнительно решается задача передачи данных между зонами, если в этом есть необходимость.

5. Динамическая балансировка вычислительной нагрузки. Балансировка вычислительной нагрузки может производиться как статически, путем первоначального распределения областей вычисления, так и динамически — во время выполнения исполняемой программы, путем перераспределения границ областей вычисления. Целесообразно использовать оба подхода по следующим соображениям. В процессе выполнения программы балансировка вычислительной нагрузки требуется в случае неоднородной вычислительной среды и/или в случае неоднородных данных для вычислительного процесса. Неоднородная вычислительная среда возникает при решении задачи одновременного использования и центрального, и графического процессоров. Зоны какого размера в области вычисления выделять центральному и графическому процессору — изначально не ясно. Соотношение производительности процессоров может быть самым разным и зависеть от характера выполняющихся вычислений.

Поэтому задачу определения границ зон предлагается решать динамически, основываясь на данных о загрузке центрального процессора во время очередного шага (или нескольких шагов) итерации, за вычетом времени, проведенном в MPI-операциях. Если загрузка центрального процессора ниже какой-то границы, то это означает, что центральный процессор постоянно ждет окончания работы графического и что зону центрального процессора можно попробовать увеличить. Если же, наоборот, загрузка центрального процессора близка к 100%, то его зону можно попробовать уменьшить. В этом процессе изменения размеров зон важно найти такие значения порогов срабатывания запуска изменения зон, чтобы достаточно быстро находить точку равновесия, когда размер зон менять не нужно, а не приходить к ситуации постоянного то уменьшения, то увеличения размеров зон.

Алгоритм, решающий задачу динамической балансировки вычислительной нагрузки, в системе НОРМА пока не реализован.

6. Результаты применения компилятора. Компилятор программ на языке НОРМА использовался для решения многих практических и тестовых задач. В качестве примера применения компилятора приведем получившиеся результаты для расчетной задачи из области газодинамики и для теста CG из пакета NPB (NAS Parallel Benchmarks) [26]. В обоих примерах применения компилятора результат, полученный различными версиями исполняемых программ, идентичен эталонному. Кроме того, была проведена оценка скорости выполнения получающихся исполняемых программ. Производительность программ для графических процессоров фирмы NVIDIA с использованием технологии NVIDIA CUDA сравнивалась с последовательной и OpenMP-версиями той же самой программы. Для теста CG из пакета NPB также приведено сравнение с оригинальными версиями теста, написанными вручную. Все запуски производились на вычислительном кластере К-100 [27]. Компиляция последовательных и OpenMP-версий программ производилась компилятором Intel версии 15.0.0, компиляция CUDA программ — компилятором nvcc версии 6.5, для MPI-версий программ использовалась Intel MPI Library 5.0 Update 1.

6.1. Прикладная задача из области газодинамики. В основном рабочем итерационном цикле данной программы производится расчет различных величин для всех точек одномерной области. Для каждой точки вызывается "плоский" внешний раздел, который на основе предыдущих значений точки и ее соседей высчитывает данные для новых значений. Все вычисления производятся с одинарной точностью.

Времена выполнения версий программы приведены в табл. 1.

Таблица 1

Время выполнения программы решения задачи из области газодинамики

Последовательная программа, 1 ядро Intel Xeon X5670	461 сек.
OpenMP программа, 12 ядер Intel Xeon X5670	45.6 сек.
CUDA программа, 1 nVidia Fermi C2050	16.9 сек.
MPI программа, 48 MPI процессов, 4 узла по 12 ядер Intel Xeon X5670	15.8 сек.
MPI+OpenMP программа, 4 MPI процесса, 4 узла по 12 ядер Intel Xeon X5670	22.1 сек.

Построенные полностью автоматически программы демонстрируют хорошие показатели распараллеливания для всех применяемых технологий. Особенно стоит отметить программу для графических процессоров, которая демонстрирует высокую скорость выполнения и эффективность примененных в компиляторе методов и решений.

