Оглавление

1. Аннотация

2. Введение

2.1 Развитие вычислительной техники. SMP-кластеры.

2.2 Параллельное программирование

2.3 Модель параллелизма DVM/OpenMP

2.3.1 Преимущества DVM/OpenMP

2.4 Актуальность работы

3. Постановка задачи

3.1 Структура "Системы автоматизации распараллеливания"

3.2 Цель работы "DVM/OpenMP-эксперт"

4. Предыдущие решения "систем автоматизации распараллеливания на SMP-кластер"

4.1 Система Parawise

5. Исследование и построение решения задачи

5.1 Автоматическое распараллеливание программ на DVM и DVM/OpenMP

5.2 Структура DVM-эксперта

5.3 Структура DVM/OpenMP-эксперта

5.3.1 Варианты распараллеливания на OpenMP

6. Практическая реализация

6.1 Список используемых терминов

6.2 Блок поиска DVM/OpenMP-вариантов

6.2.1 Краткий алгоритм работы

6.2.2 Входные данные

6.2.3 Детальный алгоритм работы

6.2.4 Оценочная функция варианта распараллеливания гнезда циклов на DVM/OpenMP.

6.2.4.1 Оценка времени выполнения цикла, не распараллеленного на OpenMP.

6.2.4.2 Оценка времени выполнения параллельного цикла без конвейера

6.2.4.3 Оценка времени выполнения параллельного цикла с конвейером

6.2.4.4 Оценка времени выполнения гнезда циклов

6.3 Блок поиска наилучшего DVM/OpenMP-варианта

6.3.1 Характеристики эффективности параллельной программы

6.3.2 Алгоритм пересчета характеристик эффективности

6.4 Особенности реализации

6.4.1 Классы решаемых задач

6.4.2 Специальные комментарии.

6.4.3 Аргументы командной строки

6.5 Результаты тестирования

7. Заключение

8. Список цитируемой литературы

Приложение А. Графики времен выполнения и ускорений распараллеленных тестовых программ

# 1. Аннотация

Целью данной работы являлась разработка алгоритма преобразования последовательной программы на языке Fortran в параллельную программу на языке Fortran-DVM/OpenMP. Это преобразование осуществляет блок ("DVM/OpenMP-эксперт") экспериментальной системы автоматизации распараллеливания, используя результаты анализа информационной структуры последовательной программы.

К моменту начала работы над поставленной задачей система автоматизации распараллеливания содержала блок, именуемый "DVM-экспертом", который автоматизирует распараллеливание Fortran-программы на кластер. В рамках дипломной работы DVM-эксперт был доработан до DVM/OpenMP-эксперта, автоматизирующего распараллеливание Fortran-программ на SMP-кластер.

Разработанные алгоритмы были реализованы, включены в состав экспериментальной системы и проверены при распараллеливании тестовых примеров.

# Введение

## 

## Развитие вычислительной техники. SMP-кластеры

Вычислительная техника в своем развитии по пути повышения быстродействия ЭВМ приблизилась к физическим пределам. Время переключения электронных схем достигло долей наносекунды, а скорость распространения сигналов в линиях, связывающих элементы и узлы машины, ограничена значением 30 см/нс (скоростью света). Поэтому дальнейшее уменьшение времени переключения электронных схем не позволит существенно повысить производительность ЭВМ. В этих условиях требования практики (сложные физико-технические расчеты, многомерные экономико-математические модели и другие задачи) по дальнейшему повышению быстродействия ЭВМ могут быть удовлетворены только путем распространения принципа параллелизма на сами устройства обработки информации и создания многомашинных и многопроцессорных (мультипроцессорных) вычислительных систем. Такие системы позволяют распараллелить выполнение программы или одновременно выполнять несколько программ.

Последние годы во всем мире происходит бурное внедрение вычислительных кластеров. *Вычислительный кластер* – это *мультикомпьютер*, состоящий из множества отдельных компьютеров (*узлов*), связанных между собой единой коммуникационной системой. Каждый узел имеет свою локальную оперативную память. При этом общей физической оперативной памяти для узлов не существует. Если в качестве узлов используются *мультипроцессоры* (мультипроцессорные компьютеры с общей памятью), то такой кластер называется SMP-кластером.

Привлекательной чертой кластерных технологий является то, что они позволяют для достижения необходимой производительности объединять в единые вычислительные системы компьютеры самого разного типа, начиная от персональных компьютеров и заканчивая мощными суперкомпьютерами. Таким образом, кластеры являются легко-масштабируемыми. Широкое распространение кластерные технологии получили как средство создания систем суперкомпьютерного класса из составных частей массового производства, что значительно удешевляет стоимость вычислительной системы.

В то же самое время, для повышения производительности кластера, в качестве узлов нередко используются мультипроцессоры. Таким образом, теперь программист имеет дело с двумя уровнями параллелизма – параллельное выполнение задач на узлах кластера, и параллельное выполнение подзадач на ядрах мультипроцессора.

## Параллельное программирование

В настоящее время практически все параллельные программы для кластеров разрабатываются с использованием низкоуровневых средств передачи сообщений (MPI). Однако MPI-программы имеют ряд существенных недостатков:

* низкий уровень программирования и как следствие, высокая трудоемкость разработки программ;
* сложность выражения многоуровневого параллелизма программы;
* MPI-программы, как правило, неспособны эффективно выполняться на кластерах, у которых процессоры имеют разную производительность.

В то же время, на текущий момент так и нет и общепризнанного высокоуровневого языка параллельного программирования, позволяющего эффективно использовать возможности современных ЭВМ.

Таким образом, разработка программ для высокопроизводительных кластеров, и в особенности SMP-кластеров, продолжает оставаться исключительно сложным делом, доступным узкому кругу специалистов и крайне трудоемким даже для них.

Попытки разработать автоматически распараллеливающие компиляторы для параллельных ЭВМ с распределенной памятью, проведенные в 90-х годах (например, Paradigm, APC), привели к пониманию того, что полностью автоматическое распараллеливание для таких ЭВМ реальных производственных программ возможно только в очень редких случаях. В результате, исследования в области автоматического распараллеливания для параллельных ЭВМ с распределенной памятью практически были прекращены.

Исследователи сосредоточились на двух направлениях:

* разработка высокоуровневых языков параллельного программирования (HPF, OpenMP-языки, DVM-языки, CoArray Fortran, UPC, Titanium, Chapel, X10, Fortress);
* создание систем автоматизированного распараллеливания (CAPTools/Parawise, FORGE Magic/DM, BERT77), в которых программист активно вовлечен в процесс распараллеливания. [3]

Остановим свое внимание на высокоуровневом языке параллельного программирования DVM/OpenMP.

## Модель параллелизма DVM/OpenMP

В последнее время для вычислительных систем, сочетающих в себе одновременно характеристики архитектур, как с общей, так и с распределенной памятью, используется так называемая "гибридная" модель. При этом программа представляет собой систему взаимодействующих процессов, а каждый процесс программируется на OpenMP. Рассмотрим вычислительную сеть, каждый узел которой является мультипроцессором или отдельным процессором:



На первом этапе определяются массивы, которые могут быть распределены между узлами (распределенные данные). Эти массивы специфицируются DVM-директивами отображения данных. Остальные переменные (распределяемые по умолчанию) отображаются по одному экземпляру на каждый узел (размноженные данные). Распределение данных определяет множество локальных или собственных переменных для каждого узла. Распределив по узлам данные, пользователь должен обеспечить и распределение вычислений по узлам. При этом должно выполняться правило собственных вычислений: на каждом узле выполняются только те операторы присваивания, которые изменяют значения переменных, размещенных на данном узле. В свою очередь, на узле эти вычисления могут быть распределены между нитями средствами OpenMP.

Основным методом распределения вычислений по узлам является явное задание пользователем распределения между узлами витков цикла. При этом каждый виток такого цикла полностью выполняется на одном узле. Выполнение по правилу собственных вычислений операторов вне распределенного параллельного цикла обеспечивается автоматически компилятором, однако это требует существенных накладных расходов.

На следующем этапе необходимо организовать доступ к удаленным данным, которые могут потребоваться при вычислении значений собственных переменных. [4]

### Преимущества DVM/OpenMP

Во-первых, DVM/OpenMP является достаточно высокоуровневым языком параллельного программирования, что придает удобство процессу написания параллельных программ.

Во-вторых, DVM/OpenMP-программа – это последовательная программа, снабженная директивами-псевдокомментариями, задающими распределение данных и вычислений. Директивы не видны обычному компилятору. Таким образом программа, написанная на Fortran-DVM/OpenMP может использоваться сразу в четырех качествах:

* Последовательная программа на языке Fortran
* Параллельная Fortran-OpenMP программа для мультипроцессора
* Параллельная Fortran-DVM программа для кластера
* Параллельная Fortran-DVM/OpenMP программа для SMP-кластера

Соответственно, для однопроцессорной системы, мультипроцессора, кластера и SMP-кластера достаточно поддерживать всего одну версию программы.

## Актуальность работы

Работа посвящена написанию системы автоматизации распараллеливания программ. Автоматическое распараллеливание последовательных программ предполагает преобразование существующих последовательных программ в параллельный код. В качестве языка параллельного программирования выбран DVM/OpenMP.

Данное направление является весьма востребованным по следующим причинам:

* Распараллеливание готовой программы требует от программиста тщательного анализа ее кода (подчас, чужого), что делает процесс распараллеливания достаточно сложным, и подчас приводит к неудовлетворительным результатам. К тому же, в процессе распараллеливания могут быть допущены ошибки. Если бы распараллеливание осуществлялось автоматически, то появление новых ошибок было бы исключено.
* Распараллеливание уже готовых последовательных программ является очень востребованным, особенно в области научного программирования, где за многолетнюю историю жизни языка Fortran была написано огромное количество программ, не потерявших свою актуальность. Таким образом отпала бы необходимость писать все эти программы заново, но уже в параллельном варианте, и появилась бы возможность использовать старые наработки.
* Для различных конфигураций системы и разного размера входных данных, оптимальное распределение вычислений может также отличаться. Человек не всегда способен учесть такие факторы Автоматический распараллеливатель же учитывает конфигурацию кластера при формировании программы, и для нового кластера может формировать наиболее оптимальный код.

