Создание эффективных параллельных программ на Фортране с использованием техники переписывающих правил

А.Е. Дорошенко[1], К.А. Жереб[1], Ю.М. Тырчак[1], А.О. Хатнюк[2]

[1] *Институт программных систем НАН Украины, Проспект Академика Глушкова 40, Киев, Украина*
[2] *Факультет кибернетики, Киевский национальный университет имени Тараса Шевченка, Владимирская 60, Киев, Украина*

doroshenkoanatoliy2@gmail.com, zhereb@gmail.com, chacke@gmail.com, anutahat@gmail.com

Аннотация. *В работе описан подход к распараллеливанию программ на Фортране, основанный на использовании техники переписывающих правил и алгебраических моделей программ. Ранее разработанный авторами инструментарий переписывающих правил был расширен с целью поддержки целевого языка Фортран. На примере конкретной задачи описаны преобразования последовательной программы в программу для параллельной системы с общей памятью, а также оптимизирующие преобразования, повышающие производительность программы. Проведено сравнение предложенного подхода с другими способами распараллеливания программ на Фортране, включающими использование распараллеливающего компилятора и платформы OpenMP. Результаты проведенных экспериментов показали эффективность разработанных преобразований.*

Ключевые слова

Фортран, OpenMP, Intel Fortran Compiler, техника переписывающих правил.

1 Введение

Фортран является одним из первых языков программирования, тем не менее, этот язык до сих пор широко используется, в особенности для решения физических, химических и других естественнонаучных задач [1]. Популярность Фортрана объясняется его относительной простотой, близостью исходного кода к математической формулировке задачи, эффективностью генерируемого бинарного кода. Кроме того, за более чем 50 лет существования языка накоплено огромное программное обеспечение в виде готовых программ и библиотек для решения разнообразных задач.

Большая часть существующего кода была разработана для последовательных вычислительных систем. Современное развитие высокопроизводительных вычислительных систем дает потенциальную возможность существенно повысить производительность таких программ за счет использования параллельных вычислений [1–2]. Однако полное переписывание существующих программ для новых аппаратных архитектур практически невозможно из-за их большого объема. Поэтому актуальной является задача распараллеливания существующих программ на Фортране, при этом особенно важно как можно больше автоматизировать процесс распараллеливания.

В данной работе описан подход к распараллеливанию программ на Фортране, основанный на использовании техники переписывающих правил и алгебраических моделей программ. Ранее разработанный авторами инструментарий переписывающих правил [3–5] был расширен с целью поддержки целевого языка Фортран. На примере конкретной задачи (алгоритм Гаусса решения систем линейных уравнений) описаны преобразования последовательной программы в программу для параллельной системы с общей памятью, а также оптимизирующие преобразования, повышающие производительность программы. Проведено сравнение предложенного подхода с другими способами распараллеливания программ на Фортране, а именно использованием распараллеливающего компилятора и платформы OpenMP [6]. Измерения производительности показали эффективность преобразований.

Данная работа описывает продолжение исследований авторов, начатых в работах [3–5]. В частности, использованный подход использования техники переписывающих правил и алгебраических моделей программ аналогичен описанному в работе [3]. Также были использованы разработанные ранее преобразования, в частности для работы с локальными копиями данных [5]. Особенностью данной работы является использование предложенного подхода для нового целевого языка – Фортрана.

Материал работы организован следующим образом. В разделе 2 приведено общее описание предложенного подхода. Раздел 3 описывает инструментальные средства, использованные для распараллеливания программ на Фортране. Раздел 4 посвящен подробному рассмотрению преобразований на примере конкретной задачи. Результаты измерения производительности преобразованных программ приведены в разделе 5. Раздел 6 содержит выводы и направления дальнейшей работы.

2 Общее описание подхода

В работе используется следующий подход для разработки эффективных параллельных программ для систем с общей памятью. В качестве входных данных используется исходный код последовательной программы на Фортране, описывающий алгоритм решения задачи. Программный код представляется в виде высокоуровневой модели с использованием алгебры алгоритмов Глушкова [7]. Далее к этому высокоуровневому представлению применяются преобразования, представленные в виде переписывающих правил. Преобразования могут быть направлены на переход от последовательной к параллельной версии программы (распараллеливающие преобразования), а также на повышение производительности программы (оптимизирующие преобразования). После применения преобразований к высокоуровневой модели программы, используется генератор кода для создания исходного кода (также на Фортране) преобразованной программы.