В то же время производительность версии программы MPI+OpenMP несколько хуже, чем производительность версий, использующих только одну технологию MPI или OpenMP на том же самом количестве вычислительных ядер. Исследование данной проблемы показало, что дело в необходимости постоянной синхронизации OpenMP-потоков перед отправкой MPI-сообщений или после принятия таковых. Синхронизация OpenMP-потоков, видимо, достаточно дорогая операция, поэтому, если в программе используется много обменов данными между MPI-процессами (а в данном примере это так — на каждом шагу итерации подкачиваются соседние точки), то и общая производительность существенно падает.

6.2. Реализация теста CG из пакета NPB. Наиболее широко тесты для измерения производительности кластерных систем представлены в пакете NAS Parallel Benchmarks [26], обладающем помимо объективных достоинств еще одним — авторитетностью программ и их авторов. Это послужило причиной выбора данного пакета для исследования возможности реализации его тестов на языке НОРМА и проверки эффективности кода, генерируемого компилятором с языка НОРМА.

После анализа исходных текстов программ тестового пакета NPB для эксперимента по портированию данного тестового пакета на язык НОРМА в первую очередь было выбрано ядро CG как наиболее подходящее под концепцию языка НОРМА. Тестовое ядро CG (Conjugate Gradient) осуществляет приближение к наименьшему собственному значению большой разреженной симметричной положительно определенной матрицы с использованием метода обратной итерации вместе с методом сопряженных градиентов в качестве подпрограммы для решения СЛАУ. Тест применяется для оценки скорости передачи данных при отсутствии какой-либо регулярности. Вычисления производятся в определенных классах (размерность основных массивов данных) задач: “Sample code”, “Class A” (маленькие), “Class B” (большие), “Class C” (очень большие), “Class D” (огромные). Все вычисления производятся с двойной точностью.

После анализа исходных текстов ядра CG было принято решение переписать на языке НОРМА основную рабочую функцию теста, а также этапы инициализации теста и основной тестовый цикл. В то же время стало очевидно, что такие действия, как начальная инициализация разреженной матрицы, плохо подходят для реализации средствами языка НОРМА. Завершающий шаг — анализ правильности полученных результатов и вывод статистических данных о прохождении теста — также было решено оставить в оригинальном виде на Си для неоспоримости факта правильности анализа полученных результатов оригинальным кодом и сохранения формата вывода. Фактически получается, что тест начинает свою работу и заканчивает ее кодом на Си, а та часть, в которой производятся все вычисления и при выполнении которой производится засечка времени и делается вывод о производительности системы, реализуется на языке НОРМА.

Среди конструкций, которые необходимо было портировать на язык НОРМА, оказались и такие, которые не могут быть выражены средствами языка НОРМА. В первую очередь речь идет о “сердце” теста — умножении разреженной матрицы на вектор. Но язык НОРМА поддерживает вызовы так называемых “внешних” функций, которые могут быть написаны на целевом языке программирования. Задача по реализации умножения разреженной матрицы на вектор (это делается с помощью косвенной адресации и цикла с переменными границами) была возложена на такую вспомогательную внешнюю функцию, а в программе на языке НОРМА в каждой точке области, соответствующей результирующему вектору, результат выполнения этой внешней функции в данной точке присваивается результирующей переменной.

Однако подобная реализация умножения разреженной матрицы на вектор (фактически оператор оставлен в том же виде, что был в оригинальном тесте), несмотря на очевидное достоинство — простоту

реализации (вся внешняя функция получилась из 8 строк) — и хорошие показатели распараллеливания в версии OpenMP, плохо подошла для выполнения на графических процессорах. Поэтому были разработаны и реализованы еще 2 варианта этой внешней функции, специально для версии для графических процессоров, которые учитывают особенности этой архитектуры. В первом из них каждая точка области рассчитывается одним блоком графического процессора. А цикл с переменными границами, в котором производится суммирование вычисленных значений, выполнен в виде редукции по нитям, где каждая нить предварительно высчитывает свое значение, соответствующее значению на определенном витке цикла. Такая реализация внешней функции получилась приблизительно в 120 строк кода. Второй вариант — использовать функцию из библиотеки cuSPARSE [28]. Такая реализация внешней функции получилась приблизительно в 60 строк кода. Результаты выполнения различных версий теста приведены в табл. 2. Во всех случаях проверка результата прошла успешно и приведенные цифры — количество миллионов операций в секунду (Mop/s), основной показатель скорости выполнения теста.