# Постановка задачи

## Структура "Системы автоматизации распараллеливания"

Данная дипломная работа посвящена разработке блока экспериментальной системы автоматизации распараллеливания, поэтому сначала ознакомимся с её структурой:

Параллельная программа с DVM/OpenMP рекомендациями

Последовательная программа

Пользователь

**program compute\_pi**

**parameter (n = 1000)**

**integer i**

**double precision w,x,sum,pi,f,a**

**f(a) = 4.d0/(1.d0+a\*a)**

**w = 1.0d0/n**

**…**

**end**



База данных

**program compute\_pi**

**parameter (n = 1000)**

**integer i**

**double precision**

**C$DVM PARALLEL**

**!$OMP PARALLEL DO**

**PRIVATE(x), SHARED(w)**

**w,x,sum,pi,f,a**

**f(a) = 4.d0/(1.d0+a\*a)**

**w = 1.0d0/n**

**…**

**end**

1

2

3

4

5

6

7

8

**Рисунок 1. Экспериментальная система автоматизации распараллеливания**

Пользователь создает последовательную программу на языке Fortran. Программа, поступая в систему автоматизации распараллеливания, проходит анализ, на основании которого формируется База данных. В Базу данных входят: дерево циклов; описания массивов, описание использования массивов в циклах; специальные пользовательские комментарии и прочее.

Пользуясь информацией из Базы данных, DVM/OpenMP-эксперт формирует варианты распределения вычислений и данных, и ищет наилучший вариант. Для поиска требуется информация о количестве вычислительных узлов, их производительности, а также латентности вычислительно сети. Далее, База данных подается на вход Генератору, который формирует параллельный код на языке Fortran-DVM/OpenMP. Код программы на выходе системы не изменяется, в него лишь добавляются директивы OpenMP и DVM.

## Цель работы "DVM/OpenMP-эксперт"

К моменту написания работы, Система автоматизации распараллеливания содержала блок, именуемый DVM-экспертом. DVM-эксперт, пользуясь результатами анализа программы, распараллеливает программу на языке Fortran-DVM. Целью дипломной работы является доработка DVM-эксперта до DVM/OpenMP-эксперта.

DVM/OpenMP-эксперт, пользуясь результатами анализа, распараллеливает программу на языке Fortran-DVM/OpenMP и заносит информацию о полученной параллельной программе в Базу данных.

# Предыдущие решения "систем автоматизации распараллеливания на SMP-кластер"

В настоящее время имеется только одна развивающаяся система автоматизированного распараллеливания для кластеров – Parawise (системы FORGE Magic/DM и BERT77 уже не развиваются и не поддерживаются). Только для нее имеется информация о применимости для реальных Fortran-приложений и эффективности выполнения распараллеленных программ.

## Система Parawise

Система Parawise является коммерческой системой, созданной компанией Parallel Software Products совместно с NASA Ames на базе системы CAPTools, разработанной в Лондонском университете Гринвича в середине 90-х годов. [3]

Общая схема процесса получения параллельной программы из последовательной состоит из следующих этапов:

* Системе Parawise подаётся на вход программа на языке FORTRAN (F77/F90/F95).
* Пользователь выбирает способы анализа данной программы.
* Parawise анализирует программу и формирует вопросы, на которые пользователь обязан дать ответ для успешного распараллеливания.
* Пользователь следует по построенному системой графу зависимостей, отвечает на поставленные вопросы и участвует в выборе варианта распределения массивов.
* Система создаёт параллельный код программы.
* Пользователь проверяет результаты распараллеливания, и, если они не удовлетворительные, заново отвечает на поставленные вопросы или изменяет программу. [7]

Главным достоинством системы Parawise являются развитые возможности межпроцедурного анализа, которые создавались в течение многих лет. Главный недостаток – программист должен сам принимать ответственное решение о том, какие массивы и каким образом требуется распределить между процессорами. [3]

В отличие от Parawise, DVM/OpenMP-экспертом будет использовать диалог с программистом только для уточнения свойств последовательной программы, а выбор наилучших решений по ее распараллеливанию осуществлять полностью автоматически, без участия программиста. К тому же, DVM/OpenMP-эксперт учитывает вычислительную мощность узлов кластера при распределении работы между процессорами.

# Исследование и построение решения задачи

## Автоматическое распараллеливание программ на DVM и DVM/OpenMP

При распараллеливании программы на DVM пользователю необходимо:

1. распределить массивы данных между узлами
2. распределить витки циклов между узлами по принципу собственных вычислений
3. организовать доступ к удаленным данным.

При распараллеливании на на DVM/OpenMP пользователю дополнительно следует выполнить еще один шаг:

1. распределить витки циклов, распределенные на узел, между ядрами на узле.

Отметим, что эти этапы можно выполнить различными способами. Полученные *варианты распараллеливания* будут иметь различную эффективность.

Рассмотрим общий алгоритм работы автоматического распараллеливателя Fortran-программ – DVM-эксперта. Пользуясь результатами анализа последовательной программы, DVM-эксперт формирует варианты распределения данных и вычислений Fortran-программы. Будем называть *DVM-вариантом* программу на языке Fortran-DVM, сгенерированную DVM-экспертом. DVM-вариантотражает один из способов распараллеливания последовательной программы на языке DVM. DVM-директивы несут для параллельного компилятора информацию о том, как следует выполнить пункты 1) – 3). Вариантов распараллеливания программы для одной Fortran-программы может быть несколько, поэтому среди всех возможных DVM-вариантов следует отыскать *наилучший*, и выдать его пользователю системы. Чтобы определить, какой из DVM-вариантов лучше, DVM-эксперт обращается к Библиотеке предсказателя производительности DVM-программ, сокращенно Библиотеке DVM-предиктора. DVM-предиктор моделирует параллельное выполнение DVM-программы и вычисляет характеристики эффективности параллельного выполнения DVM-программы. Полученные характеристики позволяют DVM-эксперту выбрать наилучший DVM-вариант.

DVM/OpenMP-эксперт является доработкой DVM-эксперта. После генерации вариантов распараллеливания программы на DVM, каждый DVM-вариант следует распараллелить на OpenMP – то есть выполнить пункт 4). Полученные DVM/OpenMP-программы будем называть *DVM/OpenMP-вариантами*.

## Структура DVM-эксперта

Для начала, рассмотрим структуру и принцип работы DVM-эксперта, существовавшего до начала работы.

База данных

Блок поиска DVM-вариантов

Блок поиска наилучшего DVM-варианта

Блок записи результатов в Базу данных

DVM-эксперт

номер

варианта

DVM-

варианты

**Рисунок 2. Схема работы DVM-эксперта**

База данных подается на вход Блоку поиска DVM-вариантов, который пользуется результатами анализа программы, и формирует варианты распараллеливания программы на DVM (DVM-варианты). Далее, DVM-варианты передаются Блоку поиска наилучшего DVM-варианта, который выбирает решетку процессоров и наилучший вариант распараллеливания программы. Затем Блок записи результатов в Базу данных записывает выбранный вариант распараллеливания в Базу данных.

Теперь рассмотрим, как следует изменить структуру DVM-эксперта, чтобы получить DVM/OpenMP-эксперт.

## Структура DVM/OpenMP-эксперта

База данных

Блок поиска DVM-вариантов

DVM-

варианты

Блок поиска наилучшего DVM/OpenMP-варианта

Блок записи результатов в Базу данных

DVM/OpenMP-эксперт

номер

варианта

Блок поиска DVM/OpenMP-вариантов

DVM/OpenMP

варианты

**Рисунок 3. Схема работы "DVM/OpenMP-эксперта"**

В DVM/OpenMP-эксперте DVM-варианты, сгенерированные Блоком поиска DVM-вариантов передаются Блоку поиска DVM/OpenMP-вариантов. Этот блок распараллеливает каждый DVM-вариант на OpenMP, выбирая при этом наиболее подходящий способ распараллеливания. Таким образом, из каждого DVM-варианта получается по одному DVM/OpenMP-варианту. Далее отрабатывает Блок поиска наилучшего DVM/OpenMP-варианта, который в качестве результата выдает наилучший DVM/OpenMP-вариант. Блок записи результатов в Базу данных записывает выбранный вариант распараллеливания в Базу данных.

Таким образом, следует разработать Блок поиска DVM/OpenMP-вариантов. Блок поиска наилучшего DVM-варианта следует доработать до Блока поиска наилучшего DVM/OpenMP-варианта. Также подлежит изменению Блок записи результатов в Базу данных. Остальные компоненты останутся без изменений.

### Варианты распараллеливания на OpenMP

Блок поиска DVM/OpenMP-вариантов получает на вход набор вариантов распараллеливания программы на кластере. Теперь нам требуется добавить еще один уровень параллелизма, и распределить работу, доставшуюся узлу кластера, между ядрами данного узла.

DVM-рекомендации позволяют распределить работу по выполнению циклов между узлами кластера. Сначала с помощью специальных DVM-директив между узлами кластера распределяются элементы массивов (блоками). Затем осуществляется распараллеливание циклов по принципу собственных вычислений. Если между витками цикла имеется зависимость по данным, цикл не распараллеливается. Исключение составляют редукционная зависимость и регулярная зависимость.

Рассмотрим несколько простых примеров.

**Пример 1**

Пусть DVM-эксперт распараллелил следующий цикл:

CDVM$ PARALLEL (i,j) ON a(i,j)

do i=1,N

do j=1,M

A(I, J) = B(J, I)

enddo

enddo

Директива CDVM$ PARALLEL (i,j) ON a(i,j) говорит о том, что виток цикла со значением итераторов (i, j) будет выполняться на то узле, на котором распределен элемента массива a(i, j). Существует два способа распараллелить на OpenMP этот многомерный цикл: распараллелить внутренний или внешний цикл.