Переход от исходного кода к высокоуровневой модели программы осуществляется в два этапа. На первом этапе работает синтаксический анализатор (парсер), который по исходному коду программы строит ее низкоуровневую (синтаксическую) модель, близкую к дереву синтаксического разбора (abstract syntax tree, AST). Далее к этой модели применяются *паттерны*, т.е. преобразования специального вида, устанавливающие соответствие между алгебраическим оператором и фрагментом кода, который его реализует (детальное описание паттернов приведено в [3]). Паттерны зависят как от языка (например, C или Фортран), так и от предметной области (например, для линейной алгебры характерны операторы работы с векторами и матрицами). Использование высокоуровневой алгебраической модели позволяет разработчику лучше воспринимать структуру алгоритма, при этом не отвлекаясь на технические детали его реализации. Однако при необходимости можно работать также на уровне низкоуровневой синтаксической модели, например, осуществляя преобразования, специфические для целевого языка или платформы реализации.

Многие этапы процесса распараллеливания выполняются в автоматическом или автоматизированном режиме. Так, переход от исходного кода программы к ее высокоуровневой алгебраической модели, а также переход в обратном направлении (генерация кода) выполняется автоматически, для заданной предметной области и языка программирования. Применение преобразований программ, представленных в виде переписывающих правил, также является автоматизированным: от пользователя требуется лишь указать, какие преобразования следует применять и на каком участке кода они должны действовать. Наиболее сложной частью описанного процесса, выполняемой вручную, является создание переписывающих правил, описывающих преобразование. Тем не менее, многие из таких правил применимы для различных программ, что способствует их повторному использованию.

3 Инструментальные средства

В данной работе используется разработанный ранее инструментарий для преобразования программ, основанный на системе переписывающих правил Termware [8–9]. Кроме собственно системы переписывающих правил, инструментарий также содержит средства работы с исходным кодом на различных языках (синтаксические анализаторы и генераторы кода), а также графический интерфейс пользователя для работы с моделями программ и редактирования переписывающих правил.

Система Termware предназначена для описания преобразования над термами, т.е. выражениями вида *f(t1,…,tn)*. Для задания преобразований используются правила Termware, т.е. конструкции вида

source [condition] -> destination [action]

Здесь *source* – исходный терм (образец для поиска), *condition* – условие применения правила, *destination* – преобразованный терм, *action* – дополнительное действие при срабатывании правила. Каждый из 4

компонентов правила может содержать переменные (которые записываются в виде *$var*), что обеспечивает общность правил. Компоненты *condition* и *action* являются необязательными. Они могут исполнять произвольный процедурный код, в частности использовать дополнительные данные о программе.

В [3–5] система Termware использовалась для работы с программами на языках C# и C for CUDA. Для использования языка Фортран был разработан синтаксический анализатор и генератор кода этого языка. Синтаксический анализатор был создан на основе открытого проекта Open Fortran Parser [10]. Исходный код этого проекта был расширен с целью генерации дерева синтаксического разбора, представленного в виде термов системы Termware. Кроме того, был разработан компонент для исправления некоторых особенностей старых версий языка Фортран, которые не соответствуют грамматике Open Fortran Parser. (Особенностью программ на Фортране является наличие большого числа диалектов и форм языка, которые часто смешиваются в пределах одной программы в процессе ее развития. Вместо того, чтобы модифицировать грамматику языка, которая и так является достаточно сложной, было принято решение использовать аналог препроцессора для исправления некоторых неподдерживаемых конструкций).

4 Распараллеливающие и оптимизирующие преобразования

Для распараллеливания программ на Фортране для систем с общей памятью в данной работе используется стандарт OpenMP [6]. Платформа OpenMP позволяет создавать многопоточные параллельные программы. Эта платформа позволяет модифицировать существующие последовательные программы на Фортране, добавляя специальные *директивы*. В частности, в данной работе используются директивы для распараллеливания циклов *!$OMP PARALLEL* и *!$OMP DO*. Кроме того, OpenMP предоставляет набор библиотечных функций, в частности для получения информации о среде выполнения параллельной программы и для синхронизации потоков.