Таблица 2

Результаты запуска теста CG (Mop/s)

	Class A		Class B		Class C	
	Си	Си + НОРМА	Си	Си + НОРМА	Си	Си + НОРМА
Последовательная программа, 1 ядро Intel Xeon X5670	1200	1175	718	716	513	615
OpenMP программа, 12 ядер Intel Xeon X5670	9737	8035	3595	3004	3240	2806
CUDA программа, простая реализация, 1 nVidia Fermi C2050	—	927	—	722	—	645
CUDA программа, реализация с редукцией, 1 nVidia Fermi C2050	—	2134	—	2178	—	1876
CUDA программа, использование cuSPARSE, 1 nVidia Fermi C2050	—	6521	—	4504	—	2704

Как можно заметить, результаты выполнения теста CG с использованием языка НОРМА практически идентичны (видимо, с точностью до погрешности измерений) соответствующим результатам оригинального теста CG, написанного вручную. Это дает нам право утверждать, что тест CG прекрасно подошел для записи его на языке НОРМА и что задачи, имитирующиеся этим тестом, также должны хорошо подходить для программирования их (или, по крайней мере, их основных вычислительных ядер) на языке НОРМА. Ничуть не проиграв по скорости выполнения получившихся программ, компилятор программ с языка НОРМА полностью взял на себя задачу по автоматическому распараллеливанию выходной программы и успешно ее выполнил.

В то же время видно, что для эффективного выполнения на графических процессорах таких задач необходимо прибегать к более тонкому ручному программированию или, когда это возможно, — использовать специальные библиотеки.

7. Заключение. Применение языка НОРМА для решения прикладных задач из области математической физики может дать не только более высокий уровень абстракции при составлении программы прикладным специалистом (что, в свою очередь, снижает время разработки и количество возможных допустимых ошибок в программе), но и позволяет получать эффективные исполняемые программы для различных видов параллельных архитектур. При этом весьма важно, что сама прикладная программа остается неизменной (за исключением, возможно, внешних функций на целевом языке программирования). В зависимости от заданной архитектуры вычислителя компилятор автоматически решает необходимые задачи по распараллеливанию программы, распределению вычислений, балансировке вычислительной нагрузки и обмену данными между вычислительными устройствами.

Работа выполнена при финансовой поддержке гранта РФФИ № 18–01–00131-а.

СПИСОК ЛИТЕРАТУРЫ

1. *Воеводин В.В.* Математические модели и методы в параллельных процессах. М.: Наука, 1986.
2. *Воеводин В.В., Воеводин Вл.В.* Параллельные вычисления. С.-Петербург: БХВ-Петербург, 2002.
3. OpenACC. http://openacc.org.