**Вариант 1.1**

CDVM$ PARALLEL (i,j) ON a(i,j)

!$OMP PARALLEL PRIVATE(i, j)

!$OMP DO SCHEDULE (STATIC)

do i=1,N

do j=1,M

A(I, J) = B(J, I)

enddo

enddo

!$OMP END DO

!$OMP END PARALLEL

Директива !$OMP PARALLEL говорит о том, что на каждом узле начинается параллельная область. То есть на узле создаются нити, между которыми будет распределяться работа. Все порождённые нити исполняют один и тот же код, соответствующий параллельной области. Предполагается, что в SMP-системе нити будут распределены по различным ядрам процессора. Клауза PRIVATE(i, j) означает, что для каждой нити выделяется локальная память под две переменные: i и j. Действительно, у каждой нити должен быть свой итератор цикла. По достижении директивы !$OMP END PARALLEL нити останавливают свою работу, и на узле остается работать только одна мастер-нить.

Директива !$OMP DO сообщает нам, что далее в программе следует цикл, витки которого будут распределяться между нитями. Клауза SCHEDULE (STATIC) гласит, что все множество итераций внешнего цикла делится на непрерывные куски примерно одинакового размера, и полученные порции итераций распределяются между нитями.

Директива !$OMP END DO сообщает об окончании параллельного цикла. Происходит неявная барьерная синхронизация нитей и неявный вызов !$OMP FLUSH. Выполнение FLUSH предполагает, что значения всех переменных (или переменных из списка, если он задан), временно хранящиеся в регистрах и кэш-памяти текущей нити, будут занесены в основную память; все изменения переменных, сделанные нитью во время работы, станут видимы остальным нитям. [9]

**Вариант 1.2**

CDVM$ PARALLEL (i,j) ON a(i,j)

do i=1,N

!$OMP PARALLEL PRIVATE(i, j)

!$OMP DO SCHEDULE (STATIC)

do j=1,M

A(I, J) = B(J, I)

enddo

!$OMP END DO

!$OMP END PARALLEL

enddo

Отличие этого варианта от предыдущего заключается в том, на каждой итерации внешнего цикла будет создаваться новая параллельная область, то есть будут создаваться нити, и выделяться локальная память для них. Затем нити будут синхронизоваться, выполнять FLUSH и останавливаться. Всё это накладные расходы, которые будут возникать N раз. Еще одно отличие заключается в том, что в этом варианте между нитями распределяются витки внутреннего цикла, а не внешнего.

Другим важным критерием, помимо накладных расходов на вызовы библиотеки OpenMP, является загруженность нитей. Если рассматривать первый вариант, то N витков внешнего цикла сначала распределяются между узлами, а потом витки, соответствующие одному узлу, распределяются между ядрами этого узла. Если количество витков цикла меньше суммарного количества ядер нашего SMP-кластера, то некоторые ядра будут простаивать. Если некоторый цикл способен загрузить не более чем по одному ядру на каждом узле, то распараллеливать на OpenMP такой цикл смысла не имеет.

**Пример 2**

Пусть DVM-эксперт распараллелил то же самый цикл следующим образом:

CDVM$ PARALLEL (i,j) ON a(\*,j)

do i=1,N

do j=1,M

A(I, J) = B(J, I)

enddo

enddo

Отличие этого примера от предыдущего заключается в распределении данных. Здесь, между узлами одномерной процессорной решетки распределено только второе измерение массива A. Соответственно, между узлами будут распределяться только итерации внутреннего цикла. Таким образом, если количество витков внешнего цикла меньше суммарного количества ядер нашего SMP-кластера, но больше количества ядер на одном узле, то распараллеливание на OpenMP внешнего цикла позволит загрузить все ядра (заметим, что в варианте 1.1 на каждом узле работало бы не более одного ядра).

Для этого примера также существует два варианта распараллеливания, аналогичных вариантам 1.1 и 1.2.

**Пример 3**

Пусть DVM-эксперт распараллелил следующий цикл:

CDVM$ PARALLEL (i,j) ON a(i,j),

\*DVM$\* REDUCTION (SUM(s))

do i=1,N

do j=1,M

S = S + A(I, J)

enddo

enddo

Клауза REDUCTION (SUM(s)) означает, что каждый узел создаст в своей локальной памяти редукционную переменную, и будет накапливать в ней суммы элементов массива. По окончанию цикла эти локальные переменные будут просуммированы и записаны в s. В таком случае мы по-прежнему имеем два варианта распараллеливания:

|  |  |
| --- | --- |
| Вариант 3.1 | Вариант 3.2 |
| CDVM$ PARALLEL (i,j) ON a(i,j),  \*DVM$\* REDUCTION (SUM(s))  !$OMP PARALLEL PRIVATE(i, j)  !$OMP\*REDUCTION(+: s)  !$OMP DO SCHEDULE (STATIC)  do i=1,N  do j=1,M  S = S + A(I, J)  enddo  enddo  !$OMP END DO  !$OMP END PARALLEL | CDVM$ PARALLEL (i,j) ON a(i,j),  \*DVM$\* REDUCTION (SUM(s))  do i=1,N  !$OMP PARALLEL PRIVATE(i, j)  !$OMP\*REDUCTION(+: s)  !$OMP DO SCHEDULE (STATIC)  do j=1,M  S = S + A(I, J)  enddo  !$OMP END DO  !$OMP END PARALLEL  enddo |

Клауза REDUCTION(+: s) означает, что каждая нить создаст в своей локальной памяти редукционную переменную, и будет накапливать в ней суммы элементов массива. По окончанию цикла эти локальные редукционные переменные нитей будут просуммированы и записаны в локальную редукционную переменную узла.

В варианте 3.1 локальные редукционные переменные нитей будут суммироваться по окончании внешнего цикла. В варианте 3.2 локальные редукционные переменные нитей суммируются на каждой итерации внешнего цикла, и это снова дополнительные накладные расходы.

**Пример 4**

Рассмотрим пример с регулярной зависимостью:

CDVM$ PARALLEL (i,j) ON a(i,j),

\*DVM$\* ACROSS (a(1:1,1:1))

do i=2,N-1

do j=2,M-1

A(I, J) = A(I-1, J) + A(I+1, J)

\* + A(I, J-1) + A(I, J+1)

enddo

enddo

Тело данного цикла примечательно тем, что невозможно независимое выполнение витков цикла, т.к. прежде чем вычислить A(I, J), необходимо вычислить A(I-1, J) и A(I, J-1).

Прежде всего, клауза ACROSS (a(1:1,1:1)) определяет точное местоположение удаленных данных (теневые грани). Также ACROSS обеспечивает сохранение порядка вычислений витков цикла.

Для распараллеливания на OpenMP циклов с регулярной зависимостью используется алгоритм конвейерного выполнения при помощи синхронизующего массива. Этот алгоритм применялся в тестах NAS [12]. Цикл с регулярной зависимостью должен содержать тесно-вложенный цикл. Вариант распараллеливания здесь всего один:

**Вариант 4.1**

CDVM$ PARALLEL (i,j) ON a(i,j),

\*DVM$\* ACROSS (a(1:1,1:1))

!$OMP PARALLEL PRIVATE(IAM, NUMT, ILIMIT, i, j)

!$ IAM = omp\_get\_thread\_num ()

!$ NUMT = omp\_get\_num\_threads ()

!$ ISYNC (IAM) = 0

!$ ILIMIT=MIN(NUMT-1, N-3)

!$OMP BARRIER

do i=2,N-1

!$ IF (IAM .GT. 0 .AND. IAM .LE. ILIMIT) THEN

!$ DO WHILE (ISYNC(IAM-1) .EQ. 0)

!$OMP FLUSH (ISYNC)

!$ ENDDO

!$ ISYNC(IAM-1)=0

!$OMP FLUSH (ISYNC)

!$ ENDIF

!$OMP DO SCHEDULE (STATIC)

do j=2,M-1

A( I, J ) = A( I-1, J ) + A( I+1, J ) +

\* A( I, J-1 ) + A( I, J+1 )

enddo

!$OMP END DO NOWAIT

!$ IF (IAM .LT. ILIMIT) THEN

!$ DO WHILE (ISYNC (IAM) .EQ. 1)

!$OMP FLUSH (ISYNC)

!$ ENDDO

!$ ISYNC (IAM)=1

!$OMP FLUSH (ISYNC)

!$ ENDIF

enddo

!$OMP END PARALLEL

Предположим, что N = 7, M = 14, а количество нитей – 4. Принцип конвейерного выполнения отображен на рисунке.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Нить 1 | | | Нить 2 | | | Нить 3 | | | Нить 4 | | |
| Массив A | J = 2…4 | | | J = 5…7 | | | J = 8…10 | | | J = 11…13 | | |
| I = 2 | Такт 1 | | | Такт 2 | | | Такт 3 | | | Такт 4 | | |
|  |  |  |  |  |  |  |  |  |  |  |  |
| I = 3 | Такт 2 | | | Такт 3 | | | Такт 4 | | | Такт 5 | | |
|  |  |  |  |  |  |  |  |  |  |  |  |
| I = 4 | Такт 3 | | | Такт 4 | | | Такт 5 | | | Такт 6 | | |
|  |  |  |  |  |  |  |  |  |  |  |  |
| I = 5 | Такт 4 | | | Такт 5 | | | Такт 6 | | | Такт 7 | | |
|  |  |  |  |  |  |  |  |  |  |  |  |
| I = 6 | Такт 5 | | | Такт 6 | | | Такт 7 | | | Такт 8 | | |
|  |  |  |  |  |  |  |  |  |  |  |  |

**Рисунок 4. Иллюстрация принципа конвейерной работы**

**Такт 1**. Нить 1 выполняет три витка цикла: (I = 2, J = 2), (I = 2, J = 3), (I = 2, J = 4). Все остальные нити ждут. Таким образом, элементы массива A(2,2), A(2,3), A(2,4) получают новые значения.

**Такт 2**. Работают 1-я и 2-я нить. Нить 1 выполняет три витка цикла: (I = 3, J = 2), (I = 3, J = 3), (I = 3, J = 4). Отметим, что A(2, 2), A(2, 3) и A(2, 4), требуемые для вычисления A(I, J) для 1-й нити в текущем такте, уже содержат новое значение. Аналогично, нить 2 выполняет три витка: (I = 2, J = 5), (I = 2, J = 6), (I = 2, J = 7). Элемент A(2, 4), требуемый для вычисления A(2, 5), был уже посчитан 1-й нитью в 1-м такте.