4.1 Структура алгоритма

В данной работе процесс распараллеливания последовательной программы рассмотрен на примере задачи решения систем линейных уравнений методом Гаусса. Последовательный код программы с помощью разработанного синтаксического анализатора языка Фортран переведен в низкоуровневую синтаксическую модель, которая затем преобразована в высокоуровневую алгебраическую модель с использованием техники переписывающих правил. Существенный для распараллеливания фрагмент кода (прямой ход алгоритма) моделируется следующим алгебраическим выражением:

```
DoCnt(K,1,N-1,
  FindMaxElement,
  CheckDetZero,
  SwapMaxRowColumn,
  CalculateRow(K),
  UpdateElements )
```

Здесь использован оператор *DoCnt(var,start,end,body)*, описывающий цикл со счетчиком. Заметим, что данный оператор является синонимом оператора *ForCnt(var,start,end,body)*, рассмотренного в [3]. Введение такого оператора обусловлено тем, что для программистов на языке Фортран более естественно именование циклов с ключевым словом Do, тогда как для языков с синтаксисом, близким к языку C (в том числе C++, Java, C#) используется ключевое слово для цикла For. Такое использование оператора предназначено лишь для синтаксических удобств, по всем свойствам данные операторы эквивалентны.

В приведенном фрагменте программы распараллеливанию подлежат два оператора. Первый из них, FindMaxElement, определяет максимальный элемент в подматрице и имеет следующий вид:

```
FindMaxElement=   DoCnt(I,K,N,
                      DoCnt(J,K,N
                          GetMaxIndex(Abs(A(I,J)),I,J,MAX,IMAX,JMAX)))
```

Вторым оператором, который подлежит распараллеливанию, является UpdateElements – вычисление обновленных элементов матрицы:

```
UpdateElements=   DoCnt(I,K+1,N,
                        Assign(S,A(I,K)),
                        DoCnt(J,K,N+1,
                              Update(A(I,J),S)))
```

Остальные операторы (CheckDetZero, SwapMaxRowColumn, CalculateRow) имеют меньшую вычислительную сложность (линейную, а не квадратичную относительно размера матрицы) и поэтому их распараллеливание является неэффективным и может привести даже к потерям производительности за счет накладных расходов.

4.2 Распараллеливание цикла

Начнем распараллеливание с оператора UpdateElements. Он имеет вид простого цикла, итерации которого являются независимыми по данным. Поэтому его распараллеливание осуществляется преобразованием последовательного цикла в параллельный:

DoCnt($var,$start,$end,$body,_MARK_Parallel)->ParallelDoCnt($var,$start,$end,$body)

Заметим, что здесь исходный цикл помечается меткой *_MARK_Parallel*, которая позволяет выделить циклы, которые должны быть распараллелены. Оператор *ParallelDoCnt* является элементом высокоуровневой модели, описывающим параллельный цикл. Его реализация для платформы OpenMP описывается следующим паттерном:

ParallelDoCnt($var,$start,$end,$body)->OmpParallelDo(DoCnt($var,$start,$end,$body))

Здесь OmpParallelDo – очередной оператор, специфичный для платформы OpenMP, но не зависящий от языка реализации. В частности, для языка Фортран этот оператор превращается в пару директив

```
!$OMP PARALLEL DO
…
!$OMP END PARALLEL DO
```

Тогда как для языка C тот же оператор имеет вид *#pragma omp parallel for*. Таким образом, достигается независимость используемых преобразований от целевого языка.

При распараллеливании оператора UpdateElements есть еще одна особенность. Внутри оператора используется временная переменная S. Эта переменная используется внутри каждой итерации цикла, и поэтому она должна быть обозначена как частная для каждого потока. Однако компилятор OpenMP не может этого определить автоматически, поэтому следует добавить параметр PRIVATE(S). Подобное преобразование может быть выполнено на уровне низкоуровневой синтаксической модели, так как оно использует особенность конкретной платформы реализации.