4. DVM-система. URL: http://dvm-system.org.
5. *Андрианов А.Н., Бугеря А.Б., Ефимкин К.Н., Задыхайло И.Б.* НОРМА. Описание языка. Рабочий стандарт. Препринт ИПМ им. М.В. Келдыша РАН. № 120. М., 1995.
6. *McGraw J.R., Skedzielewski S.K., Allan S.J., Oldehoeft R.R., Glauert J., Kirkham C., Noyce B., Thomas R.* Sisal: Streams and iterations in a single assignment language. Language Reference Manual, Version 1.2. Livermore: Lawrence Livermore National Laboratory, 1985.
7. *Касьянов В.Н., Стасенко А.П.* Язык программирования Sisal 3.2 // Методы и инструменты конструирования программ. Новосибирск: Институт систем информатики имени А.П. Ершова СО РАН. 2007. 56–134.
8. *Chamberlain B.L., Choi S.-E., Lewis E.C., Lin C., Snyder L., Weathersby W.D.* ZPL: a machine independent programming language for parallel computers // IEEE Trans. Software Eng. 2000. **26**, N 3. 197–211.
9. *Задыхайло И.Б.* Организация циклического процесса счета по параметрической записи специального вида // Журн. выч. мат. и мат. физ. 1963. **3**, № 2. 337–357.
10. *Андрианов А.Н., Ефимкин К.Н., Задыхайло И.Б.* Непроцедурный язык для решения задач математической физики // Программирование. 1991. № 2. 80–95.
11. *Задыхайло И.Б., Ефимкин К.Н.* Содержательные обозначения и языки нового поколения // Информационные технологии и вычислительные системы. 1996. № 2. 46–58.
12. *Андрианов А.Н., Бугеря А.Б., Гладкова Е.Н., Ефимкин К.Н., Колударов П.И.* Простые вещи // Суперкомпьютеры. 2014. № 2. 58–61.
13. Система НОРМА. http://www.keldysh.ru/pages/norma.
14. *Андрианов А.Н., Бугеря А.Б., Ефимкин К.Н., Колударов П.И.* Модульная архитектура компилятора языка Норма+. Препринт ИПМ им. М.В. Келдыша РАН. № 64. М., 2011.
15. *Бугеря А.Б.* Диалоговая отладка параллельных программ: распределенная схема взаимодействующих компонентов // Программирование. 2008. № 3. 42–49.
16. *Андрианов А.Н.* Система Норма. Разработка, реализация и использование для решения задач математической физики на параллельных ЭВМ. Автореф. дис. докт. физ.-мат. наук. М., 2001.
17. *Андрианов А.Н., Андрианова Е.А.* Организация циклического процесса по непроцедурной записи // Программирование. 1996. № 4. 62–72.
18. *Андрианов А.Н.* Синтез параллельных и векторных программ по непроцедурной записи в языке Норма. Дис. канд. физ.-мат. наук. 1990.
19. *Васильев М.М., Ефимкин К.Н., Иванова В.Н.* О применении метода гидродинамических потенциалов к задаче обтекания тела вязкой жидкостью // Математическое моделирование. 1994. **6**, № 10. 57–65.
20. *Андрианов А.Н., Гусева Г.Н., Задыхайло И.Б.* Применение языка Норма для расчета дозвукового течения вязкого газа // Математическое моделирование. 1999. **11**, № 9. 45–53.
21. *Андрианов А.Н., Жохова А.В., Четверушкин Б.Н.* Использование параллельных алгоритмов для расчетов газодинамических течений на нерегулярных сетках // Прикладная математика и информатика. М.: Изд-во Моск. ун-та, 2000. 68–76.
22. *Андрианов А.Н., Ефимкин К.Н.* Использование системы Норма для решения вычислительных задач на многопроцессорных системах с распределенной памятью // Вычислительные методы и программирование. 2000. **1**. 45–54.
23. *Andrianov A.N., Efimkin K.N., Levashov V.Yu., Shishkova I.N.* The Norma language application to solution of strong nonequilibrium transfer process problem with condensation of mixtures on the multiprocessor system. Vol. 2073. Heidelberg: Springer, 2001. 502–510.
24. *Андрианов А.Н.* Применение языка Норма для решения задач на вложенных сетках // Вычислительные методы и программирование. 2002. **3**. 1–10.
25. *Андрианов А.Н., Березин А.В., Воронцов А.С., Ефимкин К.Н., Марков М.Б.* Моделирование электромагнитных полей радиационного происхождения на многопроцессорных вычислительных системах // Математическое моделирование. 2008. **20**, № 3. 98–114.
26. NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html.
27. Гибридный вычислительный кластер К-100. https://ckp.kiam.ru/?hard.
28. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda.