**Такт 3**. Работают 1-я, 2-я и 3-я нить. Каждая нить выполняет по три витка. Для каждого элемента A(I, J), обрабатываемого в текущем такте, элементы A(I-1, J) и A(I, J-1) уже содержат новое значение.

**Такт 4, 5, 6, 7, 8** – аналогично.

Таким образом, порядок вычисления витков при параллельной работе нитей сохраняется.

В работе конвейера можно отметить три этапа:

* Разгон конвейера (Такты 1,2,3).
* Работа с полной загрузкой (Такты 4,5).
* Остановка конвейера (Такты 6,7,8).

**Пример 5**

CDVM$ PARALLEL (i,j,k) ON a(i,j,k),

\*DVM$\* ACROSS (a(0:0,1:1,0:0))

do i=1,N-1

do j=1,M

do k=1,P

A( I, J, K ) = (A( I, J, K ) + A( I, J, K ) +

\* A( I, J-1, K ) + A( I, J+1, K ))

enddo

enddo

enddo

Здесь регулярная зависимость присутствует только по второму измерению массива A. Соответственно, внешний или внутренний цикл можно распараллелить с помощью OMP PARALLEL DO, а для среднего можно организовать конвейер.

**Вариант 5.1**

CDVM$ PARALLEL (i,j,k) ON a(i,j,k),

\*DVM$\* ACROSS (a(0:0,1:1,0:0))

!$OMP PARALLEL PRIVATE(j, i, k)

!$OMP DO SCHEDULE (STATIC)

do i=1,N-1

do j=1,M

do k=1,P

A( I, J, K ) = (A( I, J, K ) + A( I, J, K ) +

\* A( I, J-1, K ) + A( I, J+1, K ))

enddo

enddo

enddo

!$OMP END DO

!$OMP END PARALLEL

**Вариант 5.2**

CDVM$ PARALLEL (i,j,k) ON a(i,j,k),

\*DVM$\* ACROSS (a(0:0,1:1,0:1))

do i=1,N-1

!$OMP PARALLEL PRIVATE(IAM, NUMT, ILIMIT, j, k)

!$ IAM = omp\_get\_thread\_num ()

!$ NUMT = omp\_get\_num\_threads ()

!$ ISYNC (IAM) = 0

!$ ILIMIT=MIN(NUMT-1, 11)

!$OMP BARRIER

do j=1,M

!$ IF (IAM .GT. 0 .AND. IAM .LE. ILIMIT) THEN

!$ DO WHILE (ISYNC(IAM-1) .EQ. 0)

!$OMP FLUSH (ISYNC)

!$ ENDDO

!$ ISYNC(IAM-1)=0

!$OMP FLUSH (ISYNC)

!$ ENDIF

!$OMP DO SCHEDULE (STATIC)

do k=1,P

A( I, J, K ) = (A( I, J, K ) + A( I, J, K ) +

\* A( I, J-1, K) + A( I, J+1, K ))

enddo

!$OMP END DO NOWAIT

!$ IF (IAM .LT. ILIMIT) THEN

!$ DO WHILE (ISYNC (IAM) .EQ. 1)

!$OMP FLUSH (ISYNC)

!$ ENDDO

!$ ISYNC (IAM)=1

!$OMP FLUSH (ISYNC)

!$ ENDIF

enddo

!$OMP END PARALLEL

Enddo

**Вариант 5.3**

CDVM$ PARALLEL (i,j,k) ON a(i,j,k),

\*DVM$\* ACROSS (a(0:0,1:1,0:0))

do i=1,N-1

do j=1,M

!$OMP PARALLEL PRIVATE(j, k)

!$OMP DO SCHEDULE (STATIC)

do k=1,P

A( I, J, K ) = (A( I, J, K ) + A( I, J, K ) +

\* A( I, J-1, K) + A( I, J+1, K ))

enddo

!$OMP END DO

!$OMP END PARALLEL

enddo

enddo

# Практическая реализация

## Список используемых терминов

**Тесно-вложенные циклы –** два цикла, расположенные таким образом, что один цикл является телом другого цикла.

**Гнездо циклов -** набор циклов, расположенных таким образом, что один цикл является телом другого цикла, гнездо циклов может состоять из одного цикла.

**Внешний цикл в гнезде –** цикл, который не вложен ни в один из циклов гнезда.

**Внутренний цикл в гнезде** – цикл, в который не вложен ни один из циклов гнезда.

**Гнездо циклов распараллелено на DVM –** витки циклов, принадлежащих гнезду, распределяются между узлами с помощью DVM-директив.

**Гнездо циклов распараллелено на OpenMP –** один из циклов гнезда распараллелен на OpenMP.

**Интервал –** некоторый участок выполнения программы. В дальнейшем нас будет интересовать только два вида интервалов: гнездо циклов и тело всей программы.

**Внутреннее представление DVM-эксперта** – набор структур, хранящих информацию о программе. DVM-эксперт заполняет эти структуры, пользуясь информацией из Базы данных.

**Специальные комментарий –** комментарий языка программирования, который пользователь может вставить в код программы. Специальный комментарий начинается с последовательности символов CPRG. Он сообщает о какой-нибудь особенности, относящейся к оператору, следующему после специального комментария.

## Блок поиска DVM/OpenMP-вариантов

### 

### Краткий алгоритм работы

Блок поиска DVM/OpenMP-вариантов распараллеливает на OpenMP каждый DVM-вариант. Результатом его работы являются DVM/OpenMP-варианты. При этом DVM/OpenMP-вариантов получается ровно столько же, сколько было DVM-вариантов.

Для каждого гнезда циклов, распараллеленного в DVM-варианте, Блок поиска DVM/OpenMP-вариантов выполняет следующее:

* Формирует варианты распараллеливания этого гнезда циклов на DVM/OpenMP.
* Оценивает время выполнения распараллеленного гнезда циклов в модели DVM/OpenMP.
* На основании оценки выбирает наилучший вариант распараллеливания гнезда циклов.
* Заносит OpenMP-директивы, распараллеливающие гнездо циклов выбранным способом, в DVM/OpenMP-вариант.

### Входные данные

Блок поиска DVM/OpenMP-вариантов работает не напрямую с Базой данных, а использует Внутреннее представление, сформированное DVM-экспертом. Нас будет интересовать следующая информация о программе:

* Описание SMP-кластера
  + количество узлов кластера
  + количество ядер, располагающихся на одном узле
* Описание программной единицы (модуля)
  + номер строки первого оператора
* Описание переменных (в том числе массивов):
  + текстовое представление
  + количество измерений массива (для скаляра – одно измерение)
  + количество элементов массива (для скаляра – один элемент)
* Описание циклов (отметим, что модуль программы также считается циклом – циклом верхнего уровня)
  + индексная переменная
  + начальное значение, конечное значение и шаг итератора цикла.
  + приближенное время выполнения одной итерации цикла
  + цикл, в который вложен данный цикл (родительский цикл)
  + признак тесно-вложенности данного цикла в родительский цикл
  + список циклов, вложенных в данный
  + номера строк, на которых располагаются операторы цикла
  + номер первой строки цикла (заголовка цикла)
  + список *особенностей* цикла.

Рассмотрим, какие цикл может иметь особенности:

* Цикл содержит приватные переменные. Существует три вида приватности: private, first\_private, last\_private.
* Цикл имеет редукционную зависимость (reduction).

Таким образом, *особенность* – это пара (<тип особенности>, <переменная>). Тип особенности может быть таким: private, first\_private, last\_private, reduction. Если тип особенности – reduction – то дополнительно хранится редукционная операция.

*Примечание:*

Эти особенности взаимно-однозначно соответствуют клаузам OpenMP, и подробнее о них можно прочитать в разделе *Специальные комментарии*.

Также отметим, что Внутреннее представление DVM-эксперта не содержит полную информацию о переменных с приватной особенностью. Так, индексные переменные циклов и скаляры, используемые в цикле на запись, не отмечены как приватные в особенностях циклов. Нам следует добавить эти особенности самостоятельно.

### 

### Детальный алгоритм работы

Рассмотрим более детально алгоритм работы Блока поиска DVM/OpenMP-вариантов. Его работу можно разбить на несколько этапов.

**Этап 1. Подготовительная работа**

**Шаг 1.1. Переход от DVM-вариантов к DVM/OpenMP-вариантам.**

Прежде всего, DVM-варианты следует перенести из внутреннего представления DVM-эксперта во внутреннее представление DVM/OpenMP-варианта. Внутреннее представление каждого DVM/OpenMP-варианта – это ассоциативный массив. Ключами ассоциативного массива являются номера строк программы, а значением - список DVM- и OpenMP-директив, которые следует вставить перед данной строкой. Соответственно, по окончанию данного этапа списки директив будут содержать только DVM-директивы.

Таким образом, из DVM-варианта мы получили DVM/OpenMP-вариант, в котором циклы распараллелены только на DVM. В дальнейшем мы будем добавлять в этот DVM/OpenMP-вариант директивы языка OpenMP.

**Шаг 1.2. Добавление приватных особенностей для индексных переменных**

При распараллеливании цикла на OpenMP, каждая нить должна иметь локальную копию индексной переменной. В списке особенностей цикла этот факт должен отражаться с помощью особенности типа private. Если цикл содержит вложенные циклы, то и их индексные переменные должны быть обозначены как private для данного цикла.

Во Внутреннем представлении DVM-эксперта информация об этих особенностях не отражена. Добавляем в описание каждого цикла private-особенность для индексных переменных этих циклов, а также всех вложенных циклов.

**Шаг 1.3 Добавление приватных особенностей для скаляров**

Предположим, в теле цикла в некоторую скалярную переменную происходит запись, и между витками цикла отсутствуют зависимости по данным. Если оставить эту переменную общей для всех нитей, мы получим условие гонок (race-condition). Чтобы избежать этой ситуации, в описание цикла следует добавить private-особенность для данной переменной.