4.3 Редукция

Распараллеливание оператора FindMaxElement является более сложным, поскольку между итерациями присутствует зависимость по данным. Общая форма оператора соответствует случаю редукции, т.е. вычислению некоторой величины на каждом потоке и затем объединения ее в общее значение. Для таких случаев в OpenMP предусмотрен параметр REDUCTION. Однако он работает только для предопределенных операторов. В частности, определение максимума может быть осуществлено с помощью средств OpenMP, однако для определения соответствующих максимальному значению индексов уже нельзя использовать стандартные средства.

Поэтому в данном случае используются специальные преобразования для перехода к многопоточной версии программы (рассмотренные ранее в [5]). Оператор FindMaxElement представляется в виде комбинации операторов, работающих с локальными копиями данных, и затем оператора, объединяющего их действия:

FindMaxElement=FindMaxElementLoc1… *FindMaxElementLocTN*FindMaxElementReduct*

Каждый из локальных операторов FindMaxElementLoc1,…, FindMaxElementLocTN работает со своей копией данных, поэтому они могут исполняться независимо на различных потоках. После завершения работы всех потоков производится редукция локальных результатов, т.е. из локальных максимумов и соответствующих элементов определяется глобальный максимум.

Рассмотрим более подробно оператор FindMaxElement: он состоит из двойного цикла по элементам матрицы, в котором применяется оператор

GetMaxIndex(Val,I,J, Max,Imax,Jmax)=If(Val>Max,Assign(Max,Val);Assign(Imax,I);Assign(Jmax,J))

Заметим, что оператор GetMaxIndex можно рассматривать как бинарный над множеством Real*Integer*Integer (т.е. содержащим значение и два индекса), при этом оператор является коммутативным и ассоциативным. Поэтому для операторов FindMaxElementLoc и FindMaxElementReduct можно использовать тот же оператор GetMaxIndex.

Таким образом, для распараллеливания оператора FindMaxElement используются следующие правила:

1. FindMaxElement -> Parallel(GetThreadParams; FindMaxElementLoc); FindMaxElementReduct
2. GetThreadParams->Assign(theadnum,GetThreadNum); Assign(theads,GetThreads);
3. FindMaxElementLoc-> ParallelDoCnt(I,K,N, DoCnt((J,K,N, GetMaxIndex(Abs(A(I,J)),I,J, MAXloc(threadnum),IMAXloc(threadnum),JMAX(threadnum)))
4. FindMaxElementReduct-> DoCnt(I,1,threads,GetMaxIndex(MAXloc(i), IMAXloc(i), JMAXloc(i), MAX,IMAX,JMAX)

Здесь правило 1 описывает замену последовательного оператора FindMaxElement на последовательность параллельно исполняемых операторов FindMaxElementLoc, за которым следует последовательно исполняемый оператор FindMaxElementReduct. Правило 2 описывает техническую деталь – сохранение параметров текущего потока (номер потока и общее количество потоков), необходимую для создания локальных копий данных и последующего их объединения. Правила 3 и 4 описывают операторы FindMaxElementLoc и FindMaxElementReduct – они оба последовательно применяют оператор GetMaxIndex, в первом случае к подмножеству матрицы, во втором – к локальным данным потоков.

4.4 Оптимизация доступа к памяти

Описанные выше преобразования распараллеливания выполнялись на высоком уровне алгебраической модели алгоритма. В частности, они описывали те части алгоритма которые могут выполняться в параллельном режиме (и которые могут дать ощутимое ускорение), а также особенности распределения вычислений по потокам. Однако для производительности параллельной программы большое значение могут иметь также достаточно низкоуровневые детали реализации, в частности связанные с особенностями доступа к памяти. В работе [4] авторами уже исследовались преобразования, повышающие эффективность работы с памятью для графических ускорителей, при этом было замечено, что подобные преобразования могут иметь значительно больший эффект на производительность, чем преобразования только алгоритма.