Поступила в редакцию
8.05.2019

Distribution of Computations in Hybrid Computing Systems when Translating NORMA Language Programs

A. N. Andrianov [1], **T. P. Baranova** [2], **A. B. Bugerya** [3], **and K. N. Efimkin** [4]

[1] *Keldysh Institute of Applied Mathematics, Russian Academy of Sciences; Miusskaya ploshchad' 4, Moscow, 125047, Russia; Dr. Sci., Leading Scientist, e-mail: andrianovan52@mail.ru*

[2] *Keldysh Institute of Applied Mathematics, Russian Academy of Sciences; Miusskaya ploshchad' 4, Moscow, 125047, Russia; Scientist, e-mail: bart1950@yandex.ru*

[3] *Keldysh Institute of Applied Mathematics, Russian Academy of Sciences; Miusskaya ploshchad' 4, Moscow, 125047, Russia; Ph.D., Senior Scientist, e-mail: shurabug@yandex.ru*

[4] *Keldysh Institute of Applied Mathematics, Russian Academy of Sciences; Miusskaya ploshchad' 4, Moscow, 125047, Russia; Ph.D., Senior Scientist, e-mail: bigcrocodile@yandex.ru*

Received May 8, 2019

Abstract: The methods of computational load distribution when translating programs from the nonprocedural (declarative) NORMA language into executable programs for various parallel architectures are discussed. Some brief characteristics of the NORMA language and the main features of the compiler for programs in NORMA language are given. The methods of automatic distribution of computational load when generating executable programs of the following types are described: OpenMP, NVIDIA CUDA, MPI+OpenMP, and MPI+OpenMP+NVIDIA CUDA. The problem of dynamic computational load balancing arising in the case of the heterogeneous computing environment MPI+OpenMP+NVIDIA CUDA is considered and a method of solving it is proposed. The results of practical application of the compiler for the programs in NORMA language for solving two different mathematical problems are given and the performance of the resulting executable programs is estimated for various parallel architectures.

Keywords: parallel programming, programming automation, nonprocedural specifications, hybrid architectures, load balancing, NORMA language.

References

1. V. V. Voevodin, *Mathematical Models and Methods in Parallel Processes* (Nauka, Moscow, 1986) [in Russian].

2. V. V. Voevodin and Vl. V. Voevodin, *The Parallel Computing* (BHV-Petersburg, St. Petersburg, 2002) [in Russian].

3. OpenACC. http://openacc.org. Cited June 10, 2019.

4. DVM-system. URL: http://dvm-system.org. Cited June 10, 2019.

5. A. N. Andrianov, A. B. Bugerya, K. N. Efimkin, and I. B. Zadykhailo, *Norma. Language Description. Draft*, Preprint No. 120 (Keldysh Institute of Applied Mathematics, Moscow, 1995).

6. J. R. McGraw, S. K. Skedzielewski, S. J. Allan, et al., *Sisal: Streams and Iterations in a Single Assignment Language*, Language Reference Manual, Version 1.2. (Lawrence Livermore Nat. Lab., Livermore, 1985).

7. V. N. Kasyanov and A. P. Stasenko, "Sisal 3.2 Programming Language," in *Tools and Techniques of Program Construction* (Ershov Institute of Informatics Systems, Novosibirsk, 2007), pp. 56–134.

8. B. L. Chamberlain, S.-E. Choi, E. C. Lewis, et al., "ZPL: A Machine Independent Programming Language for Parallel Computers," IEEE Trans. Softw. Eng. **26** (3), 197–211 (2000).

9. I. B. Zadykhailo, "The Organization of a Cyclical Computing Process Using a Parametric Representation of Special Form," Zh. Vychisl. Mat. Mat. Fiz. **3** (2), 337–357 (1963) [USSR Comput. Math. Math. Phys. **3** (2), 442–468 (1963)].