Возможна ситуация, что эта переменная по окончанию цикла будет использована на чтение. В этом случае, для данной переменной в описании цикла уже должна была присутствовать last-особенность.

Подытожим: если в теле цикла в какую-нибудь скалярную переменную происходит запись, для неё в описание цикла следует добавить private-особенность.

**Шаг 1.4 Добавление all\_private особенностей**

Из Базы данных может поступить информация о том, что какая-либо переменная является приватной на протяжении всей работы программы. Для отражения этого факта существует специальный комментарий all\_private (см. раздел *Специальные комментарии*).

Чтобы добавить эту информацию в описания циклов, нужно сделать следующее:

* На этапе считывания особенностей из Базы данных запомнить список all\_private переменных
* Добавить private-особенности для полученного списка переменных в описание всех циклов программы.

**Этап 2. Распараллеливание DVM-вариантов на OpenMP**

Теперь мы будем поочередно обрабатывать каждый DVM/OpenMP-вариант, который пока что распараллелен только на DVM, и пытаться распараллелить его на OpenMP. Мы будем распараллеливать только те гнезда циклов, которые распараллелены на DVM.

Итак, для каждого гнезда циклов, распараллеленного на DVM в DVM/OpenMP-варианте, выполняем следующие шаги:

**Шаг 2.1. Сбор информации о циклах в гнезде.**

Просматриваем дерево циклов программы. Нас будут интересовать только те циклы, которые распараллелены в DVM-варианте. Для формирования DVM/OpenMP-варианта распараллеливания программы, необходимо собрать следующую информацию о гнезде циклов:

* Список циклов, принадлежащих гнезду. Первый элемент цикла – внешний цикл.
* Количество итераций каждого цикла из гнезда (если точное значение вычислить невозможно, выбирается значение по умолчанию).
* Время выполнения одной итерации каждого цикла из гнезда.
* Проанализировать DVM-директиву CDVM$ PARALLEL ON и определить
  + Как отображаются измерения массива на циклы из гнезда.
  + На какие циклы из гнезда не распределены измерения массива.
  + Присутствует ли в гнезде регулярная зависимость. Если да, то определить, для каких измерений массива присутствует регулярная зависимость, и каким из циклов в гнезде эти зависимости соответствуют.

*Примечание:*

Отметим, что информацию о регулярной зависимости мы не имеем возможности получить из внутреннего представления DVM-эксперта. Поэтому в текущей реализации о том, присутствуют ли эта зависимость в цикле или нет, мы узнаем посредством анализа DVM-директив в DVM-варианте.

Этот метод не может дать точного результата. Например, если все измерение массива распределено на узел, то ни клауза ACROSS, ни клауза SHADOW\_RENEW не смогут сообщить о наличии регулярной зависимости.

Если мы не можем достоверно определить, присутствует ли в цикле регулярная зависимость или отсутствует, мы предполагаем, что она там все-таки есть. Такое решение может привести к снижению эффективности – мы реализуем конвейерное выполнение там, где оно не всегда требуется – однако, это избавляет нас от ошибок распараллеливания.

**Шаг 2.2. Формирование вариантов распараллеливания гнезда.**

Этот шаг следует выполнить для каждого гнезда циклов, которое распараллелено в DVM-варианте.

Пробегаемся по списку циклов гнезда, и пытаемся поочередно распараллелить их на OpenMP. Пусть наше гнездо содержит M циклов. Пронумеруем циклы от 1 до M. Цикл с номером 1 – внешний. Первый вариант распараллеливания гнезда у нас уже есть - распараллелить внешний цикл на DVM, и не распараллеливать на OpenMP. Формируем еще M вариантов следующего вида: внешний цикл распараллелен на DVM, а i-й цикл гнезда распараллелен на OpenMP, где i принимает значения от 1 до М.

Возможны два способа для распараллеливания i-го цикла в гнезде:

Способ 1. Распараллеливание с использованием конвейера.

Если цикл соответствует измерению массива с регулярной зависимостью, для него невозможно независимое выполнение витков. Мы можем организовать конвейерное выполнение цикла при условии, что для него есть тесно-вложенный цикл (см. варианты 4.1 и 5.2). В противном случае, мы не будем распараллеливать этот цикл. Также, мы не сможем распараллелить цикл конвейером, если для записи этих двух циклов используется одна и та же метка.

|  |  |
| --- | --- |
| Циклы нельзя распараллелить конвейером | Циклы можно распараллелить конвейером |
| DO 21 J = 1, N  DO 21 I = 1, M  <body>  21 CONTINUE | DO J = 1, N  DO I = 1, M  <body>  ENDDO  ENDDO |

Такой эффект вызван особенностью используемого алгоритма распараллеливания конвейером.

Итак, если цикл соответствует измерению массива с регулярной зависимостью, имеет тесно-вложенный цикл и для записи этих циклов не используется одна и та же метка, мы ставим для этого цикла метку "Распараллелен конвейером".

Способ 2. Распараллеливание без конвейера*.*

Если цикл не соответствует измерению массива с регулярной зависимостью, этот цикл распараллеливается без конвейера (см. варианты 1.1, 1.2, 3.1, 3.2, 5.1 и 5.3). Ставим для этого цикла метку "Распараллелен без конвейера".

Если ни один из способов не подошел, i-й цикл мы распараллеливать не будем. Также, если количество итераций i-го цикла достаточно мало (каждому узлу достанется не более одного витка цикла, то есть работать на узле будет не более одного ядра), цикл признается неэффективным для распараллеливания на OpenMP, и соответствующий вариант отбрасывается.

**Шаг 2.3. Поиск наилучшего варианта распараллеливания гнезда.**

С помощью *оценочной функции* (о ней мы поговорим далее), мы вычисляем приближенное время выполнения каждого варианта на SMP-кластере с заданным количеством узлов и ядер. Располагая прогнозируемым временем выполнения каждого варианта распараллеливания гнезда циклов, мы выбираем вариант с наименьшим временем. При оценке учитываем, каким способом мы распараллеливаем цикл – с использованием конвейера, или без.

**Шаг 2.4. Вставка OpenMP-директив в DVM/OpenMP-вариант.**

Итак, мы выбрали, как лучше всего распараллелить гнездо циклов: либо распараллеливаем на OpenMP один из циклов гнезда, либо не распараллеливаем ни один из них. Если мы приняли решение распараллелить i-й цикл в гнезде, нам нужно вставить OpenMP-директивы в DVM/OpenMP-вариант. Возможны два случая:

Случай 1. i-й цикл имеет метку "Распараллелен с конвейером".

Выполняем следующие действия:

1. Если это первый цикл, распараллеленный конвейером в DVM/OpenMP-варианте, в начало программы добавляем описание служебных переменных. Если имена служебных переменных уже заняты, генерируем для уникальные имена. Перед первым оператором в программе вставляем:

!$ INTEGER omp\_get\_num\_threads, omp\_get\_thread\_num

!$ INTEGER IAM, NUMT, ISYNC(<количество нитей>), ILIMIT

1. Добавляем в описание i-го цикла три особенности приватного типа – для переменных IAM, NUMT и ILIMIT.
2. Формируем директиву !$OMP PARALLEL. Для распараллеливания на OpenMP цикла с особенностями после директивы требуется вставить клаузы. Пробегаемся по списку особенностей.
3. Если особенность имеет тип private, first\_private или last\_private, для нее формируем клаузу PRIVATE (<список переменных>), FIRSTPRIVATE (<список переменных>) или LASTPRIVATE (<список переменных>). Отметим, что в списке особенностей одна и та же переменная может иметь одновременно две разных особенности – private и first\_private (private и last\_private). В этом случае переменную следует занести только в клаузу FIRSTPRIVATE (LASTPRIVATE).
4. Если особенность имеет тип reduction, для нее формируем клаузу REDUCTION(<операция>: <список переменных>). Для разных операций создается отдельная клауза REDUCTION.
5. Обозначаем начало параллельной области. Перед заголовком i-го цикла вставляем директиву !$OMP PARALLEL с клаузами, сформированными в предыдущем пункте.
6. Сразу после !$OMP PARALLEL <список клауз> добавляем инициализацию служебных переменных и барьерную синхронизацию нитей:

!$ iam = omp\_get\_thread\_num ()

!$ numt = omp\_get\_num\_threads ()

!$ ILIMIT=MIN(NUMT-1,<число витков i-го цикла>)

!$ ISYNC (IAM) = 0

!$OMP BARRIER

1. Перед заголовком i+1-го цикла добавляем инициализацию конвейера: ядро дожидается, пока предыдущее ядро выполнит очередную итерацию i-го цикла, и только после этого

!$ IF (IAM .GT. 0 .AND. IAM .LE. ILIMIT) THEN

!$ DO WHILE (ISYNC (IAM-1) .EQ. 0)

!$OMP FLUSH (ISYNC)

!$ ENDDO

!$ ISYNC (IAM-1)=0

!$OMP FLUSH(ISYNC)

!$ ENDIF

!$OMP DO

1. Перед ENDDO i-го цикла добавляем операции по синхронизации работы ядер

!$OMP ENDDO NOWAIT

!$ IF (IAM .LT. ILIMIT) THEN

!$ DO WHILE (ISYNC (IAM) .EQ. 1)

!$OMP FLUSH (ISYNC)

!$ ENDDO

!$ ISYNC (IAM)=1

!$OMP FLUSH(ISYNC)

!$ ENDIF

1. После ENDDO i-го цикла обозначаем завершение параллельной области.

!$OMP END PARALLEL

Случай 2. i-й цикл имеет метку "Распараллелен без конвейера".

1. Формируем директиву !$OMP PARALLEL. Выполняем действия, описанные в пункте 3 предыдущего случая.
2. Обозначаем начало параллельной области и распределение витков цикла между ядрами. Вставляем сформированную директиву !$OMP PARALLEL с клаузами, а также директиву !$OMP DO SCHEDULE(STAITC), перед заголовком i-го цикла.
3. Обозначаем конец параллельного цикла и параллельной области. После ENDDO i-го цикла вставляем директивы:

!$OMP END DO

!$OMP END PARALLEL

Если наименьшее значение оценочной функции соответствует варианту, в котором мы распараллеливаем гнездо циклов только на DVM, ничего делать не нужно.