Для рассматриваемой задачи (алгоритма Гаусса) также можно существенно повысить производительность за счет более эффективной работы с памятью (хотя используемые преобразования отличаются от рассмотренных в [4], т.к. речь идет о другой платформе – многоядерные системы с общей памятью вместо графических ускорителей). В процессе измерения производительности (см. раздел 5), было замечено, что для определенных размеров системы N (кратных 256) наблюдается резкое увеличение времени работы. Эта особенность связана с организацией памяти: при таких размерах матрицы часто используемые элементы попадают в одну позицию в кэш-памяти, поэтому каждый доступ к такому элементу вызывает обращение к оперативной памяти вместо намного более быстрого обращения к кэш-памяти.

Для устранения этой проблемы было использовано достаточно простое преобразование. При объявлении матрицы используется не ее реальный размер, а размер на 1 больший. «Лишние» элементы матрицы никак не используются в алгоритме, однако позволяют разместить данные в памяти таким образом, что повышается эффективность их использования в кэш-памяти. Преобразование применяется к низкоуровневой модели программы и описывается следующими правилами:

1. [Declaration(N,Integer,$val):$next]->[Declaration(N,Integer,$val): [Declaration(MN,Integer,$val+MShift($val)): $next]]
2. MShift($val) [$val%32==0]->1 !->0
3. Declaration(A,Array(Double,[N,N+1]))-> Declaration(A,Array(Double,[MN,MN+1]))
4. Procedure($name,[N:[A:$next]])-> Procedure($name,[N:[MN:[A:$next]]])
5. [Parameter(N,Integer,In):$next]->[Parameter(N,Integer,In):[Parameter(MN,Integer,In):$next]]

6. Call($name,[N:[A:$next]])-> Call($name,[N:[MN:[A:$next]]])

Здесь правило 1 добавляет определение нового параметра – MN, который определяет размер матрицы в памяти (а не реальный размер N, используемый в алгоритме). Правило 2 необходимо для того, чтобы исправление размера памяти срабатывало лишь тогда, когда оно необходимо. В частности, если использовать увеличение размера на 1 всегда, для размеров матрицы N=256K–1 такое исправление приведет, наоборот, к возникновению конфликтов по доступу к памяти. Правило 3 исправляет определение матрицы, так что оно использует новый размер MN. Правила 4–6 позволяют передать новый параметр MN во все процедуры, где используется матрица A.

Заметим, что правила 4–6 применяются многократно, для каждого определения процедуры (правила 4–5) и для каждого ее вызова (правило 6). Это демонстрирует преимущество техники переписывающих правил: преобразование автоматически применяется в нужных местах по всей программе, при этом исключается достаточно значительная ручная работа, а также предотвращаются возможные ошибки.

5 Измерения производительности и сравнение с другими технологиями распараллеливания

Для оценки эффективности предложенного подхода были проведены измерения производительности исходной последовательной программы и преобразованных программ. Сравнивались 4 версии программы решения систем линейных уравнений методом Гаусса: последовательная (SEQ), с распараллеленным оператором UpdateElements (PAR1), с распараллеленными операторами FindMaxElement и UpdateElements (PAR2), а также с обоими операторами и модификацией размера матрицы в памяти (MEM). Измерения проводились для размеров системы от 256 до 2048. Использовалась вычислительная система с 4 вычислительными ядрами. В табл. 1 приведены результаты измерения времени исполнения (в секундах).

Табл.1. Сравнение производительности последовательной и преобразованных программ.

Размерность	SEQ	PAR1	PAR2	MEM
256	0,056	0,054	0,030	0,020
512	0,98	0,59	0,33	0,22
768	2,56	1,74	0,90	0,85
1024	26,72	21,78	15,67	2,40
1280	28,92	25,57	18,19	4,77
1536	101,32	85,09	62,53	8,62
1792	116,39	110,50	79,74	13,74
2048	237,56	205,23	150,03	21,11

Как видно из результатов измерений, каждое из примененных преобразований повышает производительность программы, хотя их эффективность различается. Для сравнения на рис. 1 приведены графики зависимости коэффициента ускорения от размерности системы для преобразованных программ PAR1, PAR2 и MEM.

На рис. 1 видны две области в зависимости от размеров данных. Для N≤768 данные полностью помещаются в кэш-память. Это значит, что последовательная программа является относительно эффективной, и эффекты от распараллеливания различных участков программы более заметны. Распараллеливание оператора UpdateElements в данной области повышает производительность программы в 1,5 раза, добавление к этому распараллеливания оператора FindMaxElement приводит к ускорению в 2,5-3 раза. Преобразование размеров матрицы в данном случае малоэффективно, и ускорение программы MEM незначительно превышает ускорение PAR2.