10. A. N. Andrianov, K. N. Efimkin, and I. B. Zadykhailo, "Nonprocedural Language for Mathematical Physics," Programmirovanie, No. 2, 80–95 (1991) [Program. Comput. Software **17** (2), 10–22 (1992)].

11. I. B. Zadykhailo and K. N. Efimkin, "Substantive Notation and New Generation Languages," Inform. Tekhnol. Vychisl. Sistemy, No. 2, 46–58 (1996).

12. A. N. Andrianov, A. B. Bugerya, E. N. Gladkova, et al., "Simple Things," Superkomp'utery, No. 2, 58–61 (2014).

13. The NORMA System. http://www.keldysh.ru/pages/norma. Cited June 10, 2019.

14. A. N. Andrianov, A. B. Bugerya, K. N. Efimkin, and P. I. Koludarov, *Modular Architecture of NORMA+ compiler*, Preprint No. 64 (Keldysh Institute of Applied Mathematics, Moscow, 2011).

15. A. B. Bugerya, "Interactive Debugging of Parallel Programs: Distributed Scheme of Interacting Components," Programmirovanie, No. 3, 42–49 (2008) [Program. Comput. Software **34** (3), 154–159 (2008)].

16. A. N. Andrianov, *The Norma System. Development, Implementation and Usage for Solving Problems of Mathematical Physics on Parallel Computers*, Doctoral Dissertation in Mathematics and Physics (Keldysh Institute of Applied Mathematics, Moscow, 2001).

17. A. N. Andrianov and E. A. Andrianova, "Organization of Cyclic Process over Nonprocedural Structure," Programmirovanie, No. 4, 62–72 (1996) [Program. Comput. Software **22** (4), 203–212 (1996)].

18. A. N. Andrianov, *The Synthesis of Parallel and Vector Programs by the Nonprocedural Norma Specification*, Candidate's Dissertation in Mathematics and Physics (Keldysh Institute of Applied Mathematics, Moscow, 1990).

19. M. M. Vasiliev, K. N. Efimkin, and V. N. Ivanova, "On the Application of the Hydrodynamic Potentials Method to the Viscous Fluid Flow Problem," Mat. Model. **6** (10), 57–65 (1994).

20. A. N. Andrianov, G. N. Guseva, and I. B. Zadykhailo, "Norma Language Application in Calculation of Viscous Gas Subsonic Flow Model," Mat. Model. **11** (9), 45–53 (1999).

21. A. N. Andrianov, A. V. Zhokhova, and B. N. Chetverushkin, "Application of Parallel Algorithms for Gas-dynamic Flows on Irregular Grids," in *Applied Mathematics and Informatics* (Mosk. Gos. Univ., Moscow, 1973), pp. 68–76.

22. A. N. Andrianov and K. N. Efimkin, "The Use of the Norma System for Solving Computational Problems with Multiprocessor Computers with Distributed Memory," Vychisl. Metody Programm. **1**, 45–54 (2000).

23. A. N. Andrianov, K. N. Efimkin, V. Yu. Levashov, and I. N. Shishkova, "The NORMA Language Application to Solution of Strong Nonequilibrium Transfer Processes Problem with Condensation of Mixtures on the Multiprocessor System," in *Lecture Notes in Computer Science* (Springer, Heidelberg, 2001), Vol. 2073, pp. 502–510.

24. A. N. Andrianov, "Application of the Norma Language for Solving Problems with Nested Grids," Vychisl. Metody Programm. **3** (2), 1–10 (2002).

25. A. N. Andrianov, A. V. Berezin, A. S. Vorontsov, et al., "The Radiational Electromagnetic Fields Modeling at the Multiprocessor Computing Systems," Mat. Model. **20** (3), 98–114 (2008).

26. NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html. Cited June 10, 2019.

27. K-100 Hybrid Computing Cluster. https://ckp.kiam.ru/?hard. Cited June 10, 2019.

28. CUDA Toolkit Documentation v10.1.168. https://docs.nvidia.com/cuda. Cited June 10, 2019.