### 

### Оценочная функция варианта распараллеливания гнезда циклов на DVM/OpenMP.

Оценочная функция нужна для того, чтобы вычислить примерное время выполнения гнезда циклов, распараллеленного на DVM/OpenMP. Оценив время выполнения варианта распараллеливания гнезда циклов, мы сможем определить, какой из вариантов распараллеливания лучше.

Введем некоторые обозначения:

* Число\_узлов – это количество узлов кластера.
* Число\_ядер – это количество ядер на одном узле. Предполагаем, что на всех узлах одинаковое количество ядер.
* Число\_раб\_ядер – количество ядер на узле, которым достались витки параллельного цикла. Остальные ядра узла простаивают.
* Число\_редукционных\_переменных – количество редукционных переменных, находящихся в клаузе REDUCITON директивы !$OMP PARALLEL. Если клауза редукции отсутствует, то Число\_редукционных\_переменных равняется нулю.
* Ni – количество итераций i-го цикла. Напомним, что 1-й цикл в гнезде – внешний, а M-й – внутренний. Всего в гнезде M циклов.
* Блок итераций – итерации, которые достались некоторому ядру после выполнения конструкции разделения работы !$OMP DO. Если речь идет о конвейерном выполнения i-го и i+1-го цикла, то Блок итераций – это итерации i+1-го цикла, которые достанутся ядру на одной итерации i-го цикла.
* Число\_итераций\_блока – максимальное количество итераций в Блоке итераций. Отметим, что ядрам, которым достались работа, может быть распределено различное количество витков цикла. Так как мы занимаемся оценкой времени выполнения, нас будет интересовать сколько времени будет работать ядро, которому досталось больше всего работы, так как все остальные ядра при синхронизации будут ждать именно это ядро.
* ti – время выполнения одной итерации i-го цикла.
* ┌ A ┐ - округление дробного числа A в большую сторону.

Нам известно количество итераций каждого цикла и время выполнения одной итерации самого внутреннего цикла – тела цикла M. Обозначим это время за tm.

#### Оценка времени выполнения цикла, не распараллеленного на OpenMP.

Первый вариант распараллеливания гнезда, время которого нам нужно оценить,

CDVM$ PARALLEL … ON …

do IT1 = 1, N1

<…………>

enddo

При оценке мы берем в расчет только время, потраченное на вычисления. На один узел может максимум быть распределено ┌N1/Число\_узлов┐ итераций 1-го цикла. Таким образом, время параллельного выполнения цикла будет равняться ┌N1/Число\_узлов┐\* t1.

#### Оценка времени выполнения параллельного цикла без конвейера

Рассмотрим вариант распараллеливания гнезда циклов:

CDVM$ PARALLEL … ON …

do IT1 = 1, N1

<…………>

!$OMP PARALLEL PRIVATE(j)

!$OMP DO SCHEDULE (STATIC)

do ITi = 1, Ni

<………….>

do ITm = 1, Nm

<тело цикла m>

enddo

<…………>

enddo

!$OMP END DO

!$OMP END PARALLEL

<…………>

enddo

Здесь внешний цикл распараллелен на DVM, а i-й цикл – на OpenMP. При этом, i-й цикл не соответствует измерению массива с регулярной зависимостью, поэтому в организации конвейерного выполнения нет необходимости.

Посчитаем количество ядер, которым достались витки цикла i:

* Если итерации цикла распределятся между узлами кластера
  + Если Ni >= Число\_ядер \* Число\_узлов, то

Число\_раб\_ядер := Число\_ядер

* + Если Ni < Число\_ядер \* Число\_узлов, то

Число\_раб\_ядер := ┌Ni / Число\_узлов┐

* Если итерации цикла НЕ распределятся между узлами кластера
  + Если Ni >= Число\_ядер, то

Число\_раб\_ядер := Число\_ядер

* + Если Ni < Число\_ядер, то

Число\_раб\_ядер := Ni

Если Число\_раб\_ядер <= 1, i-й цикл является неэффективным для распараллеливания на OpenMP.

Посчитаем максимальное количество итераций цикла i, которое может достаться какому-нибудь из ядер SMP-кластера, следующим образом:

* Если итерации цикла распределятся между узлами кластера, то

Число\_итераций\_блока := ┌┌ Ni / Число\_узлов┐/ Число\_раб\_ядер┐

* Если итерации цикла НЕ распределяются между узлами кластера

Число\_итераций\_блока := ┌ Ni / Число\_раб\_ядер┐

Время параллельного выполнения i-го цикла (далее Время i-го цикла), складывается из нескольких составляющих:

* Время полезных вычислений
* Время барьерной синхронизации
* Накладные расходы на использование OpenMP (далее Расходы на OpenMP)
  + создание и удаление параллельной области, выделение памяти под локальные переменные. (далее Расходы на PARALLEL)
  + организация параллельного выполнения цикла (далее Расходы на DO)
  + применение клаузы REDUCTION (далее, Расходы на REDUCTION)

Время полезных вычислений := ti \* Число\_итераций\_блока

Исходим из эвристических предположений, что время барьерной синхронизации ядер, а также накладные расходы на использование OpenMP прямо пропорционально зависят от количества работающих ядер.

Введем следующие константы:

* CORE\_SYNC\_TIME – отражает накладные расходы на барьерную синхронизацию ядер,
* OMP\_PARALLEL\_OVERHEARDS – отражает накладные расходы на создание и удаление параллельной области,
* OMP\_DO\_OVERHEARDS – отражает накладные расходы на организацию параллельного выполнения цикла,
* OMP\_REDUCTION\_OVERHEARDS – отражает накладные расходы на применение клаузы REDUCTION.

Чтобы оценить временные расходы, возникающие при применение тех или иных возможностей OpenMP, нужно соответствующую константу умножить на количество работающих ядер. Накладные расходы на применение клаузы REDUCTION также зависят и от количества редукционных переменных. Таким образом:

Время барьерной синхронизации := CORE\_SYNC\_TIME \* Число\_раб\_ядер,

Расходы на PARALLEL := OMP\_PARALLEL\_OVERHEARDS \* Число\_раб\_ядер

Расходы на DO := OMP\_DO\_OVERHEARDS \* Число\_раб\_ядер

Расходы на REDUCTION := OMP\_REDUCTION\_OVERHEARDS \* Число\_редукционных\_переменных \* Число\_раб\_ядер

Расходы на OpenMP := Расходы на PARALLEL + Расходы на DO + Расходы на REDUCTION.

Просуммируем все составляющие, и получим Время вычисления:

Время i-го цикла := Время полезных вычислений + Время барьерной синхронизации + Расходы на OpenMP.

#### Оценка времени выполнения параллельного цикла с конвейером

CDVM$ PARALLEL … ON … ACROSS …

do IT1 = 1, N1

<…………>

!$OMP PARALLEL PRIVATE(IAM, NUMT, ILIMIT, i, j)

!$ IAM = omp\_get\_thread\_num ()

!$ NUMT = omp\_get\_num\_threads ()

!$ ISYNC (IAM) = 0

!$ ILIMIT=MIN(NUMT-1, Ni-1)

!$OMP BARRIER

do ITi = 1, Ni

!$ IF (IAM .GT. 0 .AND. IAM .LE. ILIMIT) THEN

!$ DO WHILE (ISYNC(IAM-1) .EQ. 0)

!$OMP FLUSH (ISYNC)

!$ ENDDO

!$ ISYNC(IAM-1)=0

!$OMP FLUSH (ISYNC)

!$ ENDIF

!$OMP DO SCHEDULE (STATIC)

do ITi+1 = 1, Ni+1

<………….>

do ITm = 1, Nm

<тело цикла m>

enddo

<…………>

enddo

!$OMP END DO NOWAIT

!$ IF (IAM .LT. ILIMIT) THEN

!$ DO WHILE (ISYNC (IAM) .EQ. 1)

!$OMP FLUSH (ISYNC)

!$ ENDDO

!$ ISYNC (IAM)=1

!$OMP FLUSH (ISYNC)

!$ ENDIF

enddo

!$OMP END PARALLEL

<…………>

enddo

Здесь внешний цикл распараллелен на DVM, а для i-го и i+1-го цикла организовано конвейерное выполнение на OpenMP. При этом, i-й цикл соответствует измерению массива с регулярной зависимостью.

Посчитаем количество ядер, которым достались витки цикла i+1:

* Если итерации цикла распределятся между узлами кластера
  + Если Ni+1 >= Число\_ядер \* Число\_узлов, то

Число\_раб\_ядер := Число\_ядер

* + Если Ni+1 < Число\_ядер \* Число\_узлов, то

Число\_раб\_ядер := ┌ Ni +1 / Число\_узлов┐

* Если итерации цикла НЕ распределятся между узлами кластера
  + Если Ni+1 >= Число\_ядер, то

Число\_раб\_ядер := Число\_ядер

* + Если Ni+1 < Число\_ядер, то

Число\_раб\_ядер := Ni

Если Число\_раб\_ядер <= 1, i-й цикл является неэффективным для распараллеливания на OpenMP.

Посчитаем максимальное количество итераций цикла i, которое может достаться какому-нибудь из ядер SMP-кластера, следующим образом:

* Если итерации цикла распределятся между узлами кластера, то

Число\_итераций\_блока := ┌┌Ni+1 / Число\_узлов┐/ Число\_раб\_ядер┐

* Если итерации цикла НЕ распределяются между узлами кластера

Число\_итераций\_блока := ┌Ni+1 / Число\_раб\_ядер┐

Время i-го цикла складывается из нескольких составляющих:

* Время полезных вычислений
* Время барьерной синхронизации
* Расходы на OpenMP:
  + Расходы на PARALLEL
  + Расходы на DO
  + Расходы на REDUCTION
  + Расходы на синхронизацию работы ядер в конвейере. Этой величиной мы будем пренебрегать, так как она очень мала.

Оценим Время полезных вычислений. Его можно разбить на три составляющие: Время разгона, Время полной загрузки, Время остановки.