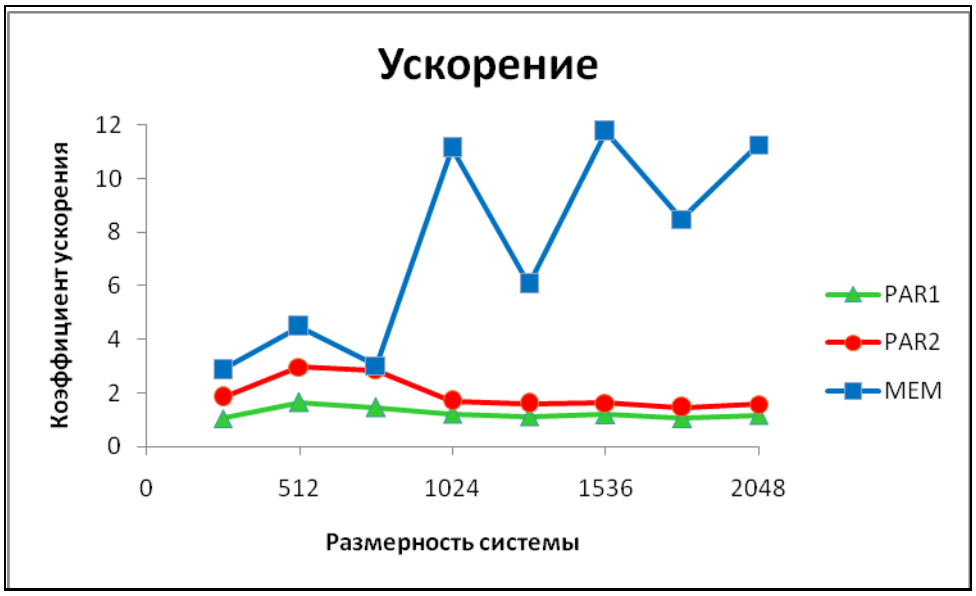

Рис. 1. Ускорение в зависимости от размеров входных данных.

Для N≥1024 ситуация меняется – теперь данные не помещаются целиком в кэш-память, и эффективность работы с памятью приобретает ключевое значение. Ускорение программы PAR1 падает до 1,1-1,2; существенно уменьшается ускорение программы PAR2 (до 1,5). Программа MEM, наоборот, становится существенно более эффективной: ее коэффициент ускорения достигает значений 6-11. Заметим, что коэффициент ускорения в этом случае превосходит количество ядер – 4. Это связано с тем, что последовательная программа использует память менее эффективно, что приводит к дополнительным задержкам. Кроме того, наблюдается значительный разброс значений коэффициента ускорения, в зависимости от степени 2 в разложении N на простые множители. Это демонстрирует сложный характер зависимости реального времени исполнения программы от размера входных данных, в отличие от теоретической оценки сложности алгоритма $O(N^3)$.

Полученные данные позволяют сравнить эффективность предложенного подхода с существующими технологиями распараллеливания. В качестве таких технологий были выбраны распараллеливающие компиляторы (Intel Fortran Compiler [11]) и использование OpenMP вручную (без дополнительных средств, таких как техника переписывающих правил). Преимуществами этих технологий являются высокая степень автоматизации и внесение минимальных изменений в исходный код программы. В частности, использование Intel Fortran Compiler позволяет полностью избежать внесения изменений в исходный код. Кроме того, распараллеливание является полностью автоматизированным и требует от пользователя лишь указания правильных ключей компилятора. Однако использование конкретного компилятора, а не стандартных возможностей языка, может быть нежелательным.