Обратимся к Рисунку 5. На нем схематически иллюстрирована конвейерное выполнение. Кружочками обозначены Блоки итераций. По вертикали отмеряются значения итератора ITi, по горизонтали - ITi+1. Блоки итераций, соединенные сплошной линией, выполняются параллельно разными ядрами. Стрелочка означает зависимость выполнения одного Блока итераций от другого. То есть стрелочками обозначено, какие Блоки итераций обязаны быть выполнены прежде чем начнется выполняться Блок итераций, из которого исходит стрелка. Оранжевым цветом обозначены Блоки итераций, соответствующие разгону конвейера, голубым – полной загрузке конвейера, желтым - остановке конвейера.

Ядра 1 2 3 4 5 6 7

ITi+1

ITi

Ядра 1 2 3 4

ITi+1

ITi

- разгон конвейера

- работа с полной загрузкой

- остановка конвейера

Рисунок 5.1

Рисунок 5.2

Рисунок 5.3

Ядра 1 2 3 4 5 6 7 8

ITi+1

ITi

**Рисунок 5. Схематическая иллюстрация конвейерного выполнения**

Существует два случая:

1. Работающих ядер меньше количества итераций i-го цикла (Число\_раб\_ядер < Ni).

Этот случай иллюстрирован на Рисунки 5.2. Для разгона конвейера требуется выполнить Число\_раб\_ядер Блоков итераций. Далее конвейер работает с полной загрузкой (Ni - Число\_раб\_ядер) Блоков итераций. Остановка займет (Число\_раб\_ядер – 1) Блоков итераций. Таким образом:

Время разгона := Число\_раб\_ядер \* ti+1 \* Число\_итераций\_блока

Время полной загрузки := (Ni - Число\_раб\_ядер) \* ti+1 \* Число\_итераций\_блока

Время остановки := (Число\_раб\_ядер – 1) \* ti+1 \* Число\_итераций\_блока

1. Работающих ядер больше или равно количества итераций i-го цикла (Число\_раб\_ядер>= Ni)

Этот случай иллюстрирован на Рисунки 5.3. Для разгона конвейера требуется выполнить Ni Блоков итераций. Далее конвейер работает с полной загрузкой (Число\_раб\_ядер – Ni) Блоков итераций. Остановка займет (Ni – 1) Блоков итераций. Таким образом:

Время разгона := Ni \* ti+1 \* Число\_итераций\_блока

Время полной загрузки := (Число\_раб\_ядер - Ni) \* ti+1 \* Число\_итераций\_блока

Время остановки := (Ni – 1) \* ti+1 \* Число\_итераций\_блока

Теперь посчитаем Время полезных вычислений как сумму его составляющих. Отметим, что для обоих рассмотренных случаев получаем один и тот же результат.

Время полезных вычислений := Время разгона + Время полной загрузки + Время полной загрузки = (Ni - 1 + Число\_раб\_ядер) \* ti+1 \* Число\_итераций\_блока.

Время барьерной синхронизации, Расходы на PARALLEL и Расходы на REDUCTION рассчитываются также, как в пункте 6.2.2.1:

Время барьерной синхронизации := CORE\_SYNC\_TIME \* Число\_раб\_ядер,

Расходы на PARALLEL := OMP\_PARALLEL\_OVERHEARDS \* Число\_раб\_ядер

Расходы на REDUCTION := OMP\_REDUCTION\_OVERHEARDS \* Число\_редукционных\_переменных \* Число\_раб\_ядер

Отметим, что параллельный цикл создается Ni раз, поэтому

Расходы на DO := OMP\_DO\_OVERHEARDS \* Ni \* Число\_раб\_ядер

Расходы на OpenMP вычисляется как сумма всех составляющих:

Расходы на OpenMP := Расходы на PARALLEL + Расходы на DO + Расходы на REDUCTION + Расходы на синхронизацию.

Просуммируем все составляющие, и получим Время i-го цикла:

Время i-го цикла := Время полезных вычислений + Время барьерной синхронизации + Расходы на OpenMP.

#### Оценка времени выполнения гнезда циклов

Итак, мы оценили Время i-го цикла. Чтобы вычислить время выполнения всего гнезда циклов, это время следует умножить на количество раз, которое этот цикл выполняется. Таким образом, время выполнения гнезда равняется N1 \* N2 \* … \* Ni-1 \* Время i-го цикла.

## Блок поиска наилучшего DVM/OpenMP-варианта

DVM-

варианты

Характеристики

эффективности

DVM-вариантов

для всех циклов

Блок записи результатов в Базу данных

Блок поиска наилучшего DVM-варианта

Библиотека

DVM-предиктора

Блок перебора вариантов и поиска наилучшего

**Рисунок 6. Принцип работы "Блока поиска DVM-варианта"**

Для начала рассмотрим принцип работы Блока поиска DVM-варианта. DVM-варианты подаются на вход "Блоку перебора вариантов и поиска наилучшего" (далее, Блок перебора вариантов). Этот блок занимается поиском оптимальной решетки процессоров, а также наилучшего варианта распараллеливания.

Для поиска оптимальной решетки и наилучшего варианта, Блок перебора пользуется Библиотекой предсказателя производительности DVM-программ, сокращенно Библиотеке DVM-предиктора.

DVM-предиктор предназначен для анализа и отладки производительности DVM-программ без использования реальной параллельной машины (доступ к которой обычно ограничен или сложен). С помощью DVM-предиктора пользователь имеет возможность получить предсказанные временные характеристики выполнения его программы на MPP или кластере рабочих станций с различной степенью подробности. [10]

Библиотека DVM-предиктора позволяет получить характеристики эффективности DVM-варианта на некоторой решетке процессоров. Характеристики эффективности подсчитываются для каждого гнезда циклов, а также для всего варианта в целом. О том, какие бывают характеристики эффективности речь пойдет ниже.

Сначала Блок перебора ищет оптимальную решетку процессоров. Он подает на вход Библиотеке DVM-предиктора некоторый DVM-вариант на различных решетках. Сравнивая полученные характеристики эффективности, Блок перебора выбирает оптимальную решетку.

Далее, Блок перебора подает на вход Библиотеке предиктора DVM-варианты только на оптимальной решетке.

Блок перебора вариантов анализирует полученные характеристики эффективности, и выбирает наилучший вариант распараллеливания. Номер наилучшего варианта распараллеливания передается Блоку записи результатов в Базу данных.

Теперь рассмотрим, какие следует внести изменения, чтобы получить "Блок поиска наилучшего DVM/OpenMP-варианта".

DVM/OpenMP

варианты

Характеристики

эффективности

DVM-вариантов

для всех циклов

Блок записи результатов в Базу данных

Блок поиска наилучшего DVM-варианта

Библиотека

DVM-предиктора

Блок перебора вариантов и поиска наилучшего

Характеристики

эффективности

DVM/OpenMP-вариантов

для всех циклов

Блок пересчета DVM-характеристик

**Рисунок 7. Принцип работы "Блока поиска DVM/OpenMP-варианта"**

Характеристики эффективности DVM-вариантов для всех циклов, посчитанные с помощью Библиотеки DVM-предиктора, подаются на вход Блоку пересчета DVM-характеристик.

Этот блок изменяет характеристики эффективности с учетом того, что в узлах кластера теперь находятся мультипроцессоры.

### **Характеристики эффективности параллельной программы**

DVM-предиктор вычисляет достаточно много характеристик эффективности для интервала. В качестве интервала может выступать гнездо циклов, а также вся программа. Нам потребуется пересчитать только некоторые из этих характеристик:

* CPU\_time\_usr - полезное процессорное время – время, потраченное на выполнение вычислений в интервале.
* CPU\_time\_sys – полезное системное время – время, проведенное процессором в системной фазе (время, затраченное Lib-DVM).
* CPU\_time – полное процессорное время. Равняется сумме CPU\_time\_usr и CPU\_time\_sys.
* Execution\_time – время выполнения интервала**.**

Остальные характеристики пересчитываются вызовом специальных функций Библиотеки DVM-предиктора, поэтому мы их рассматривать не будем.

Указанные характеристики вычисляются для всех интервалов на каждом узле. Поядерные характеристики мы вычислять не будем.

Рассмотрим интервал, соответствующий гнезду циклов. Обозначим внешний цикл гнезда как L. Значение характеристики <Имя\_ характеристики > на этом интервале на i-м узле мы будем обозначать как <Имя\_характеристики> Li.Требуется пересчитать эти поузловые характеристики с учетом того, что в каждом узле работает по несколько ядер. Поядерные характеристики не высчитываются.

### Алгоритм пересчета характеристик эффективности

Алгоритм выполняет весьма приближенную оценку характеристик эффективности DVM/OpenMP-варианта, поэтому в дальнейшем нуждается в доработке.

Будем говорить, что *интервал распараллелен на OpenMP (или DVM),* если этот интервал соответствует гнезду циклов, распараллеленному на OpenMP (или DVM).

Поочередно обрабатываем каждый интервал. Нас будут интересовать только те интервалы, которые распараллелены на OpenMP. Пусть текущий интервал – это гнездо циклов, распараллеленное на OpenMP. Обозначим внешний цикл гнезда за L. Выполняем следующие действия.

Шаг 1. Находим *коэффициент ускорения*. Коэффициент ускорения отражает во сколько раз быстрее цикл будет вычисляться на SMP-кластере, нежели на кластере с тем же количеством узлов. То есть во сколько раз полезное процессорные время DVM/OpenMP-варианта меньше соответствующего DVM-варианта. Системные затраты в расчет не берутся.

Вычисляется коэффициент ускорения следующим образом. Находим максимальное количество итераций распараллеливаемого на OpenMP цикла, которое приходится на один узел.

Затем находим, какое максимальное количество итераций этого цикла может прийтись на одно ядро этого узла. Коэффициент ускорения равняется отношению первой величины на вторую.

Шаг 2. Пересчитываем поузловые характеристики эффективности текущего интервала

* Для каждого узла кластера пересчитываем полезное процессорное время выполнения гнезда циклов, поделив старое значение параметра CPU\_time\_usrLi на коэффициент ускорения.
* Вычисляем для каждого узла разность между старым и новым значение параметра CPU\_time\_usrLi. Обозначим эту величину для i-го узла как CPU\_usr\_diffLi.
* Пересчитываем полное процессорное время:

CPU\_timeLi := CPU\_time\_usrLi + CPU\_time\_sysLi.