Производительность программы решения систем линейных уравнений методом Гаусса при использовании Intel Fortran Compiler оказывается такой же, как и у программы PAR2. Отсюда видно, что компилятор способен распознать параллелизм на уровне алгоритмических операций и эффективно использовать его. Однако преобразования, повышающие эффективность доступа к памяти, компилятором не осуществляются, поэтому для больших значений N, кратных 256, эффективность автоматически распараллеленной программы оказывается существенно ниже, чем у программы MEM. Заметим, что возможности автоматического распараллеливания могут поставить под сомнение целесообразность использования распараллеливающих преобразований, которые меняют исходный код. Однако явное выделение параллельных конструкций в исходном коде может быть полезно для тех случаев, которые не поддерживают распараллеливающие компиляторы, например при использовании платформ с распределенной памятью.

В случае использования OpenMP вручную в код программы вносятся минимальные изменения (декларативные директивы), например, такие как в программе PAR1. Такие изменения позволяют с минимальными усилиями распараллеливать программы в достаточно простых случаях. Однако, как показывает пример программы PAR2, декларативных средств OpenMP для решения некоторых задач недостаточно. В этом случае приходится использовать библиотечные вызовы, поэтому код параллельной программы существенно отличается от исходной последовательной. Преимущество предложенного подхода в этом случае заключается в использовании высокоуровневых алгебраических моделей, которые затем преобразуются в исходный код в автоматизированном режиме. Таким образом, сложная конструкция в исходном коде может быть представлена

в виде достаточно простой комбинации алгебраических операторов, что упрощает понимание и изменение программы.

6 Заключение

В работе описано применение техники переписывающих правил и алгебраических моделей программ для создания эффективных параллельных программ на языке Фортран. Ранее разработанная авторами инструментальная система переписывающих правил была расширена для поддержки программ на Фортране. На примере конкретной задачи описано применение распараллеливающих преобразований, а также преобразований для более эффективной работы с памятью. Измерения производительности показывали высокую эффективность преобразований: для наиболее эффективного варианта программы достигнут коэффициент ускорения от 3 до 11 по сравнению с последовательной программой (в зависимости от размера входных данных) на процессоре, содержащем всего 4 ядра.

Данная работа описывает первый этап в продолжающихся исследованиях в области распараллеливания программ на Фортране с использованием алгебраических методов и техники переписывающих правил. В дальнейшем планируется исследовать распараллеливание программ для систем с распределенной памятью (например, с использованием MPI). Также будут продолжены исследования гетерогенных систем, в частности содержащих графические ускорители (GPU), и возможности их использования в программах на Фортране. Кроме того, планируется использование предложенного подхода и разработанных инструментальных средств для повышения эффективности более сложных практически значимых программ.

Литература

[1] Krste Asanovic, et. al., The Landscape of Parallel Computing Research: A View from Berkeley, Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December 18, 2006. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[2] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The impact of multicore on math software. *In B. Kagstrom, E. Elmroth, J. Dongarra, and J.Wasniewski, editors, Applied Parallel Computing. State of the Art in Scientific Computing*, PARA 2006, Umea, Sweden, June 2006, LNCS 4699:1–10, 2007.

[3] Андон Ф.И., Дорошенко А.Е., Жереб К.А. Программирование высокопроизводительных параллельных вычислений: формальные модели и графические ускорители. *Кибернетика и системный анализ,* 4:176–187, 2011.

[4] Дорошенко А.Е., Жереб К.А. Разработка высокопараллельных приложений для графических ускорителей с использованием переписывающих правил. *Проблемы программирования*, 3:3–18, 2009.

[5] Дорошенко А.Е., Жереб К.А., Яценко Е.А. Об оценке сложности и координации вычислений в многопоточных программах. *Проблемы программирования*, 2:41–55, 2007.

[6] OpenMP Specifications. http://openmp.org/wp/

[7] Андон Ф.И., Дорошенко А.Е., Цейтлин Г.Е., Яценко Е.А. Алгеброалгоритмические модели и методы параллельного программирования. – К.: Академпериодика, 2007. – 631 с.

[8] Doroshenko A., Shevchenko R. A Rewriting Framework for Rule-Based Programming Dynamic Applications. *Fundamenta Informaticae,* Vol. 72, N 1–3: 95–108, 2006.

[9] TermWare. – http://www.gradsoft.com.ua/products/termware_rus.html

[10] Open Fortran Parser (OFP) http://fortran-parser.sourceforge.net/

[11] Intel Compilers. http://software.intel.com/en-us/articles/intel-compilers/.