* Пересчитываем время выполнения интервала. Execution\_timeLi складывается из полезного процессорного времени, системных затрат, простоев и пр. Мы предполагаем, что из всех этих характеристик изменяется только полезное процессорное время, и не учитываем системные затраты на использование OpenMP, а также простои ядер и время на их синхронизацию. Для каждого узла из Execution\_timeLi следует вычесть CPU\_usr\_diffLi.

Шаг 3. Пересчитываем характеристики эффективности интервалов, вложенных в текущий (гнезда циклов, вложенные, но не тесно-вложенные в текущее гнездо циклов). Для каждого вложенного интервала следует выполнить действия, описанные на Шаге 2.

Шаг 4. Пересчитываем характеристики эффективности интервалов, содержащих текущий интервал.

Пусть цикл L непосредственно вложен в цикл P. В качестве P может также выступать тело всей программы. Возможны два случая:

1. Цикл P является внешним для некоторого гнезда циклов или является телом всей программы. В этом случае, циклу P соответствует некоторый интервал, для которого DVM-предиктором подсчитаны характеристики эффективности. Эти характеристики следует пересчитать. Для каждого узла (i от 1 до Число\_узлов) выполняем следующие действия:

* CPU\_time\_usrPi := CPU\_time\_usrPi - CPU\_usr\_diffLi
* CPU\_timePi := CPU\_time\_usrPi + CPU\_usr\_sysPi
* Execution\_timePi := Execution\_timePi - CPU\_usr\_diffLi

1. Цикл P не является внешним для некоторого гнезда циклов.

Пусть Np – количество итераций цикла P. Для цикла P характеристики эффективности не вычислялись. Нам известно, что тело цикла P на i-м узле стало выполняться на CPU\_usr\_diffLi быстрее. Соответственно, весь цикл P будет выполняться на i-м узле быстрее на Np \* CPU\_usr\_diffLi. Отразим этот факт следующей записью:

CPU\_usr\_diffPi := Np \* CPU\_usr\_diffLi.

Если P является самым внешним циклом (телом всей программы), останавливаемся. В противном случае все действия, описанные в шаге 4, следует повторить для цикла, в который непосредственно вложен цикл P.

## Особенности реализации

### Классы решаемых задач

DVM/OpenMP-эксперт, имеет следующие ограничения:

* Входная программы должна быть написана на языке Фортран 77
* Входная программы не должна содержать процедуры или допускать их инлайн-подстановку.
* DVM/OpenMP-эксперт распараллеливает только циклы.
* На OpenMP распараллеливаются только те циклы, которые сумел распараллелить DVM-эксперт, а также циклы, тесно-вложенные в него.

### Специальные комментарии.

Текущая версия анализатора не предоставляет DVM/OpenMP эксперту информацию об особенностях цикла. Недостающую информацию должен сообщать пользователь, используя *специальные комментарии.* Рассмотрим их подробнее.

Если программист хочет задать дополнительную информацию о программе, он должен вставить в текст программы комментарий, начинающийся с последовательности символов CPRG. Всё, что идёт после этих символов записывается в Базу Данных в таблицу специальных комментариев. Если требуется прокомментировать свойства какого-нибудь цикла, комментарии вставляются перед заголовком цикла.

DVM/OpenMP-эксперт обрабатывает следующие специальные комментарии:

* private(<переменная>) – задаёт переменную, для которой порождается локальная копия в каждой нити; начальные значения локальных переменной не определено.
* private\_all(<переменная>) – задаёт переменную, которая будет private на протяжении всей программы.
* first\_private(<переменная>) – задаёт переменную, для которой порождается локальная копия в каждой нити; локальные копии переменных инициализируются значением этой переменной в нити-мастере;.
* last\_private(<переменная>) –переменной присваивается результат с последнего витка цикла;
* reduction(<переменная>(<оператор>)) - задаёт оператор и переменную; для переменной создаются локальные копии в каждой нити; локальные копии инициализируются соответственно типу оператора (для аддитивных операций – **0** или его аналоги, для мультипликативных операций – **1** или её аналоги); над локальными копиями переменной после выполнения всех операторов параллельной области выполняется заданный оператор [9]. В Таблице 1 отражено соответствие между текстовым представлением <оператор> и его аналогом на языке Fortran.

|  |  |
| --- | --- |
| <оператор> | Оператор языка Fortran |
| SUM | + |
| PRODUCT | \* |
| MAX | Max |
| MIN | Min |
| AND | .and. |
| OR | .or. |
| EQV | .eqv. |
| NEQV | .neqv. |

**Таблица 1. Редукционные операции.**

### Аргументы командной строки

Рассмотрим аргументы командной строки, доступные при использовании DVM/OpenMP-эксперта.

|  |  |
| --- | --- |
| Ключ | Описание |
| -omp | Отрабатывает только OpenMP-эксперт. На выходе получаем параллельную программу для SMP-системы на языке Fortran-OpenMP. |
| -dvm | Отрабатывает только DVM-эксперт. На выходе получаем параллельную программу для кластера на языке Fortran-DVM. |
| -omp -dvm | Если присутствуют оба ключа (или нет ни одного из этих ключей), отрабатывает DVM/OpenMP-эксперт. На выходе получаем параллельную программу для SMP-кластера на языке Fortran-DVM/OpenMP |
| -variants | Включается режим, в котором на выходе выдаются все варианты распараллеливания. Это могут быть DVM-варианты, OpenMP-варианты или DVM/OpenMP-варианты в зависимости от присутствия ключей  -omp и -dvm. |
| -nproc <число> | Задаем количество узлов кластера |
| -ncore <число> | Задаем количество ядер, находящихся на каждом из узлов |

**Таблица 2.Аргументы командной строки**

## Результаты тестирования

Тестирование анализатора проводилось на программах, реализующих решение уравнения Пуассона в трехмерном пространстве классическими итерационными методами: методом Якоби, методом последовательной верхней релаксации (SOR), методом красно-черного упорядочения (RedBlack), а также тесте, разработанном NASA Ames Research Center (LU). В тестах Якоби, SOR и RedBlack использовалась матрица размером 3000x3000.

DVM/OpenMP-варианты, полученные на указанных тестах с помощью DVM/OpenMP-эксперта, успешно прошли проверку на инструменте Intel Thread Checker 3.1. Intel Thread Checker – инструмент для отладки параллельных программ. Он осуществляет поиск мест с возможным недетерминированным поведением многопоточной программы, написанной как на основе библиотеки потоков (Windows или POSIX threads), так и с использованием технологии OpenMP. Intel Thread Checker рассматривает DVM/OpenMP-вариант как программу на языке Fortran-OpenMP. Отметим, что Intel Thread Checker не в состоянии проверить отсутствие ошибок в работе конвейерного алгоритма.

Тесты NAS являются самопроверяющимися, что позволяет убедиться в том, что распараллеленная программа выдает такой же результат, что и последовательная программа. Все сгенерированные DVM/OpenMP-варианты теста LU прошли проверку успешно.

Также отметим, что скорость работы DVM/OpenMP-эксперта почти не уступает скорости DVM-эксперта. Замедление в работе отражено в Таблице 3 на различных тестах.

|  |  |
| --- | --- |
| Название теста | Замедление работы эксперта |
| Якоби | 0,45% |
| SOR | 0,18% |
| RedBlack | 3% |
| LU | 0,0683% |

**Таблица 3. Замедление работы DVM/OpenMP-эксперта по сравнению с DVM-экспертом**

В Таблице 4 приведены времена выполнения DVM/OpenMP-вариантов для тестовых программ. В Таблице 5 – ускорения программ. Ускорение вычисляется как отношение времени выполнения теста на одном ядре ко времени выполнения распараллеленного теста на нескольких ядрах. Программы компилировались как Fortran-OpenMP-программы, и запускались на вычислительном комплексе IBM eServer pSeries 690 (Regatta).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество ядер | Времена выполнения тестовых программ, сек. | | | |
| Якоби | SOR | RedBlack | LU |
| 1 | 2,26 | 0,42 | 5,48 | 206,41 |
| 2 | 1,41 | 0,28 | 3,39 | 148,44 |
| 4 | 0,91 | 0,23 | 3,36 | 125,79 |
| 8 | 0,76 | 0,46 | 6,13 | 117,21 |

**Таблица 4. Времена запуска тестовых программ, в секундах**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество ядер | Ускорение тестовых программ | | | |
| Якоби | SOR | RedBlack | LU |
| 2 | 1,60 | 1,50 | 1,62 | 1,39 |
| 4 | 2,48 | 1,83 | 1,63 | 1,64 |
| 8 | 2,97 | 0,91 | 0,89 | 1,76 |

**Таблица 5. Ускорения, продемонстрированное на тестовых программах**

Графики временен выполнения и ускорений распараллеленных тестовых примеров можно найти в Приложении А.

**Заключение**

В рамках дипломной работы был реализована программа DVM/OpenMP-эксперт. Общий объем разработанного кода составил более 2000 строк на языке С++.

DVM/OpenMP-эксперт входит в состав экспериментальной системы автоматизации распараллеливания. За основу разработки был взят компонент системы, реализованный к моменту начала работы над поставленной задачей – DVM-эксперт.

Работа системы успешно прошла проверку инструментом Intel Thread Checker, а также самопроверяющимся тестом от NASA Ames Research Center. Также система была протестирована на Fotran-программах, реализующих следующие алгоритмы: классический алгоритм Якоби, алгоритм верхней релаксации (SOR), алгоритм решения системы линейных алгебраический уравнения методом Гаусса, алгоритм красно-черного упорядочения (RedBlack), а также на тесте NAS LU. Распараллеленные тесты продемонстрировали ускорение при запуске на SMP-системе.

# Приложение А. Графики времен выполнения и ускорений распараллеленных тестовых программ



**График 1.А График 1.Б**



**График 2.А График 2.Б**



**График 3.А График 4.Б**



**График 4.А График 4.Б**